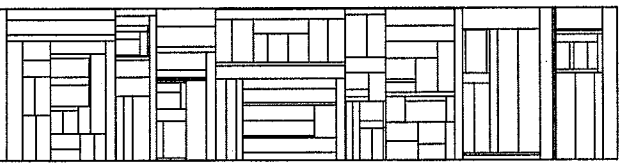
BETA LANGUAGE DEVELOPMENT SURVEY REPORT, 1. NOVEMBER 1976 (REVISED VERSION, SEPTEMBER 1977)

by
Bent Bruun Kristensen
Ole Lehrmann Madsen
Kristen Nygaard

DAIMI PB-65
September 1977

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



This report is included in the publication series from the following institutions:

- The Regional Computing Center at the University of Aarhus, Denmark as report RECAU-76-77.
- The Computer Science Department, Institute of Mathematics, University of Aarhus, Denmark as report DAIMI PB-65.
- The Norwegian Computing Center, Oslo, Norway as report NCC Publication No. 559.

This report is also referred to as:

Joint Language Project Working Note No. 2.
DELTA Project Working Note No. 3.

BETA Language Development

Survey Report, 1. November 1976

(Revised Version, September 1977)

by

Bent Bruun Kristensen, University of Aalborg, Denmark,

Ole Lehrmann Madsen, University of Aarhus, Denmark, and

Kristen Nygaard, Norwegian Computing Center, Oslo, Norway.

September 1977.

Contents

Abstract

The report describes ongoing work within the Joint Language Project (JLP). Research workers from Aarhus and Aalborg Universities, Denmark and the Norwegian Computing Center, Oslo, Norway participate in the project. The aim of the JLP is to consider new tools in programming by the development of a system programming language BETA and a high level programming language GAMMA, both related to the system description language DELTA. The state of the ideas for BETA in November 1976 is presented.

Preface to the Revised Version.

This report was prepared as a Working Note within the Joint Language Project (JLP) in November 1976. The purpose was to sum up the state of the BETA development till 1. November, 1976, and to give a survey of the JLP and the basic BETA ideas to a limited set of interested people.

We also intended to provide a published version of this Working Note. This version has been delayed for two main reasons:

- the language development proceeded rapidly, and particularly our ideas on how to approach parallelism. As a consequence, some of the proposals in the version of 1.November soon were outdated.
- when, in the end of 1976, we started the revision of the original manuscript, we found many minor points which needed a clearer exposition. Since there was no urgent need for a published version, this work was given low priority.

The Working Note, as it now is revised, still presents the state of the BETA Language development at 1.November 1976, with two modifications:

1. A few proposals, mainly relating to parallelism, have been omitted, since they soon were abandoned and also were not related to fundamental aspects of the language.
2. The syntax is modified to conform more closely to the current BETA syntax.

No ideas or porposals from after 1.November 1976 are, however, added to the manuscript. The authors want to thank the JLP team members, Birger Møller-Pedersen and Peter Jensen at the NCC, and Bruce Moon, University of Canterbury, New Zeeland. They have all contributed through discussions, ideas and comments.

September 1977.
Bent Bruun Kristensen
Ole Lehrmann Madsen
Kristen Nygaard

Preface to the Original Version.

This Working Note contains a brief description of the Joint Language Project (JLP) and a survey of the present state of one of its partial projects: the development of the BETA system programming language. The purposes of this survey are:

- to inform other participants in the JLP in order to provide a platform for our joint work.
- to make an initial evaluation and criticism by other interested parties possible.
- to serve as a partial project status report.

A number of the ideas suggested are of a tentative nature and may thus be modified, substituted or just abandoned later.

(Within the JLP, preliminary ideas and results, status reports and other information on work in progress will be presented in Working Notes. Completed work will be presented in Project Reports).

The JLP publications will be included in the publication series of the participating institutions and will for this reason carry multiple serial numbers. Also, they will be given serial numbers within the JLP and the related DELTA Project publications.

Bent Bruun Kristensen,
Ole Lehrmann Madsen,
Kristen Nygaard.

1. Introduction

In the report "The BETA Project" by Peter Jensen and Kristen Nygaard, ref. (1), the authors proposed that the NCC should establish cooperation with other partners in order to implement the BETA system programming language on micro computers. The BETA language is now being developed within the Joint Language Project (JLP) at the University of Aarhus, Denmark, with participation from research workers in Aarhus and Aalborg, Denmark and Oslo, Norway.

BETA is based upon the ideas of the DELTA system description language. This report gives an outline of the work going on within the JLP. Particularly, the report presents the JLP partial project which deals with the main framework of the BETA language.

In "The BETA Project" report, the intended uses of the BETA language were described, as well as its relation to some other languages: the DELTA system description language, the DELTA-derived GAMMA high level programming language, as well as the SIMULA high level programming language. These aspects are not repeated in this status report.

During 1977 new working notes and project reports will present research related to various aspects of BETA and, we belive, at least an initial BETA language specification. (In "The BETA Project" report the time schedule proposed calls for a firm and complete language definition at the end of 1977).

2. The Joint Language Project.

The Joint Language Project (JLP) is carried out by research workers at

- the Departement of Computer Science, Institute of Mathematics at the University of Aarhus, Denmark (DAIMI),
- the Regional Computing Center at the University of Aarhus (RECAU),
- the University of Aalborg, Denmark,
- the Norwegian Computing Center, Oslo, Norway, (NCC).

The initiative for the JLP was taken in the autumn of 1975 by Bjarner Svejgaard, Director of RECAU. The initial purpose of the JLP was twofold:

1. To develop and implement a high level programming language as a projection of the DELTA system description language into the environment of computing equipment.

2. To provide a common central activity to which a number of research efforts in various fields of informatics and at various institutions could be related.

Thus, the JLP is also a research activity aiming at contributing to the development of the field of programming research. The reports are not only to be steps in a process leading to a language implementation, but should also be regarded as attempts to explore new ideas and reorganize existing ideas of a more general interest.

The JLP is strongly related to the DELTA project at the Norwegian Computing Center, Oslo, Norway (NCC). The DELTA project is reported upon in references (2)-(7). Ref. (5) "System Description and the DELTA Language" presents the language and the reasoning behind its definition. It will also be referred to as the "DELTA-4"-report. (DELTA is a system description language which may be regarded as a generalization and development of the SIMULA high level programming language.)

The discussions within the JLP centered in the initial stage around problems of implementation and some unresolved questions related to the DELTA language. As a result six partial projects were defined.

2.1 Distribution and Maintenance.

A number of problems relate to the proper design of software which is to be distributed to a large number of computer installations of many different types, being locally installed, used and updated over an extended period of time. Questions arising in this area are, e.g., distribution formats, standardized updating procedures, documentation, interfaces to operating systems etc.

These questions are discussed in one of the partial projects of the JLP, with Jens Ulrik Mouritsen and Leif Nielsen at RECAU as the responsible sub-team. The discussion is not confined to the problems of a DELTA-derived language only.

2.2 The BETA Development.

The discussion of methods for implementing a compiler and run time system for the new language resulted in a decision to write the implementation in a (possibly augmented) subset of the language.

At this stage it became useful to introduce names for the various languages or, rather, language levels. We now discern three related languages at three different language levels:

- a DELTA language, to be used in system description.
- a GAMMA language, being the projection of DELTA into the environment of the computer and thus being a high level programming language.
- a BETA language, to be used in system programming, being developed by adapting the DELTA concepts to organize the world confronting the system programmer.

As work with the BETA language proceeded, the authors became convinced that the development of a powerful, advanced BETA system programming language was an important task in itself, and not only a convenient stepping stone to the implementation of the GAMMA language.

The development of the BETA language definition is in this first stage the responsibility of the authors of this Working Note.

2.3 Control Structures Within Entities.

The traditional control structures (conditional statements, repetition statements, case statements etc.) were given a provisional treatment in the DELTA-4 report, Section 8.2. We feel that these control structures to a very large extent may be common for both the BETA, GAMMA and DELTA levels. Particularly we feel that this should apply to the action sequencing control within an entity. A large amount of interesting work has been done in this area over the last few years, and we have to extract a useful set of structures, exploiting this effort. Also we have

to take into account that the introduction of computer-assisted program verification techniques may be made easier if control structures are selected with these techniques in mind. Within the JLP team, Benedict Løfstedt at RECAU is responsible for this partial project.

2.4 Data Types.

One main reason for the frequent references to SIMULA in recent programming language research is the properties of the SIMULA "class" concept. Often the class is used as a tool in establishing composite data types. This is, however, a doubtful approach, easily leading to conceptual confusion. (For an initial discussion of the type-quantity issue, see the DELTA-4 report, Sections 3.5.6, 7.1.) In our opinion composite quantities (described by "type declarations") basically should be introduced as items infixed ("in-line") in the entities which they characterize and only accessible as part of these entities (e.g. no references to "real" quantities). This applies both at the GAMMA and DELTA language levels. The basic tools for achieving this must be available at the BETA level, also including means for establishing the set of values which such quantities may take.

For these reasons, the whole question of the definition of types at the GAMMA and DELTA levels, and the associated questions of the means for providing the foundations for types at the BETA level, is defined as still another partial project. Morten Kyng at DAIMI is responsible for this partial project.

2.5 Contexts.

The concept of "system classes", used in SIMULA to provide a set of predefined concepts as a context for a program (e.g. the classes SIMSET and SIMULATION), has been revised and extended in a DAIMI thesis by Birger Møller-Pedersen (spring 1976). The "context" concept proposed in the thesis will be included as an important part of the structuring tools of BETA (and GAMMA and DELTA as well). Birger Møller-Pedersen (from the spring of 1977 at NCC) will be responsible for the further development and definition of this concept, which also may be regarded as a partial project in the JLP.

2.6 Representative States.

A major problem in the execution of a program by a complex of processing units working in parallel is to secure that interaction between program execution components only may result in states of variables etc. which are meaningful, or representative, in relation to the task at hand.

This problem relates to a much wider area of methods for program structuring, but is a particularly important one when parallelism and (parallel and quasi-parallel) interrupts are to be handled.

A conceptual approach to this problem, based upon the DELTA concepts (see the DELTA-4 report, Section 11.1), a report by Lars Mathiassen and Morten Kyng, ref. (9), and work by the authors of this report, is being developed by Morten Kyng and Kristen Nygaard.

The work within the JLP till now has created an active interest outside the institutions participating in implementing the BETA language, particularly as a tool for producing programs for networks comprising micro-computer technology components (but also for the implementation of SIMULA and GAMMA compilers).

3. Concepts for Description of the Computing Process.

In system programming it is, and to a much greater extent will become, necessary to control the components which are entering the program execution process: central processing units, storage media, data channels, peripheral equipment etc. This necessitates in our opinion a general conceptual approach to these apparently very diversified components. It will appear that a large proportion of the software complex now usually referred to as "basic software" and "operating system" will be regarded in our framework as being an integral part of the organization of what we may call "logical components", as opposed to the hardware components.

In the DELTA-4 report this system definition is introduced:

> "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."

The reasoning behind this definition is given in Section 2.1 in the DELTA-4 report. Systems existing in the human mind, physically materialized as states of the cells of our brains, are called mental systems. Systems external to human minds are called manifest systems.

When the term "a program execution" is used, we may either understand

- the action of executing a given program, or
- what is physically generated by this action.

In the SIMULA, DELTA and now BETA language design process the basic reasoning has been related to the structure of a program execution understood as something generated and undergoing a dynamic process of change. A program execution is regarded as a system, and more specifically as a manifest system, existing within the computing equipment.

When information about a system is transmitted from one person to another, this process may be conceived as follows: A person whom we may call the system reporter considers some system about which he wants to convey information through a system description. The system to be described is called the referent system. The system description is given in some language, understandable also to the receiver of the description. This person uses the description to generate in his mind a mental picture, a mental model system of the referent system. Since the essential feature of the receiver in this context is his ability to generate a system on the basis of a system description, he is called a system generator (see the DELTA-4 report, Chapter 2).

The above concepts do, however, apply to a much wider set of situations than just transmission of system information from one person to another. In the standard use of a computer, the system reporter is a programmer, the referent system is his conception of what should be generated within the computer, the language is a programming language, the system description is a program, the system generator is a computer and the model system is a program execution. A more detailed discussion and a number of other illustrations are given in Chapter 2 of the DELTA-4 report.

The "computer" concept is no longer useful for a precise discussion of the structure of the complex networks of interrelated computing equipment which we have to deal with in the computing systems of today and in the future. We shall regard such networks as system generators in the DELTA sense and introduce concepts in terms of which we may understand and describe such a network as a system generator.

The term "program execution" is also too narrow and sometimes misleading in a number of the situations we shall consider. As remarked above, a program execution is a system in the DELTA sense. This point of view relates naturally to our basic approach, as stated later in Section 4.1. For this reason we introduce the definition:

- a system which is being or has been generated by the execution of a program written in a language L is called an L-system.

It follows that the program execution of an ALGOL-program is an ALGOL-system, of a BETA-program a BETA-system, etc.

A system generator (computing equipment network) component upon which such systems may exist will be called a substrate. Discs, tapes, core storage, data screens etc. are examples of substrates.

A system generator component which is able to change the state within a system (in the above sense of the term) will be called a processor. A central processing unit and a disc drive unit are examples of processors.

A system generator component which provides a connecting link between two components of these categories (substrates, processors) will be called a connector. (It should be pointed out that the more complex "data channels" usually will have to be regarded as aggregates of substrates, processors and connectors at the basic "hardware level". They may be given a simpler structure at the "logical level".)

A collection of interacting substrates, processors and connectors will thus be called a system generator. We shall, in fact, understand any complex of computing equipment in these terms, any piece of equipment being regarded as belonging to one of the above component categories or as a subsystem consisting of such components. (A system generator may, of course, itself be regarded as a system.)

The set of instructions which a processor may execute as a consequence of its physical construction only (not altered or augmented by use of software) will be called its machine language or, abbreviated, its M-language. What is established on a substrate by the use of M-language alone is thus called an M-system, and the physical equipment is conceived and used as consisting of M-substrates, M-processors and M-connectors. A complex of equipment used in this manner is regarded as an M-generator.

A system generator is almost never used by prescribing its program executions in M-language. Instead we use some language, L, and we may prescribe program executions which we may conceive as L-systems. Any substrate which is organized so that it may contain an L-system will be called an L-substrate, and any processor which is organized so that it may carry out actions (changes of state) within an L-system will be called an L-processor. In the same manner we define the term L-connector.

L-substrates, L-processors and L-connectors will now no longer directly correspond to well defined pieces of equipment. They are "logical" substrates, processors and connectors, getting their defined capabilities by the use of storage, processing power and software from other components of the system generator (M-substrates, M-processors and M-connectors).

How this is achieved is not the concern of the user of the L-language, who may use the system generator as if it consisted of L-processors, L-substrates and L-connectors, generating L-systems. In this way the concept of "basic software" and "operating systems" will be dissolved, its various constituent parts being distributed and regarded as belonging to logical L-substrates, L-processors, L-connectors and L-systems.

One reason for substituting the term "program execution" by "L-system" now should be obvious: The term "L-system" will comprise a traditional program execution as well as a file on a disc unit and what is generated during a conversation between components of a network and a user, etc.

4. BETA Systems.

4.1 Basic Approach.

In the DELTA-4 report and in the teaching of SIMULA, see e.g. ref. (8), the presentation focuses upon what is generated during the execution of a system description (DELTA) or a program (SIMULA); the "model system" in DELTA, the "program execution" in SIMULA, both consisting of a collection of components called "objects".

Of course, the study and development of the tools for writing program text is important, but as soon as one leaves the simple situation of program executions consisting of one simple stack (perhaps supplemented by a set of always-passive data objects), the understanding of the dynamics of the program execution becomes the essential starting point. The program execution should be regarded as a system for which we want to prescribe precisely defined properties by writing a program text. At the higher level of generality of designing a programming language

a corresponding approach has to be used: We have to start by discussing how we want to organize the systems (program executions) which are to be generated when program texts (written in the language to be developed) are executed. The outcome of such discussions then will determine the basic features of the language. Obviously, considerations relating to clarity of textual expression, etc., enter the language design process, but are not essential in the first, basic stage.

4.2 Entities.

Our purpose now is to develop suitable properties of BETA-systems so that they may be used to organize the tasks confronting a system programmer. The next step is to develop language elements, together comprising the BETA language, to be used in system programming, that is, in prescribing specific BETA-systems.

Since BETA is based upon DELTA's system and system description concepts, in our work we are also using the DELTA terminology. In DELTA an attempt is made to create a consistent terminology relating to the very wide class of systems which may be described in that language. This class contains the class of program executions, but is much wider. The DELTA terms to a large extent have been drawn from those used in programming languages, many are new, and some are new (but hopefully improved) names for old concepts. Below we shall mention in parentheses the most closely corresponding programming language term when a DELTA term is being introduced.

As a first step we introduce the concept of an entity. (Block instances, arrays, text objects are entities). We shall build BETA on the following, well-known basic organization:

A BETA-system as existing upon some subtrate will consist of a (usually variable) collection of entities. In the corresponding system description (program) an entity is described by an entity descriptor. (Program text blocks are entity descriptors).

An entity will consist of subtrate areas containing an attribute part and an action directive. Within the entity descriptor the attribute part is described by a set of declarations and the action directive by a sequence of imperatives (a body consisting of a sequence of statements).

The attribute part consists of a data item part and a pattern part. The data item part consists of subtrate segments (storage locations) containing values of the data items defining the state of the entity at any given moment. Data items are either quantities (real and Boolean variables are examples of quantities) or references. The pattern part consists of descriptions, pattern declarations, which may be used in generating entities. (In SIMULA, e.g., procedure and class declarations correspond to pattern declarations).

The action directive consists of substrate segments containing a sequence of instructions to processors, specifying an action sequence to be executed as associated with the entity. An instruction specified in BETA will, as stated above (see also the DELTA-4 report, section 3.7.1), be called a BETA-imperative. The basic physical format of a BETA-imperative as materialized upon an M-subtrate will be, of course, a sequence of M-imperatives, i.e. instructions in M-language (possibly supplemented by local data items used in the execution of what corresponds to a BETA-imperative).

Obviously, there will exist also BETA-imperatives specifying the generation of new BETA-entities and the switching of the processor's actions from the action directive of one entity to that of another.

Considering an entity, we may distinguish between three basic aspects: its structure, its specification and its state.

The structural aspects of an entity are those properties which it has by being a BETA entity, and its mode of generation, determined by the basic definition of BETA systems and the BETA language. The specified aspects of an entity are those properties which are described within the entity descriptor, by the use of the language (BETA). All entities which have been generated in the same way, by a given constructional mode (se later) have common structural properties. Entities having a common entity descriptor have common specified properties. The state aspects of an entity are those which are associated with a specific point of time: the set of values of its data items at that point of time, the stage of execution of its action sequence, (some of these values may, of course, be constant).

When an entity is generated, its specification (entity descriptor) may be given as a part of the language element causing its generation (stating its constructional mode). This is e.g. the case with inner block instances in ALGOL and SIMULA. The entity is then said to be singular.

In other cases, an entity's properties may be specified by referring to a pattern declaration, specifying a set of properties common to a group of entities. In these cases the entity is said to be category - specified. The referring to the entity descriptor of this pattern declaration is made through an identifier associated with the declaration. This identitifer is called the title of the pattern. An identifier indicating a specific entity is called a name of that entity.

Entities having the same set of structural properties are said to be of the same kind. The basic features of a programming language are to a large extent determined by the kinds of entities available and the structural properties they are given. Using ALGOL 60 as an example, ALGOL-systems will contain three kinds of entities:

1. ordinary block instances, being either singular (inner block instances) and category-specified (procedure block instances).
2. type procedure block instances, always being category-specified.
3. arrays.

In DELTA entities of the first kind are called instances, of the second kind evaluations (since a value is associated with the entity), the third kind array entities.

In ALGOL we have also, in fact, a fourth kind of entities, associated with variables, being category-specified data items whose properties are determined by various predefined specifications called data types. In DELTA and BETA such data items are called quantities. Since ALGOL quantities all have a very simple data structure and language-defined properties, we do not usually think of them as being associated with entities which are constituent parts of other entities. In DELTA and BETA, however, the user will be able to specify more complex quantities.

In order to obtain an efficient organization of a BETA-system, the obvious solution is to separate the system in two main parts: a system descriptor and a (variable) collection of value records. The system descriptor consists of descriptors for all entities which may occur within the system, translated into a format

which may be used by the processors executing actions within the system. A value record consists of a substrate area containing the structural and specified data items of an entity. Among the structural data items are a reference to the entity's descriptor and to the imperative defining the stage of execution of the entity's action directive.

An entity thus consists of the union of a value record and a descriptor. Two or more entities may share a common descriptor.

If the attributes and actions of an entity E1 are integral parts of the attributes and actions of another entity E2, then E1 is said to be a constituent entity of E2. If the attributes and actions of an entity E1 are not parts of the attributes and actions of any other entity, E1 is said to be an autonomous entity.

In ALGOL, quantities (variables) and arrays are associated with constituent entities. Instances and evaluations are autonomous entities.

4.3 Nesting.

As another basic organizational principle we introduce the possibility of specifying that an autonomous entity may be internal to another autonomous entity, which then is called the first entity's encloser (see the DELTA-4 report, Section 3.5.4). This implies that the life span of the internal entity is confined to that of its encloser, and that it contains a constant structural attribute indicating the encloser. An entity may contain any number of internal entities, which once more may contain internal entities, etc.

The resulting interrelationship between the autonomous entities of a BETA-system will be a nested structure, or simply a nest (see the DELTA-4 report, Section 3.5.1).

Every autonomous entity will have one, and only one encloser, except one entity containing all the others. This entity will be called the system entity.

4.4 Entity Kinds and Constructional Modes.

As stated above, a high level language is defined to a large extent by the various kinds of entities available and their structural properties in terms of data items, patterns and actions.

In BETA an entity's specified properties are determined by its entity descriptor. Its structural properties are determined by the way it is generated, its constructional mode.

The BETA language notation for an entity descriptor is

BEGIN entity description END

In some cases an entity generated by some constructional mode will be anonymous, it cannot later be referred to by a name. Means do exist for the naming of autonomous entities and constituent entities.

If we want to use the descriptor to generate a group of entities sharing the same specification, we shall call the descriptor a pattern. This is indicated by the key word PATTERN followed by a title which is used to indicate the pattern, and the resulting language element is called a pattern declaration.

The BETA language notation is

PATTERN P : BEGIN entity description END P;

(In ALGOL and SIMULA the brackets BEGIN and END also are used to embrace compound imperatives. This is not the case in BETA; instead the bracket pair (* and *) is used).

In SIMULA we have both a procedure declaration, a type procedure declaration and a class declaration. In addition we have implied a type declaration in some language predefined versions (specifying the properties of REAL, BOOLEAN etc. quantities). The structural properties of entities, their kind, are given by the choice of the key words in the declaration stating its specified properties (PROCEDURE, CLASS etc.), by an indication of the predefined declaration in the generating language element as in e.g. REAL X, BOOLEAN B or by a combination (REAL PROCEDURE etc.).

In other block structured languages similar approaches are used. In general only one constructional mode is associated with a specific kind of entity declaration (the procedure imperative with the procedure declaration, the NEW imperative with the class declaration, the declarations REAL X, BOOLEAN B with the implied predefined type declaration of real and Boolean quantities).

In BETA we have only one, basic entity specification, specifying either a singular entity or a pattern. The kind of the entity is then determined by the selection of a proper constructional mode used for its generation.

In order to give the desired power and flexibility, a basic range of constructional modes are provided, combined with the possibility of building up aggregated and specialized patterns which may be used to define a range of entity kinds with the properties desired, suitable to the task at hand.

(Examples are: a set of entity kinds corresponding to the kinds of ALGOL or SIMULA, used in writing an ALGOL or SIMULA compiler and run time system; a set of entity kinds to be used in organizing an operating system).

4.4.1 Autonomous Entities and References.

An anonymous autonomous entity may be generated either as singular by the language element

NEW BEGIN entity description END

or as category specified by

NEW P

where P is the title of a pattern.

The naming of autonomous entities are achieved by references.

Data items are in BETA (as in DELTA) either references or quantities. The basic definitions and discussion of the nature of references and quantities are given in Section 3.5.6 of the DELTA-4 report, and the reader is referred to this report. In this Working Note we shall express ourselves more briefly and state that

- a reference is the association of a name and a value which is the substance of an autonomous entity (or "no entity").

(The value is the substance of the entity since it does not change even if attributes of the entity do change values. We do have also, implicit and unavailable for direct use by the programmer, structural references whose values are imperatives).

A reference attribute having a singular entity as its (always constant) value is specified by the reference declaration

REF R: BEGIN entity description END

where R is the name of the reference.

A reference attribute having a category specified entity as its value is specified by

REF R:P

where P is the title of a pattern. P is called the qualification of the reference. The implication is that the range of values of such a reference is restricted to entities specified by the same pattern. This restriction is made to guard against invalid indication of data items supposed to be attributes of the entity which is the value of R.

By this a qualified reference R is only specified and given a range of possible values. Actual assignment of a value is made by assignment imperatives as e.g.

R:- NEW P

(with the same notation as in SIMULA. The generation of constant category specified references will also be possible).

By the use of a reference we may also indicate the attributes of its (entity) value. If an attribute A is specified as an attribute of a pattern P, we may indicate the A-attribute of the P-entity which is the value of R by

R.A

(R "dot" A, or R's A).

Since we want to implement BETA in BETA itself, the question of the use of unqualified references does arise. We have not yet made any discussion as to the extent and form of unqualified references to be made available to BETA users.

4.4.2 Quantities and Infixing.

In ALGOL and SIMULA the language predefined type declarations (for real, Boolean etc.) generate contituent entities which are used to construct entities. These entities are associated with variables, and they are infixed in the surrounding entity. This is specified in the attribute speficiation part of the descriptor of the surrounding entity by variable declarations referencing to the proper types (REAL X etc.).

This constructional mode, which we shall call infixing, is in BETA related to the quantity concept, being a generalization of the ALGOL and SIMULA type variables. (The reader is once more referred to Section 3.5.6 of the DELTA-4 report for a more precise, if not complete discussion.)

As a shorter definition we shall state that

- a quantity is the association of a name and a value which is the state of the measure part of an infixed entity.

Bothe category specified and singular entities may be infixed and associated with quantities.

A quantity which is category-specified by the pattern P is specified in BETA by the quantity declaration

```
QUANTITY Q:P
```

If the quantity is singular, it is specified by

```
QUANTITY Q : BEGIN entity description END
```

In BETA the standard data types will be regarded as specified by patterns, and e.g. a real quantity variable specified by

```
QUANTITY X : REAL
```

4.4.3 Prefixing.

Another constructional mode, also available in SIMULA, is prefixing. Prefixing implies that an entity, referred to as a prefix part, becomes a permanent, constituent upper part (or "outer" part) of another entity. The organizational properties of prefixing are different from those of infixing and correspond to those of SIMULA's "subclass" concept. Prefix parts may be specified both for singular entities and patterns. Prefix parts must be category specified. A singular entity with a prefix part, specified by the pattern P, is specified by the notation:

P BEGIN entity description END

If all entities specified by the pattern Q have a prefix part specified by the pattern P, the notatation used is

PATTERN Q : P BEGIN entity description END Q;

A prefixed pattern (as Q above) is called a sub-pattern of its prefix pattern. A sub-pattern may in its turn be used to specify another prefix etc.

4.5 Repetition.

Repetition implies that a specified number (e.g. N) of references with the same qualification, or quantities being specified by the same pattern are incorporated with a common name. The specific indication of one of these references or quantities is achieved by the use of a subscript as a selector. The specification is

REF R : [N] P

and

QUANTITY Q : [N] P

More flexible array concepts may be built by the combined use of constructional modes and patterns.

4.6 The Dynamic Structure of BETA Systems.

BETA-programs will have a descriptor tree structure corresponding to that of other block structured languages, as e.g. ALGOL. The dynamic structure of BETA-systems is, however, different from that of ALGOL.

In an ALGOL-system all entities are organized into one stack. When a new entity is generated it is attached to the stack. This implies that it relates to the other entities, already being members of the stack, as a new top member at the top of the stack. The processor executing the actions within an ALGOL-system will always execute the actions associated with its top member. When the actions associated with a top member is executed, one possibility is that a new member is generated and located at the top of the stack. Another possibility is that the action directive is completed. In this latter case the entity becomes terminated and the actions associated with the previous top member are resumed. The terminated entity disappears from the ALGOL-system.

The entity at the bottom of the stack is called its basic member. When the basic member terminates, the program execution is completed and the ALGOL-system ceases to exist.

This well-known stack organization is ideally suited for the organization of what is conceived as a single action sequence, and it will be used as the basic building block for the dynamic structure of BETA-systems.

In more complex situation we may want to organize a system (program execution) as containing a number of components, with each component possibly having an associated action sequence. If two or more of these action sequences may go on in parallel, the situation is called parallelism. We will give some remarks on that situation later on.

We may also organize the action sequence of the components as a sequence of subsequences. If the action sequence of the total system consists of a sequence of subsequences, each of which is associated with one of the components, then the situation is called quasi-parallelism.

A special case of this sequencing scheme is the dynamic structure of SIMULA: a SIMULA system consists of a (variable) collection of entity stacks, called objects. (In SIMULA only object heads and prefixed instances may be basic members of objects).

A BETA-system consists of a collection of entities, organized in stacks containing autonomous entities. These stacks are called BETA-objects (or just objects).

An entity which is a non-basic member of an object (stack) is said to be in an attached state. An entity which is not in an attached state will be regarded as a basic member of an object (possibly consisting of this basic member alone), and will be said to be in a detached state.

The basic member entity of an object will be regarded as representing the object towards other parts of the system and may also be named the object entity.

Internal entities (always autonomous) may also be detached in BETA (as in SIMULA), and an entity thus may develop internal objects.

The result is that a BETA-system will be organized dynamically as a nested system of BETA-objects.

The BETA language contains imperatives which organize the insertion and deletion of autonomous entities in BETA-objects.

An object X may be attached to the object Y by e.g. the imperative

ATTACH X TO Y

(another imperative syntax may possibly be used). This implies that the entity stack of the object X is inserted at the top of the entity stack of the object Y.

If X is a reference to an entity which is a non-basic member of some object Y, then the execution of the imperative

DETACH X

will result in the deletion of X and all the entities on the top of X from Y. These entities now constitute a new object with X as its object entity (or basic member).

The imperative controlling the quasi-parallel sequencing is

RESUME X

When this imperative is executed as a part of the action directive of an object Y, the effect is that Y's stage of execution is recorded in one of Y's structural attributes, the execution of the top member of Y's entity stack is temporarily stopped and the actions associated with X's top member are initiated at the stage of execution where it was halted the last time X was active.

When an object X is attached to an object Y, both X and Y may have internal objects. This may create confusion in the desirable strict organization of action sequences by stacks, especially when many parallel processors are considered.

To remedy this situation various solutions may be proposed. At the present moment the authors tend to introduce the following restriction on the dynamic structure:

> Actions associated with internal objects cannot be executed when their encloser is in an attached state.

(The relation of objects to processor may possibly provide exceptions to this rule).

The imperative repertory to be provided to control the object organization has not yet been settled. Very important considerations in this respect relate to the execution of BETA-systems by processors working in a truly parallel mode.

4.7 Interrupts.

In this section we discuss the interrupt concept. We start by examining the information content of a so-called "hardware interrupt". Then we will extend the interpretation of such an interrupt at the M-language level to what is a more convenient "logical interrupt" concept at higher language levels. The third step is to define the meaning of a "BETA-interrupt" and a reasonable principle for the handling of such interrupts.

Let us consider a M-system MS, containing two M-subsystems MS-1 and MS-2. Assuming that our hardware complex is organized as an M-generator (i.e., using M-language), an M-interrupt (hardware interrupt) is generated by some action within MS-1. The purpose of the M-interrupt is to obtain that some specified action or action sequence be executed within MS-2.

Let us assume that the actions within MS-1 are executed by an M-processor MP-1 and those of MS-2 by an M-processor MP-2. Three different situations may now occur:

1. Both MS-1 and MS-2 have their actions at the time of the generation of the interrupt executed by the same M-processor, or, MP-1 is identical with MP-2. The joint system will then be organized by some sequencing principle, e.g. quasi-parallel. In this case we will classify the interrupt as an internal interrupt.

2. MP-1 and MP-2 are at the time of the interrupt two different M-processors. In this case we will classify the interrupt as an external interrupt.

A third conceivable case is

3. MS-2 does not at the time of the interrupt have any M-processor assigned which executes actions within MS-2, or, MP-2 is "no processor". In this case we will also classify the interrupt as an external interrupt.

Let us examine the cases 1 and 2. (Case 3 is a special case which is irrelevant at the M-level and will be excluded at the BETA-level.)

We will start with case 2. Even in its most rudimentary form, an M-interrupt must contain three pieces of information:

- a signal to MP-2 indicating that an interrupt has occurred.

- sufficient information to allow MP-2 to select and initiate some desired action sequence within MS-2. (This may imply that MP-2 has to analyze its internal registers, the state of MS-2, etc. If only one action

sequence is specified for initiation by the occurrence of an interrupt from MP-1, the possibility of identifying MP-1 as sender of the interrupt is sufficient.)

- information about the "importance" of the execution of the action sequence, i.e., some priority information. (Sometimes no such information is given, which may imply that all interrupts have equal priority.)

Regardless of the M-level implementation details we may conceive the immediate implication of an M-interrupt as:

- the transmission of a signal from MP-1 to MP-2.

- the generation of an (or identification of an existing) "M-level (autonomous) entity", available to MP-2, now containing data items and an action directive specifying the actions which are to be initiated within MS-2, as well as information about the priority of the interrupt.

At higher language levels this organization may be made more explicit, and we may e.g., speak about an "interrupt routine" etc.

In BETA we draw what we regard as the logical consequence of this way of understanding an interrupt:

A BETA-interrupt has two aspects:

- the initiation, by some action within a BETA-system BS-1, of a signal from the BETA-processor BP-1 executing the actions of BS-1 to a BETA-processor BP-2 executing the actions of a BETA-system BS-2, the signal indicating that an interrupt has occurred. (BS-1 and BS-2 are subsystems of a BETA-system BS).

- the indication (possibly also generation) of an autonomous entity E whose actions may be executed by BP-2 as a part of the actions of BS-2 (if any). E should also contain information about the priority of the execution of E if such information is available. Since E contains all available information about the actions which should be enforced upon BS-2, it will be called an interrupt entity or, for short, an interrupt.

Since every action within a BETA-system will be executed in association with an entity which is a member of a BETA-object, E must also contain the information (if any) indicating the BETA-object with which it should be associated within BS-2.

This interrupt concept corresponds to that developed in the DELTA language. The imperative within BS-1 initiating an interrupt to be executed within BS-2 will be given the format

INTERRUPT BS-2 object indication
BY interrupt entity indication

We may distinguish between two phases in the transmission of an interrupt:

1. The reception by BP-2 of a signal from BP-1 and the recording of sufficient information to be able to indicate (and possibly generate) the interrupt entity.

2. The actual indication (and possibly generation) of the interrupt entity and the execution of its actions as a part of BS-2.

Phase 1 has to be executed by BP-2 immediately upon reception of the signal. Phase 2 may be executed at a later stage, dependent upon the action sequencing organization of BP-2.

When internal interrupts are considered, their implications may be organized as a simplification of those of external interrupts:

1. No signal is really necessary. BP-1 indicates (and possibly generates) the interrupt entity as a consequence of its execution of the above interrupt imperative.

2. When a point in the action sequence is reached at which interrupt entities may be executed, BP-2 examines the list of interrupts waiting for execution and decides whether or not to execute one interrupt on this list.

The list of interrupt entities waiting to be executed will be called the agenda. The agenda may be regarded either as a part of BP-2, BS-2 or the substrate of BS-2. These alternatives have not yet been discussed in detail.

4.8 Representative States and Transitions.

The concept of representative states used in this section has been developed initially by the authors of the DELTA-4 report and Morten Kyng as a tool for understanding and describing the semantics of DELTA-systems. Dijkstra has used a related concept, "legitimate states", in another situation in ref (17). Later on Morten Kyng and Lars Mathiassen has discussed the use of the concept in connection with development of system descriptions, ref (4). The present suggestions are the result of a cooperation between Morten Kyng and Kristen Nygaard.

The main purpose of this section is to indicate a few approaches which will be used in BETA's handling of parallelism. The analysis of the associated general problem is a major task which will require much of the team efforts in 1977.

In these efforts, we will try to benefit from the large number of interesting papers which have appeared over the last years, by e.g. Brinch Hansen, Dijkstra, Hoare and Wirth. (ref. 12,13,14,15).

Some objectives of our efforts may be stated as:

- we shall attempt to classify the possible parallel interaction situations into a (hopefully small) number of qualitatively different cases, and analyse mechanisms suitable for each of these cases.

- we shall analyze these situations with reference to a general nested object structure, in order to arrive at rules covering which interactions are meaningful.

- we shall try to select from existing work and own efforts some basic properties which are incorporated as structural properties of BETA objects and which are used as the fundamental and general tools for a general organization of parallelism in BETA systems.

We belive that in this work we may make use of the approach to parallelism used in the DELTA language, even if important modifications result from the discrete nature of computer systems.

Before we start our discussion, we will introduce the notion of a subsystem.

Depending on the structure of an L-system, associated with a language L, we may give meaning to the term L-subsystem. In general an L-subsystem will be a part of an L-system, delimited according to some principle.

For the time being, let us assume that within an L-system LS we have delimited two L-subsystems LSS-1 and LSS-2, and that the actions of (associated with) LSS-1 are executed by an L-processor LP-1 and those of LSS-2 by an L-processor LP-2. We will further assume that LSS-1 and LSS-2 have no common components. (This restriction may to a certain extent be modified, as shown in the next section.)

Very often one will discuss "interaction" between LSS-1 and LSS-2 in such situations. As mentioned in the DELTA-4 report (section 3.5.9) this term is not sufficiently precise, since it implies both that LSS-1 acts upon LSS-2 and LSS-2 acts upon LSS-1. For this reason we will say that LSS-1 involves LSS-2 in its actions when these actions either make use of attributes of LSS-2 or initiate actions within LSS-2.

Such involvement may take on two main forms:

1. The sending by LP-1 of an interrupt generated within LSS-1 to LP-2 and thus initiating the execution of an interrupt within LSS-2. This has been discussed in the previous section.

2. The use within LSS-1 of attributes (by reading or assigning values) of components of LSS-2. The problem centers around the issue of "shared data items" (or "shared variables"). It is necessary to provide principles governing this kind of involvement which secures that the set of values read from LSS-2 in some sense are "meaningful" as a description of the state of LSS-2, and that the set of values assigned always establishes "meaningful" states within LSS-2.

In connection with point 1 above, the question arises: at which stages of execution of LSS-2 should it be allowed to execute an interrupt entity? A reasonable answer is: only at stages of execution of LSS-2 which have established states which in some sense are "meaningful".

A large proportion of the states which are established within an L-system cannot be given a meaningful interpretation in relation to the task which the L-system is intended to solve. E.g., in an expression when we evaluate a square root by an iteration procedure, all states of the evaluation entity used are meaningless in relation to the expression except the final one.

Similar situations are encountered very often. Intermediate states in the evaluation of algorithms are often "meaningless" in the sense that they may not be interpreted usefully in relation to the problem which is to be solved. The use of other L-system components of the values of such intermediate states may create "meaningless" states within these components.

Morten Kyng and Lars Mathiassen in a recent paper have suggested that the DELTA concept of "representative states" may be used by the programmer as a tool to organize his program to reduce the risk of "meaningless" states. It should be pointed out that it is the programmer who should indicate what he will regard as representative states. It will be seen that the concept also relates to the "action cluster" concept proposed by Peter Naur, ref (16) and the "legitimate state" concept of Dijkstra.

By a representative state we shall thus understand a state of an L-system component which has the property that it may be given a meaningful interpretation in terms of the task it performs in relation to the other L-system components.

If a means is introduced of indicating stages of execution at which the state of an L-system component is representative in the above sense, then it is reasonable to require that

- execution of interrupts are initiated within an L-subsystem LSS, whose actions are executed by an L-processor LP, only when LSS as a whole is in a representative state.

- only values obtained in such representative states are read or assigned by other L-system components.

The means of indicating that a state is arrived at in ALGOL, SIMULA and many other languages is the semicolon. The semicolon may thus be named a state indicator. It is a very rudimentary state indicator, since it only indicates that a state is obtained, but provides no further information about that state.

In BETA an additional state indicator shall be introduced, indicated by the key word STATE. When this key word appears between two imperatives, the implication is that a state is arrived at which the programmer regards as being representative.

According to this propsal, interrupts of the action sequence of a BETA-object may occur only at points of the action sequence indicated in this manner.

An action sequence specified by the execution of a corresponding sequence of imperatives, starting with an imperative preceded by STATE and ending with the first imperative executed which is succeeded by STATE, will be called a transition. A transition carries a BETA-object from one representative state to another. The intermediate states within a transition, indicated by the semicolons separating the imperatives specifying the transition will be called transient (or non-representative) states.

4.9 Parallelism.

The previous two sections, 4.7 and 4.8, shall be used as a platform for the BETA approach to parallelism in the execution of BETA-systems. However, this section will contain no solutions but just indicate the problems as we see them now.

As stated earlier, a BETA-system will be structured dynamically as a nested structure of BETA-objects. A BETA-system may be executed by a BETA-generator containing a number of BETA-processors. A BETA-object will execute actions if it has been assigned a BETA-processor. At a given moment of time a number of BETA-objects may have assigned a BETA-processor and in this way execute actions in parallel. In general there will not be a BETA-processor available for each BETA-object so the execution of actions has to be organised as a combination of quasi-parallelism and parallelism.

We must distinguish between two kinds of quasi-parallel execution:

- in one situation a number of objects operate in quasi-parallel because there is not enough available processors, and they could in fact operate in parallel,

- in another situation a number of objects operate in quasi-parallel by sharing one processor, and at most one of the objects may operate at a time. Even if there were more processors the organisation of actions in objects should not take place in parallel. (This mode of operation corresponds to Simula).

In BETA it should be possible to organize a set of objects in both ways. This means that a BETA-system consists of a set of nested parallel sub-systems which all may operate in parallel and inside each parallel sub-system only one object at a time may have assigned a processor.

The important and difficult case is of course the organization of interaction between parallel sub-systems.

BETA is intended for programming at several levels:

- at one level the programmer may have complete knowledge and control over the BETA-generator, and in this way control the assignment of BETA-processors to BETA-objects.

- at a higher level, the programmer may have little information about the BETA-generator which executes his program. This means that he should be able to organize his program as several parallel sub-systems and whether or not they operate in parallel depends on the actual processor-situation - unknown to the programmer.

Communication between BETA-systems existing on separate BETA-substrates will be organized by principles corresponding to those developed in relation to communication between parallel sub-systems.

5. BETA Language Constructs.

In Chapter 4 the basic features of BETA-systems were briefly discussed and developed. In accordance with the "basic approach" defined in Section 4.1, the BETA language elements were derived from decisions on BETA-system features.

In this chapter some important language element proposals will be presented, and the material will be structured in a way more suitable to a language introduction. At this stage large sections of the language are still open to further discussions and modification, before a more comprehensive proposal is made. For this reason we feel that it is not useful to invest much

effort in stating details of syntax and semantics. The presentation will be informal and sketchy, often only with the purpose of conveying an impression about what kind of elements will be proposed.

The notation used for syntax description will be a simplified version of that introduced in Section 4.5 in the DELTA-4 report. It should be self-explanatory as used here. As usual the brackets < and > are used to embrace a syntactical element specification. We will in addition use the syntactical key word optional to indicate that an element may be omitted and the brackets ⊰ and ⊱ to embrace a composite syntactical element.

5.1 The Entity Descriptor.

The basic syntactical element used when an entity is to be specified is the entity descriptor. The format is

```
<entity descriptor> is
     optional <prefix>
     BEGIN
     <interface specification part>
     <attribute specification part>
     <measure specification part>
     <action directive>
     END
```

The <prefix> is an (entity) pattern title. The <prefix> specifies the entity category to which the entity's prefix (if any) belongs.

In the <interface specification part> we should be able to specify

- the exclusion of specified language elements from use within the entity (the prefix entity excepted) and all the entity's internal entities.

(Whether or not it should be possible to specify re-inclusion of earlier excluded language elements is not yet decided upon).

In the <attribute specification part> we may specify

- singular (autonomous) entities (with associated infixed singular references) and entity patterns being internal to the entity,
- singular infixed entities (associated with quantities),
- category defined infixed entities (associated with quantities) and category qualified references,
- virtual attributes.

In the <measure specification part> we should be able to specify

- a set of possible values,
- a method for assigning one of these values as the current value of the entity,
- operations upon such values,
- a method for assigning an initial value of the entity.

In the <action directive> we may specify

- a set of possible action sequences which is associated with the entity. The specification is made by a sequence of BETA-imperatives (statements).

5.2 Construction al Modes.

An entity pattern P is specified by a pattern declaration :

PATTERN P: < entity descriptor> P

This declaration does not in itself generate any entity. The generation of an autonomous entity, category defined by P is specified by

NEW P

if singular the specification is

NEW <entity descriptor>

The generation of a singular, autonomous entity with the singular reference (name) E is specified by

REF E: <entity descriptor>

A reference R whose set of possible values is restricted to autonomous entities belonging to a category defined by the pattern P or a pattern prefixed by P is specified by the reference declaration

REF R: P

The possibilities may be extended, in the same way as for quantities, to include repetition and initialization.

We may generate two kinds of constituent entities (non-autonomous), by infixing and prefixing.

The infixing of a single entity Q (thus being associated with a quantity) belonging to a category defined by a pattern P is specified by the declaration

QUANTITY Q: P

The notation for the infixing of a singular quantity is

QUANTITY Q: <entity descriptor>

We may, of course, specify a list of quantities by the standard format

QUANTITY Q1, Q2, ..., QN: P

A sequence of quantities may be specified by supplementing the declaration by a repetition clause

QUANTITY Q: <repetition clause> P

In this case a particular P quantity in Q is indicated by

Q <subscript clause>

It should also be possible in some way to specify the initial value of a quantity.

Prefixing is obtained by specifying the pattern title of the entity category to which the prefix entity belongs. Prefixes may only be category defined, but may be used to prefix both singular entities and patterns.

5.3 Constants and Restricted Involvement.

It has to be possible to specify that the value of a reference or the value of an entity is to be kept constant. If the entity is infixed, this will imply that we may specify that a quantity has a constant value.

The possibility of one entity involving other entities in its actions has to be clearly defined in terms similar to those discussed in Section 3.5 in the DELTA-4 report. In addition

to the structurally defined restrictions defined by scope rules, we may also want to extend the degree of restriction to include, e.g., the following possibilities:

- an attribute is only made observable by other entities except those internal to the entity (observable implies that the value is available for reading, but not modification). Correspondingly an attribute may be specified as modifiable only. Or the attribute may be specified as internal, only observable and modifiable by those entities which are internal to the entity.

- an attribute is made unavailable in entities with some specified prefix.

- restriction of the use of an attribute to its entity only.

No definite set of possibilities has yet been worked out. Obviously, the use of attributes with restricted availability in the specification of unrestricted patterns may present some problems.

5.4 Interface Specification.

The set of possibilities available in the <interface specification part> has not yet been discussed in any detail. Among the possibilities which may be useful we may mention, e.g.,

- the exclusion of any use of attributes of or references to entities external to the entity (only the entity itself and its internal entities being available).

- the exclusion of the use of a specified list of identifiers (names and titles).

- the restriction to the use of only a subset of the imperatives of the language.

Interface specifications will be particularly useful when contexts are to be specified. An interpretation of the language elements discussed in the previous section into the <interface specification part> will be considered. The same does apply to virtual specifications.

5.5 Imperatives, Expressions and States.

The BETA imperatives (statements) may be classified as follows:

1. Generator imperatives
   - 1.1 Substrate imperatives
   - 1.2 Processor imperatives
   - 1.3 Connector imperatives

2. System imperatives
   - 2.1 Entity external imperatives
   - 2.2 Entity internal imperatives

The generator imperatives are those which contain explicit references to the BETA generator components. The interrupts treated in Section 4.7 relate to BETA-system components and only implicitly to processors. Of course, the task of keeping track of these interrupts, assigning them to their appropriate BETA-objects and executing their actions is the task of the BETA-processor executing the actions of the BETA-subsystem in question. But the programmer should not have to specify anything which may as well be regarded as being part of the internal task of the logical component (a BETA-processor).

We have not yet done work on discussing the generator imperatives. Some ideas are mentioned in Section 5.8. At this point we should only mention that we should like to preserve the above

classification of imperatives also in, e.g., situations in which a BETA-system BS-1 on one BETA-substrate S1 interrupts another BETA-system BS-2 on another BETA-substrate S2, and S1 and S2 are not both sub-substrates of an enclosing BETA-substrate S.

The entity external imperatives are those which relate to

- generation of new autonomous entities (their internal structure determined by their declaration, by e.g., use of the available construction modes).

- the organization of BETA-objects by insertion and deletion of entities from the BETA-object stacks.

- the organization of the sequence of the object phases, i.e., the switching of the processor's attention from one BETA-object to another.

Generation of autonomous entities is achieved in five different ways:

1. By singular (internal) entity specifications in the entity's attribute part, the syntax being

   REF E: <entity descriptor>

   The entity is generated as part of the generation of its encloser (in which the above specification is inserted). The singular and constant reference E is at the same time becoming a data item of the encloser. The internal entity will have the same life span as its encloser, and it is upon generation detached. This implies that it may also be regarded as a new BETA-object.

2. Category-specified detached entities are generated by the construct

NEW P

where P is a pattern title. If the entity is not immediately attached to a BETA-object (or assigned as a subsystem to another processor) it will be anonymous and unavailable from other BETA-objects and thus regarded as non-existent. For this reason it is necessary to use a reference assignment (assign a name)

X: - NEW P

to keep a new detached category-specified entity in the BETA-system. (X is a reference qualified by P or by a pattern which has P as a sub-pattern.)

Another possibility is to include the generation in the attribute specification part as

REF X : CONSTANT P

where X now becomes a constant reference to the new entity.

3. Anonymous singular entities may be generated by the construct

NEW <entity descriptor>

Obviously, the reasoning about existence given above applies also in this case. Since no name may be assigned to such entities, these entities have to attached immediately to a BETA-object. This attachment has to be specified by an imperative and will be described later in this section.

4. Anonymous category-specified entities may be generated within expressions. Let X be a quantity described by a pattern with the title PL and let PR be the title of another pattern. Then the imperative

X: = PR

is valid if certain relations between PL and PR holds.

There are two cases, either PL=PR or PR is a subpattern of PL. In both cases the implication is

- a new autonomous PR-entity is generated,
- this PR-entity is attached immediately,
- when the PR-entity terminates its value is determined, the part of the value defined in the PL-prefix (this is the whole value if PL=PR) is assigned as the new value of X, and
- finally the PR-entity disappears.

(In the above assignment example we have assumed the existence of an ":="-operator for the pattern PL.) This kind of entity will correspond to "type procedures" in ALGOL.

5. Anonymous singular entities may also be generated within expressions if the entities do possess a prefix part satisfying the same requirement as PR in pt. 4 above We may e.g. write

X: = PR <entity descriptor>

(If such a prefix is lacking, only values of attributes of the anonyms singular entity may be used, assuming that they satisfy the prefix part requirement.)

The basic format of the imperatives which control entities' membership of the stacks of BETA-objects is as follows

ATTACH X TO Y
DETACH X

The implication of the first imperative is that the object X (perhaps a single entity) is inserted at the top of the object stack of the BETA-object Y. (The effect of the imperative if X is not detached (not an object) should probably be "no effect".) At present we feel that the internal BETA-objects of X (if any) should not execute any actions as long as X is in an attached state. If it is desired to attach an object to the object executing the ATTACH-imperative, this is achieved by the simplified version

ATTACH X

which has two important cases

ATTACH NEW P

corresponding to a "procedure call" in other languages, and

ATTACH NEW <entity descriptor>

corresponding to an "inner block".

ATTACH/DETACH is a manipulation of stacks and the object executing the ATTACH/DETACH statement continues to execute actions. Note that an object may attach an object to itself and in the way shift the execution of actions to another entity (now being member of itself).

When the action directive of an attached entity reaches the final END, the entity becomes terminated and can execute no more actions. The processor will detach the entity and resume the actions of the entity below it in the BETA-object

stack. If the entity is detached, the arrival at the final END of the action directive also causes termination. Probably the processor then should resume the enclosing entity. Inspection of this and other cases demonstrates that there is a close connection between the ATTACH-imperative and the INTERRUPT-imperative which has to be explored.

The DETACH-imperative has the converse effect. If X is in an attached state, it now becomes a BETA-object having in its object stack all those entities which were above it in the stack.

In order to stop the actions of an active object and resume the action of another object X by the imperative which follows after the one last executed in X's action directive, the imperative used is

RESUME X

The INTERRUPT-imperative has been discussed briefly in Section 4.7. All these imperatives need a thorough study in order to arrive at a suitable sufficient set of imperatives to cover all relevant situations. (All cases cannot be covered by those mentioned here.) In particular, it should be explored to what extent the ATTACH-imperative may be regarded as a special case (within a quasi-parallel subsystem) of the INTERRUPT-imperative.

The entity internal imperatives will not be discussed in this report. As mentioned in Chapter 2, the entity internal control structures are made the subject of a JLP partial project.

In relation to expressions, some proposals have been made earlier in this section. We only want to add that the value of an autonomous entity, referred to by the reference X, can be indicated by the notation

X. VALUE

So far we have not discussed the subject of establishing initial values of attributes of entities. Some aspects of this question will be discussed in the next section. At this point it only should be mentioned that an associated question has to be resolved in a manner which does not destroy the principles developed in this section: that of transmitting values from a terminating entity to the entity below it in the BETA-object stack.

In Section 4.8 we pointed out that the symbol semicolon ";" may be regarded as a state indicator, and that we need an additional state indicator to be used when what we choose to regard as a representative state is arrived at, and interrupts are allowed to occur. At the time being, we want to introduce the notation

```
STATE <state descriptor>
      <priority clause>
      <exit clause>
      <reentry clause>
```

where any combinations of these clauses may be empty.

The <state descriptor> may be used to describe details of the state obtained and is related to the "invariant" concept. The <priority clause> will indicate the "resistance" of the object against letting itself be interrupted in this state.

It will be compared with a corresponding "power" priority of the interrupts indicating their importance. The <exit clause> may specify actions which are executed before an interrupt is executed, if accepted. Correspondingly, the <reentry clause> may specify actions executed when returning to this point of the action directive from an interrupt. (The clauses correspond to similar clauses in the "time concurrency imperatives" of DELTA, see Section 8.3 of the DELTA-4 report).

Finally, it should be remarked that the proposals made here regarding values and quantities owe much to discussions going on within another JLP partial project - that relating to "types" (see Chapter 2).

5.6 Parameters and Virtual Attributes.

Until now we have been discussing only closed entity descriptions. For a number of reasons, it is desirable to be able to leave parts of an entity specification open in order to close it at a later stage. In ALGOL and SIMULA the parameters of procedures and class declarations are tools which may be used for this purpose. In SIMULA we have in addition the "virtual" concept as a powerful tool, allowing us to specify virtual attributes which may be bound ("matched") or even redefined at lower subclass levels.

We may have a number of parameter transmission modes.
In SIMULA we have three modes

"value", "reference", "name".

For values of quantities and references we want to adopt the DELTA proposal (DELTA-4 report, Sections 8.2.3, 8.2.4.2, 8.3.2.2 and 8.4.1) except the proposal relating to "functions" (DELTA-4 report, Section 8.4.1). This proposal implies that values of quantities and references may be transmitted at entity generation and values retransmitted upon termination of the entity. (This retransmission in BETA, as mentioned at the end of the previous section, may necessitate some consideration.)

Values can be transmitted to an entity by means of a PUT-clause

```
PUT (*A:= E; T:-F; ...*)
```

If R is a reference to the entity then the construct corresponds to the following sequence of assignement statements

```
R.A:=E; R.T:-F; ...
```

Values can be retransmitted from an entity by means of a GET-clause

```
GET(*X:=A; Y:-T; ... *)
```

If again R is a reference to the entity this corresponds to

```
X:=R.A; Y:-R.T; ...
```

In BETA, PUT/GET clauses are relevant in connection with the imperatives NEW, ATTACH, DETACH and RESUME. Presently we consider the following possibilities

```
NEW P    <PUT-clause>
ATTACH X<PUT-clause> TO Y
ATTACH X<PUT-clause><GET-clause>
DETACH X<GET-clause>
RESUME X<PUT-clause>
```

All the PUT/GET clauses are optional.

Name parameters do not appear in DELTA, and will not appear in BETA. In the case of virtual pattern attributes we will adopt the solution offered in SIMULA (and copied in DELTA).

The important problem is to develop a suitable "pattern parameter" concept. The solutions offered in ALGOL (carried over to SIMULA) by its "procedure parameter" concept, has a number of disadvantages. Some of these disappear because of

our abolishment of name parameters. A further improvement of the situation is achieved by requiring that parameters to pattern parameters should be specified.

We intend to include a pattern parameter concept in BETA, preferably implying a unification of the "virtual" and "parameter" concepts.

5.7 Contexts.

In SIMULA it is possible to use the concept of an external system class to provide a context of predefined concepts to be used in the specification of a SIMULA-system (in writing a SIMULA program). The classes SIMSET and SIMULATION are examples of such language-defined contexts.

Useful as it is, the SIMULA solution still has a number of deficiencies:

- it is not possible to create a union of such classes except by their organization in a class hierarchy.

- sufficient mechanisms are lacking for the protection of the concepts of these classes from incorrect manipulation by the user. (Even with the newly added "hidden" and "protected" mechanisms.)

Birger Møller-Pedersen in his Master's thesis (ref. 10) has discussed the problem of providing contexts and has developed solutions which seem very powerful. The above deficiencies have been removed, and his "context", "subcontext" and "context set" concepts will allow the establishment of one-level (non-hierarchical) libraries of contexts, still keeping the power of SIMULA's "subclass" concept (which may, e.g., be used in resolving name conflicts when a union of contexts is specified).

He also proposes the introduction of specifying attributes as "auxiliary". Auxiliary attributes are freely available for use in the definition, (also for cross-referencing between class specifications) but are completely shielded off from the program using the context.

The proposal also has a number of other interesting features which will not be mentioned in this report. We intend to adapt the context concept as what is, we believe, a very useful part of the BETA-language.

As an example of the use of a context in system programming we may mention the writing of ALGOL, SIMULA and GAMMA compilers. The approach would be to describe the structural properties of the various kinds of entities (object heads, tasks, instances, prefixed instances, evaluations) by patterns collected in a context. The structural attributes and their manipulation may then be protected as auxiliary attributes and by local language restrictions. A declaration of, e.g., a "procedure P" may then, by using the context, be specified as

PATTERN P: PROCEDURE BEGIN ..... END P

A syntax transforming program may then carry the ALGOL format into the one given above. Of course there are aspects of ALGOL and SIMULA which cannot be handled in this simple manner (name parameters, connection imperatives etc.) and which have to be handled with additional BETA programming. Still, the task of writing such compilers should be reduced drastically.

5.8 Generator Specification.

In order to be able to control the components of the BETA-generator (and thus the computing equipment) its structure must be specified and this specification must be available within a BETA-system (and to the programmer writing a BETA-program). We have as yet, not addressed ourselves to this task and only a few remarks may be made.

As a basic approach we shall try to use the DELTA concepts in understanding, describing and organizing the properties of substrates, processors and connectors.

The degree of detail possible in the specification has to be considered. We should like to simplify the concept of a connector as much as possible, and hopefully reduce it to the nature of a qualified substrate or processor reference.

With regard to processors, we believe that it may be useful to supply the basic processor specification with a number of virtual attributes which may or may not be matched in sub-concepts of the processor concept. Also, the "basic" processor may be given a complete BETA language capability which may be narrowed in particular processors (e.g. those handling discs etc.) by local language restrictions. It also has to be decided to what extent (if any) and in what form the registers and other attributes of the processor should be made available explicitly, while still protecting the structural properties of BETA-systems.

We may list some desirable capabilities for the substrate concept:

- it should be possible to specify sub-substrates of a substrate, and sub-substrates of sub-substrates etc.

- it should be possible to specify the borders of sub-strates and sub-substrates, and to modify the borders of sub-substrates.

- it should be possible to specify the available methods of allocating entities to positions on the substrate: fixed, relative according to some list-based scheme etc. (This may imply that a specification of entity generation in some cases has to include a substrate allocation clause.)

- it should be possible to manipulate the location of the descriptor parts of entities, keeping protected and updated the descriptors' relations to other descriptors and the value records' references to their descriptors.

- it should be possible to load a tested out BETA-system in a given state (e.g. an "initial" one) together with the appropriate run-time modules (internal parts of a BETA-processor) into another substrate (also physically separate). This is necessary if BETA is to be useful in developing programs on a program development computer for dedicated microcomputers with small substrate areas. (See "The BETA Project", ref. (1).)

In Chapter 3 we remarked that large sections of what is usually called "basic software" and "operating systems" should be considered by our definitions of substrates, processors and connectors as belonging to such generator components as internal organizing parts. For the time being we believe that this should not be the case with those parts of the operating systems which handle the distribution of resources between the BETA-systems (e.g. "user programs") existing on a substrate.

We feel that any substrate should have a "resident entity" which may be specified and developed into a more complex BETA-system. This BETA-system should carry out these tasks of the operating system and be able to receive, establish and execute a stream of users' programs as subsystems.

5.9 ALPHA-Level Considerations.

Even if we want to implement BETA in BETA, we have to do a certain amount of initial programming in assembly level languages (which may be named appropriately the ALPHA language level).

When this ALPHA-level "kernel" is established, we still cannot write BETA in "pure" BETA. One important reason is that we have to be able to operate upon the structural attributes of entities in order to write e.g. substrate management programs, processors' interrupt handling programs etc. Other reasons are that we may want to specify M-processor register operations, and perhaps want to include occasional ALPHA-level code to speed up execution.

The question now is: Should BETA users be allowed, by explicit specification, to use this extended but less secure level of BETA?

(It should be noted that rather strong features already are available in "strict" BETA. It will, e.g., be possible to define types directly operating upon "words", "bytes" and "bit sequences".)

If this extended, unsecure BETA level is made available, it should at least be done with some safety considerations, such as imposing the confinement of as much as possible of this sort of programming to the writing of contexts.

6. Conclusion.

We hope that this report may convey an impression of the ideas governing the BETA language development, and that some of the ideas may prove useful to other research workers. We will welcome any suggestion and criticisms, which will be considered in the further BETA development.

7. References.

1. Peter Jensen and Kristen Nygaard:

"The BETA Project"
Joint Language Project Working Note No. 1, 1976
Norwegian Computing Center, Oslo, Norway.

2. Kristen Nygaard:

"On the Use of an Extended SIMULA in System Description"
DELTA Project Report No. 1, 1973
Norwegian Computing Center, Oslo, Norway.

3. Roar Fjellheim, Petter Håndlykken and Kristen Nygaard:

"Report from a Seminar on Systems Description at "Skogen" Røros"
DELTA Project Report No. 2, 1974
Norwegian Computing Center, Oslo, Norway.

4. Morten Kyng and Birger Møller-Pedersen:

"Description of a Model of a Single Helix Pomatia Brain Neuron and an Associated Neurophysiological Experiment"
DELTA Project Report No. 3, 1974
Department of Computer Science, University of Aarhus, Denmark.

5. Erik Holbæk-Hanssen, Petter Håndlykken and Kristen Nygaard:

"System Description and the DELTA Language"
DELTA Project Report No. 4, 1975,
Norwegian Computing Center, Oslo, Norway
(The "DELTA-4 report").

6. Kristen Nygaard:

"DELTA-prosjektet og dets tilknytning til problemene i systemutvikling"
DELTA Project Report No. 5, 1976
Norwegian Computing Center, Oslo, Norway
("The DELTA Project and its Relation to Problems in System Development", in Norwegian).

7. Erik Holbæk-Hanssen, Petter Håndlykken and Kristen Nygaard:

"A Brief Survey of the DELTA Project"
DELTA Working Note No. 1, 1975
Norwegian Computing Center, Oslo, Norway.

8. Graham M. Birtwistle, Ole-Johan Dahl, Bjørn Myhrhaug and Kristen Nygaard:

"SIMULA BEGIN"
Studentlitteratur, Lund, and Auerbach, New York 1973.

9. Morten Kyng and Lars Mathiassen:

"Application Oriented Discussion of Formalized Language Tools" (In Danish).
Department of Computer Science, University of Aarhus, Denmark, 1976.

10. Birger Møller-Pedersen:

"Communicating Concepts in an interdisciplinary Project - Four Models of a Lake described in the DELTA Language".
DAIMI PB-77, DELTA Project Report No. 7.
Department of Computer Science, University of Aarhus, Denmark (May 1977).

"Proposal for a Context Concept in DELTA"
DAIMI PB-83, DELTA Project Report no. 8.
Departement of Computer Science, University of Aarhus, Denmark (September 1977).

11. Morten Kyng:

"Implementation of the DELTA Language Interrupt Concept within the Quasi-parallel Environment of SIMULA"
Department of Computer Science, University of Aarhus, Denmark, 1976.

12. P. Brinch Hansen:

"Operating System Principles"
Prentice Hall, England Cliffs, N.J., 1973.

13. E.W. Dijkstra:

"Cooperating Sequential Processes"
in Programming Languages, F. Genuys, Ed.,
Academic Press, New York, 1968.

14. C.A.R. Hoare:

"Monitors: An Operating System Structuring Concept"
Comm. ACM 19, 5 (May 1967), 273 - 279.

15. N. Wirth:

"Modula: A language for Modular Multiprogramming"
Software - Practice and Experience, 7, 1 (Jan. 1977), 3-35.

16. P. Naur:

"Programming by Action Clusters"
BIT 9 (1969), 250 - 258.

17. E.W. Dijkstra:

"Self-stabilizing Systems in Spite of Distributed Control"
Comm. ACM 17, 11 (Nov. 1974), 643-644.