DELTA PROJECT
REPORT NO. 11

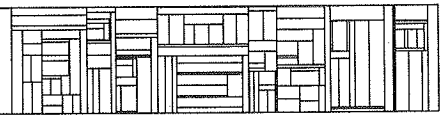
ISSN 0105-8517

CONSTRUCTION OF SYSTEM DESCRIPTIONS IN SIMULA AND DELTA

by
Niels Erik Andersen
and
Morten Kyng

DAIMI PB-81
April 1978
(Revised version)

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



Abstract

In the first part of the paper we discuss the task of constructing a system description. We emphasize,

- that the part of the world which we want to regard as a system is delimited through our selection of the components, their structural relationships and their capabilities to affect each other.
- that there are different ways of conceiving a given part of the world as a system.
- that the terms in which the people, who know the system speak of it define the concrete starting point from which the system description should be developed.

The two construction strategies, decomposition and composition ("top-down" and "bottom up") are discussed. We consider the concepts of subsystems and aggregates as developed in the DELTA-project, which may assist us when using the two construction strategies. We emphasize that the properties of an aggregate require a set of attributes to conceive of and describe those properties. Such attributes are not developed in the DELTA-project.

In the second part of the paper we give a preliminary proposal for conceiving of and describing a simple aggregate concept. We describe the aggregate concept by means of the class concept. So we do not need to add new language elements to the SIMULA/DELTA-languages. The problem of the identity of an aggregate is discussed briefly.

In the appendix we give a formalized DELTA-description of some of the attributes proposed in the second part.

CONTENTS

PREFACE

This paper is a slightly elaborated version of the manuscript from a lecture given by Morten Kyng at the fourth SIMULA Users' conference in Noordwijkerhout, Netherlands, 8th - 10th September 1976.

The associated abstract and minipaper are printed in the proceedings of the conference (Norwegian Computing Center - publication no. S-81) with the title : "Decomposition and aggregation in SIMULA and DELTA".

We want to thank Ejvind Lynning at the Norwegian Computing Center for his comments on the manuscript.

Aarhus, August 1977

Niels Erik Andersen and Morten Kyng

PREFACE TO THE REVISED VERSION

In this version a number of minor changes have been made to increase the readability, and a sketch of a possible implementation in SIMSET has been added to the appendix.

Aarhus, March 1978

Niels Erik Andersen and Morten Kyng

PART 1

CONSTRUCTION OF SYSTEM DESCRIPTIONS

The subject of this paper is some of the concepts and language tools available in the task of constructing a system description and in communication about systems.

Let us as an example consider a railroad system.

Example : A railroad system

We want to construct a description with the purpose of clarifying the relations between service to customers and different repairing and maintenance strategies.

When we make this description we may do it in cooperation with several different groups of people.

From the planners point of view and in the context of planning the work of repairing and maintenance, we may identify different kinds of locomotives and railroad cars, the rails with shunts, workshops, and railroad stations. Within the context of the customers we will talk about the railroad in terms of railroad lines, departure and arrival times, the condition of the cars, ticket prices, first and second class, the possibilities to eat and sleep, punctuality and the frequency of accidents.

□

In different contexts we may identify different components and different relations between the components. Components, which in one description of a part of the world as a system are considered as important and characterized in a certain way, may in another description of the same physical part of the world be of minor importance and characterized by different attributes.

This illustrates that we do not consider any part of the world to be per se a system with a well-defined inherent structure.

Definition of a system

We use the term system as defined in the DELTA project (DELTA 75 p. 15):

"A system is a part of the world

> which we choose to regard as a whole, separated from the rest of the world during some period of consideration, a whole which we choose to consider as containing a collection of components, each characterized by a selected set of associated data items and patterns, and by actions which may involve itself and other components".

We emphasize that a system is only defined when we have chosen the components, and the structure of the components. We may thus look at the construction of a system description as the definition of the described system.

This is in contrast with some common system description methodologies, which regard the construction of a system description as a stepwise uncovering of an a priori given system structure. This is exemplified by the definition of "system analysis" in a widely used Danish edp-dictionary (GYLDENDAL 75):

> "System analysis : a process that includes the decomposition of a system into parts. The purpose is to aquire knowledge about the way in which the system functions ....".

From this point of view system analysis may not take into account different ways of conceiving a given part of the world as a system.

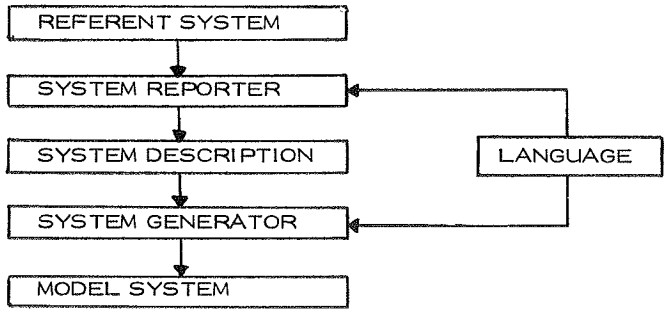
Communication about systems

Using the terminology of the DELTA project the system described will be called the referent system. The person making the description will be called the system reporter (or just reporter).

In a communication process the system description will be used by a system generator to make a model system.

The generated model system will in some sense be similar to the referent system. The similarity results primarily from the reporter's and the system generator's knowledge of the system description language.

We may illustrate this in the following way:



The problems in the communication process are discussed in more detail by Niels Erik Andersen and Niels Karsten Thorhauge in Rapport 76. In this paper we only want to comment upon some of the well known languages or description methods used in the construction and documentation of large edp-programs, such as block diagrams, state descriptions, users' manuals, programmers' guides, etc.

The starting point in the named description methods or descriptions is the edp-programs.

The environment in which the edp-programs are going to be used is hardly ever mentioned. At the same time the edp-programs may be based on rather rigid assumptions about this environment.

In the communication about the construction and the use of edp-programs (that is communication about a system which contains the edp-programs) the descriptions of the relations between the edp-programs and the environment are important. We are interested in system description methods allowing a joint description of all components of a system – both those which are computer based and those which are not.

The description of the relations between the edp-program and the environment is often reduced to so-called "I/O specifications". In GYLDENDAL 75 (p. 318) it is said that:

> "System users (make) a number of demands on the description. It is e. g. necessary to know which input the system needs, which output it produces, and how the output is to be treated and interpreted".

In practice the input specifications are often separated from the output specifications. This separation further obscures the relations between the edp-programs and the environment although it reflects a division of labour.

In the next section we consider the task of constructing a system description and study some construction strategies.

Construction of system descriptions

The construction consists of the identification of

- the components of the system and
- the attributes and actions of the components.

The part of the world which we want to regard as a system is delimited through our selection of the components and our stipulations of their structural relationship and capabilities to affect each other. Some of the properties of the components may be regarded as system components themselves whereas others are regarded as data items.

Before we procede with the discussion of how to identify and characterize components, we will once more consider the railroad example.

Concrete, representative and abstract elements

The components identified in the referent system (in this example the railroad system) and the attributes and actions characterizing these components we will call concrete (referent) system elements (or just concrete elements). In the railroad example we have trains, locomotives, railroad cars, departure and arrival times, and ticket prices as examples of concrete elements.

The elements of the model system representing the concrete elements we will name representative (model) system elements (or just representative elements). Locomotives and railroad cars may be represented by objects and ticket prices by integers.Furthermore, locomotive- and railroad car-objects may be grouped together to represent a train.

The representative elements may in turn be built from elements which are not directly comprehensible in relation to the referent system. We say that these elements are abstract elements. When we organize a queue we may do this by means of "predecessor" and "successor" references. In relation to the queue in the referent system, these references are abstract.

In the literature dealing with the ideas and concepts of abstract data types another terminology (and thus another way of thinking) is used. In this framework some of the components in the referent system may be modelled by objects of different abstract types, which then may be implemented using the concrete data types of the language.

Our point here is that although the goal by using abstract data types is to enable the user to define his own quantities, the conceptual framework is that of the computer scientist:

- a user-defined queue is abstract
- integers and references of the programming language are concrete.

In order to develop an application-oriented framework we should start from the other end:

> The terms in which the people who know the system speak of it are concrete, and these define the starting point from which the system description should be developed.

We may illustrate our concepts once more in terms of a post office example:

Example: A post office system

In the referent system a concrete element may be the action:

> a customer delivers a parcel.

In the model system this may be represented by:

> a CUSTOMER-object executes the action DELIVER A PARCEL.

The action DELIVER A PARCEL may be built from subactions such as:

```
NUMBER OF PARCELS:=
    NUMBER OF PARCELS - 1;
COUNTER.NUMBER OF PARCELS:=
    COUNTER.NUMBER OF PARCELS + 1;
```

which are abstract elements .

□

The concepts concrete, representative and abstract elements are discussed in more detail by Morten Kyng and Lars Mathiassen in Formal 76.

Let us now return to the task of identifying and characterizing the components of a part of the world considered as a system - the task of defining a system or constructing a system description.

Construction strategies

We may use at least two strategies: decomposition and composition. In practice we will alternate between the strategies.

Decomposition

The decomposition strategy is often named analysis, and in computer programming terminology it is usually referred to as the "top-down approach".

Example: In the detailed analysis of a component we may find that several components are needed to portray its behaviour. In the rail road system we may describe a train-component as a TRAIN-object containing inner objects representing the locomotive, carriages, goods-vans etc.

□

When the components of the system which we consider are identified and characterized, then it is common to describe the construction-task as a decomposition and this often leads to very nice and clearly structured descriptions. Different degrees of detail may be reflected in the degree of decomposition, i.e. in the kind and number of components identified and in the amount of detail in the characterization of the components. In the literature on programming methodology we find examples of this kind of descriptions where the construction-task is named structured programming or stepwise refinement. (See e.g. Dijkstra 72 and Wirth 71).

However, if we read such a top-down description carefully, we will most likely find evidence to support the point of view that the description is not a true description of the task of construction. An example of this is given in Naur 72. We will give a simple example later as a comment on the decomposition of the post office described in DELTA 75.

Composition

The composition strategy is the reverse of decomposition and is often named synthesis. In programming terminology it is referred to as the "bottom-up approach".

Example:

Working "bottom-up" we may want to consider a collection of components as a meaningful whole.

When we have identified locomotives, carriages and goods-vans in the rail-road system we may want to group such components together to from trains.

□

In section 1.1 and 1.2 we consider some of the DELTA-concepts and language tools, which may assist us when using the two construction strategies.

1.1 Decomposition in DELTA

In the DELTA-project a conceptual framework and some language tools have been developed to assist in the construction of system descriptions. The concepts are mainly described and discussed in connection with the decomposition strategy. In DELTA 75 (p. 66) some of the desired properties of the concepts are expressed in this way:

"The user of the language must be allowed to understand the system at a number of different levels of analysis. A part of the system which at a given step of decomposition is conceived as a meaningful whole in relation to other parts of the system, should at later steps still be identifiable as a system element to which an inner structure may be related."

and this may be achieved in at least two ways (DELTA 75 p. 67):

"By considering the part identified as a subsystem which, like the system as a whole, is represented by a component which contains internal components whose existence is confined to the enclosed world of the subsystem.
Or, alternatively, by considering the part as an aggregate consisting of a group of components, some of which have permanent membership of the aggregate while others move in and out of one or more aggregates."

The Subsystem Concept

Subsystems are given the following structural properties:

- the constituent parts of earlier steps of decomposition are meaningful system elements to be referred to within this world.

- the possibility of directly interacting from the outside with the internal components of a subsystem is strictly limited by the rules imposed (see DELTA 75 p. 133 f.f.).

The subsystem concept is not used very much in SIMULA-programming. One reason is that it is not possible to describe the sequencing inside a subsystem according to simulated time.

In SIMULA we have to represent activities, processes etc. by objects at the same system level, that is objects declared in the block prefixed by SIMULATION, SIMON, DRAFT or whatever it may be.

In DELTA we can describe the sequencing inside a subsystem according to TIME.

The Aggregate Concept

Contrary to the restrictive structural properties of the subsystems an aggregate is a more loosely organized system element. The aggregate

offers the possibility of portraying a group (of components) of changing membership, with almost any kind of involvement between the members being specified. The possibilities of changing memberships (the dynamic structures) require a set of attributes to make the changing possible and some possibilities of specifying restrictions on the use of attributes to achieve a desirable separation of an aggregate from the rest of the system.
These problems are not discussed in detail in DELTA 75 while the structural properties of the subsystem concept are analysed and described in detail.

Decomposition Tools

NESTING:

When a component is decomposed into a subsystem having internal components we will say that the component is decomposed by nesting. The first step of decomposition is always a nesting.

SPLITTING:

When a component is decomposed into an aggregate we will say that the component is decomposed by splitting.

In DELTA 75 the two decomposition tools are used to decompose a post office system (p. 151 - 157). We will give a summary of this decomposition and the explanation in DELTA 75. Afterwards we will give our comments on the example.

Example:

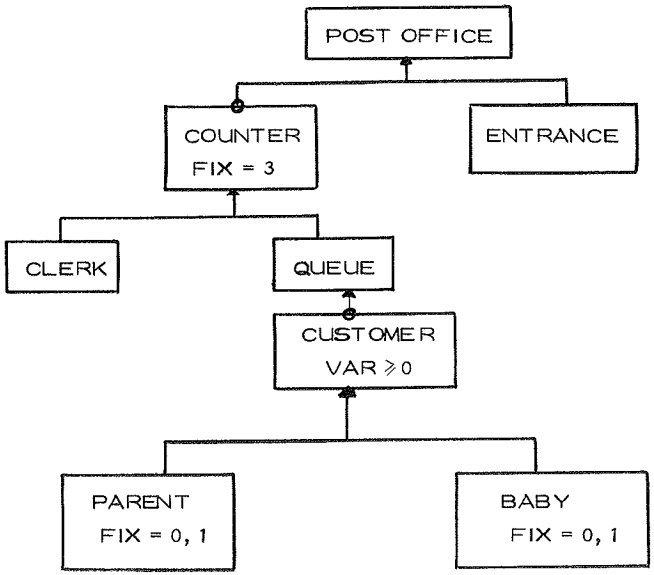
Decomposition of a post office system

The purpose of the decomposition and the system description is to study:

- the behaviour of the queues
- the service towards the customers
- the work load of the clerks.

The decomposition is described in two phases. In the first phase only nesting is used and in the second phase the kind of interaction between the components is studied and splitting is applied.

After the first phase which consists of four steps of nesting the result of the decomposition is illustrated by this figure (fig. 3.29 p. 155 in DELTA 75):



The post office system contains a set of service counters - in this case three. The separate action sequence which introduces new customer components into the system is associated with an "entrance" - component.

Each counter component is decomposed into a subsystem containing a clerk and a queue component, and the queue is decomposed into a subsystem containing a number of customer components. Some of the customers may consist of a

parent and a baby, therefore we may have a parent component and a baby component as internal components of a customer component. The small circle enclosing the connection point of the "counter" rectangle and the directed line into the "post office" rectangle indicates that the "counter" rectangle represents zero or more similar components contained within the same "post office" component. The variables FIX and VAR are used to denote the number of similar components. The graphical notation is explained in detail in DELTA 75.

Some of the consequences of only using nesting as decomposition tool are:

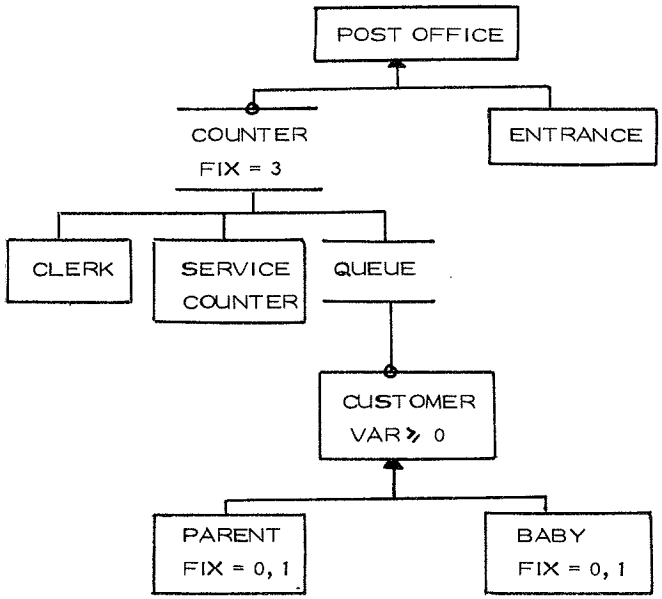
- customer components cannot move from queue to queue (or from counter to counter) because they are internal components of a specific queue component.
- clerk components cannot operate customer components.

Therefore

the counters and the queues must be decomposed into aggregates.

But we have to decide whether we want to reintroduce the component being decomposed as a component within the aggregate, or not. If we reintroduce the component it provides a link to the earlier steps of decomposition. (DELTA 75 uses the word "retain" instead of "reintroduce").

After the second phase in DELTA 75 the counter component is reintroduced and renamed to "service counter" to avoid confusion with the counter aggregate, and the queue components are dropped. Aggregates are represented in the same way as components, only with the borderline of the rectangle removed. The result of the decomposition is now illustrated by (fig. 3.30 page 157 in DELTA 75) :



□

Comments on the decomposition example

- It is very unnatural to introduce the entrance component in the first step of decomposition. Especially when the arguments are in terms of components (the customers) which are not yet identified. This indicates that the construction is not really a strict decomposition as described in DELTA 75.

- Elements which most people would mention first when discussing a post office such as letters and parcels are not mentioned at all. This is probably due to the fact that DELTA 75 does not describe the construction of a post office system description but rather presents an existing system description in a top down way (cf. page 8).

- the DELTA language provides no language tools for the description of aggregates. Such language tools may solve the problem of whether the component which is decomposed into an aggregate should be reintroduced or not.
  In part 2 of this paper we give a proposal for language tools for describing a simple aggregate concept.

1.2 Composition in DELTA

The composition strategy is the reverse of decomposition. It is used when we want to consider a collection of components as a meaningful whole. Using composition as a strategy we may start by identifying and characterizing components and structures that are easy to understand and capture. Resting on this basic platform we may stepwise build up descriptions, which progressively covers a larger part of a complex totality. This can be done by

ENCLOSING:

Composition of a subsystem, i.e. conceive of and describe some components as parts of a subsystem either as data item attributes or as internal components enclosed by the subsystem. This is the counterpart of nesting.

Example: If we have introduced letters, parcels, and customers as components in a post office system, we may later want to conceive of letters and parcels as parts of a customer subsystem and represent them as data item attributes of customer components (e.g. integers: number of letters and number of parcels).

□

GROUPING:

Composition of an aggregate, i.e. conceive of and describe some components as members of an aggregate. These memberships may change during time. This is the counterpart of splitting.

Example: If we have introduced customers, service counters and clerks as components in a post office system we may want to conceive of customers as members of some aggregates called queues and conceive of a clerk, a service counter and a queue as members of an aggregate called counter. A customer may move from one queue to another and therefore from one counter to another.

□

The two composition tools, enclosing and grouping, are not dealt with in DELTA 75. They name the use of the concepts of subsystem and aggregate in a composition strategy.

We resume the construction tools in this diagram:

CONSTRUCTION STRATEGY / SYSTEM ELEMENT	DECOMPOSITION (INTO)	COMPOSITION (OF)
SUBSYSTEM	NESTING	ENCLOSING
AGGREGATE	SPLITTING	GROUPING

PART 2

A SIMPLE AGGREGATE CONCEPT

As stated in DELTA 75 and earlier in this paper the dynamic aspects of the aggregate concept require a set of attributes for their conception and description. This includes language tools which may be used to form system structures different from the tree-structures of subsystems. Such attributes or language tools are not developed in the DELTA-project. In this part of the paper we give a preliminary proposal for conceiving of and describing a simple aggregate concept.

At first we consider an unsorted list of those desired properties of a simple aggregate concept, which we will stress.

2.1 Desired properties

1. An aggregate should be easily identifiable as a system element.
2. It should be possible to group together similar aggregates and treat them as belonging to a common category.
3. The possible and actual members of an aggregate should be easily identifiable.
4. It should be possible to specify a lower and upper bound on the number of the possible members from different categories.
5. It should be possible for an aggregate to contain both components and other aggregates as members.
6. An aggregate should not be a part of itself.
7. All parts of an aggregate should be system elements at the same level.
8. It should be possible for an element to be a member of more than one aggregate at a time.

9. An element may during its lifetime change its membership of aggregates.

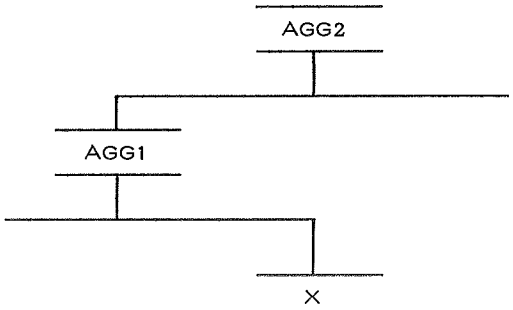
2.2 A preliminary proposal

In order to be able to discuss the aggregate concept more precisely we introduce some terminology

Terminology

Those elements into which an aggregate is decomposed or from which it is composed are called the members of the aggregate. And we say that an aggregate contains its members.

If an element X is a member of an aggregate AGG1 which in turn is a member of an aggregate AGG2 we say that X is a part of AGG2. We may illustrate this in the following way:



In general we say that an element, X, is a part of an aggregate, A, iff there exists a sequence of elements X_1 to X_n such that $X = X_1$, $A = X_n$ and X_{i+1} contains X_i for $i = 1, \ldots, n-1$.

We say that an aggregate includes its parts.

Property number six in the preceding section states that we do not allow circularity in an aggregate.

Language tools based on the class concept

We will describe an aggregate concept by means of the class concept. The advantage is obvious: we do not need to add new language elements. Reasonable syntax may be provided by some macro facility. One drawback is that aggregates may be used only on system level one, since the classes used to implement them are attributes of the system object. In SIMULA, however, nesting of objects is rare, and in DELTA we may regard each object as containing the necessary class declarations as attributes.

In order to describe an aggregate we have to describe the set consisting of its members.

In SIMULA the class SIMSET gives us facilities for manipulation of circular two-way lists. Such a list may be used to describe the member set of an aggregate. An approach in which we prefix all objects representing aggregates by the class HEAD of SIMSET and all members of aggregates by the class LINK of SIMSET will, however, have several drawbacks:

- an object can only be a member of one aggregate at a time, since an object prefixed by LINK may appear in at most one two-way list.
- an aggregate can only contain objects prefixed by LINK as members. Other aggregates can not be members, since these will be prefixed by HEAD.

To avoid the drawbacks we need some kind of multi-set facilities. These are discussed in the following subsection. A possible implementation using SIMSET is sketched in the last part of the appendix.

The classes MEMBER and AGGREGATE

In a class MEMBER we include the attributes common to all aggregate members. This class is then used as a prefix to all possible members.

The outline of the class with its functional and procedural attributes is as follows

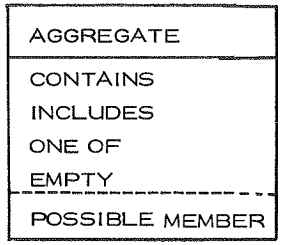
Example:

Using our railroad example one description of a class of locomotives may look like this:

```
CLASS LOCOMOTIVE : MEMBER OBJECT
      BEGIN
          description of attributes and actions common
          to all LOCOMOTIVEs
      END LOCOMOTIVE OBJECT
```

□

In the same way we group together the attributes common to all aggregates and describe them in a class declaration. The outline is as follows:



Example:

In the railroad example we may describe a TRAIN aggregate in the following way:

```
CLASS TRAIN : AGGREGATE OBJECT
      BEGIN
          specification of possible members
      END TRAIN OBJECT
```

□

In the cases where an aggregate may itself be a member of another aggregate we would like to be able to write something like

```
CLASS TRAIN : MEMBER & AGGREGATE OBJECT
      BEGIN
          ⋮
      END TRAIN OBJECT
```

This is, however, not possible. Instead we make AGGREGATE a subclass of the class MEMBER.

Let us consider the attributes of the MEMBER and AGGREGATE classes in more detail. In the appendix we give a formalized DELTA-description of some of the attributes.

```
CLASS MEMBER: OBJECT
BEGIN
    FUNCTION MEMBER OF (AGG) : BOOLEAN BEGIN test whether THIS
                                             MEMBER is a member
                                             of AGG or not
                                       END*** MEMBER OF ***;

    FUNCTION PART OF (AGG) : BOOLEAN BEGIN test whether THIS
                                           MEMBER is a part of
                                           AGG or not
                                     END***PART OF***;

    PROCEDURE INTO : BEGIN make THIS MEMBER a member of a
                           specified AGGREGATE, AGG, if
                           THIS MEMBER is not already a member of
                           AGG and no circularity is introduced
                     END*** INTO ***;

    PROCEDURE LEAVE : BEGIN make THIS MEMBER leave a
                            specified AGGREGATE, AGG, if
                            THIS MEMBER is a member of AGG
                      END*** LEAVE ***;

    PROCEDURE GOTO : BEGIN make THIS MEMBER leave all the
                           AGGREGATEs of which it is a member
                           and then make it a member of a specified
                           AGGREGATE, AGG
                     END***GOTO***;

END MEMBER OBJECT;
```



```
CLASS AGGREGATE : MEMBER OBJECT
BEGIN
    FUNCTION CONTAINS (M) : BOOLEAN BEGIN test whether THIS AGG
                                          contains M as a member
                                          or not
                                    END*** CONTAINS ***;

    FUNCTION INCLUDES (M) : BOOLEAN BEGIN test whether THIS AGG
                                          includes M as a part or not
                                    END*** INCLUDES ***;

    FUNCTION ONE OF : REF MEMBER  BEGIN IF THIS AGG has any members
                                        THEN ONE OF is a reference
                                             to one of them
                                        ELSE ONE OF is NONE
                                  END *** ONE OF *** ;

    FUNCTION EMPTY : BOOLEAN BEGIN test whether THIS AGG has
                                   any members or not
                             END*** EMPTY ***;

END AGGREGATE OBJECT;
```

Specifying possible members.

We would like to be able to specify the classes[1] of the possible members as parameters to the aggregate, i.e. in our rail road example we would like to write something like:

```
TRAIN (LOCOMOTIVE, ....      ) :
    AGGREGATE OBJECT
        BEGIN ....
```

This is, however, not possible - without a macro facility. In any case we have to end up with something like:

---

(1) In DELTA it shall of course also be possible to specify that a given singular object is a possible member of some aggregate.

```
CLASS AGGREGATE : MEMBER OBJECT
BEGIN
    :
    FUNCTION POSSIBLE MEMBER : BOOLEAN VIRTUAL;
    FUNCTION POSSIBLE MEMBER (M): BOOLEAN
              BEGIN M : REF MEMBER;
                    RESULT : = TRUE
              END*** POSSIBLE MEMBER ***;
    :
END AGGREGATE OBJECT;


CLASS TRAIN : AGGREGATE OBJECT
BEGIN
    FUNCTION POSSIBLE MEMBER (M): BOOLEAN
              BEGIN M : REF MEMBER;
                    RESULT : = M IN LOCOMOTIVE OR
                               M IN CARRIAGE    OR
                               M IN GOODS-VAN
              END*** POSSIBLE MEMBER ***;
END TRAIN OBJECT;
```

In a similar way we may specify lower and upper bounds on the number of possible members from each class.

With these additions and the obvious modifications of the class MEMBER procedures INTO, LEAVE and GOTO of the class MEMBER, our preliminary proposal has the desired properties listed in section 2.1.

2.3 The identity of an aggregate - a topic for further work

When decomposing into aggregates we have to decide whether we want to reintroduce the component being decomposed as a component within the aggregate or not (cf the post office example in part 1).

If we do reintroduce the component it provides a link to the earlier steps of decomposition. The component may then be used in a number of ad hoc ways to represent the aggregate . But the proposed aggregate concept does not distinguish between the members of an aggregate, and there are no special language elements to denote the reintroduced component.

If we do not reintroduce the component being decomposed this possibly implies that none of the members are used to represent the aggregate as a whole.

Whether to reintroduce the component or not is one aspect of the problem of the identity of an aggregate. Another aspect of this problem concerns the empty aggregate.

When decomposing a component into an aggregate the component will always be split into other components (two or more). An empty aggregate will not be constructed. The structure of the subsystems and aggregates resulting from a decomposition is thus always a tree structure where the leaves (and the root) are components.

From one point of view we may want an aggregate to cease to exist when it becomes empty (or to regard operations involving an empty aggregate as errors).

From another point of view we may want to supply the aggregates with attributes and possibly some actions. In a composition we may want to specify the possible members of an aggregate and then use their attributes in the specification of the aggregate and vice versa.

REFERENCES

DELTA 75 — Erik H. - Hansen, P. Håndlykken, Kristen Nygaard: "System description and the DELTA language",DELTA Report No. 4, Norsk Regnesentral 1975.

Dijkstra 72 — E.W. Dijkstra: "Notes on structured programming" in "Structured Programming" by Dahl, Dijkstra and Hoare. - Academic Press, London 1972.

Formal 76 — Morten Kyng and Lars Mathiassen: "Arbejdspapir om anvendelsesorienteret diskussion af formaliserede sproglige udtryksmidler". DAIMI - april 1976 (unpublished).

GYLDENDAL 75 — J. Nielsen and H.J. Helms: "Gyldendals edb-leksikon", Gyldendal, København, 1975.

Naur 72 — Peter Naur: "An experiment on program development" - BIT 12 (1972), 347 - 365.

Rapport 76 — N.E. Andersen and N.K. Thorhauge: "En rapport om et specialearbejde - emne: "Kommunikationsproblemer under udvikling af edb-systemer". DAIMI, februar 1976.

Wirth 71 — N. Wirth : "Program development by stepwise refinement". - Comm. ACM 14 (april 1971) 221 - 227.

APPENDIX

A formalized DELTA-description of some of the concepts proposed in part 2.

In part 2 we introduced the class AGGREGATE as a subclass of the class MEMBER. We also introduced some attributes to assist us in describing the relationships of the MEMBER and AGGREGATE objects.

In order to describe the attributes we need some list structures to keep track of the relationships.

1.
For each AGGREGATE object we need a list of the members of this AGGREGATE and

2.
For each MEMBER object we need a list of the aggregates of which this MEMBER is a direct member.

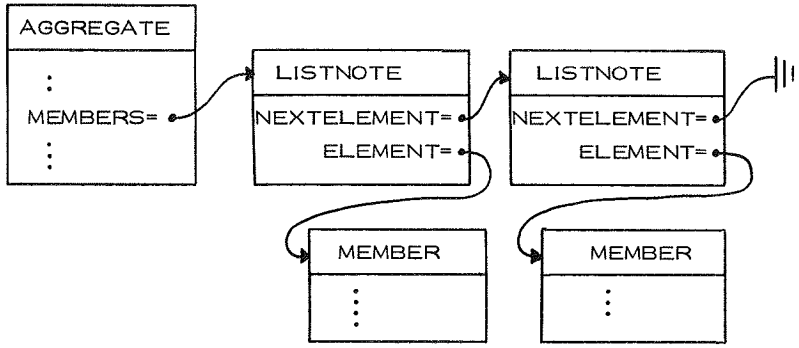
A MEMBER object may be a direct member of more than one aggregate at a time. That is a MEMBER object may be an element of more than one list of MEMBERS.

Similarly an AGGREGATE object may be an element of more than one list of "is member of" AGGREGATEs.

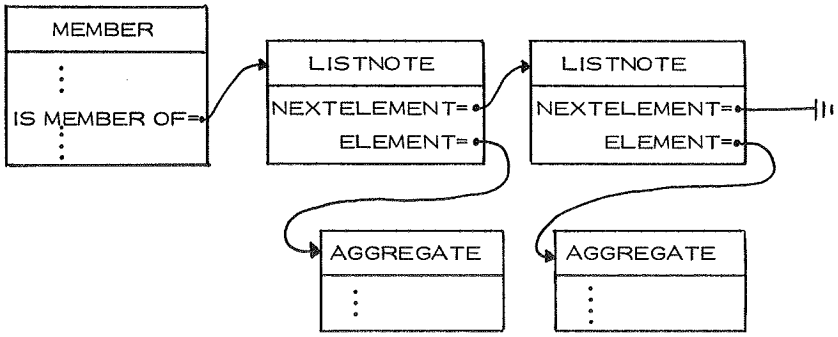
We will describe the two lists as lists of LISTNOTE objects with the AGGREGATE and MEMBER objects as listheads. The two listheads contain the list references, MEMBERS and IS MEMBER OF respectively.

We now have the following structure of the lists:

1. List of MEMBERS



2. List of "is member of" AGGREGATEs:



The concepts proposed in part 2 may be described by:

```
CLASS MEMBER :
    OBJECT BEGIN
        IS MEMBER OF : REF LISTNOTE;
        FUNCTION MEMBER OF (AGG) : BOO B E;
        FUNCTION PART OF (AGG) : BOO B E;
        PROCEDURE INTO : B E;
        PROCEDURE LEAVE : B E;
        PROCEDURE GOTO : B E;
    END MEMBER OBJECT;

CLASS AGGREGATE:
    MEMBER OBJECT BEGIN
            MEMBERS : REF LISTNOTE;
            FUNCTION CONTAINS (M) : BOO B E;
            FUNCTION INCLUDES (M) : BOO B E;
            FUNCTION ONE OF : REF MEMBER B E;
            FUNCTION EMPTY : BOO B E;
        END AGGREGATE OBJECT;

CLASS LISTNOTE :
    OBJECT BEGIN
        NEXTELEMENT : REF LISTNOTE;
            ELEMENT : REF MEMBER;
    END LISTNOTE OBJECT;
```

We give a formalized DELTA description of

- the MEMBER procedures INTO and LEAVE and
- the AGGREGATE functions CONTAINS (M) and INCLUDES (M).

We do, however, not include tests for possible violation of the upper and lower bounds of the number of possible members.

```
PROCEDURE INTO:
    BEGIN make THIS MEMBER a member of a specified AGGREGATE,
          AGG. That is, if THIS MEMBER is not already a member
          of AGG and no circularity is introduced, then AGG into THIS
          MEMBER's "is member of" list and THIS MEMBER into
          AGG's "members" list.

    AGG : REF AGGREGATE;
    FUNCTION CIRCULARITY : BOOLEAN
            BEGIN
              IF THIS MEMBER IS AGGREGATE
              THEN
                  (* RESULT : = THIS MEMBER QUA
                       AGGREGATE. INCLUDES (AGG)
                  *)
            END *** CIRCULARITY *** ;

    IF MEMBER OF (AGG) OR CIRCULARITY
    THEN (*ERROR*)
    ELSE (* IS MEMBER OF : - NEW LISTNOTE
                  PUT (* ELEMENT : - AGG;
                    NEXTELEMENT : - IS MEMBER OF *);

           AGG. MEMBERS : - NEWLISTNOTE
                  PUT (* ELEMENT : - THIS MEMBER;
                    NEXTELEMENT : - AGG. MEMBERS *);
         *)
END *** INTO *** ;
```



```
PROCEDURE LEAVE :
    BEGIN make THIS MEMBER leave a specified AGGREGATE, AGG.
          That is, if THIS MEMBER is a member of AGG, then AGG
          out of THIS MEMBER's "is member of" list and THIS MEMBER
          out of AGG's "members" list.

    AGG : REF AGGREGATE ;

    PROCEDURE OUT OF IS MEMBER OF LIST :
       BEGIN LAST, NEXT : REF LISTNOTE ;
          NEXT : - IS MEMBER OF . NEXTELEMENT;
          IF IS MEMBER OF . ELEMENT = = AGG
          THEN (* IS MEMBER OF : - NEXT *)
          ELSE (* WHILE NEXT. ELEMENT = /= AGG REPEAT
                        (* LAST : - NEXT ;
                           NEXT : - NEXT.NEXTELEMENT
                        *);
                   LAST. NEXTELEMENT : - NEXT. NEXTELEMENT
                *)
       END *** OUT OF IS MEMBER OF LIST ***;


PROCEDURE OUT OF AGG MEMBERS LIST :
    BEGIN LAST, NEXT : REF LISTNOTE;
       NEXT : - AGG. MEMBERS. NEXTELEMENT;
       IF AGG. MEMBERS. ELEMENT = = THIS MEMBER
       THEN (*AGG. MEMBERS : - NEXT*)
       ELSE (*WHILE NEXT. ELEMENT =/= THIS MEMBER REPEAT
                 (* LAST : - NEXT;
                    NEXT : - NEXT. NEXTELEMENT
                 *) ;
              LAST. NEXTELEMENT : - NEXT. NEXTELEMENT
            *)
    END *** OUT OF AGG MEMBERS LIST *** ;

IF MEMBER OF (AGG)
THEN (* OUT OF IS MEMBER OF LIST;
        OUT OF AGG MEMBERS LIST
      *)
ELSE (* ERROR *)

END *** LEAVE *** ;
```

```
FUNCTION CONTAINS (M) : BOOLEAN
    BOOLEAN BEGIN   test whether THIS AGGREGATE contains M
                    as a direct member. That is, test whether M
                    is an element of THIS AGGREGATE's "members"
                    list.
            M : REF MEMBER;
            NEXT : REF LISTNOTE;

            NEXT : - MEMBERS;
            WHILE NOT RESULT AND NEXT =/= NONE REPEAT
                (* RESULT : = NEXT.ELEMENT == M;
                   NEXT : - NEXT.NEXTELEMENT
                *)
            END *** CONTAINS *** ;
```

```
FUNCTION INCLUDES (M) : BOOLEAN
    BEGIN   test whether THIS AGGREGATE includes M as a part.
            That is, test whether M is an element of THIS AGGREGATE's
            "members" list or one of the members of THIS AGGREGATE
            includes M as a part.
        M : REF MEMBER;
        FUNCTION MEMBERS INCLUDE M : BOOLEAN
            BEGIN NEXT : REF LISTNOTE;
                NEXT : - MEMBERS;
                WHILE NOT RESULT AND NEXT =/= NONE REPEAT
                    (* IF NEXT.ELEMENT IS AGGREGATE
                       THEN
                         (* RESULT : = NEXT.ELEMENT QUA
                                AGGREGATE.INCLUDES (M)
                         *);
                       NEXT : - NEXT.ELEMENT
                    *)
            END *** MEMBERS INCLUDE M *** ;
        IF CONTAINS (M)
        THEN (* RESULT : = TRUE *)
        ELSE (* RESULT : = MEMBERS INCLUDE M *)
    END *** INCLUDES *** ;
```


Implementation in SIMSET

The list structures could also have been described by means of the SIMSET facilities as sketched below.