ISSN 0105-8517

# BETA LANGUAGE PROPOSAL

as of April 1979

by

Bent Bruun Kristensen
Ole Lehrmann Madsen
Birger Møller-Pedersen
Kristen Nygaard

DAIMI PB-98
June 1979

Computer Science Department
**AARHUS UNIVERSITY**
Ny Munkegade – DK 8000 Aarhus C – DENMARK
*Telephone: 06 – 12 83 55*

ISSN 0105-8517

Joint Language Project
Working Note No. 5

# BETA LANGUAGE PROPOSAL

as of April 1979

by

Bent Bruun Kristensen *
Ole Lehrmann Madsen **
Birger Møller-Pedersen ***
Kristen Nygaard ***

DAIMI PB-98
June 1979

Also available from the Norwegian Computing Center, Oslo, as report NCC Publication No. 635.

* Aalborg University Center, Aalborg, Denmark
** Aarhus University, Aarhus, Denmark
*** Norwegian Computing Center, Oslo, Norway

## Preface

The BETA language is being developed as a part of the Joint Language Project (JLP). The JLP is described in JLP Working Note No. 2 ref. [1] (the Regional Computing Center at Aarhus University has also participated in the JLP in addition to the institutions of the authors). The Norwegian Computing Center (NCC) has decided to implement BETA as an integral part of a large project aimed at developing improved tools for software production. This project is named SCALA (System Construction and Application LAnguages).

The *development* of BETA has been a joint *research effort* by the authors, assisted and stimulated by colleagues at the institutions mentioned above, as well as many others. The *implementation* of BETA will be a major *project component* in NCC's SCALA software tool project, which in its turn will contain additional research efforts.

The implementation of a language necessitates that it is given a firm, exact and complete definition. The carrying out of this process for BETA is intended to follow approximately the same procedures as those which were used for SIMULA 67, (see ref. [4]).

This Working Note contains the authors' proposal for the definition of the BETA language. It is building upon earlier JLP Working Notes (ref. [1]) as well as the concepts of the DELTA system description language (ref. [2]), which in its turn is related to SIMULA (ref. [3]). It will be apparent to the reader that we are influenced by a large number of other research workers within informatics (some names will be obvious), even if our basic "systems approach" — with its emphasis on the substance of program executions — differs from the "algorithmic and data structuring approach" which has been dominating programming language research.

We find it unnecessary in this Working Note to refer in detail to all influences we are aware of. This will instead be done in the report containing the BETA Common Base Definition, being written after the BETA Common Base Conference later this year.

Up to the Common Base Conference this Working Note and additional written material by the authors and others will be the basis for discussions on the BETA language definition.

The Working Note may be regarded as definitive in its proposal of basic language layout and concepts: The use of *singular* or *category-defined descriptors* combined with *constructional modes* to generate *entities*. The entities will be of three *autonomous kinds: objects, instances* and *contexts*, these possibly containing three *constituent kinds* of entities as building blocks: *prefixes, infixes* and *insertions*. Also, the approach to sequencing, in particular to concurrency of action sequences each associated with an object, is not going to be modified in its essential structure, whereas details may be modified as the outcome of further discussions.

Another important feature is settled: the control of the transition and branching of action sequences between entities through **ENTRY**- and **EXIT**-parts. This feature allows us to see in a common light a number of apparently different situations usually referred to as: parameter transmission, assignment of composite data types, interrupts.

This last-mentioned feature has been developed since the beginning of February this year. It has led to a drastic simplification and, we feel, still an important increase in the power of the language. The present Working Note contains our current version of this feature. We are convinced, however, that its semantics (and perhaps its syntax) may be improved upon.

As a result of these basic design properties, BETA should give its users the capability of a very detailed specification of the structural properties of the substance of the joint execution of a program collection by a collection of varied hardware — still remaining within the scope of a high-level programming language. This capability may be exploited in many ways, e.g. in terms of simplified storage management when this is desirable.

Another result should be the capability of a very direct specification of the action sequences. The mode of specification is oriented towards the actual computing processes, at the expense of (in our opinion misleading) similarity to mathematical notation. It is our belief that this mode of specification may turn out to be an equally powerful (hopefully even better) platform for the application of mathematics to the analysis of properties of programming languages and the executions of programs.

Finally, the association of a compiler-generator with the BETA language is intended to make it very easy to use BETA as a tool for the implementation of other languages which are either

— already defined (as e.g. SIMULA), or

— designed for specific jobs, crafts or professions, to make it possible for information system users to program their own working procedures, and thus their use of the system, or

— reflecting alternative views upon programs and program development (as e.g. interactive program proving, or extracting an electronic information system subset from a general system description comprising both people, machines and information storage, processing and communication equipment).

According to current plans, this compiler-generator will be based upon a design to be developed at DAIMI.

As stated above, this Working Note gives proposals for the main parts of the BETA language. It also contains sections which indicate possible alternatives to the proposals described.

In addition the Working Note presents briefly our current views upon the concepts as yet undefined: the autonomous entity kind named *context*, the *descriptor* being an executable program in some language (or language version) or a possible part of such a program, and the *layer* of a program execution, a program in a given layer regarding those at earlier layers as its *system generator*.

The semantic definition given is mainly informal, using English. However, the static semantics is, as an experiment, defined by a semi-formal notation based on attribute grammars. The notation used is described in ref. [5].

The Working Note describes what may be termed "basic BETA". Any implementation will be supplemented, e.g. by the standard types and their associated operators. This version which may be named "standard BETA" is briefly discussed in section 5.

## References

[1] JLP Working Notes:
WN 1 The BETA Project. Norwegian Computing Center, Oslo, 1976.

WN 2 BETA Language Development. Survey Report, November 1, 1976 (revised version, September 1977). Norwegian Computing Center, Oslo, publ. no. 559, 1977. (Also available as DAIMI PB-65).

WN 3 DRAFT PROPOSAL for Introduction to the BETA Programming Language as of August 1, 1978. Norwegian Computing Center, Oslo, publ. no. 620, 1978.

WN 4 A Definition of the BETA Language. DAIMI TR-8, February 1979.

[2] E. Holbæk-Hanssen, P. Håndlykken, K. Nygaard: System Description and the Delta Language. Norwegian Computing Center, Oslo, 1975.

[3] O.-J. Dahl, B. Myhrhaug, K. Nygaard: Simula 67. Common Base Language. Norwegian Computing Center, Oslo, 1970.

[4] K. Nygaard, O.-J. Dahl: The Developments of the Simula Languages. ACM SIGPLAN, History of Programming Languages Conference. SIGPLAN Notices vol. 13, no. 8, August 1978.

[5] O. L. Madsen: An introduction to Attribute Grammars, by Examples. DAIMI, 1979.

## Contents

# 1. INTRODUCTION

BETA is a general block-structured language in the style of Algol, Simula and Pascal. It is intended to be used for programming computer systems like operating systems, data base systems, etc. It is also intended to be used for defining and implementing application oriented languages. BETA contains a few general concepts which may be used to define more aggregated concepts. Most of the possibilities of Algol-like sequential languages are present. In addition it contains facilities for nondeterminism and concurrent execution of actions. BETA may be used to organize the operating system, file system, program libraries etc. of a computer system into an integrated system all programmed in BETA.

## Abstraction

An important aspect is the abstraction mechanism of BETA (called a *pattern*). A pattern is no specific kind of abstraction but may be used to model abstractions like the procedure, function, type and class concept of the above mentioned languages. Concepts like process, monitor, class, module, cluster, etc. of languages like Concurrent Pascal, Modula, CLU, etc. may also be modelled by patterns. In addition a pattern may be used to define e.g. control structures and what is often called abstract data types.

By block-structured we mean that the language is based on units similar to an Algol block: **BEGIN** declarations; statements **END**. In this connection the important aspects of block-structuring are the possibility of using nested blocks and thus localizing the use of names, the static name binding and the stack organization. Furthermore the use of blocks to define the data part and the possibility of organizing actions as several stacks (coroutines) as in Simula are considered important. In BETA the unit corresponding to a block instance is an example of an *entity*. From a BETA point of view some of the differences between Algol like languages may be understood as different ways of describing, generating and manipulating entities.

A pattern defines a category of entities (with a common structure). Entities in such a category may then be generated by referring to the pattern.

In Algol a procedure declaration is a pattern declaration which describes a category of structurally identical procedure activations. In Pascal a type declaration is a pattern which describes a category of structurally identical variables. A class in Simula describes a category of objects with a common structure.

## Action Structure

Pascal and Algol are sequential languages using one (run time) stack to describe the stage of execution of the program. Simula is a multistack language where a number of objects (stacks) may operate as coroutines. In addition to these modes of operation, BETA contains possibilities for truly concurrent execution of objects. Objects may communicate by means of various synchronization primitives, e.g. by exchange of information and by interrupts. A few basic control structures are defined for sequential programming. However, by using patterns, more advanced control structures may be defined (normally associated with a specific class of data structures).

## BETA Systems

A BETA program execution (called a *BETA system*) is a (possibly varying) collection of entities that exists during a (possibly indefinite) period of time. Some of the entities in a BETA system may execute actions that influence the state of the BETA system and the state of a possible environment of the BETA system.

An environment is a collection of contexts. Besides influencing the states of its contexts, a BETA system may use concepts defined in its contexts.

A collection of BETA systems may exist within a set of computer components. One of these BETA systems will be identified with the computer components.

Some BETA systems may contain BETA system descriptors and generate and control BETA systems according to these descriptors. Such a BETA system is called a BETA system generator. A BETA system generator may dynamically be extended with new BETA system descriptions. Similarly it may dynamically remove BETA system descriptions.

Consequently a BETA program execution is not viewed as a function evaluation which on the basis of some input values delivers some output values.

A traditional program is then a BETA system description that is "given" to a BETA system generator and used to generate a BETA system which has an effect upon some input/output context. The reason for considering systems instead of usual function-like program executions is because of the need to cover operating systems, file systems, data base systems, program libraries, etc. We want to view all information (including programs being executed) on a computer as an integrated system where dynamic exchange of information may be described in the BETA language.

## Hardware Control

A collection of BETA systems existing within a set of computer components is hierarchically organized. The system at the lowest layer will be identified with the computer components. The task of describing the computer components using BETA and of bootstrapping the corresponding BETA system is the job of implementing an initial BETA system on a set of computer components. The initial BETA system contains descriptions of the different physical units in the configuration. These descriptions determine the ways that the hardware can be controlled and used by the collection of BETA systems.

A BETA system existing on some layer will only have partial information about BETA systems at lower layers. Consequently BETA systems at a higher layer have less and less information about the hardware.

If BETA is to be run as part of an existing operating system the initial BETA system must contain a description of the part of the operating system with which it can communicate. As a minimum I/O must be described.

## On Defining New Languages

BETA is also a tool for implementing new languages. The idea is that the semantics of a language is defined using BETA. The syntax of the language will be defined using a meta language like BNF. A syntax transformation from the syntax of the language into BETA then connects the syntax with the semantics. Associated with BETA there will be a compiler generator that, given such a language specification, is able to generate a compiler for the language.

An example of this is the class "Simulation" of Simula. "Simulation" may be viewed as a class that defines the semantics of a simulation language. However, Simula syntax has to be used in order to express simulation programs (the syntax of Simula has been slightly extended in order to ease this). Using BETA it will be possible to invent and use a special syntax for such a simulation language.

In many applications the user of a computer system will need a special language for his kind of application. Such languages should contain constructs that are based on concepts from the given application. We think that there is (or will be) a great need for such application oriented languages. It is in this connection important that such languages are easy to implement.

## On Subsets of BETA

A language like Simula is often inconvenient for systems programming as it contains too much dynamics in its storage allocation. Algol, Pascal and Fortran are gradually less dynamic. Fortran has a completely static (compile time known) storage structure. In BETA it is possible to use exactly the kind of dynamics which are needed.

An important aspect is that it is possible to define a static subset of BETA where no dynamic storage allocation is available. Instead of Algol-like procedure calls one may use Fortran-like subroutines, or inline procedure calls. Instead of having dynamically created Simula like objects one may specify a fixed number of (inline) variables and (offline) objects at compile time. The objects may still operate as coroutines or in concurrency.

It is likely that for many purposes this static subset is sufficient, e.g. when programming dedicated processors, like I/O controllers etc.

# 2. BETA SYSTEMS

## Entities and Patterns

A BETA system is a (possibly varying) collection of *entities*. The properties of an entity are described by an *(entity) descriptor*. Entities are generated according to *constructional modes* which in turn generate entities of different *kinds*. A kind determines the possible ways an entity may be manipulated. Each entity consists of an *attribute part* and an *action part*. In the descriptor the attributes are described by a set of declarations and the action part by a sequence of imperatives.

The attribute part consists of a set of *data items* and a set of *patterns*. A data item may be either the association of a *name* and an entity (of kind *infix*) or a *reference* which is an association of a name and a system element capable of denoting an entity (of kind *object*).

The action part represents actions associated with the entity. The execution of an action may imply state changes, generation of new entities and transfer of control inside the action part or to the action part of another entity. The action part is composed of some basic actions and entities, the latter being of kind *insertion*.

The pattern attributes are a set of possibilities associated with the entity. A pattern is the association of a name (the *title* of the pattern) and an entity-descriptor. A pattern describes a category of entities having a common structure.

A constructional mode must be supplied with a specification of the entity to be generated. A specification may be either

- the descriptor of a pattern, in which case the entity is said to be *category defined*, or
- an entity descriptor, in which case the entity is said to be *singular*.

The lifetime of an entity is restricted to that of the entity containing its descriptor (the *origin* of the entity). If an entity is category defined, its origin is the entity containing the pattern according to which it is generated. If an entity is singular, its origin is the entity containing the constructional mode that generated it. A BETA system starts with one entity, thus a BETA system is organized as a nested structure.

The *state* of an entity is the states of its attributes and the state of execution of its action part. The state of a pattern is a descriptor, and the state of a reference is the substance of the entity being denoted.

The *substance* of an entity is the entity itself. If two entities have the same substance, then they are the same entity.

An entity may be a subentity of a pattern. If PP is a subentity of P, then PP will have all the attributes and actions of entities generated according to P. These attributes and actions are collected within PP in an entity of kind *prefix*.

Entities of kind prefix, infix and insertion are called *constituent entities*. A constituent entity is a fixed part of another entity. The lifetime of a constituent entity is the same as that of the entity of which it is a part. Entities which are not constituent are called *autonomous*.

## Value and Assignment

An entity may have an associated value which is represented by parts of its state. The value of an entity may be observed and assigned to another entity. Observation/assignment of the value of an entity implies that clauses within its action part will be executed.

## Objects, Instances and Stacks

A BETA system may be regarded as a nested system of entities of kind *object*. To each object there may be associated a *stack* of entities of kinds *infix, insertion* or *instance*. The object is the bottom member of the stack.

An object in BETA may be *passive*, operate *concurrently* with other objects, (i.e. in so-called "physical parallel" or "true parallel"), or organized to operate in so-called "quasi-parallel" with a group of other objects (as in e.g. Simula).

The object stack will have the object itself as its bottom member, its infixes and insertions as possible members. The dynamic generation of an instance makes this entity the top member of the stack of the object to which the instance's descriptor belongs. The instance's infixes and insertions become new possible members of this object's stack. *No entity may be represented more than at most once at any time in its object stack.* (No entity may have multiple membership in an object stack at any time.)

The stacks organize actions in *action sequences*, each sequence thus associated with an object. At a given moment, the "object's action" will be the action currently being executed by the top member of its stack, all other stack members being passive. The action (execution of an imperative) of the top member of a stack may imply that another entity is "pushed on the stack" and the execution shifts to this entity.

If the top entity finishes the execution of its action part, it is "popped off the stack" and execution shifts to the entity which then is the top member - whose actions thus are being resumed.

An object which finishes its action part can never again execute actions and is said to be *terminated*. Objects are generated either as the result of executing an imperative or together with the generation of a reference data item.

## Concurrent Execution and Communication between Objects

Objects in a BETA system may execute their actions independently, which means that they may possibly execute actions in concurrency. Whether or not this is the case depends on the actual linking of the BETA-system to a BETA system generator. Initially a fixed number of objects are connected with objects in the BETA system generator corresponding to (physical) processors. The BETA system may then itself share the processors between its objects. How to do this is not treated in this report (see also section 4).

Objects may communicate by means of *signals*. As part of an action, an object may generate a signal requiring another object to execute a specific entity. Similarly, as part of an action, an object may be willing to accept a signal. If an object is generating a signal at the same time as another object is willing to accept it, then the signal is *communicated* which implies that the receiver of the signal executes the entity specified by the signal. An object may, while it is executing an entity, generate one or more signals to different objects and be willing to accept more than one signal from different objects. If a signal may be communicated then the execution of the entity is interrupted and the signal is communicated whereafter the execution of the interrupted entity is resumed.

# 3. LANGUAGE DEFINITION

The *BETA language* is a set of symbol sequences. A symbol sequence will be called a *BETA system description* and describes a BETA system. The BETA language definition defines:

(1) The set of symbols that may be used in a BETA system description.
(2) The set of symbol sequences that are valid BETA system descriptions, and
(3) The meaning of a BETA system description.

(1) defines the lexical level of the BETA language. The symbols are only defined implicitly as those appearing in the syntax rules. Lexical conventions such as possible external representations of the symbols are not treated in this report.

(2) defines the syntax of the BETA language. The context-free part of the syntax is defined using an extended BNF (see below). The context-sensitive part of the syntax (hereafter called *static semantics*) is defined in a semi-formal way using a mixture of attribute grammars and English (see below).

(3) defines the semantics of the BETA language. The semantics of a BETA system description defines the BETA system it describes and the way it behaves. Among other things the semantic definition focuses upon what is generated during the execution of a BETA system description.

A BETA system consists of a dynamic varying nested structure of entities each having an associated descriptor. A BETA system description defines a static nested structure of prototype descriptors. Each descriptor in a BETA system corresponds to a unique prototype descriptor. The differences between a descriptor and its corresponding prototype descriptor are minor. The prototype description contains a description of all the properties according to which the entities may operate or be operated. The additional properties contained in the descriptor have to do with the actual format of the entities, such as virtual bindings and the range of repetitions (cf. sections 3.4-3.6). The BETA system description is isomorphic to the nested structure of prototype descriptors and we shall not distinguish between these. In the rules which define the static semantics the word descriptor is used instead of the more correct prototype descriptor.

## The extended BNF-notation

Besides the usual BNF-notation we use

- *list*{A}*sep*{B} which stands for A or ABA or ABABA, etc. If B is the empty string then *sep*{B} will be deleted,
- *list*$_0${A}*sep*{B} which stands for the empty string or *list*{A}*sep*{B}, and
- *opt*{A} which stands for the empty string or A.

## The semiformal notation for defining static semantics

To describe the static semantics we use the following concepts more or less informally:

*Declaration and application of names*

A major part of the static semantics comes from the rules for *declaring* a name and in *applying* a name declared elsewhere (as mentioned earlier the names of patterns are called titles but will here be referred to as names).

Certain BETA *system elements* may have *names*. In the corresponding BETA system description these names are associated with the elements in the *declaration* of the elements, and they are *applied* in order to indicate the elements. System elements which may have names are: infix entities, references, infix repetitions, reference repetitions, patterns, virtual patterns and imperatives. When declaring a system element of this kind and associating a name with it, we will say that a *meaning* is associated with the name. The possible meanings will be denoted: infix, ref, infixrep, refrep, pattern, virtual and label, respectively. In addition the system element description (of the declaration) is associated with the name.

— *Environments*

An environment is a set of pairs (name, meaning). If two or more different pairs have the same name part then only one of the pairs will be **visible**. In fact an environment is a partial function from the set of visible names into meaning.

— *Inherited environments*

Each nonterminal of the grammar is defined relatively to an *inherited environment*. This environment will contain all declared names that are directly applicable in the system description part that may be generated from the nonterminal. The actual set of names in this environment is determined by the set of declarations in the part of the system description surrounding the nonterminal.

— *Synthesized environments*

Some nonterminals may have an associated synthesized environment whose set of names is determined by the system description part that is generated from the nonterminal (in the given inherited environment). Nonterminals having synthesized environments will typically be some that generate declarations, the synthesized environment will then be the set of declared names (and their meaning).

— *Naming of environments*

From now on inherited environment (synthesized environment) will be abbreviated IE (SE). Let <block> be a nonterminal, then <block ↓ G> is the nonterminal <block> and the IE of block is referred to by the name G. If <block> in addition has an SE, then <block ↓ G ↑ D> will be used as a name for <block> with its IE called G and its SE called D.

— *Productions and environments*

Each production is defined relatively to the IE of the nonterminal of the leftside and the possible SEs of nonterminals on the rightside. Rules will then be given for defining the IEs of the rightside and the possible SE on the leftside.

Consider the production

```
<block ↓ G ↑ D> ::=
        BEGIN
                <dcl ↓ DE ↑ DS>;
                <stmt ↓ SE>
        END
```

This corresponds to the BNF-production

```
<block> ::=
        BEGIN
                <dcl>;
                <stmt>
        END
```

G and DS are treated as the (variable) environments at which the static semantics is relatively defined, and the values of DE, SE and D will then be defined.

"A production with environments may informally be viewed as a procedure declaration. The leftside is the procedure heading and the rightside the procedure body. Nonterminals on the rightside will then correspond to (recursive) procedure calls. IEs may then be viewed as input parameters and SEs as output parameters."

— *Manipulation of environments*

If the same name is used for more environments in a production, then these two environments are identical.

Let E, D be two environments, then E ∪ D is a new environment consisting of all pairs from E and D. The same name may not be visible in both E and D. ∪ is called *disjoint union*.

E\D is a new environment consisting of all pairs from E and D. If a name is visible in both E and D, then only the one from D will be visible in E\D. \ is called *overriding* of E by D.

— *Partitions of inherited environments*

An inherited environment is partitioned into three distinct parts, the *global IE (GE)*, the *prefix IE (PE)*, and the *local IE (LE)*. Thus in general

$$\text{IE} = \text{GE}\backslash\text{PE}\backslash\text{LE}$$

If E is an IE, then E.GE is the global IE of E, etc.

— *Declared names and applied names*

An instance of a name in a production will be either a declaration or an application referring to a corresponding declaration. We use different nonterminals to generate a name depending on whether it is a declaration or an application and depending on the meaning of the declaration or application.

Nonterminals of the form <d-M-name> where M is a meaning are used in declarations, and similarly we use <a-M-name> in applications. If e.g. <d-infix-name> appears, it means that the name will be declared with meaning infix and the name will be added to the SE of the leftside of the production containing <d-infix-name>. Similarly <a-infix-name> means that the name refers to a declaration. The name must be in the IE of <a-infix-name> and with the meaning infix (see also 3.19).

## 3.1 Entities and patterns

*Syntax*

```
B     <BETA-system-descriptor> ::=
          <entity-descriptor ↓ G>

E     <entity-descriptor ↓ G> ::=
      opt {   <prefix ↓ G ↑ P>}
              BEGIN
                  opt {<attribute-part ↓ E ↑ L>}
                  opt {<value-part ↓ E ↑ V>}
                  opt {<action-part ↓ E>}
              END

P     <pattern-declaration ↓ G ↑ L> ::=
          PATTERN <d-title ↓ G> : <entity-descriptor ↓ E>

S     <entity-specification ↓ G> ::=
S1        <entity-descriptor ↓ G>
S2    |   <pattern-denotation ↓ G>

I     <pattern-denotation ↓ G> ::=
I1        <a-title ↓ G>
I2    |   <a-virtual ↓ G>
```

*Examples*

```
      BEGIN ... END
      PATTERN A : BEGIN ... END
```

*Static Semantics*

Rule B: G consists of all predeclared names. The actual set of names in G depends on the context concept (see section 4).

Rule E: $E = G\backslash P\backslash(L \cup V)$, i.e. $E.GE = G$, $E.PE = P$, and $E.LE = L \cup V$. $P\backslash L$ is called the *attribute environment* of the descriptor. Rule P is included here and is defined in section 3.6.

*Semantics*

Rule B: A BETA system is initiated by generating an entity (a *system entity*) as described by <entity-descriptor> and starting the execution of the actions of this entity.

Rule E defines an (entity-)descriptor. An entity generated according to a descriptor will have properties as described by the descriptor. These properties are an attribute corresponding to each declaration described by <attribute-part>, a value-part described by <value-part>, and an action-part described by <action-part>. These properties constitute the *main-part* of the entity. The possible <prefix> of the descriptor describes additional properties of the entity to those of the main-part. A prefixed entity will have a *prefix-part* containing properties which it has in common with a set of other entities. The *origin* of a descriptor and an entity generated according to the descriptor is the entity containing the descriptor.

Rule P declares a pattern-attribute. A *pattern* is the association of a *title* (a name described by <title>) and a descriptor (described by <entity-descriptor>). A pattern may be used to generate entities which will have the properties described by the descriptor associated with the title of the pattern.

Rule S defines an *entity-specification* which is used in constructional modes to specify the properties of an entity to be generated. Rule S1 implies the generation of a *singular* entity as described by <entity-descriptor>. Rule S2 implies the generation of a *category* defined entity as specified by <pattern-denotation>.

Rule I: A pattern is denoted by its title.

## 3.2 Prefixing

*Syntax*

```
P      <prefix ↓ G ↑ P> ::=
P1         <pattern-denotation ↓ G>
P2     |   (<d-prefix-name ↓ G> : <pattern-denotation ↓ G>)
```

*Examples*

**PATTERN** B : A **BEGIN** ... **END**
(X : A) **BEGIN** ... **END**

*Static Semantics*

P = A ∪ V where A is the attribute environment and V the value-list (see 3.7) of the descriptor of the pattern referred to by <pattern-denotation>. Using P2, P will be overridden by the declared name with meaning prefix.

*Semantics*
The properties of an entity consist of
a) properties belonging to the main part,
b) properties described by the <prefix> of the descriptor associated with the entity.

A prefixed entity will have a *prefix-part* which is a constituent entity (of kind *prefix*) with properties described by <pattern-denotation>. The attributes, the value, and the actions of the main-part will be *prefixed* with the corresponding parts of the prefix:
— *Prefixing the attributes.* The attributes will be the union of the prefix and main part attributes.
— *Prefixing the value-part.* This is described in section 3.7.
— *Prefixing the actions.* This is described in section 3.8.

An entity prefixed by a pattern P is a *sub-entity of P.* If the <entity-descriptor> is part of a pattern-declaration, then this pattern is a *sub-pattern of P.*

## 3.3 Attributes

*Syntax*
```
<attribute-part ↓ E ↑ L> ::=
  list {<data-item-declaration ↓ E ↑ D>
       | <pattern-declaration ↓ E ↑ P>
       } sep {;}

<data-item-declaration ↓ E ↑ D> ::=
       <infix-declaration ↓ E ↑ D>
  |    <reference-declaration ↓ E ↑ D>
```

*Static Semantics*
Each declaration declares a set of names. The same name may only be declared once in an attribute-part. Thus L is the disjoint union of all D's and P's in the list.

*Semantics*
The attributes of the main part are specified by a sequence of declarations.

## 3.4 Infix-declarations

*Syntax*

```
I       <infix-declaration ↓ E ↑ L> ::=
                <simple-infix ↓ E ↑ L>
        |       <infix-repetition ↓ E ↑ L>

S       <simple-infix ↓ E ↑L> ::=
            list{<d-infix-name ↓ E>}sep{,} : <entity-specification ↓ E>

R       <infix-repetition ↓ E ↑ L> ::=
            list{<d-infixrep-name ↓ E>}sep{,} : [<integer-expression ↓ E.GE>]
                <entity-specification ↓ ES>
```

*Example*

X,Y : REAL; N : [10] INTEGER; Z : B **BEGIN ... END**;

*Static Semantics*

L is the disjoint union of all name pairs being declared in the lists (rules S and R). Rule R: ES is E overridden by two integer attributes RANGE and INDEX.

*Semantics*

An infix-declaration specifies a number of infix-attributes.

Rule S: An *infix* is the association of a name and an entity of kind *infix*. Each name in the list implies an infix with that name. The associated entities have properties as described by <entity-specification>.

Rule R: An *infix-repetition* is the association of a name, a *range*, and a sequence of entities of kind *infix*, each having an associated *index*. Range is an integer value which is the number of entities in the sequence. <integer-expression> is evaluated when the origin of the descriptor containing the declaration is generated and range will have this value. The index of an entity in the sequence is an integer value in the interval [1,range]. No two different entities in the sequence have the same index.

In each entity in the sequence, the values of RANGE and INDEX are the value of range and the entity index respectively. If x is an infix-repetition then the entity with index I may be denoted by x[e] if e is an integer expression which evaluates to I.

## 3.5 References

*Syntax*

```
<reference-declaration ↓ E ↑ L> ::=
        <simple-reference ↓ E ↑ L>
|       <reference-repetition ↓ E ↑ L>

<simple-reference ↓ E ↑ L> ::=
        REF list {<d-reference-name ↓ E>}sep{,} :
           <entity-specification ↓ E>

<reference-repetition ↓ E ↑ L> ::=
        REF list {<d-refrep-name ↓ E>}sep{,} :
                 [<integer-expression ↓ E.GE>] <entity-specification ↓ E>
```

*Examples*

```
REF R,S : A; REF T : B BEGIN ... END;
REF U : [M] B;
```

*Semantics*

A reference declaration declares a number of *references*. A reference is the association of a name and a system element capable of denoting an entity of kind *object*. A reference is declared for each name in the list. The state of a reference is said to be the *substance* of the entity being denoted.

The set of objects that may be denoted are determined by <entity-specification> that may be

— a pattern, say PT. Each reference may denote any PT-entity, any sub-entity of PT and no entity (NONE). The reference is *variable* and its state may be changed by reference assignment. Initially it has the state NONE.

— an entity descriptor. Each reference will constantly denote an entity generated according to the descriptor. Each such (singular) entity is generated when the entity containing the declaration is generated.

## 3.6 Patterns

*Syntax*

```
P       <pattern-declaration ↓ G ↑ L> ::=
P1          PATTERN <d-title ↓ G> : <entity-descriptor ↓ E>
P2          PATTERN <d-virtual ↓ G> : VIRTUAL <a-title ↓ G>
P3          BIND <a-virtual ↓ G.PE> : <entity-specification ↓ E>
P4          BIND <a-virtual ↓ G.PE> : VIRTUAL <a-title ↓ G>
```

*Examples*

```
PATTERN Q: BEGIN PATTERN C: VIRTUAL A; ... END;
PATTERN Q1:  Q BEGIN BIND C: A BEGIN ... END; ... END;
X: Q BEGIN BIND C: B; ... END;
PATTERN Q2:  Q BEGIN BIND C: VIRTUAL B; ... END;
REF R: Q2 BEGIN BIND C: B BEGIN ... END; ... END;
```

*Static Semantics*

Note: In rules P1 and P2, the names are declared. In P3 and P4 the names refer to names declared in the prefix environment of G. Rules P3 and P4 are not allowed if the descriptor containing the declaration is prefixed by a virtual pattern.

In rule P1/P2 L consists of the title/virtual being declared. In P3 and P4 the applied virtual names are redeclared. In P3 the meaning of the name is changed to title. In P4 the meaning of the name is still virtual. In both cases L contains the name pair.

In P1/P3 E is identical to G except that some names with meaning title or virtual have a marking indicating a restricted usage of the corresponding patterns. These patterns must not be used to specify constituent entities in the descriptor associated with the pattern defined in P1/P3. A pattern, P, may be recursively defined by using P (either directly or indirectly) to define its associated descriptor. However P must not be used to specify constituent entities in this descriptor as this will lead to infinite entities. Consequently P and patterns using P to define constituent entities in their associated descriptors have this marking in E.

*Semantics*
A *pattern* is the association of a *title* (a name) and a *descriptor.* A pattern specifies the common properties of a category of entities. The title of a pattern may be used in constructional modes and will then imply the generation of an entity according to the descriptor associated with the title.

A descriptor and entities generated according to the descriptor are *qualified by* a pattern P, if
- the descriptor is the one associated with P, or
- the prefix of the descriptor is qualified by P.

A pattern S is qualified by P if the descriptor associated with S is qualified by P.

There are four kinds of pattern-declarations:

P1 declares a pattern with a title described by <title> and a descriptor described by <entity-descriptor>.

P2 declares a *virtual pattern.* The title of the pattern is described by <virtual>. The descriptor of the pattern is only partially described. It will be one that is qualified by <title>. <title> is called the *qualifying-title* of the virtual pattern. Consequently the declared virtual pattern describes entities which have all the properties of the pattern referred to by <title>.

BIND (P3,P4) extends the descriptor of a virtual pattern declared in the prefix-entity of the entity containing the declaration. <virtual> specifies the title of the pattern. <entity-specification> (P3) and <title> (P4) must be qualified by the qualifying title of the virtual pattern referred to by <virtual>.

P3: The descriptor of the virtual pattern will be the one described by <entity-specification>. *The pattern is no longer virtual.*

P4: The descriptor of the virtual pattern will be one that is qualified by <title> *The pattern is still virtual*, and <title> is now its qualifying-title.

When an entity is generated then all of its virtual pattern attributes will be associated with the descriptor of their qualifying title.

## 3.7 Value-part

*Syntax*

<value-part ↓ E ↑ L> ::= **VALUE** $list_0${<infix-declaration ↓ E ↑ D>;}

*Example*

**VALUE** I:INTEGER; X,Y:REAL;

*Static Semantics*
L is the disjoint union (∪) of all Ds in the list. The *value-list* of the descriptor is the list of declarations following **VALUE** prefixed by a possible value-list of the prefix. Prefixing means concatenating the value-list of the prefix with the declared list. If X is an infix in the value-list then X must be assignment-compatible with X (see 3.14) and X must be value-compatible with X (see 3.11).

*Semantics*
The values of the data-items in the value-list represent the parts of the value of the entity that is transferred between two entities as a part of an assignment.

The *value-list* of an entity is the expression-list (see 3.14) <D1, D2, ... Dn> where each D*i* (i ∈ [1,n]) is the denotation of the corresponding infix declared in the value-list of the descriptor.

## 3.8 Action-part

*Syntax*

<action-part ↓ E> ::=
*opt*{<entry-part ↓ E>}
*opt*{**DO** <imperative-list ↓ E>}
*opt*{<exit-part ↓ E>}

n <entry-part ↓ E> ::=
**ENTRY** *opt*{< <expression-list ↓ E> >} <imperative-list ↓ $E_1$>

x <exit-part ↓ E> ::=
**EXIT** <imperative-list ↓ $E_1$> *opt*{< <expression-list ↓ E> >}

*Examples*

```
BEGIN ... DO ... END;
BEGIN I,J : INTEGER; X : REAL;
ENTRY <I,J> ...
EXIT ... <X>
END
```

*Static Semantics*
$E_1$ in rule n is E extended with the name PRED. PRED denotes the predecessor (see 3.14) of the entity containing the ENTRY-part.

$E_1$ in rule x is E extended with the name SUC which similarly denotes the successor (see 3.14). PRED and SUC may be used as remote-names (see 3.18), (i.e. as PRED' and SUC') and in qualification-expressions (see 3.11).

*Semantics*
The EXIT(ENTRY)-list of the entity is the <expression-list> following EXIT(ENTRY) prefixed by a possible EXIT(ENTRY)-list of the prefix. The EXIT(ENTRY or DO)-action of the entity is the <imperative-list> following EXIT(ENTRY or DO) prefixed by a possible EXIT(ENTRY or DO)-action of the prefix respectively. Prefixing EXIT(ENTRY)-lists means concatenation as with value-lists (3.7). Prefixing <imperative-list>s means that the execution of an INNER imperative that is textually contained in the <imperative-list> of the prefix implies an execution of the corresponding <imperative-list> in the main-part.

The action-part of an entity describes the actions associated with the entity. An execution of the entity implies an execution of its DO-action. If the entity is assigned a new value then its ENTRY-action is executed prior to its DO-action. Similarly if the entity is assigned to another entity then its EXIT-action is executed after the DO-action (see entity assignment section 3.14).

## 3.9 Imperatives

*Syntax*

```
<imperative-list ↓ E> ::=
    list {<imperative ↓ E>} sep {;}

<imperative ↓ E> ::=
    <empty>
|   <repetition-imperative ↓ E>
|   <selection-imperative ↓ E>
|   <break-imperative ↓ E>
|   <inner-imperative ↓ E>
|   <entity-imperative ↓ E>
|   <reference-imperative ↓ E>
|   <communication-imperative ↓ E>
|   <d-label-name ↓ E> : <imperative ↓ E1>
```

*Static Semantics*

$E_1$ is E overridden by the declared label. This means that a label is only visible inside the <imperative> being labelled.

*Semantics*

An <imperative> specifies an action. An <imperative-list> specifies a sequence of actions to be executed in the order of appearance.

## 3.10 Repetition

*Syntax*

```
R       <repetition-imperative ↓ E> ::=
               [<integer-expression ↓ E>] <imperative ↓ E1>
```

*Example*

```
        [100] ...;
```

*Static Semantics*

$E_1$ is E overridden by two integer attributes RANGE and INDEX.

*Semantics*

The <imperative> will be executed a number of times (possibly zero). The execution starts by evaluating the expression and the value of the expression determines the number of times to execute the <imperative>. The repetition may be stopped if the <imperative> contains a break-imperative (see 3.12).

The value of the attribute RANGE is the number of times the <imperative> will be executed. The value of INDEX is the number of the current repetition.

## 3.11 Selection

*Syntax*

```
S       <selection-imperative ↓ E> ::=
S1          IF <entity-expression ↓ E>
                    list {<entity-selection ↓ E>}
            ENDIF
S2      |   IF <reference-expression ↓ E>
                    list {<reference-selection ↓ E>}
            ENDIF
S3      |   IF <qualification-expression ↓ E>
                    list {<qualification-selection ↓ E>}
            ENDIF

I       <entity-selection ↓ E> ::=
            = list {<entity-expression ↓ E>} sep {OR}
              THEN <imperative-list ↓ E>

R       <reference-selection₁ ↓ E> ::=
            = = list {<reference-expression ↓ E>} sep {OR}
              THEN <imperative-list ↓ E>

Q       <qualification-selection ↓ E> ::=
            {IS <pattern-denotation ↓ E> | IN <pattern-denotation ↓ E>}
              THEN <imperative-list ↓ E>

X       <qualification-expression> ::=
X1          <pattern-denotation>
X2      |   <reference-denotation>
X3      |   <infix-denotation>
X4      |   <prefix-denotation>
```

*Examples*

```
        IF X
        = E1 THEN ...
        = E2 OR E3 THEN ...
        ENDIF;
        IF R
        IS P1 THEN ...
        IN P2 THEN ...
        ENDIF
```

*Static Semantics*
Below we use concepts defined in 3.14.

Let EE1 and EE2 be two <entity-expression>s. EE1 and EE2 are *value-compatible* if (1) EE1 and EE2 have both an EXIT-part, (2) the EXIT-lists of EE1 and EE2 have the same length and corresponding elements in these lists are value-compatible, and (3) if EE1 and EE2 have value-lists then they have the same length and the descriptors of their associated action-entities have a common prefix containing the value-lists. All the <entity-expression>s in rules S1 and I must be value-compatible.

*Semantics*
A <selection-imperative> selects between a set of <imperative-list>s for execution. The selection is based upon the state of an expression. The state of the expression following **IF** is evaluated. Next the states of the expressions in the selection-clauses are evaluated in some nondeterministic order. (Not all of the expressions are necessarily evaluated.) If the state of one of the expressions in a selection-clause is equal to the state of the expression following **IF**, then the <imperative-list> following **THEN** may be selected for execution. Exactly one of the <imperative-list>s that may be selected will be selected. At least one must exist.

The selection may be based upon three kinds of state concepts:
— *entity values* (rule S1, I).
  An <entity-expression> EE, is *evaluated* by executing EE, and then executing its EXIT-action, and then evaluating the expressions in its EXIT-list. The *value* of EE is the values of the elements in its value-list and EXIT-list. If EE1 and EE2 are value-compatible, then the values of EE1 and EE2 are *equal* if values of corresponding elements in their value-lists and EXIT-lists are equal.
— *reference states* (rule S2, R).
  The state of a reference is the substance of an (autonomous) entity. Two reference states are equal if they are the substance of the same entity.
— *qualifications* (rule S3, Q, X).
  The state of a qualification-expression is a descriptor which is: The descriptor associated with the pattern (X1), or the descriptor associated with the entity being denoted (X2, X3, X4). A (qualification) state is equal to **IS** P if the state is the descriptor of the pattern P. A state is equal to **IN** P if the descriptor is qualified by P. Two descriptors are equal if they have the same origin and are defined by the same <entity-descriptor> (3.1).

### 3.12 Break-imperatives

*Syntax*

```
<break-imperative ↓ E> ::=
    RESTART <a-label-name ↓ E>
|   LEAVE <a-label-name ↓ E>
```

*Examples*

```
L1: BEGIN ... L2:BEGIN ... LEAVE L2;
                 ... RESTART L1; ...
                 END; ...
    END
```

*Semantics*
Both imperatives have the effect that the execution of the imperative labelled by the label will be stopped. **RESTART** means that this labelled imperative will be executed again. **LEAVE** means that the imperative following the labelled imperative will be executed.

### 3.13 Inner-imperative

*Syntax*

```
<inner-imperative> ::=
       INNER
```

*Semantics*
See the description of action-part.

## 3.14 Entity-imperative, -assignment, and -expression

*Syntax*

```
i       <entity-imperative ↓ E> ::=
            <action-entity ↓ E>
        |   <entity-assignment ↓ E>

a       <entity-assignment ↓ E> ::=
            <entity-expression ↓ E> → <compound-entity ↓ E>

e       <entity-expression ↓ E> ::=
            <compound-entity ↓ E>
        |   <entity-assignment ↓ E>

c       <compound-entity ↓ E> ::=
c1          <action-entity ↓ E> opt{< <expression-list ↓ E> >}
c2      |   < <expression-list ↓ E> >

l       <expression-list ↓ E> ::=
           list{<entity-expression ↓ E> | <reference-expression ↓ E>} sep{,}
```

*Examples*

```
        X → Y → Z; <A,B> → C;
        D → <X,Y,Z>; E<A,B> → F<X,Y,Z>;
```

*Static Semantics*

The action-entity (-descriptor), EXIT/ENTRY-actions, EXIT/ENTRY-lists, value-list and expression-list of an <entity-assignment> are the ones of the <compound-entity> on the rightside of rule a. The same notions are defined for <entity-expression> and are the ones defined by <compound-entity> or <entity-assignment> on the rightsides of rule e. The action-entity (-descriptor) of a <compound-entity> is the one described by <action-entity> (rule c1) or empty (rule c2). The EXIT/ENTRY-actions, EXIT/ENTRY-lists and the value-list of a <compound-entity> are the ones of its action-entity. The expression-list is defined by <expressin-list>.

Rule a: We abbreviate <entity-expression> to EE and <compound-entity> to CE. The action-entities of EE/CE are denoted EE.A/CE.A respectively.

EE.A is called the *predecessor* of CE.A and CE.A is called the *successor* of EE.A. EE.A is *assignment compatible* with CE.A if EE.A has an EXIT-part and CE.A has an ENTRY-part and if CE.A has a value-part then the descriptors of

EE.A and CE.A must have a common prefix, P, such that the value-part of CE.A is contained in P.

Let L1 and L2 be two lists, either expression-lists, EXIT/ENTRY-list, or value-lists. L1 is *assignment-compatible* with L2 if the length of L1 is *greater than or equal* to the length of L2 and corresponding elements in the lists are assignment-compatible.

EE is *assignment-compatible* with CE if

- EE.A and CE.A are assignment-compatible,
- the EXIT-list of EE is assignment-compatible with the expression-list of CE, and
- the expression-list of EE is assignment-compatible with the ENTRY-list of CE.

*Semantics*
The execution of an <entity-imperative> (i) consists of executing the <action-entity> or <entity-assignment> on the rightside.

The execution of an <entity-expression> (e) consists of executing the <compound-entity> or <entity-assignment> on the rightside.

The execution of a <compound-entity> (c) consists of executing the <action-entity> (c1) or is an empty action (c2).

Rule a:
Let <e1,e2,...,em> and <f1,f2,...,fn> be assignment-compatible lists ($m \geqslant n$).

*Assigning* <e1,e2,...,em> *to* <f1,f2.,,,fn> takes place by executing the following sequence of assignment-imperatives: e1 *to* f1; e2 *to* f2; ...; en *to* fn; where ei *to* fi ($i \in [1,n]$) means ei → fi or ei = => fi if ei,fi are entity-expressions or reference-expressions respectively.

A *value-transfer from* EE *to* CE takes place by:

- executing the EXIT-action of EE and then assigning the EXIT-list of EE to the expression-list of CE,
- assigning the value-list of EE to the one of CE, and
- assigning the expression-list of EE to the ENTRY-list of CE and then executing the ENTRY-action of CE.

*An execution of an assignment-imperative* consists of executing EE, executing a value-transfer from EE to CE, and then executing the action-entity of CE.

## 3.15 Action-entity

*Syntax*

```
        <action-entity ↓ E> ::=
            <infix-denotation ↓ E>
        |   <insertion ↓ E>
        |   <instance ↓ E>

F       <insertion ↓ E> ::=
            opt{<remote-name ↓ E ↑ E1>} <entity-specification ↓ E2>

C       <instance ↓ E> ::=
            INSTANCE opt{<remote-name ↓ E ↑ E1>}
            <entity-specification ↓ E2>
```

*Example*

X; P; **PBEGIN ... END; INSTANCE** R'Q;

*Static Semantics*

For the definition of $E_2$ see 3.16, rule o.

*Semantics*

The execution of an <action-entity> consists of executing the DO-action of some entity. There are three kinds of action-entities:

— *infix* (rule E).

The action entity is the one denoted by <infix-denotation>.

— *insertion* (rule F).

An entity specified by <entity-specification> will be a constituent entity (of kind *insertion*) of the entity containing the imperative. This constituent entity is the action-entity.

— *instance* (rule C).

Each execution of an <instance> will result in the generation of a new entity (of kind *instance*) as specified by <entity-specification>. This new entity is the action-entity.

The execution of an action-entity may imply the execution of an action-entity belonging to an "enclosing" object. Let M be such an action-entity belonging to enclosing object R. The execution of M will then take place by executing R.M (see 3.17).

## 3.16 Reference-imperative, -assignment and -expression

*Syntax*

i &lt;reference-imperative ↓ E&gt; ::=  
  &lt;reference-assignment ↓ E&gt;

a &lt;reference-assignment ↓ E&gt; ::=  
  &lt;reference-expression ↓ E&gt; ==&gt; &lt;reference-denotation ↓ E&gt;

e &lt;reference-expression ↓ E&gt; ::=  
e1  &lt;reference ↓ E&gt;  
e2 | &lt;reference ↓ E&gt; **AS** &lt;pattern-indication ↓ E&gt;

r &lt;reference ↓ E&gt; ::=  
r1  &lt;reference-denotation ↓ E&gt;  
r2 | &lt;reference-value ↓ E&gt;  
r3 | &lt;reference-assignment ↓ E&gt;

v &lt;reference-value ↓ E&gt; ::=  
v1  **NONE**  
v2 | **THIS OBJECT**  
v3 | &lt;object-generation ↓ E&gt;

o &lt;object-generation ↓ E&gt; ::=  
**OBJECT** *opt*{&lt;remote-name ↓ E ↑ $E_1$&gt;}  
  &lt;entity-specification ↓ $E_2$&gt;

*Example*

**NONE** ==&gt; R ==&gt; S; **OBJECT** P ==&gt; R;

*Static Semantics*

The elements on the leftside of rules i, a, e, r and o have an associated descriptor defined as follows: Rule o: described by &lt;entity-specification&gt;; rule v1: 'NONE'; rule v2: 'THIS OBJECT'; rule e2: described by &lt;pattern-indication&gt;; rule a: the one associated with &lt;reference-denotation&gt;; for all other rules the one associated with the element on the rightside.

Rule a: The descriptor associated with &lt;reference-denotation&gt; must be part of a (possible virtual) pattern, say P. The descriptor associated with &lt;reference-expression&gt; must be qualified by P or the descriptor 'NONE'; the &lt;reference-expression&gt; is then *assignment-compatible* with the &lt;reference-denotation&gt;.

Rule o: If no <remote-name> is present then $E_2 = E$. If a <remote-name> is present then the use of $E_2$ is informal:

- if <entity-specification> is <pattern-denotation> then $E_2 = E_1$.
- if <entity-specification> is <entity-descriptor> then $E_2 = E$, but a possible prefix-pattern of the entity-descriptor must be defined in $E_1$.

*Semantics*
Rule i: The execution of a <reference-imperative> consists of executing the <reference-assignment>. Rule a: The <reference-expression> is evaluated yielding a reference-state which is transferred to the reference denoted by <reference-denotation>, and this reference state is the state of the <reference-assignment>.

Rule e: The state of a <reference-expression> is the substance of an entity and is the one of <reference>. Rule e2: The state of <reference> must be the substance of an entity which is qualified by <pattern-indication>. Rule r: The state of a <reference> is the one of the element on the rightside. The state of a <reference-denotation> is the substance of the entity denoted by the reference.

Rule v: Case v1: The state is **NONE** which is the substance of no-entity. Case v2: the imperative referring to **THIS OBJECT** is part of the action part of some autonomous entity (possibly indirectly through some constituent entities). This autonomous entity will be the member of a stack at the time that **THIS OBJECT** is referred. The substance of the object-entity (bottom) of this stack will be the state of **THIS OBJECT.** Case v3: the execution of an <object-generation> will yield a state.

Rule o: The execution of an <object-generation> implies the generation of a new entity of kind object. The substance of this new entity is the state yielded by the <object-generation>.

The predefined name SENDER is a reference-denotation denoting a reference to the object from which the latest signal has been accepted.

## 3.17 Communication-imperative

*Syntax*

```
c       <communication-imperative ↓ E> ::=
c1          WHILE <entity-denotation ↓ E>
            list{ON <communication ↓ E> THEN <imperative-list ↓ E> }
            ENDWHILE
c2      |   <communication ↓ E>

        <communication ↓ E> ::=
             <input-communication ↓ E>
        |   <output-communication ↓ E>

i       <input-communication ↓ E> ::=
            <sender ↓ E> ? <input-message ↓ E>
            opt{→ <entity-expression ↓ E>}

o       <output-communication ↓ E> ::=
o1          opt{<entity-expression ↓ E> →} <receiver ↓ E ↑ A> ! <message ↓ A>
o2      |   opt{<entity-expression ↓ E>  →} <receiver ↓ E ↑ A> .  <message ↓ A>
            opt{→ <entity-expression ↓ E>}

s       <sender ↓ E> ::=
s1          <object-denotation ↓ E>
s2      |   SENDER opt{AS <pattern-indication ↓ E>}

e       <input-message ↓ E> ::=
e1          <message ↓ E>
e2      |   MESSAGE opt{AS <pattern-indication ↓ E>}

m       <message ↓ E> ::=
m1          <empty>
m2          <infix-denotation ↓ E>
m3      |   INSTANCE <pattern-indication ↓ E>

r       <receiver ↓ E ↑ A> ::=
            <object-denotation ↓ E> ::=

b       <object-denotation ↓ E> ::=
            <reference-denotation ↓ E>
            list₀{ . <simple-reference-denotation ↓ E> }
```

*Examples*

```
Z → R!M; S?N → X;

WHILE A
   ON Z → R!M THEN ...
   ON S?N → X THEN ...
   ON T.INSTANCE P THEN ...
   ON SENDER AS Q?N THEN ...
ENDWHILE
```

*Static Semantics*
Rule b: Consider an <object-denotation> RD.R1.R2, then R1 must be a reference attribute of the object denoted by RD and R2 must be a reference attribute of the object denoted by R1. This is quite similar to <remote-name> (see 3.18). Rule r: A is the attribute environment of the object denoted.

*Semantics*
Rule c1: The entity described by <entity-denotation> (ED) is executed. During this execution an attempt is made to establish a *communication* with one or more objects. If such a communication is established then the execution of ED will be interrupted and a communication will take place (see below). The execution of the whole <communication-imperative> is finished when the execution of ED is finished.

Each ON-clause describes an attempt to establish a communication. Communication between objects take place by means of *signals*. A signal is a triple (sender, receiver, message) where sender and receiver are references and message is an entity "belonging to" the receiver object.

There are two kinds of ON-clauses:
— input-communication (i): the object is willing to accept a signal described by (<sender>, **THIS OBJECT**, <message>).
— output-communication (o): the object generates a signal described by (**THIS OBJECT**, <receiver>, <message>).

<sender> (s) will match either a particular object (s1) or any reference to an object qualified by <pattern-indication> (s2).

<input-message> (e) will match either a particular message (e1) or any message (entity) which is qualified by <pattern-indication> (e2). <message> (m) will match either the empty message (m1) or a particular infix (m2) or an instance (m3).

<receiver> (r) is always a particular object. An <object-denotation> (b) RD.R1.R2...RN is a reference to the object denoted by RN.

A signal (S,R,M) is *communicated* if
- object S is executing a communication imperative (C*S*) that generates (S,R,M),
- object R is executing a communication imperative (C*R*) that will accept (S,R,M).

Let O*S* and O*R* be the ON-clauses in C*S* and C*R* that generated/accepted the signal, and let O*S* be defined by rule o1.

The *communication* takes place as follows:

(1) The EDs of C*S* and C*R* are interrupted, (see below).

(2) The possible <entity-expression> (EE) of O*S* is executed and a value transfer between EE and M is carried out.

(3) S and R will then independently (and possibly concurrently) continue with:

S: The <imperative-list> of O*S* is executed and then the execution of EDs is resumed (see below).

R: (a) M → <entity-expression> (of O*R*) is executed, then
(b) <imperative-list> of O*R* is executed, and then the execution of ED*R* is resumed (see below).

When an ED is interrupted it will terminate at the next semicolon in the imperative-list of its DO-action and then execute its EXIT-action. When it later is resumed it will start by executing its ENTRY-action.

Let O*S* be defined by rule o2 and have the form **ON** EE → R.M → $EE_1$ **THEN** ....
The semantics is as above except that (3) is changed to:

(3') R will execute M; then a (possible) value-transfer between M and <entity-expression> (of O*R*) is executed; then a (possible) value-transfer between M and $EE_1$ is executed. S and R will then independently continue with:

S: $EE_1$ is executed; then S continues as in (3).

R: <entity-expression> of O*R* is executed; then R continues as in (3b).

Rule c2: An execution of a single <communication> is considered as a special instance of rule c1. If com is a <communication> then the execution of com means

L: **WHILE** WAIT **ON** com **THEN LEAVE** L **ENDWHILE**

where WAIT is an entity that loops infinitely.

## 3.18 Infix- and reference-denotation

*Syntax*

```
q       <infix-denotation ↓ E> ::=
            opt{<remote-name ↓ E ↑ E₁>}
              <simple-infix-denotation ↓ E₁>

i       <simple-infix-denotation ↓ E> ::=
            <a-infix-name ↓ E>
        |   <a-infixrep-name ↓ E> [ <integer-expression ↓ E> ]

r       <reference-denotation ↓ E> ::=
            opt{<remote-name ↓ E ↑ E₁>}
              <simple-reference-denotation ↓ E₁>

f       <simple-reference-denotation ↓ E> ::=
            <a-ref-name ↓ E>
        |   <a-refrep-name ↓ E> [ <integer-expression ↓ E> ]

p       <prefix-denotation ↓ E> ::=
            <a-prefix-name ↓ E>

d       <remote-name ↓ E ↑ E₂> ::=
            opt{<remote-name ↓ E ↑ E₁>}
                {<simple-infix-denotation ↓ E₁> '
                | <prefix-denotation ↓ E₁> '}
```

*Examples*

X; Z[I]; H'A; U[3]; L'M'Y

*Static Semantics*

Rule q, r, p: Each denotation refers to a name visible in E or $E_1$. Besides its meaning, this name has an associated descriptor appearing in the declaration of the name. This descriptor (which may be empty (see below)) is associated with <infix-denotation>, <reference-denotation>, or <prefix-denotation> respectively.

Rule q, r, d: If no <remote-name> is present then $E_1 = E$. If a <remote-name> is present, then $E_1$ is a *modified* version (see below) of the attribute environment of the descriptor associated with the name denoted by <remote-name> on the rightside.

Rule d: $E_2$ is a *modified* version of the attribute-environment of the descriptor accessed through the denotation.

Rule i, f, p: The descriptor accessible through the denotation is the one associated with the respective applied name.

*Modified attribute environment*
A descriptor is *remote-visible* if it has a prefix, P, and

- P is not defined in the attribute-environment of the origin of the descriptor, or
- P is remote-visible.

A descriptor is *remote-hidden* if it is not remote-visible.

An attribute is *remote-accessible* if its associated descriptor is remote-visible.

An attribute is *partially-remote-accessible* if its descriptor is remote-hidden.

Let D be a descriptor with attribute-environment A. The *modified* attribute-environment of D is identical to A except that for all partially-remote-accessible attributes in A, the descriptor will be empty.

*Semantics*
Rule q, r: A denotation refers to a data-item in some entity. This data-item has an associated descriptor, defined by an <entity-specification>. This descriptor specifies the properties that are known about the data-item, i.e. an additional piece of information about the meaning of the data-item. If the data-item is accessed through a <remote-name> then its descriptor may be unknown (see below).

Rule d: A <remote-name> will be the name of a constituent-entity, either infix or prefix. The purpose of the <remote-name> is to access an attribute in that entity.

Rule i, f: The applied name refers to a data-item.

## 3.19 Application of Names

*Syntax*
For each M in {infix, ref, prefix, infixrep, refrep, pattern, virtual, label} there is a syntax rule: <a-M-name ↓ E> ::= <name>.

The actual representation of names is not considered here. Let A be the name generated by <name>.

*Static Semantics*
A must be visible in E. The meaning of A must be M. Let D be the descriptor for the entity containing the construct <a-M-name> that generates A.

As mentioned, E is partitioned into three parts, the local environment (LE), the prefix environment (PE) and the global environment (GE). A is said to be a

- *local attribute*, if A is visible in E.LE,
- *prefix attribute*, if A is visible in E.PE and not visible in E.LE,
- *global attribute*, if A is visible in E.GE and A is not visible in E.PE\E.LE.

*Semantics*
All entities generated according to the descriptor D can refer to an attribute with the name A and meaning M.

Let T be an entity generated according to D. Let TP be the prefix part of T. Let the entity containing the descriptor for T (i.e. D) be T*n*, and for i = 1,2,...,n-1 let T*i-1* be the entity containing the descriptor for T*i*. TP may be empty and T may be the BETA system entity in which case T*n* does not exist.

A will be either
- an attribute of T, or
- an attribute of some T*i*, i in [1,n].

If A is a
- local attribute, then A refers to the main-part attribute in T with the name A and meaning M,
- prefix attribute, then A refers to the attribute in the TP-part of T with the name A and meaning M,
- global attribute, then there exists a T*i*, $1 \leqslant i \leqslant n$, such that A is an attribute of T*i* and no attribute in T*i+1*,...,T*n* has the name A. A refers to the attribute in T*i* with the name A and meaning M.

## 3.20 Predefined Patterns

Some entities within a BETA system will not have a description in the BETA language. Among these are entities generated according to *predefined patterns*. There are predefined patterns for the specification of bit- and integer-entities.

*Bit-entities* are defined by the bit-patterns BITVALUE, ZERO, ONE, and BIT. Bit-entities have the following properties:

— Bit-entities are assignable to BIT-entities,
— ZERO, ONE, and BIT are subpatterns of BITVALUE,
— bit-entities are value-compatible with bit-entities,
— the values of two ZERO-entities are always equal,
— the values of two ONE-entities are always equal,
— the value of a ZERO-entity is not equal to the value of a ONE-entity,
— the state of a bit-entity can only be changed by assigning a new value to it. Thus the states of BITVALUE-, ZERO- and ONE-entities are constant,
— if a BIT-entity, A, is assigned the value of a bit-entity, B, then the value of A is equal to the value of B until either A or B is assigned a new value.

Following the same lines as above, integer-entities are defined by the integer-patterns INTEGERVALUE, INTEGER and one for each integer-value (0, 1, -1, 2, -2, etc.).

An <integer-expression> is an <entity-expression> consisting of integer-entities.

# 4. THE NOTIONS OF BETA SYSTEM GENERATOR, DESCRIPTOR AND CONTEXT

The introduction gives an outline of the concepts BETA system, BETA system generator, descriptor and context, and of the relations between them. This section treats these concepts in more detail and gives an indication of how they may be included in the BETA language. Arguments for inclusion of the concepts and desirable properties of the concepts are also given.

A *BETA system* is a system which is being or has been generated by the execution of a BETA system description. A BETA system consists of a collection of entities. During the lifetime of a BETA system the states of the entities may be changed, new entities may be generated and entities may be removed from the system. Other terms for BETA system and BETA system descriptions are "BETA program execution" and "BETA program", respectively.

A collection of computer components organized (and programmed) so that it may execute BETA system descriptions and thus generate BETA systems is called a *BETA system generator*. A system generator may be regarded as a system and we want to be able to describe BETA systems which act as system generators. The *initial BETA system generator* on a set of computer components must of course be implemented in an underlying language, e.g. the machine language and/or an assembly language, but it may still be regarded as consisting of BETA system elements.

The wish to be able to describe system generators as BETA systems is essential, e.g. in order to be able to program operating systems entirely in BETA. It has the implication that the BETA language shall contain a *descriptor* concept and mechanisms for handling these, e.g. for translating and executing them. BETA system descriptions are descriptors giving rise to BETA systems.

In order to organize the possibilities of a system generator, e.g. descriptors which are provided for the users of the system generator, it is the intention to introduce a new kind of entities, the *context entities*. Contexts should also provide *links* between system generators and systems.

## Descriptors

By a descriptor we will understand a sequence of symbols describing an entity. In the present version of BETA there is a descriptor concept with the following characteristics:

— A descriptor may describe a singular entity (with a name) or if it is associated with a pattern (with a title) it may describe a category of entities.

— On descriptors there are operations which generate entities according to the descriptor. Given e.g. a pattern P, **OBJECT** P specifies the generation of an object entity. The generated object entity will have the descriptor of the pattern associated with it and use it in its behaviour as an object.

— It is possible to give a partial specification of a descriptor of a pattern (virtual patterns) and to bind a partially specified descriptor to a descriptor which is either partially or completely specified.

— There are some possibilities for aggregation of a descriptor by using other descriptors (infixing, prefixing and insertion), but more editing-oriented operations will be needed: addition/removal of descriptors to/from the system generator and modifications of descriptors, e.g. substitution of symbol sequences.

In addition to the already provided operations on descriptors it should be possible to specify transformations on descriptors, e.g. translation from one language to another or from one language level to another

**TRANSLATE** <descriptor>

It should also be possible to specify execution of a descriptor, e.g.

**EXECUTE** <descriptor>

## System Generators

The initial BETA system generator has been mentioned. It is the intention that it should not be necessary to go below the language of this generator; the set of computer components should be regarded as a "BETA machine".

A system generated by the initial BETA system generator may in turn act as a system generator, and a system generated by this may in its turn act as a system generator. In this way it should be possible for a set of computer components to contain a hierarchy of systems. In general we will say that the systems generated by a system generator constitute a *layer* of systems. Some of the systems of a layer may generate new systems, constituting a new layer, while other systems ″are just systems″. Note that we will consider this hierarchy of layers of systems as being of another kind than that of nesting. Consider a system SG acting as the system generator for a system S in the next layer. In order for SG to carry out the specified state transformations in S, by executing the descriptors of the entities of S, SG must have access to all entities of S, independently of their nesting within S. A nesting within the system only affects the ways in which entities of the system may involve each other.

The purpose of introducing a new layer is normally to provide sets of concepts which are aggregated from the concepts below. It may be pure aggregations, or a syntax for using the new concepts and a compiler may be provided so that a system generator for a new language is obtained. Some of the physical units of the configuration may have their descriptions in the initial BETA system exchanged with descriptions in terms of aggregated concepts. The purpose will be to provide new ways of operating the units (from the system). As examples we may take:

- In system descriptions for the initial BETA system generator the association of a processor to an object may be explicitly specified. The present language definition states that objects may have their actions executed independently and possibly concurrently. One of the first layers may then provide a time-sharing system generator so that objects which may be executed concurrently share the available processors.

- A device may be characterized by its ability to read and write characters. A next layer may provide concepts for reading and writing lines.

Addition of a new layer often implies loss of information about the hardware. A typical high level system generator is a generator for a high level language, where almost no information about the hardware is present and where system descriptions are submitted to the generator without caring about how the specific computer components are involved in their execution.

## Contexts

We have now given a description of how a set of computer components may contain a hierarchy of layers of systems. Some of these may be system generators.

When submitting a system description to such a set of system generators it should be specified by which system generator it shall be generated A system description that assumes some particular specified properties of its system generator, can only be submitted to system generators having these properties.

In addition to (or as part of) this linking up to the system generator, a system should be linked up to its context(s). The approach for the introduction of contexts in BETA discussed in the following is:

> *Context entities* are autonomous entities of a special kind (different from the kinds object and instance). When they are components of a system generator, a system generated by this generator may have some of these as context entities. This will probably be obtained by references. A system with a context will get the attributes of the context entity available.

As contexts are regarded as entities of a special kind, the specification of a context entity could have the form

C: **CONTEXT** <entity specification>

Contexts may be regarded according to any descriptor, and may thus have data items and patterns as attributes and may have action parts.

The possibilities for a system to use and/or influence its system generator will be provided by the contexts. When using the pattern attributes of a context for the specification and generation of entities within the system, the context entity is used as a library component. When using the data items of a context it is used as a data base component. Context entities survive the single systems and modifications made by one system may be observed by another. It should also be possible for more than one system to have context references to the same context entity (at the same time). On the other hand it should also be possible to obtain individual copies of contexts for each system. This is especially important in concurrency situations where a concurrent system should not have to wait for other systems in order to use important or frequently used concepts. An example of a context which will always be present is the context containing definitions of integer, real, etc. Even if they may be considered defined in a common context, a "copy" will be associated with each processor of the generator. A context entity with an action part which e.g. contains specifications of translation and/or execution of descriptors will play the role of an operating system component.

The context concept is to cover many desirable properties of the interface between system generator and the systems generated by it. In addition to the language of a system generator it is desirable to be able to provide it with sets of related concepts, described in the language and available to the users of the generator. As examples may be mentioned a set of standard mathematical functions, a set of additional I/O operations etc. This use of context focuses on the pattern attributes of contexts, but data items may also be useful in this connection, e.g. for holding statistics on the use of the context. Another example on context of this kind will be the contexts used for definition of "Standard BETA" (see section 5). There will e.g. be a context containing definitions of the standard types. Together with a syntax for expressions involving these types and a compiler, this provides a Standard BETA system generator. Note that although we will regard the standard types as defined by patterns, described in the present version of BETA, the arithemtic capabilities of the hardware will of course be used in an actual implementation of an initial system generator on that hardware. Non-standard types as e.g. DATE, COMPLEX, etc. should also be defined within contexts.

We believe that associated with the introduction of a context concept there will be an introduction of new and/or additional scope rules. We see the context entity as a natural means for obtaining a protection concept with other properties than the one obtained by nesting. Contexts will be the units which are designed for use in programs not yet written or even conceived, often defining sets of concepts which should only be used in a specified manner. We believe that the virtual concept also may be useful in connection with contexts, but we give an indication of a possible kind of protection for contexts:

> Consider a context C with a definition of a type T (by a pattern), with the functional operators defined by patterns not defined locally to T. This is desirable, because then the functional form, e.g. <A,B> → TPLUS → <C> may be used instead of the form <B> → A'TPLUS → <C>. In the present version of BETA the value part is not remotely accessible, so in order to operate on the value part *within* the context, it should be (and is) accessible from main parts of entities prefixed by T. When *using* the context, however, prefixing with T should perhaps not give access to the T-value part, in order to protect the type. This calls

> for a specification of different scope rules for prefixing inside and outside a context. Alternatively it would be a solution to open up for remote access to the value part and then provide the possibility of specifying auxiliary attributes within a context, that is attributes which are only known within the context. A drastic alternative is to open up for access into nested entities within a system and then only have the present scope rules for access from system to context.

As mentioned it is convenient to be able to collect related concepts and define them within one unit. If e.g. one of several related concepts is used it is likely that some of the others also are used. In any case there may be concepts which are used to define the one used, so that these will be used implicitly.

On the other hand it should also be possible to separate independent sets of concepts, and a system should only need to refer to concepts which are actually needed. It is therefore the intention that a system should be able to obtain references to more than one context. The syntax for context-specification is of course not settled, but for the time being we may assume that an entity has a context part consisting of a set of context declarations. A context declaration of the form

```
C: CONTEXT <entity specification>
```

has been mentioned. This should declare a "local" context, generated according to the descriptor of the entity specification. The specification of a context reference denoting a context entity of the system generator could be

```
M: CONTEXT == MATLIB
```

It should also be possible to declare anonymous contexts:

```
CONTEXT <entity specification>
CONTEXT == MATLIB
```

The effect should be that the attributes of the contexts are available without any remote denotation. In the first example the pattern SIN has to be denoted by M.SIN.

When using more than one context there may be name conflicts between the attributes of the contexts. This may be solved by requiring a remote denotation of the attributes. We would however like to provide a kind of renaming mechanism so that it should suffice with declaration of anonymous contexts.

A context for a system should give possibilities which are provided for all entities in the system, independently of their places in the nested structure. Given a MATLIB context for a system it should e.g. be possible for more than one entity to generate instances of a pattern SIN of the context. Infixes of a context should be executable by any entity, but of course still only by one at a time.

## 5. COMMENTS TO THE PROPOSAL

The development of a comprehensive programming language proposal will at some time enter a stage where the language is exposed to comments and criticism. It may be possible to postpone that time, in order to polish the proposal and guard against errors in e.g. language details and examples. The risk is that useful discussion which should influence the design details may appear too late, either delaying the implementation or simply being neglected. On the other hand early publication aimed at a wider audience may result in criticism for using peoples' time on unfinished presentation of ideas whose values have not yet been proved, etc.

The authors have followed a kind of intermediate approach: We have published three Working Notes describing the state of the development:

- WN-2, September 1977, contained our initial set of ideas,
- WN-3, August 1978, presented the development in the setting of a short tutorial,
- WN-4, February 1979, presented a definition of those language aspects which were well developed at that time.

These Working Notes were circulated to a group of people which had an active interest in commenting upon the language development. The next Working Note was planned to appear in April and should contain minor revisions of WN-4, but extended with contexts and those parts of BETA which relate to concurrency.

Immediately after the publication of WN-4, the extended usee of thee **ENTRY**- and **EXIT**-clauses were invented. This required a complete re-working of the language, now also taking concurrent action sequences into account as the normal case. As a result, this Working Note is less definite and less polished than planned.

## 5.1 General Comments

As stated in the preface, the main features of BETA are fixed. What remains are:

1. Correction of errors and inconsistencies.
2. Improvement of proposals.
3. Extensions of the language by ″contexts″ and the ″layer″ concept.
4. The definition of a ″standard BETA″, consisting of BETA augmented by a standard context containing normally used features as e.g. the patterns REAL, INTEGER, and their associated operators.

With regard to pt. 1, hopefully all major errors and inconsistencies are removed in the course of the language ″freezing″ process. (It is well known that minor faults are often discovered during implementation, and that some faults stay with a language even after it is commonly used. We hope of course to avoid this, and the development of formalized BETA definitions should be helpful also for this purpose.)

### Improvements of the proposal

1. The semantics related to sequencing is based upon the stack principle described in section 2. There are two main points which need further study:

   — The transition from concurrency into quasi-parallel (coroutine) sequencing. Our approach is to make the initial assumption that all objects may act concurrently, and to arrive at rules for quasi-parallelism by ″collapsing″ concurrency, instead of trying to ″generalize″ from single-stack sequencing through quasi-parallel to concurrency. This approach has resulted in departures from e.g. Simula's sequencing principles. The main structure of BETA's parallelism is clear, but there are still details to be considered for extension and revision.

   — The details of the stack discipline relating to prefix entities. The point in question is the possibility of, within an entity, to assign a value to its constituent prefix entity. Assignment to individual items on the prefix's value-list is possible, but it is desirable to make direct assignment to the ″total value″ as defined by the category of the prefix. (E.g. we want to handle an entity of category ″SINUS″ as having an associated REAL value, not only producing a REAL value through its EXIT-list.) Proposals for achieving this exist, but need further evaluation.

— The transfer of action and branching of action sequences between objects may be specified in detail by **EXIT**- and **ENTRY**-clauses. If this kind of transfer or branching is initiated by an object S and involves another object R, then the execution of the **EXIT**-clause of S is carried out with the stack of R operating as a *slave stack* in relation to the stack of S which operates as a *master stack*. When the **EXIT**-clause of S is finished, the **ENTRY**-clause of R is executed, now with R's stack as the master and S's stack as the slave. When the **EXIT-ENTRY**-clauses are finished the stacks of S and R operate as normal stacks.

A master stack may cause the expansion of its slave stack by additional stack member entities. These entities will have return points within corresponding entities in the master stack in addition to the return points referring to the local stack organization.

2. Both the syntax and semantics of the **ENTRY**- and **EXIT**-clauses are very recent, and we have not yet succeeded in giving them a form which is satisfying. The problems relate to the syntactical status of the <*a1,a2,...,an*> list notation and to the proper way of concatenating lists and defining correspondence between lists in more complex situations (as well as in some very simple ones, e.g. <a*1*,...a*n*> → <b*1*,...,b*n*>).

The "arrow operator", "→", is in the present proposal being used in a large variety of cases. It has been discussed whether it would be advisable to introduce the operator "=>" for those situations where X → Y (present notation) imply a transmission of values from X's value part to Y's value part.

3. The notion of concurrency has in some respects been modified from our earlier ideas (WN-2), whereas certain basic views remain constant. Simula's "remote procedure calls" have several counterparts in BETA. They are now handled by a common syntax which generalizes (a modified version of) the corresponding Simula notion. The new format of the clauses specifying conditions for acceptance of interrupts are of course influenced by the recent discussion of "message passing".

The notation we now propose and its possibilities are new to us, and we are exploring their use. This may result in modifications.

4. The BETA syntax is now being examined in an attempt to obtain an improved and more systematically constructed version. In a number of frequent special cases the present syntax is somewhat cumbersome. It will be considered to introduce standard abbreviations for these cases. A more systematic syntax relates in particular to the declarations and notations for binding of category, substance and/or value of entities. A by-product seems to be increased power of the virtual concept.

   By extending the virtual and reference concepts it is possible to arrive at a large variety of binding and referencing situations — analogous to e.g. ALGOL68, or PASCAL, or other languages. Our approach has been to restrict the use of explicit references to object and context entities, and also to restrict aliasing relating to constituent entities (even if there exist some possibilities, e.g. through SUC and PRED). We hope that the discussion of BETA will bring comments on these issues.

## Extensions of the language

The most important extensions of BETA are being discussed in section 4 of the Working Note. It should be observed that the design of the context concept to a large extent becomes a deduction to arrive at properties which are needed but not naturally offered by the other structuring tools (as e.g. nesting).

Similarly, the descriptor concept has to be further developed together with a careful study of a proper decomposition and restructuring of those pieces of software which today is referred to as "compilers", "runtime systems", "operating systems", "linkers", "loaders", "file systems", etc.

It should also be observed that a descriptor at one system level may be regarded as an entity (probably an object) with a piece of program text as a (possibly structural) attribute, plus attributes indicating the language of the program text etc. The same descriptor may at later system layers function in its capacity of program text only, as descriptor attribute of entities.

Initialization of values are not discussed in this report. Some solutions have been evaluated, but they must now be reevaluated together with the final settling of binding rules in general. (If special language tools are given for initialization of values, then it may be argued that the values assigned to data items (e.g. variables) should *not* be dependant upon the declaration sequence within an entity's attribute part.)

### Standard BETA

The BETA of this report is a "basic BETA" in terms of which other standard concepts may be aggregated. We will of course provide, as stated above, the standard types with a "standard BETA", implemented by efficient use of the available hardware. In the examples of the next section we have assumed the availability of obvious parts of a "standard BETA".

## 5.2 Specific Comments

In this subsection we enumerate a list of either alternative language constructs to be considered or extensions to one of the proposed constructs. The purpose is to indicate more specifically where there are alternatives to the present proposal. It should be obvious that such a list cannot be exhaustive.

*INNER* (3.13)
The INNER concept needs to be reconsidered as the proposed generalization is problematic when INNER is applied in the action part of an entity being inserted. In this case it will be possible to obtain multiple membership in an object stack.

*Repetition and selection imperatives* (3.10, 3.11)
In the present semantics of a selection-imperative it is required that at least one <imperative-list> may be selected for execution. It is considered to drop this requirement. Variants of the repetition and selection imperatives may be included, i.e. an ELSE-part of the selection imperative, an unlimited repetition, etc. It is however *not* intended to include special purpose variants of these imperatives, e.g. for repetition to include WHILE-DO, REPEAT-UNTIL, FOR-TO-DO, FOR-DOWNTO-DO imperatives as known from Pascal.

*Naming of INDEX and RANGE* (3.4, 3.5, 3.11)
The possibility of associating names with INDEX and RANGE in repetitions may be included. The problems may arise when nesting of repetitions is involved.

*Repetition-assignment* (3.14)
An extension of assignment to be defined between infix/reference repetitions may be considered. An assignment of this kind is presently only possible by assigning the sequence, data-item by data-item.

*Signal-specification* (3.17)
The specification possibilities of the communication imperative may be extended or reduced (i.e. conditions on sender, receiver, message etc.), implying respectively a stronger or weaker communication control.

*Nesting of WHILE-imperatives* (3.17)
The entity, executed as part of the execution of a **WHILE**-imperative, may itself execute a **WHILE**-imperative. In the present semantics all such nested **WHILE**-imperatives are effective in the sense that an interrupt may take place at each level. It is being considered that only the innermost (latest) **WHILE**-imperative should be effective in the sense that only signals specified by this **WHILE**-imperative may be communicated.

*Interruption of entities* (3.17)
The entity which is executed as part of a **WHILE**-imperative may be interrupted at semicolon of that entity (not a semicolon of entities above it in the object stack). The current semantics may be reconsidered and the introduction of e.g. a **STATE**-imperative discussed. The purpose of **STATE** would be to indicate the points at which the entity could be interrupted.

The possible points of interruption of an entity (either at semicolon or **STATE**) applies when the signal is coming from another object in the same system. In the situation with layers of systems (see section 4) it may be considered that a signal from a lower system level may cause an interrupt at any stage of the execution of the entity.

Another possibility is that only objects and not entities may be interrupted: Let object S execute an imperative

**WHILE** X **ON** R?M **THEN** ... **ENDWHILE**

It is considered to change the **WHILE**-imperative such that X has to be a reference. The semantics should then be: If a processor is connected to S then the execution of **WHILE** X ... implies that this processor is now connected to X; the object denoted by X will then be interrupted if a signal may be communicated to S. Among others this has the advantage that the execution of M cannot interfere with X (except through signals).

*Internal objects* (3.17)
Internal objects to a given object can be involved in a communication using the signal concept. An interpretation of this situation may be that the object has a number of devices which independently may registrate, execute or emit signals. The implications of this possible configuration must be carefully investigated.

*Nondeterministic processing* (3.11, 3.17)
Presently nondeterminism is achievable through three kinds of specifications based on internal conditions — and a single specification based on conditions external to the object. It may be desirable to combine all kinds of conditions, possibly by an integration of the existing possibilities. A further generalization of this idea will be an imperative supporting general nondeterministic processing.

*Initial values* (3.4, 3.5)
The problem concerning initial values of the data items has not yet got a proper solution. One possibility is to introduce a special initialization-clause in the entity-descriptor. Alternatively an initialization-part may be added to each data item declaration. In both of these cases it may be admissible to omit initial values.

Another possibility is to exclude special initialization possibilities entirely. Then it must be decided upon whether the data items will have standard default values/states or will be undefined.

*Initial values to objects* (3.16)
The EXIT/ENTRY-parts have no meaning in objects. It should however cause no problems to allow a construction of the form

E → **OBJECT** P = => R

where after generation of a P-object it is initialized (via ENTRY) by the expression E.

*Prefixing, infixing and insertion by a pattern referred to by an infix- or reference-denotation* (3.2, 3.4, 3.8, 3.16)
The specification of constituent entities by means of a pattern title referred to through an infix-denotation implies only minor problems and is a likely language improvement. More doubtful is the similar specification through a reference-denotation because of the dynamic aspect of references.

*Prefixing by references* (3.2)
The possibility of using a reference as prefix-part of an entity may be considered. A consequence of this may be that a reference itself has to be

regarded as an entity. One way of obtaining this situation is to introduce special *reference values* to be included in the value list and then to define references as entities with such occurrences in their value list.

*Default binding of virtual patterns* (3.6)
As proposed virtual patterns are bound by default to the currently valid virtual specification. It is possible to extend the pattern declaration to involve the existing virtual specification as well as a supplying default possibility.

*Generalization of the virtual concept to infixes and references* (3.4, 3.5, 3.6)
It may be considered to have virtual infixes and references with respect to state, substance (or better: name) and qualification. As for state this possibility is related to the initial value problem and the specification of constant infixes and references. Concerning substance this possibility may correspond to language concepts as indirect references and variable parameters for respectively references and infixes. Extension with respect to qualification seems doubtful.

*Reference assignment* (3.16)
Involvement of attributes of objects is obtained by communication. Messages may only be infixes and instances which implies that reference attributes may not be involved directly (but they may indirectly by infixes and instances). Thus a remote reference R.S cannot be assigned to directly and its value cannot be obtained directly. This has to be reconsidered. Solutions may be to allow references as messages or to allow remote reference involvement as a special feature.

*Existence*
The problems related to "existence" or "lifespan" of entities are not discussed in this report. They are strongly related to the set of storage management procedures made available. They have, however, also implications on language design. The possibility of prefixing with a pattern at levels different from the level where it is declared open up for references denoting internal objects. This calls for a careful definition of when entities may disappear (or may be removed) from a BETA system and of reference qualification.

## APPENDIX — Examples

The purpose of this appendix is to give a set of examples demonstrating most of the constructs in BETA. As mentioned in section 5 we use obvious parts of a "standard BETA" such as commonly used data types and their operations. The use of "standard BETA" is however limited as much as possible as the purpose of the examples is to demonstrate BETA as it is and we do e.g. not use other control structures than those in BETA.

*Example A* is the BETA version of Hoare's smallintset. The properties common to the operations HAS, INSERT and REMOVE are collected in the pattern OPERATION. The BETA version is extended with a pattern ERROR and an operation FORALL. If an insertion is tried when there is no more place an ERROR instance is generated. ERROR is a virtual pattern which means that the user of SMALLINTSET may provide the actual set with a specific ERROR pattern. The operation FORALL may be used to scan the set and perform some actions for each set member.

```
PATTERN OPERATION:
   BEGIN E: INTEGER
     ENTRY <E>
   END;

PATTERN SMALLINTSET:
   BEGIN
     REP: BEGIN S: [100] INTEGER; SIZE: INTEGER END;
     PATTERN ERROR : VIRTUAL EXCEPTION;
     PATTERN HAS:
           OPERATION
           BEGIN B: BOOLEAN;
           DO FALSE → B;
              L: [REP'SIZE]
                 IF (E = REP'S [INDEX])
                    = TRUE THEN TRUE → B; LEAVE L
                    = FALSE THEN ENDIF
           EXIT <B>
           END (*HAS*);
```

```
    PATTERN INSERT:
        OPERATION
        BEGIN C: BOOLEAN;
        DO <E> → HAS → <C>;
          IF C  = FALSE THEN
                IF (REP'SIZE < 100)
                  = TRUE THEN REP'SIZE + 1 → REP'SIZE;
                      E → REP'S [REP'SIZE]
                  = FALSE THEN INSTANCE ERROR ENDIF
              = TRUE THEN ENDIF
        END (*INSERT*);

    PATTERN REMOVE:
        OPERATION
        BEGIN DO
          L: [REP'SIZE]
            IF (REP'S[INDEX] = E)
              = TRUE THEN
                BEGIN J: INTEGER;
                DO INDEX → J;
                  [REP'SIZE-INDEX]
                    BEGIN DO
                      REP'S[J+1] → REP'S[J];
                      J+1 → J
                    END;
                  REP'SIZE-1 → REP'SIZE;
                  LEAVE L
                END
              = FALSE THEN ENDIF
        END (*REMOVE*);

    PATTERN FORALL:
        OPERATION
        BEGIN DO
          [REP'SIZE]
            BEGIN DO
              REP'S[INDEX] → E;
              INNER
            END
        END (*FORALL*)
END (*SMALLINTSET*)
```

Examples on use:

```
S1: SMALLINTSET;
<5> → S1'INSERT;
S1'FORALL
BEGIN DO
   <E> → PRINT
END;

S2: SMALLINTSET
BEGIN
   BIND ERROR: ERRORROUTINE
END
```

*Example B* is a generalization of the SMALLINTSET of example A. The types of the members are not settled (but required to be subtypes of TYPE) and the maximum size of the set is open. The operations will have the same specifications as in SMALLINTSET.

```
PATTERN OP: BEGIN END;

PATTERN SET:
   BEGIN
      PATTERN SETTYPE: VIRTUAL TYPE;
      PATTERN MAX: VIRTUAL INTEGERVALUE;
      PATTERN OPERATION: OP BEGIN E: SETTYPE; ENTRY <E> END;
      REP: BEGIN S: [MAX] SETTYPE; SIZE: INTEGER END;
      .
      .
      .
   END (*SET*)
```

Examples on use:

```
PATTERN INTSET: SET BEGIN BIND SETTYPE: INTEGER END
PATTERN SMALLSET: SET BEGIN BIND MAX:100 END
PATTERN SMALLINTSET:
   SET
   BEGIN BIND SETTYPE: INTEGER;
      BIND MAX:100
   END
```

*Examples C, D, E* are examples on the use of the value part to represent the part of the state which is to be transferred in an assignment.

*Example C* defines an enumeration and demonstrates how to define constants (literals) which may not be assigned to (no ENTRY) and variables which may be assigned to:

```
PATTERN COLOUR:
   BEGIN
      PATTERN VP: VIRTUAL INTEGERVALUE;
      VALUE V: VP
      EXIT
   END;

PATTERN RED:  COLOUR BEGIN BIND VP: 1 END;
PATTERN BLUE:   COLOUR BEGIN BIND VP: 2 END;
PATTERN WHITE:  COLOUR BEGIN BIND VP: 3 END;

PATTERN COLOURVAR: COLOUR BEGIN ENTRY END
```

Examples on use:

```
      C1, C2: COLOURVAR;
      RED → C1; C1 → C2;
      IF C1  =RED THEN ...
             = BLUE THEN ...
             = WHITE THEN ... ENDIF
```

*Example D* demonstrates how to obtain a datatype with different concrete and abstract representations. The operations CONVERT, CHECK CONSISTENCY and NEWYEAR are not specified here. CONVERT converts a date of the form (DAY, MONTH, YEAR) to a Julian date and CHECK CONSISTENCY checks whether a date is consistent or not. NEWYEAR replaces the year part of a date with a new one.

```
PATTERN DATE:
   BEGIN
      VALUE R: INTEGER; (* JULIAN date *)
      ENTRY EXIT
   END
```

```
PATTERN NEWDATE:
   BEGIN DAY, MONTH, YEAR: INTEGER;
      D: DATE
         BEGIN
         DO <DAY,MONTH,YEAR> → CONVERT → R
         END
   ENTRY <DAY,MONTH,YEAR>; CHECK CONSISTENCY
   EXIT <D>
   END;

PATTERN THIS-DAY-YEAR-N:
   DATE
   BEGIN N: INTEGER; (* YEAR *)
   ENTRY <N>
   EXIT <N> → NEWYEAR → <R>
   END
```

Examples on use:

```
      D1, D2: DATE;
      <17,5,1814> → NEWDATE → <D1>;
      D1 → D2;
      D2 <1918> → THIS-DAY-YEAR-N → D1
```

*Example E* defines a bounded real and a widest real. In every assignment to a bounded real the nearest real value within an interval is obtained. A widest real will have an associated interval which is the widest of all the bounded reals which have been assigned to it.

```
PATTERN BOUNDED:
   BEGIN LOWER, UPPER: REAL;
      SETINT:
         BEGIN
            ENTRY <LOWER,UPPER>
         END;
      BOUND:
         BEGIN DO
            IF TRUE
               = (R < LOWER) THEN LOWER → R
               = (R > UPPER) THEN UPPER → R
               = (LOWER ≤ R AND R ≤ UPPER) THEN ENDIF
         END;
      INREAL: BEGIN ENTRY <R>; BOUND END;
      OUTREAL: BEGIN EXIT <R> END;
      VALUE R: REAL
      ENTRY INNER BOUND
      EXIT
   END (* BOUNDED *)

PATTERN WIDEST:
   BOUNDED
   BEGIN
      ENTRY
         IF (PRED'LOWER < LOWER)
            = TRUE THEN PRED'LOWER → LOWER
            = FALSE THEN ENDIF;
         IF (PRED'UPPER > UPPER)
            = TRUE THEN PRED'UPPER → UPPER
            = FALSE THEN ENDIF
   END (* WIDEST *)
```

Examples on use:

```
       B1, B2: BOUNDED; W1: WIDEST;
       <5,15> → B1'SETINT; <1,9> → B2'SETINT;
       <7,12> → W1'SETINT;
       <4> → B1'INREAL; B1 → B2; B1 → W1; B2 → W1;
       W1'OUTREAL → <S>
```

*Example F* is an example on objects acting as coroutines, activating each other by sending messages. It is the BETA version of the well-known example problem of transferring a stream of characters read from cards (of 80 characters) to lines of 125 characters, replacing double occurrences of '*' by '↑'. The disassembler (DA) reads a card from INCARD (not specified) and sends the characters individually to the squasher (SQ) which replaces '**' by '↑' and sends the characters to the assembler (AS). The assembler assembles the characters in lines of 125 characters. A line is printed by activating LINEOUT. STOP will have the effect of stopping the assembler. Note that the imperative INCARD?C is not presently allowed in BETA.

```
REF DA:
   BEGIN C: [80] CHAR;
   DO
   L: BEGIN DO INCARD?C;
        [80] C[INDEX] → SQ!CH;
        ' ' → SQ!CH; RESTART L
      END
   END (* DA *);

REF SQ:
   BEGIN CH: CHAR;
   DO
   L: BEGIN DO DA?CH;
        IF (CH = '*')
          = TRUE THEN DA?CH;
            IF (CH = '*')
              = TRUE THEN '↑' → AS!CH
              = FALSE THEN '*' → AS!CH; CH → AS!CH ENDIF
          = FALSE THEN CH → AS!CH ENDIF;
        RESTART L
      END (* L *)
   END (* SQ *);

REF AS:
   BEGIN CH: CHAR; L: [125] CHAR;
   DO
   A: BEGIN DO
        [125]
          BEGIN DO SQ?CH → L[INDEX];
            IF (CH = 'end')
              = TRUE THEN
                BEGIN I: INTEGER;
                DO INDEX → I;
                  [RANGE-I] ' ' → L[INDEX];
                  LINEOUT!; STOP
                END
              = FALSE THEN ENDIF
          END;
        LINEOUT!; RESTART A
      END
   END
```

*Example G* is a simple clock. A clock object will repeatedly perform the action of adding a unit to time. At any time it may be interrupted in order to deliver the current value of time. The time used for interruption (TI) is for each reentry into the interrupted action added to TIME. Similarly the time used for reading TIME (TR) is added to TIME.

```
PATTERN CLOCK:
   BEGIN
      TIME, UNIT, TI, TR: REAL;
      NR: INTEGER;
      SETCLOCK:
         BEGIN
         ENTRY <TIME>
         END
   DO 0 → NR;
      SENDER?SETCLOCK;
      WHILE
         BEGIN
         ENTRY TIME + TI → TIME
         DO TIME + UNIT → TIME
         END
      ON SENDER?TIME THEN NR+1 → NR; TIME+TR → TIME
      ENDWHILE
   END (* CLOCK *)
```

Example on use:

```
      REF CL: CLOCK;
      <0> → CL!SETCLOCK;
      CL.TIME → T
```

*Example H* specified a bounded buffer object. A buffer object has two operations which always may be applied. It has two internal objects which will add/remove an element in the buffer provided that the buffer is not full/empty.

```
PATTERN BUFFER:
  BEGIN
    S: [N] CHAR;
    IN,OUT: INTEGER;
    PATTERN COND: BEGIN B: BOOLEAN EXIT <B> END;
    PATTERN FULL: COND BEGIN DO (OUT = IN ⊕ 1) → B END;
    PATTERN EMPTY: COND BEGIN DO (OUT = IN) → B END;

    REF PUT:
      BEGIN NEW: CHAR; B: BOOLEAN;
      DO
      L: BEGIN DO FULL → <B>;
            IF B = FALSE THEN SENDER?NEW → S[IN]; IN ⊕ 1 → IN
                 = TRUE THEN ENDIF; RESTART L
        END
      END

    REF GET:
      BEGIN B: BOOLEAN;
        NEXT: BEGIN EXIT <S[OUT]> END
      DO
      L: BEGIN DO EMPTY → <B>;
            IF B = FALSE THEN SENDER?NEXT; OUT ⊕ 1 → OUT
                 = TRUE THEN ENDIF; RESTART L
        END
      END

  DO
  L: BEGIN DO
      SENDER?MESSAGE AS COND;
      RESTART L
    END
  END (* BUFFER *)
```

Examples on use:

```
    REF B: BUFFER;
    'A' → B.PUT!NEW; B.GET.NEXT → <CH>;
```

*Example I* demonstrates how to define suitable control structures using the prefix/inner mechanism.

```
PATTERN for:
   BEGIN
      X: [N] INTEGER;
      PATTERN N: VIRTUAL INTEGERVALUE;
      INX: INTEGER
      ENTRY <X>
   DO [N]
      BEGIN DO
         X[INDEX] → INX;
         INNER
      END
   END
```

Example on use:

```
       <1,27,5,33,11,13> →
       (F: FOR)
          BEGIN
             BIND N:6
          DO F'INX → INSTANCE Q
          END
```

*Example J* demonstrates the use of the virtual concept in obtaining "procedures as procedure parameters". A pattern SUM is defined. It computes the sum F(A[1]) + F(A[2]) + ... + F(A[A'RANGE]) where A is a repetition of REALs and F is a REALFUNCTION:

```
A: [N] REAL;
PATTERN REALFUNCTION:
    BEGIN R: REAL
        ENTRY <R> EXIT <R>
    END;

PATTERN SUM:
    BEGIN
        S,R: REAL:
        PATTERN F: VIRTUAL REALFUNCTION;
    DO 0 → S;
        [A'RANGE]
            BEGIN DO
                <A[INDEX]> → F → <R>;
                S + R → S
            END
        EXIT <S>
    END
  .
  .
  .
DO INSTANCE SUM
    BEGIN
        BIND F: SIN
    END → <S1>; ...
```

*Example K* describes a resource object. At any time at most one object owns the resource and may operate the resource. However any object may at any time request the resource and a reference to that object will then be put into a queue. When the current owner releases the resource, a new owner will be selected and acknowledged. The objects that may use the resource have to be prefixed by PROCESS which is not specified here.

```
PATTERN RESOURCE:
   BEGIN AGENDA: QUEUE; (* waiting requests *)
      REF OWNER: PROCESS; (* current owner of the resource *)
      REQUEST: BEGIN DO <SENDER> → AGENDA'ENTER END;
      RELEASE: BEGIN END;
      PATTERN OPERATION: (* to prefix all other RESOURCE operations *)
         BEGIN
         DO WHILE INNER
            ON SENDER AS PROCESS?REQUEST  THEN
            ENDWHILE
         END;
      NEWOWNER:
         BEGIN DO AGENDA'NEXT → <OWNER>;
            IF (OWNER == NONE) = FALSE THEN OWNER!
                               = TRUE THEN ENDIF
         END
DO NONE ==> OWNER;
   WHILE WAIT
   ON SENDER AS PROCESS?REQUEST THEN NEWOWNER
   ON OWNER?RELEASE THEN NEWOWNER
   ON OWNER?MESSAGE AS OPERATION THEN
   ENDWHILE
END (* RESOURCE *);
```

Examples on use:

```
REF PRINTER:
    RESOURCE
    BEGIN
        WRITECH: OPERATION BEGIN ... END;
        NEWLINE: OPERATION BEGIN ... END;
        .
        .
        .
    END

REF A:
    PROCESS
    BEGIN DO
        PRINTER!REQUEST;
    L: WHILE X
        ON PRINTER? THEN LEAVE L
        ENDWHILE
        <'*'> → PRINTER!WRITECH; PRINTER!NEWLINE;
        PRINTER!RELEASE;
    END (* Q *);
```

TRYK: DAIMI/RECAU