TRYK: RECAU (06) 12 83 55

PB - 210 K.S. Thomsen: Inheritance for Termination Detection

ISSN 0105-8517

Inheritance Used to Factorize Distributed Termination Detection Algorithms

Kristine Stougård Thomsen

DAIMI PB - 210
April 1986

DATALOGISK AFDELING
Bygning 540 - Ny Munkegade - 8000 Aarhus C
tlf. (06) 12 83 55, telex 64767 aausci dk
Matematisk Institut Aarhus Universitet

Abstract

A multiple inheritance mechanism on processes is introduced. Processes are described in classes, and the different action parts of a process inherited from different classes are executed in a coroutine-like style called alternation.

The inheritance mechanism is a useful tool for factorizing the description of common aspects of processes. This is demonstrated within the domain of distributed programming by using the inheritance mechanism to factorize the description of distributed termination detection algorithms from the description of the distributed main computations for which termination is to be detected.

The factorization is obtained by programming the termination detection algorithms in separate classes. The main computations are programmed in classes that use appropriate termination detection classes as superclasses. A clear separation of concerns is obtained, and arbitrary combinations of termination detection algorithms and main computations can be formed.

The same termination detection classes can also be used for more general purposes within distributed programming, such as detecting termination of each phase in a multi-phase main computation.

1 Introduction

In [15] a multiple inheritance mechanism on processes was introduced. The idea is that not only data but also processes can be organized in an inheritance hierarchy like the subclass hierarchies of Simula [3], Smalltalk [7] and Beta [10].

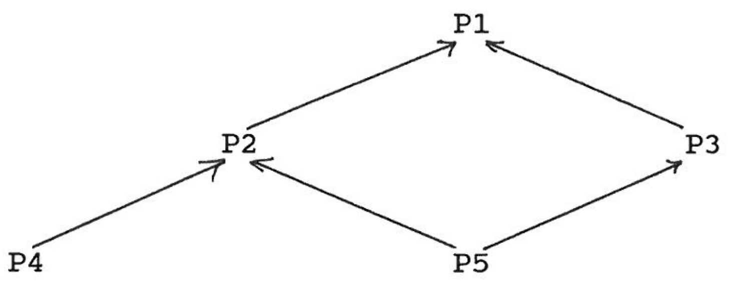

fig. 1

The process P5 in fig. 1 inherits declarations and action parts from P1, P2 and P3. Execution of a P5 process implies execution of four different action parts in a coroutine like style called alternation.

The most important advantage of inheritance on processes is that it allows factorization of common process parts that can be described once and used by different subclasses. Inheritance makes it possible to separate different concerns and thus simplify the involved algorithms.

In [15], the examples used to illustrate the use of inheritance with alternation on processes were mainly chosen from the database area. The purpose of this paper is to show some larger examples from another application area, namely distributed programming.

When programming a distributed algorithm, it is often so that the algorithm can be more clearly and simply formulated when ignoring how the processes should agree to finish the computation or different phases of it. It then remains to detect when certain kinds of stable states in the distributed system are reached. On detection of a stable state, the appropriate reaction may be just to terminate, or to report some state of the system, and/or to initiate a new phase of computation.

A number of stability detection algorithms, often called termination detection algorithms, have been developed for detection of certain kinds of stability in a distributed system. They are general in the sense that they can be expressed independently of the main computation for which they detect stability.

We will talk about "termination detection" because the algorithms discussed in this paper are presented by their authors as termination detection algorithms, although they detect a certain kind of stability which could cause other reactions than termination, e.g. initiation of a new computational phase.

Since the main computation and the termination detection algorithm can be conceptually separated, it is desirable to have language constructs that make it possible to program them separately, instead of integrating them textually. Such a separation will make the conceptual structure of the algorithms more clear. Moreover, it should make it possible to use the same termination detection algorithm in different main computations or vice versa, without duplicating the description of either of them.

We will show that such separation of concerns can be obtained through inheritance with alternation on processes. The coroutine like alternation mechanism fits nicely with the nature of the interaction between a termination detection algorithm and a main computation.

Organization of the paper

The reader is expected to be familiar with object oriented languages like Simula [3] and Smalltalk [7]. Hopefully, however, people whose main interest is in distributed programming could also read the paper with profit.

In section 2, we will present the distributed termination problem in detail and discuss the overall idea of using inheritance for describing termination algorithms.

In section 3, the termination detection algorithm by Dijkstra, Feien and Van Gasteren [4] is presented and programmed by means of the language outlined in Appendix A. A distributed algorithm computing the shortest paths in a graph is programmed to illustrate the use of the termination detection algorithm. The algorithm by Dijkstra, Feien and Van Gasteren has been chosen because it is intuitively simple and easy to explain also to the reader who is unfamiliar with distributed programming.

Section 4 contains brief presentations and realizations of other termination detection algorithms, to convince the reader who is familiar with distributed programming that the technique of using inheritance with alternation is applicable to a big class of termination detection algorithms. This section could be skipped by the reader who is only interested in the main points of the paper.

In section 5 we evaluate the results obtained and discuss some more general applications of inheritance. Moreover, we comment on future language design for the purpose.

Appendix A outlines a language with inheritance on processes. The language serves as a framework for presenting inheritance on processes and is in most other respects very rudimentary.

2 Factorization of Termination Detection

In this section we will present the problem of detecting termination in a distributed system, and we will show the overall idea of using inheritance to separate the description of the termination detection and the main computation.

2.1 Termination Detection

The classical way of presenting the problem of distributed termination detection is as follows:

We have a distributed system consisting of a finite set of processes and a set of communication channels each connecting two distinct processes. A distributed computation called the main computation is performed in the system. Each process carries out some local computation and can communicate with neighbour processes by means of asynchronous message passing. The messages related to the main computation are called primary messages. Communication channels are assumed to be error free and deliver messages in the order sent, and message buffers are assumed to be infinite. From the point of view of the main computation, a process is at any moment either active or passive. Only active processes send primary messages. Reception of a primary message makes a process change from passive to active. An active process may become passive spontaneously, whereas passive processes remain passive until they receive a primary message.

The state in which all processes are passive from the main computations point of view, and no primary messages are in transit is a stable state. Often the system is intended to terminate when such a stability is reached. The problem of detecting this kind of stability is therefore usually called termination detection, although other reactions than termination are possible. We will adopt the conventional terminology in the following.

Termination should be detected by a distributed algorithm based on control messages sent between the processes. A process can participate in the termination detection algorithm when it is passive from the main computation's point of view.

2.2 Common Structure of Termination Detection Algorithms

Many different algorithms have been developed for this purpose, e.g. [4], [6], [12], [16]. Most of them assume that the main computation is programmed in some high level language similar to CSP [9] with guarded commands. The typical structure of the termination detection algorithms is that each process administrates some status variables that are updated each time the process receives and/or sends a primary message. Moreover, a control communication part consisting of a number of guarded commands is superimposed on the main computation of each process in such a way that the control communication part is only active when the main computation is passive.

Franzec and Rodeh [6] explicitly describe how their algorithm can be incorporated into a main computation by means of a series of textual changes to the main computation algorithm (provided that the main computation has a specific structure).

The technique of textually modifying the main computation is not very appealing from a programming methodological point of view, where abstraction and separation of concerns are important principles. The termination detection algorithm can be used for many different main algorithms, so the termination detection algorithm should be factorized out by means of an abstraction which could be used in different contexts. By separating the description of the main algorithm and the termination detection algorithm, the structure of each could be easily comprehended without being obscured by irrelevant details conceptually belonging to the other algorithm.

The point of this paper is that the inheritance mechanism for processes presented in the next section and Appendix A is a high level language construct that is well suited for this purpose.

2.3 A Small Language With Inheritance

A small language with inheritance on processes is outlined in Appendix A. The language is based on classes that contain declarations and a statement list called an action part. The most important statements are assignment statements, communication statements for asynchronous message passing (denoted by ? and !), and alternative and repetitive guarded commands (denoted if...fi and do...od).

Classes are organized in a subclass hierarchy (multiple inheritance hierarchy) such that a subclass inherits all properties from its superclasses.

An instance of a class is an autonomous process that executes in parallel with other processes. A process has all the properties described in its class and its superclasses, including an action part from each class. The different action parts of a process are executed alternately such that only one action part is active at a time, and control alternates between them at specific changeover points denoted by "*". Each changeover point specifies a resumption condition that describes under what circumstances the action part can be resumed for execution.

As a starting convention all the action parts of a process are executed until their first changeover point in a top down sequence in the inheritance hierarchy. After this initial activity, the alternation mechanism works as follows: When an active action part reaches a changeover point, control is transferred to an action part with satisfied resumption condition and highest priority (possibly the currently active action part). As a default convention, priorities are associated with classes in a bottom-up manner such that the action part of a subclass has higher priority than the action parts of its superclasses.

2.4 Use of Inheritance in Termination Detection

The inheritance mechanism suits the problem of factorizing termination detection algorithms from main computations in that it allows the termination detection algorithm to be programmed in a class (actually two classes) and the main computation to be programmed as a subclass of this class. See fig. 2.

```
 Termination Class:
-------------------------
| Status variables        |
|                         |
| Abstractions for        |
|   communication         |
|                         |
| Control communication   |
|   action part           |
-------------------------
            |
            |
            |
 Main Computation Class:
-------------------------
| Declarations            |
|                         |
| Main computation with   |
|   use of abstractions   |
-------------------------
```

fig. 2

The actual distributed processes will be instances of the main computation subclass, implying that they each have all the properties described in both classes, including an action part for the main computation and an action part for the termination detection algorithm. The termination class describes the status variables, and its action part constitutes the control communication part of the termination detection algorithm (see section 2.2). Moreover, the termination class defines some abstractions - procedures or guarded command abstractions - that enforces the preconditions (if any) for primary communication, enforces a changeover point when no message is available (with resumption condition "message available"), and updates the status variables after the communication. These abstractions are inherited by the main computation subclass and are used for all primary communication. Thus in a process that is an instance of the main computation class, the two action parts will alternate in such a way that the main computation has highest priority but gives control to the termination detection algorithm when it is waiting for a message.

3 A Termination Detection Algorithm

In section 3.1, the distributed termination detection algorithm developed by Dijkstra, Feien and Van Gasteren [4] will be described. We will use the name DFVG for the algorithm. In section 3.2 the algorithm is realized by means of the language that was briefly introduced in section 2.3. The language is described in more detail in Appendix A. Section 3.3 describes an example of use of the realized algorithm.

3.1 Presentation of DFVG

In addition to the assumptions already mentioned in the general problem formulation, the DFVG algorithm assumes that the processes are connected in a ring with respect to control communication, that consists of simple black and white tokens. That is, the processes can be numbered such that P0 can send tokens to Pn-1 and Pi can send tokens to Pi-1 for i=1..n-1. Moreover, message passing is assumed to be instantaneous such that when a message is sent, it arrives immediately at the buffer of the recipient. The consequence of this assumption is that termination can be concluded when knowing that all processes are passive, since there are never any messages in transit between two processes.

The DFVG algorithm consists of a number of probes started by P0. The processes and the token that constitutes the probe have colours black or white, and the token is circulated according to the following rules (taken directly from [4]):

- Rule 1:
  A process sending a primary message makes itself black.
- Rule 2:
  P0 initiates a probe by making itself white and sending a white token to Pn-1
- Rule 3:
  When active, Pi+1 keeps the token; when passive, it hands

over the token to Pi.

- Rule 4:

  When Pi+1 propagates the probe, it hands over a black token if it is black itself, whereas being white it leaves the colour of the token unchanged.

- Rule 5:

  Upon transmission of the token to Pi, Pi+1 becomes white. (Its original colour may have influenced the colour of the token because of rule 4.)

- Rule 6:

  After the completion of an unsuccessful probe, P0 initiates a next probe.

When P0 receives a token that has remained white all the way round the ring, and P0 itself is white, it can conclude that no processes have sent any primary messages after the previous (unsuccessful) probe. Since message passing is instantaneous, there can be no messages in transit. Thus all processes are passive, and termination has occurred. On the other hand, if the token has become black when received by P0, some other process may be active and termination cannot be concluded. P0 then initiates a new probe.

In [4] the algorithm is proven to be correct provided the above 6 rules are obeyed.

In [4] the spreading of the knowledge of termination from P0 to the other processes is ignored. For completeness we include this spreading and realize it as a special termination wave initiated by P0 after P0 has detected termination. The wave is passed all the way round the ring to inform all processes that termination has been detected.

3.2 Realization of DFVG

The asymmetry between processes - i.e. the difference between the behaviour of P0 and the other processes - makes it necessary to realize DFVG by means of two different classes, DFVG-0 for P0 and DFVG-i for Pi, i<>0. However, these two classes have some common declarations that we can factorize into a common superclass DFVG. The three classes are shown in fig. 3. Some remarks are included in the following concerning the individual classes and the interaction between termination classes and main computation classes. However, the reader who wants a deeper understanding of the realization must consult appendix A.

DFVG contains the status variables "colour" (the colour of the process) and "token" (the colour of the token). Moreover, two abstractions, "Sendmsg" and "Receivemsg" are declared. Receivemsg enforces a changeover point with resumption condition "appropriate message available" in the action part that invokes it. Sendmsg updates the status variables by making the process black. Sendmsg and Receivemsg are intended to be used to program all primary communications in the main computation. Note that Receivemsg is a guarded command abstraction and may be used in if and do commands just as the primitive "?" command, which can be considered overloaded by Receivemsg in the subclasses.

DFVG-0 consists only of an action part with a cycle in which P0 sends a white token to its left neighbour in the ring and waits (at a changeover point) until the token is returned from its right neighbour. If both the colour of the token and the colour of the process are white, termination has been detected. When termination is detected, a termination wave is sent to the left. When it has been all the way round the ring, the procedure "Termination" is called. Procedure "Termination" is incompletely specified, and further specification is deferred until the main computation subclass. The main computation subclass determines in its specification of the procedure "Termination" what should be the consequence of the termination detection. Thus the "Termination" procedure provides the means for spreading the knowledge of the termination detection from the termination action part of a process to the main computation action part of the same process.

```
class DFVG (left,right: DFVG);

  Type Control = (black, white);
  Type T = (t); {termination wave}

  var colour, token: Control;
      term: T;

  Procedure Termination;
  {deferred specification}

  Guard-cmd Receivemsg (in P: DFVG ; out c: Simple);
     * P ? c -> skip
  end;

  Procedure Sendmsg (in P: DFVG ; in c: Simple);
  begin
    P ! c;
    colour:= black
  end;

  action

end;


class DFVG-0 is-a DFVG;

  action

    do true -> colour:= white
               left ! white;
               * right ? token;
               if (token=white) and (colour=white) ->
                      {termination detected; send termination wave}
                      left ! t;
                      right ? term;
                      Termination
               fi
    od
end;


class DFVG-i is-a DFVG;

  action

    colour:= white;
    do *
       right ? token -> if (colour=black) ->
                           token:= black
                        fi;
                        left ! token;
                        colour:= white

     / right ? term  -> {termination wave received}
                        left ! term;
                        Termination
    od
end;
```

fig. 3

DFVG-i also contains an action part with a cycle in which the process receives a token or a termination wave from its right. In case of a token, it passes it on to its left according to the colouring rule - i.e. making the token black if the process is black. In case of a termination wave, it passes it on and calls the procedure "Termination", which serves the same purpose here as described for DFVG-0. The "*" after "do" specifies a changeover point with resumption condition "some guard is open" - in this case "either a token or a termination wave has arrived".

Note that the purpose of the types Control and T is to avoid confusion between control messages and primary messages. The DFVG-0 or DFVG-i action part takes care of arrivals of tokens and termination waves, whereas the main computation action part is to take care of arrival of primary messages. "Token" is not assignable to a boolean or vice versa, and thus arrival of a token can always be distinguished from a boolean primary message, provided that the type Control is not used by the main computation. Similarly for the termination wave, which moreover cannot be confused with the token.

To see how the termination action part interacts with the main computation action part in a process, we assume that we have a specific main computation that we want to combine with the DFVG termination detection algorithm. For simplicity, we assume that the main computation also has a leader process and a number of symmetric processes. We then program two classes, a leader class as a subclass of DFVG-0 and a non-leader class as a subclass of DFVG-i. The class hierarchy is shown in fig.4.

$$\begin{tikzcd}
 & \text{DFVG} & \\
\text{DFVG-0} \arrow[ur] & & \text{DFVG-i} \arrow[ul] \\
\text{Leader} \arrow[u] & & \text{Non-Leader} \arrow[u]
\end{tikzcd}$$

fig. 4

Class Leader and Non-Leader each contain declarations and an action part related to the main computation. The main computation classes Leader and Non-Leader are assumed to obey the following disciplinary rules (which could be enforced by the language):

1) All primary communication is programmed by means of the inherited abstractions "Sendmsg" and "Receivemsg".
2) None of the status variables from the termination detection class are touched.
3) The type declarations in the termination class are not used.

The distributed computation will consist of an instance of Leader and a number of instances of Non-Leader. Each instance will have actual parameters denoting their left and right neighbours in the ring. (We ignore how the circularity is established.)

If we consider an instance of Leader or Non-Leader, the top-down starting convention for action execution ensures that the process is cleaned by the DFVG-0 or DFVG-i action part as the first thing. After this, the bottom-up priorities and the use of the inherited Receivemsg ensures that the Leader or Non-Leader action part is active, and that the DFVG-0 or DFVG-i action part receives control only when the main computation is passive. In an instance of Non-Leader, the DFVG-i part will execute one iteration of its loop indivisibly (provided a token or a termination wave has arrived) and then it will be ready to give back the control to the main computation in case a primary message has arrived in the meantime. In an instance of Leader, the DFVG-0 part behaves similarly, except that its changeover point is placed in the middle of the loop.

To argue more formally that the realization is correct, we can argue that the rules 1 to 6 for process behaviour in section 3.1 are satisfied:

The disciplinary rule 1 for the main computation, described above, ensures that sending of a primary message is done by means of the abstraction Sendmsg which makes the process black. Thus rule 1 is obeyed.
Rule 2 follows immediately from the action part of DFVG-0 which receives control when the Leader action part reaches a changeover point - e.g. in connection with waiting for a primary message, or after reaching its final end (where there is always an implicit changeover point).
Rule 3: Because of the bottom-up priorities, the main computation part has control when active, so the DFVG-i part only gets control when the main computation is passive. If a token has arrived, the resumption condition of DFVG-i is satisfied and the token is received and handed over to the left, according to the colouring rule corresponding to rule 4. After sending the token, the process becomes white and thus satisfies rule 5. (Then a changeover point is reached and control may return to the main computation.)
Rule 6 is satisfied since the DFVG-0 part of P0 starts over again when an unsuccessful probe is completed - i.e. when a token is received from the right and termination cannot be concluded.

3.3 Use of DFVG in Shortest Paths Algorithm

In this section, we will show a complete example of a main computation in a distributed system. The example illustrates the separation of concerns obtained by factorizing the termination detection algorithm into a superclass.

The example is a distributed graph algorithm that computes the lenghts of the shortest paths from a special vertex v0 to all other vertices vi, i<>0, in a weighted oriented graph. The graph is assumed to have no cycles with negative total length.

More formally, the problem can be stated as follows: A graph is a tuple (V,E), where V is a set of vertices and E is a set of weighted edges each connecting two vertices. If (vi,vj,w) belongs to E, there is a simple path of length w from vertex vi to vertex vj. In general, there is a path from vertex vi to vertex vj of length L if there is a sequence of edges in E (vi,vi1,w1),(vi1,vi2,w2),...,(vin,vj,wn+1) and the sum of the weights on the edges, w1 + w2 + ... + wn+1 is L. Given a special vertex, v0, the shortests paths problem is for each vertex vi, i<>0, to find the shortest path from v0 to vi.

The algorithm consists of one process for each vertex in the graph. Each process knows its successors and predecessors in the graph and the weights on all outgoing edges. P0 corresponds to the special vertex v0 and is the leader process in the computation. P0 starts by sending a message to each of its successors containing the weight on the edge from P0 to the successor. Each Pi process (i<>0) behaves as an iterative process that repeatedly improves its current estimate of the shortest path from P0 to Pi by minimizing the lengths of paths that go via predecessors to Pi. That is, Pi accepts messages from all predecessors Pj denoting estimated pathlengths from P0 to Pj. Then Pi determines whether the message provides a basis for improving the estimated minimal pathlength from P0 to Pi. If this is the case, it sends its new estimate to all successor processes. Each process starts with an estimate of positive infinity. The algorithm should terminate when no more improvements are reported; that is, all processes are passive waiting for messages. Each process should then report its result and terminate.

A realization of this algorithm is given in fig. 5 as two subclasses of DFVG-0 and DFVG-i respectively. The class Vertex serves the purpose of giving a common denotation for all the processes. This denotation is used for formal parameter specification.

```
class Vertex(succ,pred: sequence of Vertex;
              weights: sequence of Integer);
end;


class V0 is-a Vertex, DFVG-0;

  Procedure Termination;
  begin
    halt {terminate all action parts of the process}
         {including those from sub- and superclasses}
  end;

  action

    {send weights to all successors}
    for i:=1 to succ.length do
      Sendmsg(succ.get(i), weights.get(i))
end;


class Vi is-a Vertex, DFVG-i;

  var shortest-found: Boolean;
      shortest,L: Integer;

  Procedure Termination;
  begin
    shortest-found:= true
  end;

  action

    shortest:= maxinteger;
    shortest-found:= false;
    do *
      {receive estimate from any predecessor}
      (i=1..pred.length) Receivemsg(pred.get(i), L) ->
            if L<shortest -> shortest:= L;
                             {send new estimate to all successors}
                             for j:= 1 to succ.length do
                               Sendmsg(succ.get(j),
                                  shortest + weights.get(j))
            fi

    / shortest-found -> {Report(shortest)}; halt
    od
end;
```

fig. 5

The system consists of P0 that is an instance of V0, and a number of Pi's that are instances of Vi.

The V0 action part of process P0 will send the relevant weights to its successors and then reach the implicit changeover point at the final end of the action part, thus leaving control to the DFVG-0 action part. When the DFVG-0 action part detects termination, it terminates the process by means of a "halt" in the procedure "Termination". The Vi action part of a Pi process specifies the procedure "Termination" to update a variable "shortest-found" that is used in a guard in the main loop of the algorithm such that deadlock is broken when termination has occurred. The process can then report its result and terminate.

The "*" after "do" in class Vi could be omitted since Receivemsg contains a changeover point that changes a "do" in which Receivemsg is a guard to a "do *" with resumption condition "some guard is open".

Note that the algorithm described in class Vi very clearly shows the basic ideas without being confused with details concerned with termination detection. This was exactly the purpose of separating the concern of termination detection from the main computation. The reaction on the detection of termination, however, must of course be decided in relation to the main computation. This is reflected in that procedure "Termination" is specified and its result is used in class Vi.

4 Other Algorithms

In this section three additional distributed termination detection algorithms are presented and realized by means of the language outlined in Appendix A. Section 4 mainly serves as a supplement to section 3 to show that the technique described in detail there is more generally applicable. The reader of this section is assumed to be familiar with distributed programming, and the algorithms are therefore presented in less detail than the DFVG algorithm in section 3.

Three algorithms are treated: one developed by Kumar, one by Dijkstra and Scholten, and one by Topor.

4.1 Kumar's Algorithm

Kumar presents several classes of algorithms and possible optimalizations [12]. We concentrate on the class 2 algorithm and for simplicity ignore the optimalizations.

Kumar's algorithm is based on the following assumptions:

- Communication is asynchronous and transmission time is finite.
- A ring can be found in the communication graph, that includes each process at least once.

The algorithm is very similar to the DFVG algorithm in that a marker is sent round the ring. However, Kumar does not assume instantaneous message passing (in fact not even first sent first received). Thus there may at any time be messages in transit between two processes, and termination cannot be concluded just on the basis of knowledge of passivity of all processes. Therefore the marker must carry more complex information than in the DFVG algorithm.

The marker consists of three components. The first two components carry information about the total number of messages sent and the total number of messages received respectively in the whole network of processes. When receiving the marker, a process contributes to the first two components of the marker by adding counts of its own activities (messages sent and received) since the last visit of the marker. After having contributed to the third component of the marker as described below, it passes on the marker. Termination can be concluded when a full traversal of the ring has taken place in which there has been stability in the sense that the total number of messages sent has been constant and equal to the total number of messages received, and no processes has added anything to the marker. Since the ring may contain the same process several times, it is not necessarily so that the process that initializes the marker is the one that can detect termination. Instead a counter is used as the third component of the marker to see when stability has lasted for a full traversal of the ring. Each process increments the counter or sets it to zero before passing on the marker, depending on whether there is stability or not. When the counter reaches a certain value C (the total number of nodes involved in the ring), termination can be concluded by whichever process that has the marker.

In the realization of Kumar's algorithm, we assume for simplicity that the ring is a simple ring - i.e all processes are visited only once in one traversal of the ring. (The general algorithm can also be realized but will be more complicated. For instance, the parameters of the classes must be sequences of processes, and the algorithm for passing on the token must be changed from a simple sending to the successor, to a cyclic passing the token to different successors each time.)

The algorithm is realized in fig. 6 by means of two classes, K and K0. K0 only adds some initialization to take place before the first changeover point. This is the only difference between the leading and the non-leading processes.

```
class K(pred,succ: K; C: Integer);

  Type Control isnew Integer;
       T = (t); {termination wave}

  var B: Boolean;
      sntp,recp,sntm,recm: Control;
      count: Integer; term: T;

  Procedure Termination;
  {deferred specification}

  Guard-cmd Receivemsg(in P: K; out e: Simple);
     * P ? e -> recp:= recp + 1
  end;

  Procedure Sendmsg(in P: K; in e: Simple);
  begin
     P ! e;
     sntp:= sntp + 1
  end;

  action

     sntp:= 0; recp:= 0;
     do *
        pred ? recm -> pred ? sntm;
                       pred ? count;
                       {all components of marker received}
                       B:= (sntp=0) and (recp=0);
                       sntm:= sntm + sntp;
                       recm:= recm + recp;
                       sntp:= 0; recp:= 0;
                       if
                          B and (recm=sntm) -> {stability}
                                               count:= count + 1

                       / not B or (recm<>sntm) -> count:= 0
                       fi

                       if
                          count >= C -> {termination detected}
                                        succ ! t;
                                        pred ? term;
                                        Termination

                        / count < C  -> {send marker}
                                        succ ! recm;
                                        succ ! sntm;
                                        succ ! count

                       fi
      / pred ? term -> succ ! term;
                       Termination
     od
end;


class K0 is-a K;

  action
    sntm:= 0; recm:= 0; count:= 0;
    {start circulation of the marker}
    succ ! recm;
    succ ! sntm;
    succ ! count;
end;
```

fig. 6

The variables "sntp" and "recp" keep count of the messages sent and received by one process since the last visit of the marker. "Sntm", "recm" and "count" are the values that constitute the marker.

It should be noted that the structure of the termination classes is the same as for the DFVG termination detection algorithm. This means that the shortest paths algorithm programmed in section 3.3 could use Kumar's termination detection algorithm instead of the DFVG algorithm, just by using other superclasses to the main computation classes. The main computation classes V0 and Vi should then have K0 and K respectively as superclasses instead of DFVG-0 and DFVG-i. No changes need to be made to the contents of any classes.

4.2 Dijkstra and Scholten's Algorithm

The algorithm by Dijkstra and Scholten [5] solves a slightly different problem than the other algorithms discussed so far. It is assumed that the main computation is a diffusing computation. That is, a computation in which one process, called the environment, is the initiator of all activity and otherwise does not participate in the computation. All other processes are initially passive until they receive a primary message. Except from this, the problem is as before. The environment must detect when all other processes are passive and no messages are in transit.

The idea in the algorithm is to ensure that when a computation has terminated, each channel has carried as many primary messages in the one direction as it has carried control signals in the opposite direction. The environment will receive the last signal and can conclude termination when the number of messages and signals sent and received on all its channels are equal.

In general, each process keeps count in a variable D of the total number of primary messages it has sent minus the total number of control signals it has received. Moreover, it maintains a data structure called a cornet containing the identities of all neighbour processes from which it has received primary messages but not yet sent control signals to. The same neighbour may occur several times in the cornet. At any time, one neighbour in the cornet is marked as the very first that entered the cornet. The cornet is administrated according to the rule "very first in, very last out". The very first process P in the cornet of another process Q is the process that most recently made Q change from passive to active by sending it a primary message.

All processes send and receive control signals and primary messages in such a way that they maintain the invariant

(C >= 0) and (D >= 0) and ((C > 0) or (D = 0))

where C is the number of elements in the cornet.

This enables the environment to conclude termination when its C = D = 0. The reader is referred to [5] for proofs.

The realization of Dijkstra's and Scholten's algorithm is shown in fig. 7. The realization consists of three classes: a common superclass, a class for internal processes and a class for the environment. Further comments are given after the figure.

```
class DS(neighb: Sequence of DS);
  Type S = (s); {signals}
       T isnew Integer; {termination wave}
  var D: Integer;
      signal: S;
      term: T;
  Procedure Termination;
  {deferred specification}
  action
    D:= 0;
    term:= 1
end;
```

```
class DS-Internal is-a DS;

  var C: Integer;
      cornet: Sequence of DS;
      dummy: DS;
      te: T

  Guard-cmd Sendmsg(in P: DS; in e: Simple);
    * C>0 => P ! e;
             D:= D+1
  end;

  Guard-cmd Receivemsg(in P: DS; out e: Simple);
    * P ? e -> cornet.append(P);
               C:= C+1
  end;

  action

    C:= 0;
    cornet:= empty;
    do *
        (i:1..neighb.length) neighb.get(i) ? signal -> D:= D-1

     / (C>1) or ((C=1) and (D=0)) ->  cornet.get(C) ! s;
                                      cornet.reduce(dummy);
                                      C:= C-1

     / (i:1..neighb.length) neighb.get(i) ? te ->
                         if te=term -> for j:= 1 to neighb.length do
                                         neighb.get(j) ! term;
                                       term:= term+1;
                                       Termination

                          / te<>term -> skip {ignore redundant
                                               termination waves}
                         fi
      od
end;


class Environment is-a DS

  Procedure Sendmsg(in P: DS-Internal; in e: Simple);
  begin
    P ! e;
    D:+ D+1
  end;

  action

    *(true); {initiating primary messages sent by subclass}
    do
        (i:1..neighb.length) neighb.get(i) ? signal -> D:= D-1

     / D = 0    ->  {termination detected}
                    for j:= 1 to neighb.length do
                      neighb.get(j) ! term;
                    term:= term+1;
                    Termination
      od
end;
```

fig. 7

We have chosen a simplified realization of a cornet as a stack (by means of a sequence). This has been done for simplicity and does not significantly change the algorithm. The major difference from the algorithm as presented in [5] is that signals are now always sent to the process that sent the last primary message. In DS, there is a non-deterministic choice between all not "very first" processes in the cornet, when sending a signal.

Note that in class DS-Internal, Sendmsg is defined as a guarded command abstraction with a changeover point that will make any if or do command that uses Sendmsg in a guard change to an if* or do* command. The reason for this is that Sendmsg has a precondition C>0 that must be guaranteed to be true when sending a message.

The role of the integer value of the termination wave and the variables "te" and "term" in class DS-internal, is to enable a process to distinguish the first reception of a specific termination wave from later arrivals of the same termination wave via other paths in the communication graph. The value of the termination wave identifies it such that a late arrival of an old termination wave can be distinguished from a first arrival of a new termination wave (belonging to a new computational phase).

The environment class defines no Receivemsg abstraction, since the environment process takes only part in the main computation by sending primary messages that initiate the computation. Thereafter, it only waits for termination. That is, the main computation subclass of class Environment consists only of a number of initiating Sendmsg invocations. The "*(true)" changeover point in the Environment process implies that the main computation part gets control (because of the bottom-up priorities). Thus, the initiating primary messages are sent, and the final (implicit) changeover point is reached, giving back control to the termination part of the process.

It should be mentioned that the algorithm by Dijkstra and Scholten was originally intended to be a secondary computation that should be superimposed as a concurrent process on the main computation. However, in this realization of the algorithm, the coroutine like sequencing called alternation is used without changing the basic idea of the algorithm.

The shortest paths algorithm in section 3.3 is a diffusing computation. All that is needed in order to change this algorithm to use the termination detection algorithm by Dijkstra and Scholten instead of the DFVG algorithm is to let the main computation classes V0 and Vi use the classes Environment and DS-internal respectively as superclasses instead of DFVG-0 and DFVG-i.

4.3 Topor's Algorithm

Topor's algorithm [16] resembles the DFVG algorithm but differs in that it is based on a spanning tree in the communication graph instead of a ring.

The idea of using a spanning tree was first used by Francez and Rodeh (FR) [6] who assumed synchronous message passing like CSP [9]. The FR algorithm is rather complicated and is omitted from this paper. However, it can be realized by means of inheritance with alternation, in particular if a language framework is used that supports synchronous message passing including send statements in guards.

Topor's algorithm combines Francez and Rodeh's idea of using a spanning tree with the idea of black and white tokens and processes from DFVG. In Topor's algorithm, token waves are sent from the leaves to the root in the spanning tree, and repeat waves are sent in the opposite direction. The root is able to determine whether any activity has taken place during the time passed between two upwards waves. A process is white if it has sent no messages since the last upwards wave, and tokens moving upwards in the tree change colour according to the colours of the processes visited like in DFVG. A token is passed on upwards by a process when the process is passive and has received a token from each of its sons in the tree. When a white token reaches the root and the root is itself white, the root concludes termination. When a black token reaches the root or when the root is black on arrival of the token, a repeat wave is started from the root towards the leaves, and a new token wave can begin.

We will present two different realizations of Topor's algorithm, one in section 4.3.1 where we assume that the spanning tree is given in advance, just as we did with the ring in the previous algorithms, and another realization in section 4.3.2 where the computation of the spanning tree is performed in a separate class.

4.3.1 Static Spanning Tree

The first realization of Topor's algorithm is given in fig. 9. It consists of four classes: a common superclass T and three subclasses describing the behaviour of the leaves, the internal nodes and the root of the spanning tree of processes.

Use of this realization of Topor's algorithm requires programming subclasses of T-leaf, T-internal and T-root, describing the main computation. The actual processes are instances of these subclasses with actual parameters supplied in such a way that a spanning tree is formed.

If we consider the shortest paths algorithm discussed in section 3.3, the algorithm could be combined with Topor's termination detection algorithm instead of the DFVG algorithm as follows: T-root is substituted for DFVG-0 in the list of superclasses of class V0. Two versions (called Vi and Vj) of class Vi are made, one with T-internal substituted for DfVG-i in the list of superclasses, and one with T-leaf substituted for DFVG-i in the list of superclasses. The contents of the classes V0 and Vi (Vj) need not be changed. Fig. 8 shows the class hierarchy.

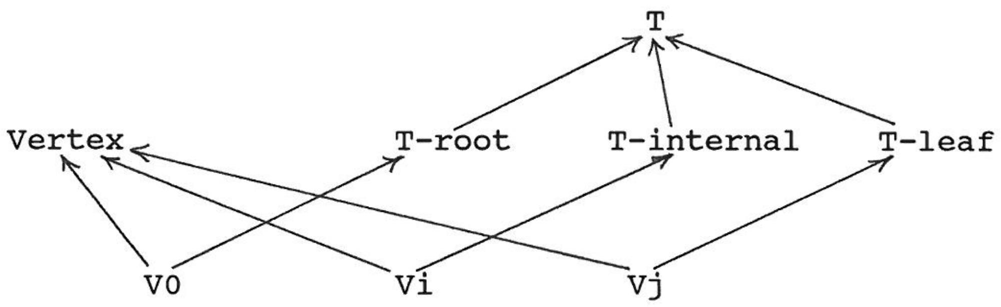

fig. 8

It is clearly a disadvantage that two copies of the main computation class Vi, Vj are needed just because there are three classes involved in the termination detection algorithm. Therefore we present another realization of Topor's algorithm in section 4.3.2.

```
class T (sons: sequence of T; father: T);

  Type Control = (undef, white, black);
       T = (t); {termination wave}
       R = (r); {repeat wave}

  var token, colour: Control;
      repeat: R;
      term: T;

  Procedure Termination;
  {Deferred specification}

  Procedure Sendmsg(in P: T; in c: Simple);
  begin
    P ! c;
    colour:= black
  end;

  Guard-cmd Receivemsg(in P: T; out c: Simple);
    * P ? c -> skip
  end;

  Function all-equal(in s: sequence of Simple;
                     in c: Simple) : Boolean;
  {gives true if all elements in s equal c}
  begin...end;

  Function some-equal(in s: Sequence of Simple;
                        in c: Simple) : Boolean;
  {true if some element in s equals c}
  begin ... end;

  action

    colour:= white
end;


class T-leaf is-a T;

  var hastoken: Boolean;

  action
    hastoken:= true; token:= white;
    {when hastoken, the leaf starts token wave}
    {when not, it waits for wave from father}
    do *
       hastoken          -> if (colour=black) -> token:= black fi
                            father ! token;
                            hastoken:= false;
                            colour:= white

     / father ? repeat -> hastoken:= true;
                            token:= white

     / father ? term   -> Termination
    od
end;
```

```
class T-internal is-a T;

  var tokens: sequence of Control;
      {the tokens received from sons}

  action

    for i:= 1 to sons.length do
      tokens.append(undef); {no tokens from sons have arrived}
    do *
       (i:1..sons.length) sons.get(i) ? token ->
                          tokens.put(i,token);
                          if  not some-equal(tokens,undef) ->
                             {tokens received from all sons}
                             token:= white;
                             if not all-equal(tokens,white)
                                or (colour=black) -> token := black
                             fi
                             father ! token;
                             colour:= white
                          fi

      / father ? repeat -> for i:= 1 to sons.length do
                          begin
                            sons.get(i) ! repeat;
                            tokens.put(i,undef)
                          end

      / father ? term   -> for i:= 1 to sons.length do
                            sons.get(i) ! term;
                          Termination
    od
end;


class T-root is-a T;

  var tokens: sequence of Control;

  action;

    for i:=1 to sons.length do
      tokens.append(undef);
    do *
      (i: 1..sons.length) sons.get(i) ? token ->
          tokens.put(i,token);
          if not some-equal(tokens,undef) ->
                {all tokens from sons have arrived}
                token:= white;
                if not all-equal(tokens,white)
                  or (colour=black) -> token:= black
                fi
                if
                   token = white -> {termination detected}
                                    for j:= 1 to sons.length do
                                      sons.get(j) ! t;
                                    Termination
                 / token = black ->
                                for j:=1 to sons.length do
                                begin
                                  tokens.put(j,undef);
                                  sons.get(j) ! r {repeat}
                                end
                fi
    od
end;
```

fig. 9

4.3.2 Dynamic Spanning Tree

Instead of assuming that the spanning tree is known in advance as a static structure, it could be found in the dynamic computation. Provided that it is given which process is to become the root in the spanning tree, a distributed algorithm for finding a tree can be realized as follows: The root starts by sending a question to all its neighbours to become its sons. Any other process accepts as its father only the first neighbour process who sends it such a question. When receiving a question, a process sends an answer telling whether it accepts the sender as its father or not. When a process accepts a father, it sends a question to all its other neighbours to become its sons. When a process has received answers from all the neighbours that it has asked to become its sons, it knows its own position in the spanning tree. That is, it knows its father and its sons.

This spanning tree algorithm can be programmed by means of two classes, ST-root and ST-others for the root and for the other processes, respectively. A common superclass, ST, is included to contain the declarations common to all processes. The classes are shown in fig. 10. The guarded command abstractions in class ST are intended for use in ST-root and ST-others only and represent a factorization of some of their common properties. Each process has variables "position", "father" and "sons". "Position" is set to "undefined" initially and set to "root", "internal" or "leaf" when the process knows its position, i.e. its father and sons in the spanning tree. This information is to be used by subclasses.

```
class ST (neighb: sequence of ST);

  Type Question = (q);
       Answer = (yes, no);
       Pos = (undef, root, internal, leaf);

  Var  position: Pos;
       sons: sequence of ST;
       father: ST;
       qu: Question; ans: Answer; no-of-answers: Integer;

  Guard-cmd Receiveanswer;
    (i:1..neighb.length) neighb.get(i) ? ans ->
        if ans = yes -> sons.append(neighb.get(i)) fi;
        no-of-answers:= no-of-answers + 1
  end;

  Guard-cmd Reject;
    (i:1..neighb.length) neighb.get(i) ? qu ->
                               neighb.get(i) ! no
  end;

  action
    position:= undef; sons:= empty
end;


class ST-root is-a ST;

  action
    for i:=1 to neighb.length do
      neighb.get(i) ! q;
    no-of-answers:= 0;
    do *
      Receiveanswer ->
         if no-of-answers = neighb.length ->
                                position:=root
         fi

    / Reject -> skip
    od
end;


class ST-others is-a ST;

  action
    if *
     (i:1..neighb.length) neighb.get(i) ? qu ->
                          father:= neighb.get(i);
                          neighb.get(i) ! yes;
                          for j:=1 to neighb.length do
                            if j<>i => neighb.get(j) ! q fi;
    fi;
    no-of-answers:= 0;
    do *
      Receiveanswer ->
         if no-of-answers = (neighb.length - 1) ->
                     if sons=empty  -> position:= leaf
                     / sons<>empty -> position := internal
                     fi
         fi

    / Reject -> skip
    od
end;
```

fig. 10

Now Topor's algorithm can be realized by means of only two classes that use the spanning tree classes as superclasses. These two classes also have a common superclass. The structure of the inheritance hierarchy is shown in fig. 11 and the classes that realize Topor's algorithm are sketched in fig. 12.

$$\begin{tikzcd}
& \text{ST} & & & & \\
\text{ST-root} \arrow[ur] & & \text{ST-others} \arrow[ul] & & \text{T} & \\
& & & \text{T-root} \arrow[ulll] \arrow[ur] & & \text{T-others} \arrow[ulll] \arrow[ul]
\end{tikzcd}$$

fig. 11

Class T is like in section 4.3.1, except that it has no parameters. Information about father and sons is now inherited from class ST.

```
class T-root is-a ST-root,T;
  Var {as in T-root in section 4.3.1}
  action
    *(position = root)
    {awaits that position in spanning tree is known}
    ...
    {action part from T-root in 4.3.1}
    ...
end;


class T-others is-a ST-others,T;
  Var {union of variables from T-internal
        and T-leaf in section 4.3.1}
  action
    if * {awaits that position in spanning tree is known}
       position = leaf -> {action part as in T-leaf in 4.3.1}

     / position = internal -> {as in T-internal in 4.3.1}
    fi
end;
```

fig. 12

Now, the shortest paths algorithm from section 3.3 can use Topor's algorithm in exactly the same way as it could use the other termination detection algorithms. Class V0 should use T-root as superclass instead of DFVG-0, and Vi should use T-others instead of DFVG-i. The class hierarchy is shown in fig. 13.

$$\begin{tikzcd}
 & \text{ST} & & & & \\
\text{ST-root} \arrow[ur] & & \text{ST-others} \arrow[ul] & & \text{T} & \\
 & \text{T-root} \arrow[ul] \arrow[urrr] & & \text{T-others} \arrow[ul] \arrow[ur] & & \text{Vertex} \\
 & & \text{V0} \arrow[ul] \arrow[urrr] & & \text{Vi} \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

fig. 13

If we consider a process that is an instance of Vi, it will contain 5 non-empty action parts, two of which serve only initialization purposes (ST and T). The top-down starting convention ensures that initialization will be done appropriately. After initialization, the three action parts from ST-others, T-others and Vi will alternate in such a way that the Vi part has highest priority and thus executes whenever possible. In the beginning, the T-others part is at a changeover point whose resumption condition is that the position of the process in the spanning tree is known. Therefore, when the Vi part is passive, i.e. waiting for a primary message, the ST-others part will execute. After the ST-others part (in alternation with the Vi part) has found the position of the process in the spanning tree, the T-others action part can be resumed (when the Vi part is passive) and start termination detection in alternation with the Vi part.

5 Evaluation

A number of different algorithms for detecting distributed termination have been programmed in a small language with inheritance on processes. The inheritance mechanism has made it possible to factorize the description of each termination detection algorithm from the main computations that use it.

Each termination detection algorithm has been realized as essentially two classes describing the termination behaviour of a leader process and the other processes in the distributed system. A main computation that uses a specific termination detection algorithm can be programmed as two classes that are subclasses of appropriate termination classes. In this way, the same termination classes can be used by many different main computations.

The choice of termination detection algorithm does not affect the structure of the main computation. (Except that use of Dijkstra and Scholten's algorithm requires the main computation to be a diffusing computation.) Different termination detection algorithms can be chosen by a main computation just by choosing different superclasses. This is true also for termination detection algorithms that are based on a spanning tree in the communication graph. In these algorithms, the computation of the spanning tree can be factorized into yet another superclass, such that only a class for the leader process and a class for the other processes are needed to represent the termination detection algorithm. (As opposed to three classes representing root, internal nodes and leaves in the tree.)

An example of a main computation, a shortest paths algorithm, has been programmed. The example shows how the termination detection concerns and the main computation concerns have been separated. The structure of the shortest paths algorithm is very clear and not confused with irrelevant details concerned with termination detection.

In conclusion, the termination detection algorithms have been nicely factorized from the main computations. The alternation mechanism between the different action parts of a process has been demonstrated to fit naturally with the kind of interaction that is wanted between a termination detection algorithm and a main computation.

The termination detection classes that have been programmed can be used for more general purposes than just to ensure proper termination in a distributed system. Each termination class contains a procedure "Termination", which is activated when termination is detected. The specification of the procedure is deferred until the main computation subclass and thus provides a flexible tool that enables main computations to make different choices of what to do when termination is detected. In the shortest paths example, the action to do on termination detection was simply to report a result and terminate. However, there are other interesting possibilities:

- In the kind of main computations where global deadlocks are difficult to avoid but easy to break when they occur, the "Termination" procedure can be specified in the main computation class to break a deadlock. Afterwards computation can continue. The termination action part of each process is then ready to detect the next global deadlock.

- A main computation that is structured as a multiphase algorithm can use the termination detection algorithm to detect termination of each computational phase. The procedure "Termination" can then be specified to exit the current computational phase and initiate the next phase.

In both the above situations, it would be more appropriate to talk about "stability detection" rather than "termination detection", where stability is defined as in section 2.1.

Other kinds of stability detection are relevant in a distributed system. A general algorithm for detecting stable states in a distributed system by means of socalled "distributed snapshots" is presented by Chandy and Lamport in [2]. An obvious idea is to factorize this algorithm into a class that can be used as a superclass by different main computations. Inheritance with alternation would be appropriate to describe the interaction between the distributed snapshots algorithm and a main computation. However, the distributed snapshots algorithm is so general that programming it independently of what kinds of "states" and "stabilities" are considered, requires a more powerful language than the one we have been using in this paper. Provided that a language with more powerful abstraction mechanisms were given, inheritance promises to be well-suited to factorize also this very general algorithm.

This leads to some comments on language design. The language used in this paper merely provides a framework for presenting the inheritance mechanism. In a complete language design, the abstraction mechanisms should be carefully designed, and the semantic principles of Tennent [14] should be used. That is, it should be possible to make abstractions of all semantically meaningful syntactic categories in the language. (The guarded command abstraction introduced in this paper can be considered a single use of this principle.) This would increase the expressive power of the inheritance mechanism since different kinds of abstractions could be specified in a superclass and inherited by subclasses.

The possibility of deferred specification should be extended to apply to all kinds of abstractions. It could also be generalized to cover the concept of "virtuals" in Beta [10].

- oOo -

It has been demonstrated that inheritance with alternation on processes is a useful structuring mechanism that enables factorization of termination detection algorithms, or rather certain kinds of stability detection algorithms. Moreover, in a language with powerful abstraction mechanisms, inheritance promises to be a valuable factorization mechanism also in relation to more general applications.

Acknowledgement

The initial inspiration to this work was provided by Erik Meineche Schmidt who also made many valuable suggestions during the production of the paper. Thanks are also due to Ole Lehrmann Madsen for useful discussions and comments.

6 References

[1]: A.Albano, L.Cardelli, R.Orsini: "Galileo: a Strongly-Typed, Interactive Conceptual Language", ACM Transactions on Database Systems, vol.10, no.2, June 1985, p. 230-260

[2]: K.M.Chandy, L.Lamport: "Distributed Snapshots: Determining Global States of Distributed Systems", ACM Transactions on Computer Systems, vol.3, no.1, Feb. 1985, p 63-75

[3]: O.J.Dahl, B.Myhrhaug, K.Nygaard: "SIMULA Information, Common Base Language" Norwegian Computing Center, Oct. 1970

[4]: E.W.Dijkstra, W.H.J.Feien, A.J.M.vanGasteren: "Derivation of a Termination Detection Algorithm for Distributed Computations", Information Processing Letters, 16, 1980, p 217-219

[5]: E.W.Dijkstra, C.S.Scholten: "Termination Detection for Diffusing Computations", Information Processing Letters, 11, 1980, p 1-4

[6]: N.Francez, M.Rodeh: "Achieving Distributed Termination Without Freezing" IEEE Transactions on Software Engineering, vol.8, no.3, 1982, p 287-292

[7]: A.Goldberg, D.Robson: "Smalltalk-80, The Language and its Implementation", Addison-Wesley, 1983

[8]: P.B.Hansen: "Distributed Processes: A Concurrent Programming Concept", Communications of the ACM, vol.21, no.11, Nov. 1978, p 934-941

[9]: C.A.R.Hoare: "Communicating Sequential Processes", Communications of the ACM, vol.21, no.8, Aug. 1978, p 666-677

[10]: B.B.Kristensen, O.L.Madsen, B.M.Pedersen, K.Nygaard: "Abstraction Mechanisms in the Beta Programming Language", Proceedings of the 10'th ACM Symposium on Principles of Programming Languages, Austin, Texas, 1983

[11]: B.B Kristensen, O.L.Madsen, B.M.Pedersen, K.Nygaard: "Multi-sequential Execution in the Beta Programming Language", Sigplan Notices, vol.20, no.4, April 1985

[12]: D.Kumar: "A Class of Termination Detection Algorithms For Distributed Computations",
Lecture Notes in Computer Science 206, p 73-100

[13]: "Rationale for the Design of the ADA Programming Language",
ACM Sigplan Notices, vol.14, no.6, June 1979, part B

[14]: R.D.Tennent:
"Language Desing Methods Based on Semantic Principles",
Acta Informatica 8, 1977, p 97-112

[15]: K.S.Thomsen: "Multiple Inheritance, A Structuring Mechanism for Data, Processes and Procedures",
Dept. of Computer Science, Aarhus, DAIMI PB-209

[16]: R.W.Topor:
"Termination Detection for Distributed Computations",
Information Processing Letters, 18, 1984, p 33-36

APPENDIX A

Language Presentation

A.1 Classes and Objects

A class is a template that describes the structure and behaviour of all objects created from it (called instances of the class). Classes are used to describe both processes and data structures. A class definition contains:

1) The name of the class.

2) A list of value parameters of the class.

3) A list of superclasses from which the class inherits parameters, attributes and action parts.

4) A number of type declarations common to all instances of the class and its subclasses.

5) A number of data attributes private to each instance of the class. There are two different kinds of data attributes:
   - Variables:
     var x:T, where T is an ordinary enumeration type, scalar type or sequence type.
   - Variable references:
     var x:C, where C is a class name. The variable x is a reference to an instance of class C.

6) A number of abstracted attributes attached to each instance of the class. The abstracted attributes are procedures, functions and guarded command abstractions. The procedures have value and result parameters, local variables and bodies that can manipulate the attributes of the surrounding ob-

ject. Functions have value parameters only and can inspect attributes of the surrounding object and manipulate local variables. Functions cannot call procedures and thus have no side effects. Abstracted guarded commands are discussed in section A.5.

7) An action part related to each instance of the class. An action part has a priority associated with it. The action part may manipulate the attributes, invoke the abstractions and communicate with other objects by means of asynchronous message passing (section A.3).

A class that has one or more superclasses inherits all parameters, attributes and action parts from its superclasses (and their superclasses in turn etc). Each superclass contributes only once to the class regardless of the number of paths in the inheritance hierarchy from the class to the superclass. This means that objects generated from the class will have structure and behaviour as described by the combination of all the classes in the inheritance hierarchy above the class:

Parameters and attributes that a class inherits from its superclasses are visible within the class just as if they had been locally declared.

An object has its own version of parameters, attributes and action parts. The different action parts of an object inherited from different classes are executed alternately according to their relative priorities and ability to proceed, as will be described in detail in section A.6. The action parts can be considered as forming a single autonomous process that starts executing when the object is generated and executes in parallel with other objects. An object can only communicate with other objects through message passing.

Objects are dynamically generated from classes. A variable reference declared as x:C gets a new C-object (or C1-object) assigned to it by the statement:

x:=new C1(aparam)

where C1=C or C1 is-a C. (The is-a relation is the transitive closure of the "subclass of" relation.) Aparam is a list of actual parameters to the class C1.

Example

{curly brackets surround comments}

```
class A                              class A1 is-a A
  var i: Integer;                      var j: Integer;
      b: B; {class name}                   c: C; {class name}

  Procedure P;                         Function F: Integer;
  begin ... end;                       begin ... end;

  action                               action

    S1                                   S2
end                                  end
```

Assume that x is declared as "x: A1" and created by "a:= new A1", then x will contain the variables i,j,b,c and the routines P and F. Moreover, x will execute the two statement lists S1 and S2 alternately as described in section A.6.

A.2 Types

The language is typed and allows static type checking.

For the purpose of this paper, the possible types are: class types, sequence types, scalar types and enumeration types. Traditional scalar types and enumeration types like Integer, Boolean and Char will be used together with their traditional operators.

The sequence types are specified as

   sequence of Element

where Element is any other type.

We assume the existance of at least the following functions and procedures on sequence types (and ignore error situations):

   function length -> Integer

      s.length gives length of the sequence s

```
function get(in i:Integer) -> Element
   s.get(j) gives j'th element of s

procedure put(in i:Integer;in e:Element)
   s.put(i,e) sets i'th element of s to e

procedure append(in e:Element)
   s.append(e) adds e to the right end of s

procedure reduce(out e:Element)
   s.reduce(e) removes right element from s
   and returns it in e
```

The type system allows parametric polymorphism as described below. The first language with this kind of type system was Simula [3], but a similar mechanism is included in recent languages - e.g. Beta [10],[11] and Galileo [1].

Parametric polymorphism of the type system is obtained by means of a partial <= relation on types. For class-types, the <= relation is the transitive closure of the is-a relation (the sub-class-of relation). For sequence-types the <= relation is the <= relation on the element types. For scalar- and enumeration-types the <= relation is the ordinary subset-of relation. Two special type expressions are available for formal parameter specification: "Simple" and "Object". All simple types - i.e. enumeration and scalar types - are <= Simple, and all class types are <= Object.

That is, the <= relation is implicitly defined for simple types and sequence types, and explicitly for class types. It is possible to define a derived type (similar to Ada's [13])

```
   Type T isnew T1
```

where T1 is a simple type or a sequence type. The purpose of this mechanism is to allow definition of new types that do not participate in the <= relation with other types. Only T <= T is valid for such a type, and T <= Simple if T is a simple type.

Type compatibility related to assignment and parameterization is defined as follows:

1) An expression of type T is assignable to a variable of type S iff T <= S.
2) Actual parameters of type T are accepted for formal parameters of type S iff T <= S.

The parametric polymorphism of the type system gives the flexibility that routines written to manipulate objects of type T can also be used to manipulate all objects of more specialized types without introducing any risk of runtime errors. However, the routine is only allowed to manipulate a parameter in accordance with its formal parameter specification - i.e. only properties defined by type T can be assumed although the actual parameter may have additional properties.

A.3 Communication

Objects communicate with each other by means of asynchronous message passing. Communication is described by means of send statements:

```
partner ! exp
```

and receive statements:

```
partner ? var
```

where "partner" is a reference to an object and may be omitted from a receive statement, "exp" is an expression of some type T and "var" is a variable of some type S. T and S are simple types.

Execution of the send statement implies evaluating the expression and sending the value to the specified partner.

Execution of the receive statement implies waiting until a matching message is available. If a partner is specified, only the message first arrived (but not yet received) from this partner is

considered, otherwise the first message arrived (but not yet received) from all objects are considered.
An arrived message "val" of type T matches "var" of type S if "val" is assignable to "var".
When a matching message is available, it is received - i.e the value is assigned to the variable.

A.4 Control structures

The language includes an alternative and a repetitive command similar to CSP's [9]. The alternative command has the form:

```
if
   guard-1 -> statement-list-1
   ...
/  guard-n -> statement-list-n
fi
```

and the repetitive:

```
do
   guard-1 -> statement-list-1
   ...
/  guard-n -> statement-list-n
od
```

where a guard can contain a boolean expression (including function calls) followed by a receive statement. The boolean expression or the receive statement in a guard may be absent, but not both.
A guard G with a boolean expression B and a receive statement R is said to be

- closed if B is false,
- indetermined if B is true and no message is available that matches R
- open if B is true and a message that matches R is available.

Absence of B or R makes the value of the guard depend only on the present part.

The semantics of the alternative if command is:

```
      evaluate all guards
      if all guards are closed then skip
      else if an open guard exists
              choose an open guard
           else wait until a matching message is available
                     in one of the indetermined guards
               choose an open guard
           receive message and perform statement-list
           related to the chosen guard
```

This is similar to the CSP semantics except that communication is asynchronous and closed guards do not imply failure. The semantics of the repetitive do command is to execute the corresponding if command repeatedly until all guards are closed.

We allow guards of the form

```
  (i:1..n) Bi, Ri -> Si
```

as an abbreviation for n guards with different values substituted for i in Bi, Ri and Si.

Besides the alternative and repetitive commands, the language includes a simple for-to statement.

A.5 Guarded command abstractions

Besides the ability to make abstractions over statement sequences and expressions in terms of procedures and functions, the language allows abstractions over guarded commands. The language construct for this is called Guard-cmd and is specified as follows:

```
   Guard-cmd G (in and out parameters)
     B, R -> S
   end
```

where B is a boolean expression, R is a receive statement and S is a statement list. Either B or R may be absent but not both.

Parametric polymorphism is used for guarded command abstractions just as for procedures and functions.

A guarded command abstraction can be invoked in an alternative or repetitive command in the class in which it is declared or in subclasses of the class:

```
if
   B1, R1  -> S1
   ...
 / Bi, G(apl)  -> Si
   ...
 / Bn, Rn  -> Sn
fi
```

Where apl is a list of actual parameters to G. The i'th guard is equivalent to

```
Bi and B(apl), R(apl) -> S(apl); Si
```

G(apl) can be used as an abbreviation for the alternative command with G as the only choice.

The guarded command abstraction is intended to abstract over communication with some precondition and some subsequent action.

A.6 Multiple action parts

As already mentioned, an object gives rise to an autonomous process that starts executing when the object is created and executes concurrently with other objects. The process originating from one object consists of all the action parts described in the class of the object and all its superclasses. In the following, we will focus on the control aspect within a single object.

The different action parts of an object may have different priorities associated with them (not to be discussed in detail) and the strategy is to execute the action parts alternately according to their ability to proceed and their relative priority. Control shifts from one action part to another at well-defined points in the action parts called changeover points. That is, the action parts are executed in a kind of programmer controlled interleaving or coroutine sequencing.

Changeover points are specified by means of the symbol "*". Each changeover point has a socalled resumption condition associated with it. The resumption condition expresses under what circumstances the action part is able to continue execution. Changeover points may appear in the following contexts:

1) *(B) may be used as a statement, where B is a boolean expression explicitly denoting the resumption condition of the changeover point.

2) * may be used as a prefix of a receive statement meaning that there is a changeover point immediately before the message reception. The resumption condition is that a matching message is available.
   Alternation by means of only such changeover points corresponds to the alternation mechanism in Beta [11].

3) * may be used in an alternative or repetitive command as follows:

```
        if *                   do *
          ...                    ...
        fi                     od
```

   The semantics of an if* command can be expressed using the notation in 1) above for a changeover point:

```
      *(existence of an open guard)
      choose an open guard
      receive message and perform statement-list
        related to the chosen guard
```

   Note that the if* command will always imply choice of a guard as opposed to the if command. If* also waits if all guards are closed, since some guard may become true later, not only as a result of the arrival of messages but also as a result of the alternation with other action parts of the same object that change the values of some boolean expres-

sions.
The semantics of do* is an infinite repetition of the corresponding if*.
If* and do* are generalizations of the when and cycle statements in Distributed Processes [8].

4) An implicit changeover point is placed after the last statement of each action part. After such a changeover point, the action part is terminated and not reconsidered for resumption.

Each action part of an object has a local program counter lpc. When the object is created, all action parts are executed until their first changeover point in a top down sequence in the inheritance hierarchy. This starting convention enables proper initialization of common variables. Hereafter all lpc's are located at their first changeover point of their action part. The alternation mechanism between the different action parts of the object then proceeds as follows:

```
While not all action parts have terminated do
    1) Wait until the resumption condition of some action part is
       satisfied.
    2) If several are satisfied, one of the action parts with
       highest priority and satisfied resumption condition is
       chosen. If no priorities are specified, default is that a
       subclass has higher priority than its superclass - i.e.
       bottom up priorities
    3) Resume execution of the chosen action part until its next
       changeover point.
```

In general, a changeover point specified in an abstracted attribute will imply a changeover point in the action part that invokes the abstraction.

A guarded command abstraction that contains a changeover point in its guard "infects" the whole if or do command in which it is used as a guard. That is, any use of the guarded command ab-