INTERNATIONAL TELECOMMUNICATION UNION

CCITT

THE INTERNATIONAL
TELEGRAPH AND TELEPHONE
CONSULTATIVE COMMITTEE

RED BOOK

VOLUME VI – FASCICLE VI.10

FUNCTIONAL SPECIFICATION AND DESCRIPTION LANGUAGE (SDL)

RECOMMENDATIONS Z.100-Z.104

VIIITH PLENARY ASSEMBLY
MALAGA-TORREMOLINOS, 8-19 OCTOBER 1984

Geneva 1985

INTERNATIONAL TELECOMMUNICATION UNION

CCITT

THE INTERNATIONAL
TELEGRAPH AND TELEPHONE
CONSULTATIVE COMMITTEE

RED BOOK

VOLUME VI – FASCICLE VI.10

FUNCTIONAL SPECIFICATION AND DESCRIPTION LANGUAGE (SDL)

RECOMMENDATIONS Z.100-Z.104

VIII^TH PLENARY ASSEMBLY
MALAGA-TORREMOLINOS, 8-19 OCTOBER 1984

Geneva 1985

ISBN 92-61-02231-6

CONTENTS OF THE CCITT BOOK
APPLICABLE AFTER THE EIGHTH PLENARY ASSEMBLY (1984)

RED BOOK

Volume I	–	Minutes and reports of the Plenary Assembly. Opinions and Resolutions. Recommendations on: – the organization and working procedures of the CCITT (Series A); – means of expression (Series B); – general telecommunication statistics (Series C). List of Study Groups and Questions under study.
Volume II	–	*(5 fascicles, sold separately)*
FASCICLE II.1	–	General tariff principles – Charging and accounting in international telecommunications services. Series D Recommendations (Study Group III).
FASCICLE II.2	–	International telephone service – Operation. Recommendations E.100-E.323 (Study Group II).
FASCICLE II.3	–	International telephone service – Network management – Traffic engineering. Recommendations E.401-E.600 (Study Group II).
FASCICLE II.4	–	Telegraph Services – Operations and Quality of Service. Recommendations F.1-F.150 (Study Group I).
FASCICLE II.5	–	Telematic Services – Operations and Quality of Service. Recommendations F.160-F.350 (Study Group I).
Volume III	–	*(5 fascicles, sold separately)*
FASCICLE III.1	–	General characteristics of international telephone connections and circuits. Recommendations G.101-G.181 Study Groups XV, XVI and CMBD).
FASCICLE III.2	–	International analogue carrier systems. Transmission media – characteristics. Recommendations G.211-G.652 (Study Group XV and CMBD).
FASCICLE III.3	–	Digital networks – transmission systems and multiplexing equipments. Recommendations G.700-G.956 (Study Groups XV and XVIII).
FASCICLE III.4	–	Line transmission of non telephone signals. Transmission of sound-programme and television signals. Series H, J Recommendations (Study Group XV).
FASCICLE III.5	–	Integrated Services Digital Network (ISDN). Series I Recommendations (Study Group XVIII).

Volume IV – *(4 fascicles, sold separately)*

FASCICLE IV.1 – Maintenance; general principles, international transmission systems, international telephone circuits. Recommendations M.10-M.762 (Study Group IV).

FASCICLE IV.2 – Maintenance; international voice frequency telegraphy and fascimile, international leased circuits. Recommendations M.800-M.1375 (Study Group IV).

FASCICLE IV.3 – Maintenance; international sound programme and television transmission circuits. Series N Recommendations (Study Group IV).

FASCICLE IV.4 – Specifications of measuring equipment. Series 0 Recommendations (Study Group IV).

Volume V – Telephone transmission quality. Series P Recommendations (Study Group XII).

Volume VI – *(13 fascicles, sold separately)*

FASCICLE VI.1 – General Recommendations on telephone switching and signalling. Interface with the maritime mobile service and the land mobile services. Recommendations Q.1-Q.118 *bis* (Study Group XI).

FASCICLE VI.2 – Specifications of Signalling Systems Nos. 4 and 5. Recommendations Q.120-Q.180 (Study Group XI).

FASCICLE VI.3 – Specifications of Signalling System No. 6. Recommendations Q.251-Q.300 (Study Group XI).

FASCICLE VI.4 – Specifications of Signalling Systems R1 and R2. Recommendations Q.310-Q.490 (Study Group XI).

FASCICLE VI.5 – Digital transit exchanges in integrated digital networks and mixed analogue-digital networks. Digital local and combined exchanges. Recommendations Q.501-Q.517 (Study Group XI).

FASCICLE VI.6 – Interworking of signalling systems. Recommendations Q.601-Q.685 (Study Group XI).

FASCICLE VI.7 – Specifications of Signalling System No. 7. Recommendations Q.701-Q.714 (Study Group XI).

FASCICLE VI.8 – Specifications of Signalling System No. 7. Recommendations Q.721-Q.795 (Study Group XI).

FASCICLE VI.9 – Digital access signalling system. Recommendations Q.920-Q.931 (Study Group XI).

FASCICLE VI.10 – Functional Specification and Description Language (SDL). Recommendations Z.101-Z.104 (Study Group XI).

FASCICLE VI.11 – Functional Specification and Description Language (SDL), annexes to Recommendations Z.101-Z.104 (Study Group XI).

FASCICLE VI.12 – CCITT High Level Language (CHILL). Recommendation Z.200 (Study Group XI).

FASCICLE VI.13 – Man-Machine Language (MML). Recommendations Z.301-Z.341 (Study Group XI).

Volume VII – *(3 fascicles, sold separately)*

FASCICLE VII.1 – Telegraph transmission. Series R Recommendations (Study Group IX). Telegraph services terminal equipment. Series S Recommendations (Study Group IX).

FASCICLE VII.2 – Telegraph switching. Series U Recommendations (Study Group IX).

FASCICLE VII.3 – Terminal equipment and protocols for telematic services. Series T Recommendations (Study Group VIII).

Volume VIII – *(7 fascicles, sold separately)*

FASCICLE VIII.1 – Data communication over the telephone network. Series V Recommendations (Study Group XVII).

FASCICLE VIII.2 – Data communication networks: services and facilities. Recommendations X.1-X.15 (Study Group VII).

FASCICLE VIII.3 – Data communication networks: interfaces. Recommendations X.20-X.32 (Study Group VII).

FASCICLE VIII.4 – Data communication networks: transmission, signalling and switching, network aspects, maintenance and administrative arrangements. Recommendations X.40-X.181 (Study Group VII).

FASCICLE VIII.5 – Data communication networks: Open Systems Interconnection (OSI), system description techniques. Recommendations X.200-X.250 (Study Group VII).

FASCICLE VIII.6 – Data communication networks: interworking between networks, mobile data transmission systems. Recommendations X.300-X.353 (Study Group VII).

FASCICLE VIII.7 – Data communication networks: message handling systems. Recommendations X.400-X.430 (Study Group VII).

Volume IX – Protection against interference. Series K Recommendations (Study Group V). Construction, installation and protection of cable, and other elements of outside plant. Series L Recommendations (Study Group VI).

Volume X – *(2 fascicles, sold separately)*

FASCICLE X.1 – Terms and definitions.

FASCICLE X.2 – Index of the Red Book.

PAGE INTENTIONALLY LEFT BLANK

PAGE LAISSEE EN BLANC INTENTIONNELLEMENT

TABLE OF CONTENTS OF FASCICLE VI.10 OF THE RED BOOK

Recommendations Z.100 to Z.104

Functional specification and description language (SDL)

PRELIMINARY NOTES

1 The Questions entrusted to each Study Group for the Study Period 1985-1988 can be found in Contribution No. 1 to that Study Group.

2 In this Fascicle, the expression "Administration" is used for shortness to indicate both a telecommunication Administration and a recognized private operating agency.

FASCICLE VI.10

Recommendations Z.100 to Z.104

FUNCTIONAL SPECIFICATION AND DESCRIPTION LANGUAGE (SDL)

PAGE INTENTIONALLY LEFT BLANK

PAGE LAISSEE EN BLANC INTENTIONNELLEMENT

INTRODUCTION TO SDL

1 Introduction

This Recommendation is a general explanation of and introduction to the CCITT *S*pecification and *D*escription *L*anguage (SDL). The language is defined in detail in Recommendations Z.101 to Z.104.

1.1 *General*

The purpose of recommending SDL is to provide a language for unambiguous *specification* and *description* of the behaviour of telecommunications *systems.* The *specifications and descriptions* using SDL are intended to be formal in the sense that it is possible to analyse and interpret them unambiguously.

A *system specification* consists of a *specification* of the *functional behaviour* and a set of *general parameters* of the *system.* SDL aims only to describe the behavioural aspects of a *system*; the *general parameters* describe properties like capacity and weight which have to be described using different techniques.

The terms *specification* and *description* are used with the following meaning:

– a *specification* of a *system* is the description of its required *behaviour,* and

– a *description* of a *system* is the description of its actual *behaviour.*

SDL describes the *behaviour* in a stimulus/response fashion, assuming that both the stimuli and responses are discrete entities. The approach is based on the concepts of *extended finite state machines.*

The *behaviour* of the *system* as described in SDL is the sequence of responses to any given sequence of stimuli as seen from outside the *system.* Any two *systems* described in SDL having the same *behaviour* in this respect are said to be *functionally equivalent.* The concept of *functional equivalence* is used to compare *systems* with each other, e.g. a *specification* of a required *system* with a *description* of an offered *system.* The SDL User Guidelines, which are appended to the Recommendations contain a discussion on the criteria and methods for this comparison.

SDL also provides structuring concepts which allow a *system* to be partitioned so that it can be defined, developed and understood one part at a time.

These concepts are of value both initially in specifying a *system,* when different aspects can be independently dealt with, and later in describing a *system,* when the description structures should match the *system* structure.

1.2 *Objectives*

The general objectives when defining SDL have been to provide a language that:

– is easy to learn, use and interpret in relation to the needs of an operating organization;

– provides unambiguous specifications and descriptions for ordering and tendering;

– may be extended to cover new developments;

– is able to support several methodologies of system specification and design, without assuming any one of these.

1.3 *Scope*

The main area of application for SDL is the description of the behaviour of aspects of telecommunications *systems.* Applications include:

– call processing (e.g. call handling, telephony signalling, metering) in SPC switching systems;

– maintenance and fault treatment (e.g. alarms, automatic fault clearance, routine tests) in general telecommunications systems;

– system control (e.g. overload control, modification and extension procedures);

– data communication protocols.

SDL can of course also be used for the description of any behaviour capable of being described using a discrete model, i.e. communicating with its environment by discrete messages.

2 Language survey

The following survey of SDL is intended as an introduction to Recommendations Z.101 to Z.104. The explanations given here are, except for § 2.1, not formal definitions, but are intended as just tutorial explanations.

2.1 *Some basic definitions*

Some general concepts and conventions are used throughout Recommendations Z.100 to Z.104. Their definitions follow here:

Type, definition and instance

In the Recommendations, an entity is strictly separated from its *definition.* The schema and terminology defined below and shown in Figure 1/Z.100 are used.

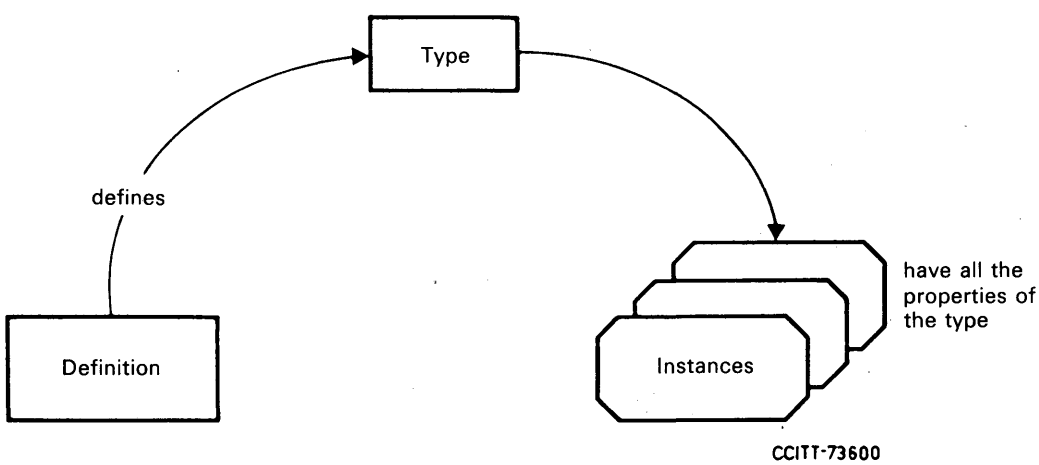


FIGURE 1/Z.100

The Type concept

A *type* is defined by its *definition,* which defines all properties associated with that *type.* A *type* may be *instantiated* in any number of *instances.* Any *instance* of a particular *type,* has all the properties defined for the *type.*

The schema applies to several SDL concepts, e.g. we have *system definitions* and *system instances, process definitions* and *process instances.*

The *instances* of a *type* may be *types* themselves. The properties associated with *types* are inherited according to the hierarchical application, e.g. the properties of the generic type *system* are inherited by all *instances* of specific *system definitions.*

To avoid cumbersome text, the convention used is that whenever it is obvious from context that the *definition* or the *instance* of a *type* is meant the attribute is omitted.

Names

The following naming conventions and terminology are adopted to identify *definitions* and *instances*:

All *identifiers* used in a representation are unique. The identification of an entity consists of two parts: one *name part* and one *qualifying part*:

$$\underbrace{\underbrace{\langle \text{System name} \rangle \quad \langle \text{Block name} \rangle \quad \langle \text{Signal type} \rangle}_{\textit{Qualifying part}} \qquad \underbrace{\text{OFFHOOK}}_{\textit{Name part}}}_{\text{Identification}}$$

The *qualifying part* is derived from the complete hierarchical context the entity is defined in and the *type* of the entity. The *name part* should be a meaningful *name* in relation to the purpose or effect of the named entity.

When the identification appears in the concrete syntax, the obvious *qualifiers* may be implied only; i.e. when it is not ambiguous only the *name part* should be used.

Visibility rules

When an *identifier* is introduced into a *specification* or *description* that *identifier* is *visible* at the point of introduction and at levels directly below the point of introduction in the *specification* or *description* hierarchy. The highest level at which the *identifier* is visible is indicated by the lowest hierarchical level given in the *qualifying part* of the *identifier.*

Specification error

Where properties of an SDL *specification* are inconsistent or ambiguous, that *specification* is *invalid.* Where the interpretation of an SDL *specification* violates properties of a *specification* which is valid, that *system* interpretation is in *error.* An interpretation of a *system* which leads to an error means that the future behaviour of the *system* cannot be predicted from the specification.

2.2 *The basic SDL*

The dynamic *behaviour* of an SDL *system* is generated by *process instances,* acting concurrently. A *process* is modelled as an *extended finite state machine.* It will only act in response to external discrete stimuli, and may then generate discrete responses back to its *environment.* Other *processes* may accept the generated responses as stimuli.

In SDL a *process* will wait in a *state* until it receives a valid *signal* from its *environment.* Then it will perform a *transition* to another *state.* During the *transition* it may perform *actions*; these may either be manipulation of information local to the *process* or *sending signals* to other *processes* or to the external *environment* of the system.

All *processes* in a *system* or in its *environment* have access to the "absolute time", and may perform time measurements and timing.

Process instances may be *created* and *terminated* dynamically. Any *instance* 4 may be created by another *instance* or it may exist at *system* initialization time. A *process instance* may only be *terminated* by an explicit *stop action* performed by itself.

Several *signal instances* may be waiting to be accepted by a *process.* In order to handle this *signal* queue, an *input port* is associated with each *process instance.* The queueing discipline is basically first in first out, but the *process* may itself manipulate this by a *save-signal-set* associated with a *state.*

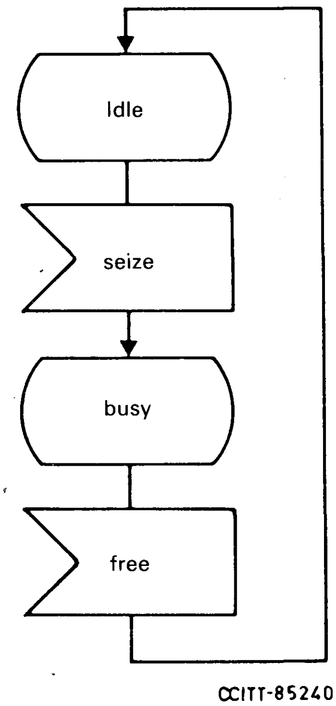


FIGURE 2/Z.100

A process definition

Signal instances may carry *data values*, which will be available to the receiving *process instance* when the *signal* is received. The *data values* may be stored in, and retrieved from, local *variables* of the *process*.

The static *structure* of an SDL *system* is described in terms of a *system*, *blocks* and *channels*, as shown in Figure 3/Z.100.

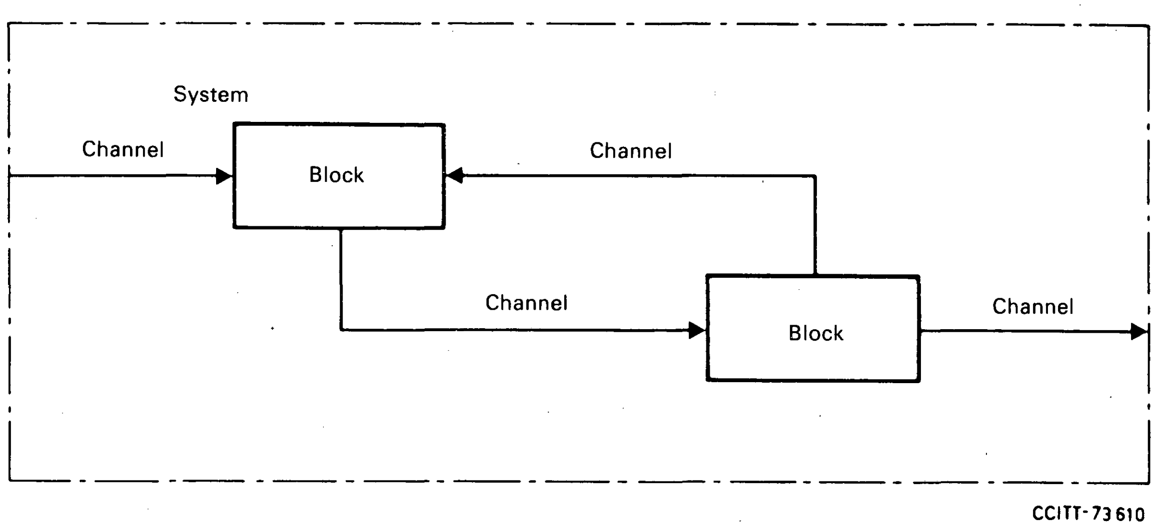


FIGURE 3/Z.100

Static structure of an SDL system

The *system* is a composition of *blocks*, connected to eachother and to the *system boundary* via *channels.* The *system boundary* separates the *system* from its *environment.* The *environment* is assumed to "behave in an SDL-like manner"; i.e. it will send *signals* to the *system* and accept responses in the form of *signals* from the *system.*

The *channels* act as the transport media for *signals* between *blocks* and between *blocks* and the *system boundary. Channels* are unidirectional. The *blocks* are containers for the *processes,* and serve to structure the *system* in building blocks.

2.3 *Data in SDL*

SDL processes may retain and manipulate *data values.* The *data values* are *bound* to *variables,* which are local to a *process.* However, *data values* may be sent between *processes,* and to and from the *environment,* by means of *signals.* Local *variables* for a *process* are defined in the *process definition.*

The data treated is *typed.* A number of *predefined data types* are defined in SDL. In addition the user may define new *data types.* The *predefined data types* are:

- *Boolean* having the *values* TRUE and FALSE and the normal logical operations on these *values.*
- *Integer* having its normal mathematical meaning.
- *Natural* having the normal mathematical meaning of natural numbers.
- *Real* having the normal mathematical meaning of real numbers.
- *Character* having the normal (as in programming languages) meaning and having the *values* and representation as in the CCITT alphabet number 5.
- *Array* an indexed collection of items which are all of the same *data type.* This is an extension of the normal meaning of array as used in programming languages.
- *Structure* a composition of a number of *data types* which may be different from one another.
- *String* a list of items of any *data type* and of arbitrary length.
- *Charstring* a list of characters of arbitrary length. Note a *charstring* is a *string* of characters formed from the *data type string* with the *data type* character.
- *Powerset* having the normal mathematical meaning such that a *powerset value* is an ordered set.
- *PId* used to identify *process instances.*
- *Time* absolute time.
- *Duration* the interval between two instants in absolute time.
- *Timer* the *data type* used for timers and which defines the operations SET and RESET for TIMER variables.

The STRUCT concept allows further data types containing composite values composed from a number of (possibly different) data types to be constructed.

The user-defined *data types* may be defined in terms of the *predefined data types* together with restrictions like *range* of an *integer,* or they may be additional *abstract data types. Abstract data types* are defined using an "axiomatic" method.

Data types may be generic for a *system,* or contained in a *block definition,* a *channel definition* or in a *process definition.* The *definition* is *visible,* and thus usable, within the scope of the object they are contained in. e.g. If a *type definition* is contained in a *block, data items* of that *type* may be used in all *processes* contained in the *block* and in the *sub-blocks* of the *block.*

2.4 *Structural concepts in SDL*

Structural concepts are provided in SDL in order to facilitate descriptions of complex and/or large *systems.* The concepts are so defined that they can support:

- the *partitioning* of large descriptions into modules so that parts can be dealt with and understood independently;
- the description of a *system,* or parts of a *system,* on several levels of abstraction;
- the description of the actual structure of a *system.*

When using these structural concepts it should be made clear if they represent the required or actual *structure*, or are just used to facilitate the representation.

In the basic SDL, a *system* is composed of *blocks*, *channels* and *processes*. These components may be further structured, i.e. *blocks* into *sub-blocks* and *sub-channels*, *channels* into *blocks* and *channels*, and processes into *sub-processes*. A *process definition* may also be further detailed in a number of steps by using *procedures* and *macros*.

When *partitioning blocks* into *sub-blocks* and *sub-channels* a hierarchical multi-level description of the *system* is obtained. The information contained in the upper *levels* should be contained in the interfaces of the lower *levels*. The obtained structure may be illustrated by a *block tree diagram* as shown in Figure 4/Z.100.

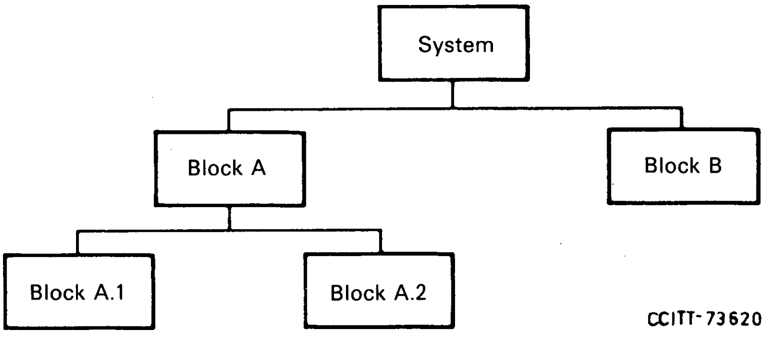


FIGURE 4/Z.100

A block tree diagram

It may also be described in more detail by a *block interaction diagram* as shown in Figure 5/Z.100.

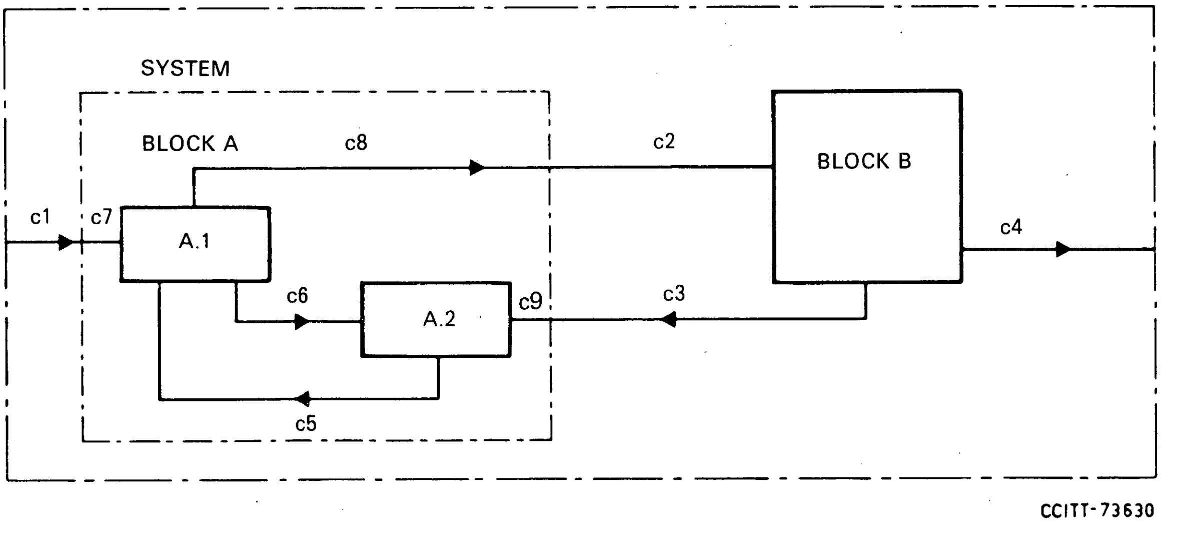


FIGURE 5/Z.100

A block interaction diagram

This latter diagram also shows the *channels* and *sub-channels* connecting the structure of *blocks* and *sub-blocks.* Using this *block partitioning,* it is only necessary, for the description to be meaningful, that the most detailed *sub-blocks* (the "leaf-blocks" in the *block tree diagram*) contain *processes.*

Partitioning of *channels* into *channels* and *blocks* will also result in a hierarchical structure. A simple example is given in Figure 6/Z.100.

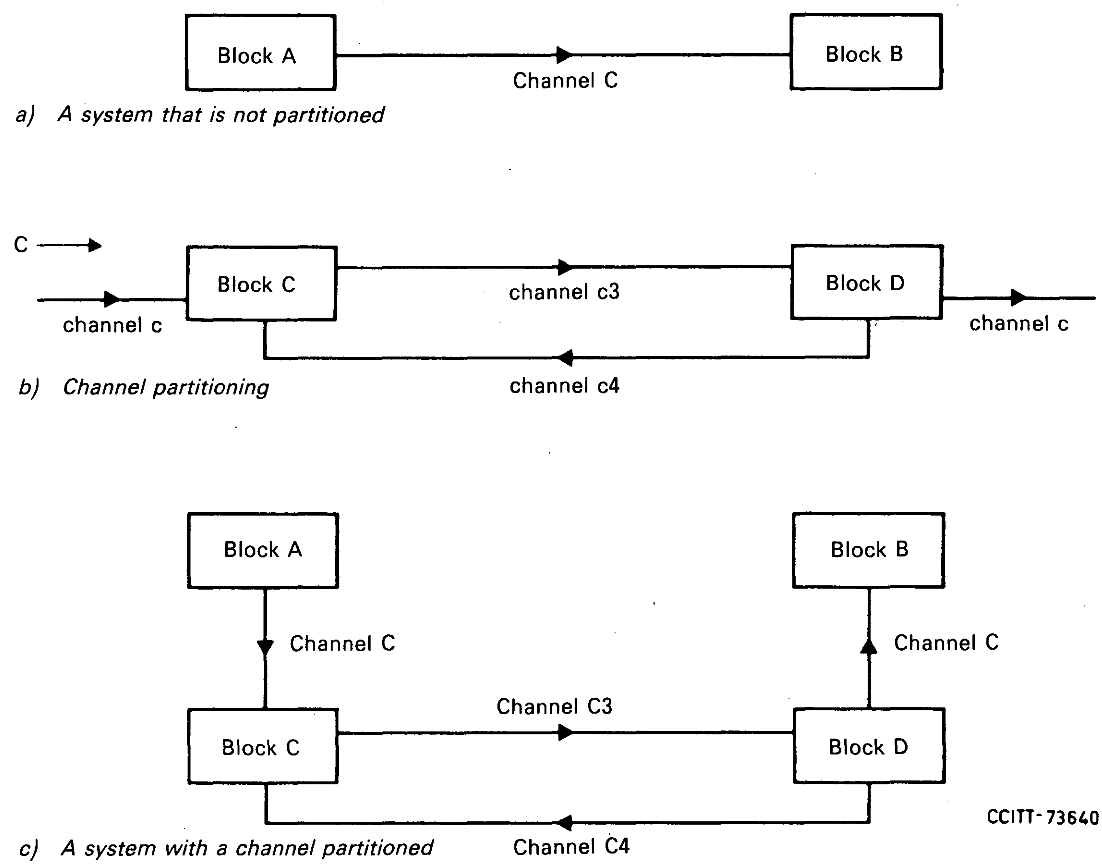


FIGURE 6/Z.100

Channel partitioning

The *partitioning* of a process into a set of *sub-processes* is related to *block partitioning,* as the *sub-processes* of a *process* must always be located in the *sub-block(s)* of the *block* containing the *process.* This is shown in Figure 7/Z.100.

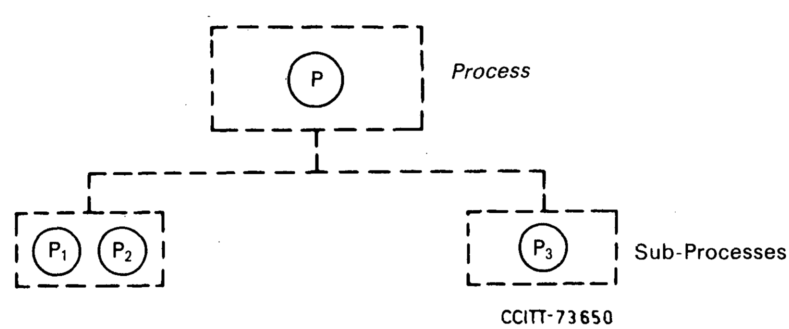


FIGURE 7/Z.100

Process partitioning

The *sub-processes* of a process represent together an alternative, more detailed, description of the behaviour described by the *process.* Thus when having a description as in Figure 7/Z.100, a choice has to be made if the detailed *behaviour,* represented by the *sub-processes,* or the less detailed one, represented by the *process,* should be interpreted. In SDL the alternative descriptions are said to support different *levels of abstraction.*

The *partitioning* of a *process* is illustrated by a *process tree diagram* as shown in Figure 8/Z.100.

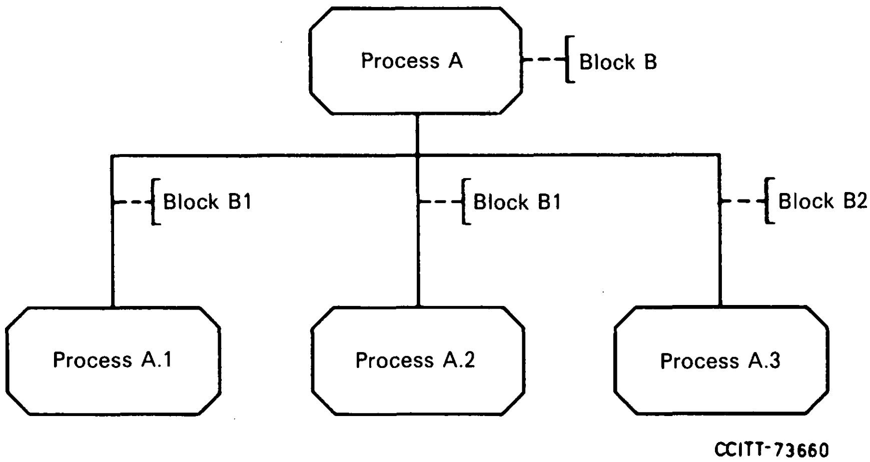


FIGURE 8/Z.100

A process tree diagram

A *process* may also be structured and detailed by the use of *procedures.* A *procedure* in SDL is similar to procedures in programming languages. It represents a parameterised and predefined *behaviour* which may be invoked by any *process* having access to its definition. The *procedures* may be predefined in libraries, or user-defined.

A *procedure* is defined as a set of *actions* and may include *states,* in a similar manner to a *process.* An example of the use of procedures is shown in Figure 9/Z.100.

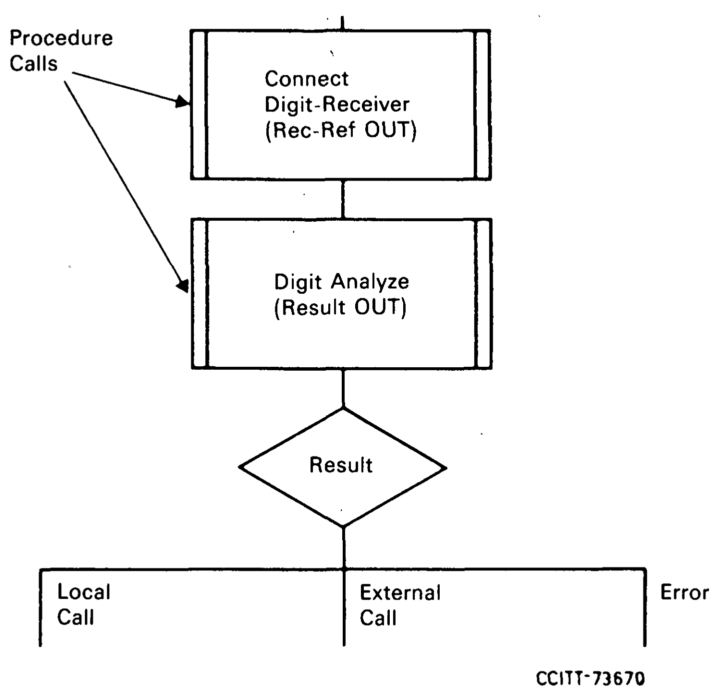


FIGURE 9/Z.100

Example of use of procedures in a transition

SDL representations may also be structured by the use of *macros.* A *macro* is a syntactical means to ease the drawing/writing of SDL representations, and to ease the understanding of them. A *macro* is a named collection of syntactical items, defined by the user. Whenever a reference to a *macro* appears in a description, it can be understood by replacing it with the items it is defined as, i.e. it has no semantics of its own. *Macros* may be used in any SDL representation, e.g. for *process* representations, for structural diagrams, etc. An example of the use of a *macro* is shown in Figure 10/Z.100.

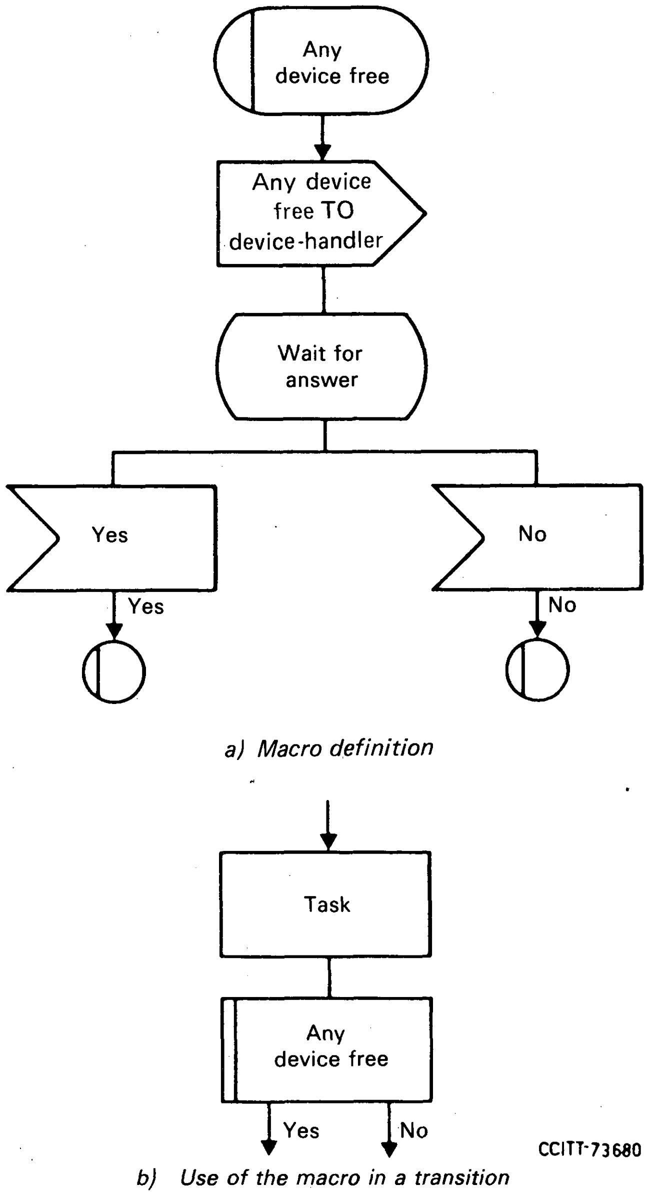


FIGURE 10/Z.100

Example of the use of a macro

All the structuring concepts in SDL may of course be used in combination with each other.

2.5 *Composite operations*

Composite operations in SDL are standard shorthand notations provided in the language to simplify the design of SDL *processes.* They are defined in terms of other SDL operations, and should be interpreted as if they were replaced by those operations.

Normally, the *composite operations* imply hidden *states* and *signal* interchange with other processes, which is why sometimes care should be taken of side effects.

The *composite operations* provided are:

Import/Export of *data values*

A shorthand notation for accessing *values* of *data items* local to the other processes via an implied *signal* interchange.

Enabling condition

A shorthand notation for the capacity either to accept a *signal* as an *input* or to *save* the *signal* depending on a *condition,* which may contain *imported values.* This is modelled as having several *states* in which the *signal* is accepted as *input* and other in which the *signal* is saved. The implied *state* is chosen after evaluation of the condition. The operation may also imply *signal* interchange.

Continuous signals

A shorthand notation for leaving a *state* and entering a *transition* when a condition, which may use *imported values,* becomes true. A *continuous signal* has lower *priority* than "normal" *signals,* and may be used to stimulate a transition when an external *data value* changes. The operation implies a number of *states* and *signal* interworking.

By using *composite operations,* the number of *states* and *transitions* in a process definition may be reduced.

2.6 *The concept of option in SDL*

When several similar applications are *specified* or *described* by using SDL often the same *process definition* may be applicable in several applications if only slightly modified. The *OPTION* concept makes it possible to have optional parts in a *process definition.*

Also, in a *specification,* it allows the representation of equally acceptable *behaviours* from the specifier's viewpoint. The actual *system* will implement one of these alternatives.

3 **Preliminaries to the language specification**

The language specification of SDL is contained in Recommendations Z.101, Z.102, Z.103 and Z.104. In the following preliminaries the methods and strategies used when defining the language are explained and guidelines on how to read the Recommendations are given.

3.1 *Strategy used in the language specification*

SDL gives a choice of two different syntactic forms to use when representing SDL descriptions; a Graphic Representation (SDL/GR), and a textual Phrase Representation (SDL/PR). As they both are concrete representations of the same SDL semantics, they are equivalent from a semantic point of view. In order to ensure that they are equivalent to each other, and thus transformable into each other, the definition of the semantics of SDL is strictly separated from the definitions of the different concrete syntaxes. The relations between the semantics definition of SDL and the concrete syntactical forms may be represented as shown in Figure 11/Z.100.

The semantics of SDL are defined by using an *abstract syntax,* with no concrete representation, associated with rules for well-formedness and interpretation. This definition is called the Common Language Model. Each of the concrete syntactical forms have then a definition of its own syntactical form and of its relationship to the *abstract syntax* (i.e. how to transform into the abstract form). Using this approach there is only one definition of the semantics of SDL; each of the concrete representation forms will inherit the semantics via its relations to the *abstract syntax.* The approach also ensures that the concrete syntactical forms are equivalent. As the transformations work both ways, either representation may be transformed into the other form via the *abstract syntax.*

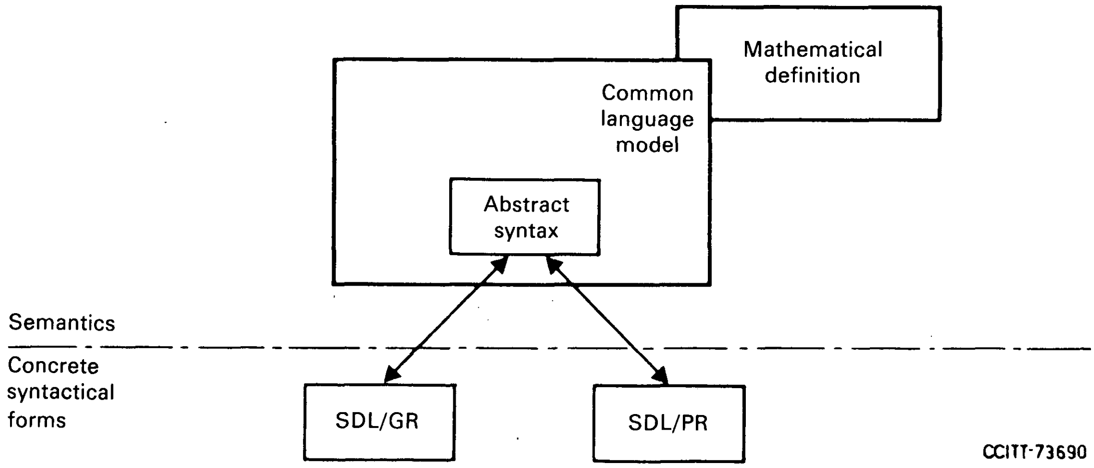


FIGURE 11/Z.100

Structure of SDL language specifications

The interpretation rules of the common language model are defined in an operational manner. i.e. The definition describes how instances of SDL concepts interpret their definition as an "abstract SDL machine". In addition, a mathematical definition of SDL in denotational semantics is also provided (but is not part of the Recommendations). The mathematical definition is closely related to the *abstract syntax* and structure of the common language model.

Following this strategy in Recommendations there will be, for each SDL concept, first a definition of the *abstract syntax* and its rules, then the rules of how to interpret the concept. Finally, the concrete syntactical forms to represent the concept will be given.

3.2 *Terminology*

In the Recommendations, for each SDL concept, the same term will be used consistently throughout. To distinguish occasions when a term refers to an SDL concept from those when a more general sense is meant all SDL concept terms will be in *italics.*

A glossary of all SDL terms is appended to the Recommendations as Annex A. The glossary contains a short explanation of each term, and a reference to where it is defined.

3.3 *Definition of the SDL/GR*

The SDL/GR is defined following the schema below:

– First the shape and content of the symbols are defined.
– Connectivity rules follow, when appropriate, i.e. what compositions of symbols are allowed.
– Lastly, the relations to the *abstract syntax* of the common language model are given.

3.4 *Definition of the SDL/PR*

The *SDL/PR* is defined using *syntax diagrams* and additional rules in natural language. The definitions follow the schema below:

– First the syntax is defined by diagrams and text.
– Then the relations to the *abstract syntax* of the common language model are given.

3.4.1 *Syntax Diagrams*

A *syntax diagram* consists of terminal and non-terminal symbols connected by flow lines.

A terminal symbol contains a character or a sequence of characters, and the generation rules are that when passed in a path those characters should appear in the *SDL/PR* text.

A non-terminal symbol is a reference to another *syntax diagram*, having the name appearing in the symbol. The generation rules are that when a non-terminal symbol appears in a diagram, the path leads into the referenced diagram, and the path will not leave the non-terminal symbol until the referenced diagram is left.

All terminal symbols, non-terminal symbols and *syntax diagrams* have exactly one flow line leading to them and exactly one flow line leading from them.

The graphic symbols used are shown in Figure 12/Z.100.

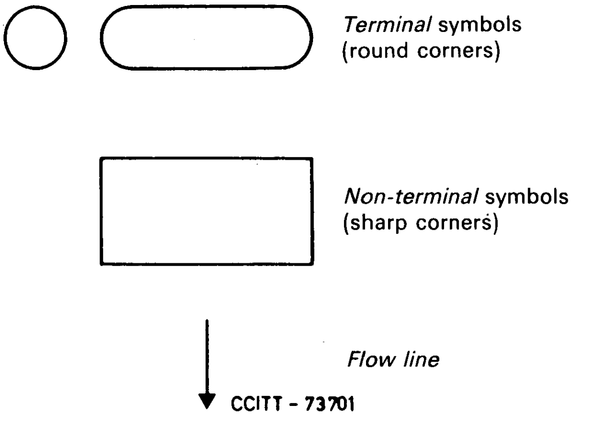


FIGURE 12/Z.100

Symbols in syntax diagrams

The SDL/PR syntax definition consists, on the "highest" level, of one syntax diagram, the SYSTEM. This diagram refers further, via non-terminals, to a set of other diagrams. Any path, starting with the flow line leading into this diagram and coming out of the diagram will on its path generate an *SDL/PR* text. The text will be syntactically correct if the SDL syntax rules have been followed.

3.5 *Common syntactic elements for SDL/GR and SDL/PR*

For certain syntactic elements, the same concrete syntax is used for both SDL/GR and SDL/PR.

3.6 *Structure of the SDL Recommendations*

Four Recommendations (Z.101 to Z.104) follow this Recommendation. The intention is to guide users to select a sub-set of SDL appropriate for their applications and methodology.

SDL may be applied in many fashions, supporting different purposes and methods. The minimum SDL sub-set which can be chosen is given in the Recommendation Z.101.

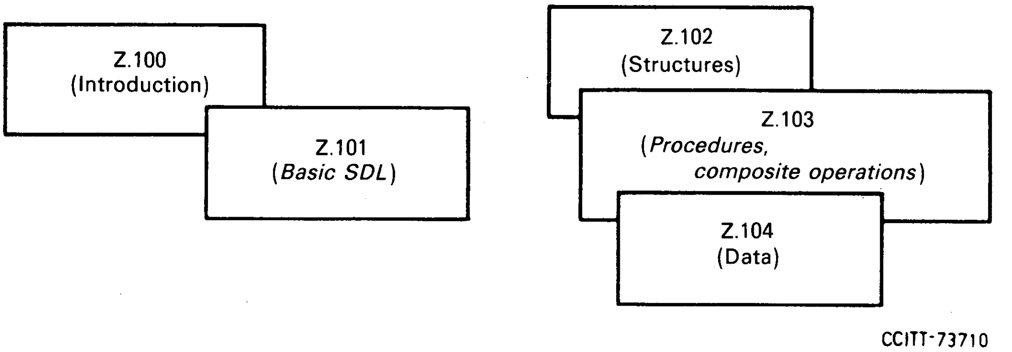


FIGURE 13/Z.100

Structure of SDL Recommendations

Recommendations Z.102 to Z.104 contain extensions to the minimum sub-set of SDL, and should be applied when appropriate. The extensions may be applied in any combination.

Examples of the use of the concepts defined in the Recommendations are contained in Annexes. More comprehensive examples of the use of SDL are also contained in the SDL User Guidelines.

The contents of the following Recommendations may be briefly summarized:

Z.101 Defines the basic concepts of SDL. The Recommendation forms the minimum sub-set of SDL to be applied. This sub-set of SDL is sufficient for describing the *behaviour* of *systems.*

Z.102 Defines additional structural concepts, used to describe large and/or complex *systems.* They can be used both for describing the actual *structure* of a *system* and to describe the *system* on several *levels of abstraction.*

Z.103 Defines the *procedure* concept, the *composite operations,* the *macro* concept, the *option* concept and the state orientated pictorial extensions *(SDL/PE).* These concepts are defined independently of each other, and may be applied in any combination.

Z.104 Certain data concepts which are considered predefined in Z.101 to Z.103 are defined in Z.104. It should be noted that *data* may be used completely informally in SDL.

Z.104 also contains a definition for the SDL *abstract data type* concept.

In addition to these Recommendations, a set of auxiliary documents is available, describing and explaining the language without having the status of being a Recommendation. Some of these documents are annexed to the Recommendations. The auxiliary documents are:

The formal definition of SDL

This document contains the mathematical definition of the semantics of SDL. The definition is expressed in denotational semantics (VDM, META IV). It will shortly be available as a separate document. [1]

The SDL glossary

This document contains all SDL terms. Each term has a short explanation and reference to where it is defined in the Recommendations. The document is Annex A to the Recommendations, and appears in Fascicle VI.11 of the Red Book.

The SDL abstract syntax summary

This document contains a summary of the complete *abstract syntax* for the language. The *abstract syntax* is described in a short BNF-like form. The document is Annex B to the Recommendations, and appears in Fascicle VI.11 of the Red Book.

The SDL concrete syntax summary

This document contains a summary of all the concrete syntaxes of SDL; i.e. the Graphical Representation *(SDL/GR),* the state orientated Pictorial Extension to the graphical form *(SDL/PE)* and the textual Phrase Representation *(SDL/PR).* The document is Annex C to the Recommendations, and appears in Fascicle VI.11 of the Red Book.

The SDL user guidelines

This document exemplifies and explains the use of SDL (without defining the language). It contains a number of examples and discussions on different usages of SDL. Some of the concepts defined in the Recommendations which should be used with special care are also discussed in the User Guidelines. The document is Annex D to the Recommendations and appears in Fascicle VI.11 of the Red Book.

The SDL course

This document is intended as training in the use of SDL. It is available as a separate document. [2]

Finally, in the inside back cover of this fascicle, two templates are enclosed for use in drawing the graphical forms of SDL. They contain all the recommended graphical symbols, in their recommended format.

---

[1] The formal definition of SDL will be obtainable from the International Telecommunication Union, General Secretariat – Sales Section, Place des Nations, CH-1211 Genève 20 (Switzerland).

[2] The SDL course will be available through the International Sharing System for Training, ITU Secretary General – Technical Cooperation Department, Training Division, Place des Nations, CH-1211 Genève 20 (Switzerland).

BASIC SDL

1 Introduction

This Recommendation defines the basic CCITT Specification and Description Language (SDL). The basis for SDL is the concept of communicating finite state machines called *processes.* An SDL *system* is a set of *blocks. Blocks* are connected to each other and to the *environment* by *channels.* Within each *block* there are one or more *processes.* These *processes* communicate with one another by *signals* and are assumed to execute concurrently.

In defining the SDL, it was found useful to first define a common SDL model. This common model forms an abstract basis for the *concrete syntaxes* and ties them together. The *concrete syntaxes* are just alternate means of representing the concepts of SDL. Currently, there are two *concrete syntaxes*: SDL/GR and SDL/PR. Since these two syntaxes represent the same SDL concepts, it is possible to map a *system* defined using one form of SDL *concrete syntax* to the other *concrete syntax.*

In this Recommendation, the language is defined by first defining the common language model and then defining the SDL/GR and SDL/PR *concrete syntaxes.*

2 Common language model

2.1 *An introduction to the common model*

In SDL, a *system* is a set of *blocks* connected to each other, and to the *environment* of the *system,* by means of *channels* (see Figure 1/Z.101). The *channels* are unidirectional.

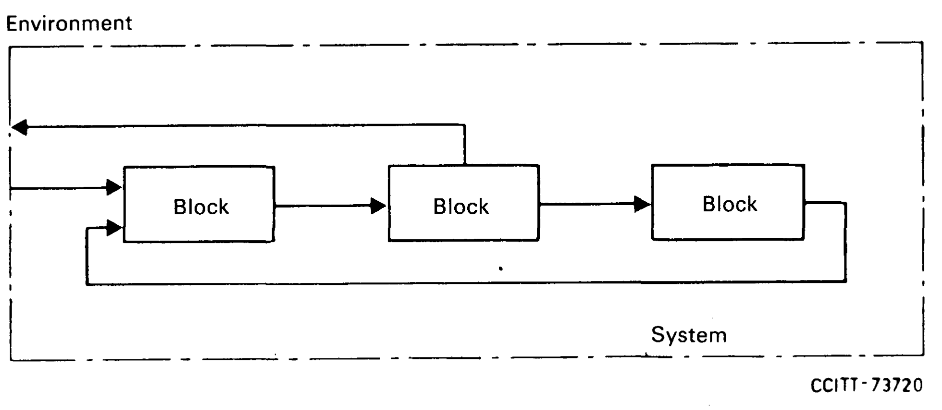


FIGURE 1/Z.101

SDL Model

The behaviour of each *block* is modelled by one or more *processes.* A *process* is defined by a *process definition.*

Processes interact with other *processes* or the *environment* by means of *signals.* A *signal* is a flow of *data conveying* information between *processes.* When a *process* outputs a *signal,* the *signal* will be transported to the *process* it is directed to. The transportation mechanism for *signals conveyed* between *processes* in the same *block* is the same as for *signals conveyed* between *processes* in different *blocks. Channels* represent the transportation route for *signals* exchanged between *blocks.*

When a *signal* arrives at the *process* it is directed to, it will be *retained* outside the *process* until the *process* is ready to receive the *input signal.* A *signal* will be consumed when the destination *process* receives the *signal.*

SDL models open *systems*, this means that the *system* may interact with its *environment.* This interaction takes place solely by means of *conveying signals* via the *channels* leading to and from the *environment.* The *environment* is assumed to act in an SDL-like fashion, i.e., the *environment* can be considered to contain a *process* which outputs signals to the *channels* leading into the *system* and receives *signals* from the *channels* leading out of the *system.*

During its lifetime, a *process* is either in a *state* (waiting to receive one of a set of *signals*) or in a *transition* (performing a sequence of actions). When in a *state* only a specified set of *signals* can be received by a *process.* If one of these *input signals* is *retained* outside the *process*, it is received by the *process.* The receiving of the *input* makes the *data* carried by the *signal* accessible for the *process* and *starts* a *transition.* During a *transition*, the *data* of the *process* may be manipulated and *signals* may be output. The *transition* will end with the *process* entering a new *state* or with a *stop.*

A *stop* causes the *process* to cease to exist.

Common to the *system* and its *environment* is the concept of absolute time which is the same throughout the *system* and the *environment.*

2.2 *Abstract syntax*

System definition

A *system definition* contains a *system name*, one or more *block definitions*, a set of *channel definitions* and a set of *signal definitions.*

A *system definition* contains the *signal definition* for each *signal name* contained in the *signal list* associated with each *channel definition.*

Block definition

A *block definition* contains a *block name*, one or more *process definitions* and may contain *signal definitions.*

Each *block definition* contains the *signal definition* for each type of *signal* interchanged between *processes* within the *block.*

Channel definition

A *channel definition* contains a *channel name*, an origin *block definition identifier*, a destination *block definition identifier* and a *signal list.* The *signal list* contains the *identifier* of each *type* of *signal* that may be *conveyed* through the *channel.*

The origin *block definition identifier* and the destination *block definition identifier* associated with a *channel definition* must be different and each must be the *identifier* of a *block definition* in the *system definition* or must be the *environment.*

The *signal list* associated with the *channel definition* contains at least one *signal identifier.*

Signal definition

A *signal definition* contains a signal *name* and may contain a list of *data type names.*

Process definition

A *process definition* contains a *process name*, a pair of *integers* and a *process graph* and may contain a *formal parameter list*, *variable definitions* and *viewing definitions.*

Process graph

A *process graph* is a graph whose *nodes* are connected by directed *arcs.* An *arc* entering a *node* is called an incoming *arc* and an *arc* exiting the *node* is called an outgoing *arc.* A *node* having an *arc* as an incoming *arc* follows the *node* having the same *arc* as an outgoing *arc.*

The following categories of *node* exist:

State node
Input node
Task node
Output node
Decision node
Start node
Stop node
Create request node

The following rules define the connectivity of a *process graph*:

1) Each *process graph* contains one and only one *start node*. The *start node* is followed by a *transition string*. The *start node* does not follow any other *node*.

2) A *transition string* can be one of the following:

   a) null followed by either a *state node* or a *stop node*;
   b) an *action* string followed by a *transition string*;
   c) a *decision node*.

3) An *action* string can be one of the following:

   a) a *task node*,
   b) an *output node*,
   c) a *create request node*.

4) A *decision node* is followed by two or more *decision arcs*.

5) A *decision arc* is a named *arc* followed by a *transition string*.

6) A *state node* is followed by one or more *input nodes*.

7) An *input node* follows one and only one *state node*.

8) An *input node* is followed by a *transition string*.

9) The *stop node* has no *nodes* following it.

10) Each *process graph* has at most one *stop node*.

11) Each *node* is reachable from the *start node*.

State node

A *state node* contains a *state name* and may contain a *save-signal-set*. *State nodes* within a *process graph* have different names.

Input node

An *input node* contains a *signal identifier* and may contain an ordered set of *variable identifiers*.

Save-signal-set

A *save-signal-set* contains *signal identifiers*.

Task node

A *task node* contains either a sequence of *statements* or *informal text*.

Statement

A *statement* is either a *set statement*, a *reset statement* or an *assignment statement*.

Set statement

A *set statement* contains a *time expression* and a *timer identifier*.

Reset statement

A *reset statement* contains a *timer identifier*.

Assignment statement

An *assignment statement* contains a *variable identifier*, an *assignment operator* and an *expression*.

Output node

An *output node* contains a *signal identifier*, a destination *expression* and may contain an *actual parameter list*.

Decision node

A *decision node* contains a *question* and has at least two outgoing *arcs*. Each outgoing *arc* has associated with it a set of one or more *answers* to the *question*. Every possible *answer* to the *question* should be associated with one and only one *arc*. A *question* is either an *expression* or *informal text*. An *answer* is either a *value identifier* or *informal text*.

Create request node

A *create request node* contains a *process definition identifier* and may contain an *actual parameter list.*

Data type

Predefined *data types* exist for *types*: *natural, integer, real, character, charstring, Boolean, time, duration, timer* and *process instance identifier.* They are defined in Recommendation Z.104.

Variable definition

A *variable definition* contains a *variable name* and a *data type identifier* and may contain a *reveal attribute.*

Formal parameter list

A *formal parameter list* is an ordered set of *formal parameters.*

Formal parameter

A *formal parameter* contains a *formal parameter name* and a *type identifier.*

Actual parameter list

An *actual parameter* list is an ordered set of *actual parameters.*

Actual parameter

An *actual parameter* is an *expression.*

View definition

A *view definition* contains a *variable identifier,* a *data type identifier* and a *process definition identifier.*

The *variable* must have a *reveal attribute* in the *process definition* referred to by the *process definition identifier.* The revealing *process definition* referred to must belong to the same *block* as the viewing *process definition.*

Expression

An *expression* is either a *value identifier* or a *variable identifier* or an *operation.*

Operation

An *operation* contains either a *viewing expression* or an *operator* and a list of one or more *expressions.*

Viewing expression

A *viewing expression* is composed of a *viewing operator* plus a *variable name* and *process instance identifier.*

2.3 *Interpretation rules*

2.3.1 *System*

A *system* is a concrete entity, such as a telephone exchange, and is an *instantiation* of a *system type* defined by a *system definition.* A *system* is separated from its *environment* by a *system boundary* and contains a set of *blocks.* Communication between the *system* and its *environment* or between *blocks* within the *system* can only take place using *signals.* Within a *system,* these *signals* are conveyed on *channels.* The *channels* connect *blocks* to one another or to the *system boundary.*

The *system* possess a parameterless function of *type time* called NOW, yielding the current *time.* The current *time* is immediately available throughout the *system* and its *environment.* NOW can be used in *expressions* in any *process* in the *system.*

2.3.2 *Channel*

Within the *system,* there is a *channel* for each *channel definition* in the *system definition.* A *channel* is a transportation route for *signals.* The route is unidirectional. The end points of the *channel* are either a *block* or the *system boundary.* At least one of the end points of the *channel* must be at a *block.* If both end points are at *blocks,* the *blocks* must be different. The *channel definition* contains the list of all *signals* that may be *conveyed* on the *channel.*

When a *signal* is *output* to the *channel*, the *signal* is conveyed to the destination *block*. The order of the *signals* output to the *channel* and the order of the *signals* received from the *channel* is the same. If two or more *signals* arrive at a *channel* simultaneously, they are arbitrarily ordered.

2.3.3 *Block*

Within the *system*, there is a *block* for each *block definition* in the *system definition*. A *block* is an object of manageable size in which one or more *processes* can be interpreted. There are 2 communication mechanisms between *processes* within the same *block*: *signals* and *shared values*.

When a *signal* arrives at the *block* from a *channel*, the *block* delivers the *signal* to the *input port* of the *process* addressed by the *process identifier* in the *signal*.

When a *signal* is output by a *process* within the *block*, it is delivered to the *process* addressed by the *process identifier* in the *signal*. If the addressed *process* is within the same *block*, the *block* delivers the *signal* to its *input port*. If the addressed *process* is in another *block*, the *block* delivers the *signal* to the *channel* able to convey that *signal*.

Shared values allow a *process* to *view* a revealed *variable* in another *process*. Only the revealing *process* is allowed to change the *value* of the *variable*. The viewing *process* receives the current *value* of the revealed *variable* by using a *viewing operator*.

2.3.4 *Signal*

A *signal* is a flow of *data* conveying information between *processes* and is an instantiation of a *signal type* defined by a *signal definition*. A *signal* can be sent by either the *system environment* or a *process* and is always directed to either a *process* or the *environment*.

Each *signal* contains the *signal identifier* in the *signal* definition, an *origin process instance identifier* and a *destination process instance identifier*. In addition, other *values* may be conveyed by *variables* in a *signal*. In a *signal*, there is one *variable* for each *name* in the *data type* list in the *signal definition*.

2.3.5 *Process*

A *process* is a communicating finite state machine and is an instantiation of a *process type* defined by a *process definition*. Within a *block* there may be zero or more *processes* for each *process definition*. *Processes* can exist from the time that a *system* is created or can be created by *create request actions* and may cease to exist by performing *stop actions*. A *process* executes independently from and concurrently with other *processes* in the *system*.

All *processes* in the *system* possess four predefined *variables* of *process identifier type* called: SELF, PARENT, OFFSPRING and SENDER. These *variables* are *process instance identifiers* for:

- the *process* (SELF);
- the creating *process* (PARENT);
- the most recently created *process* (OFFSPRING);
- the *process* from which the last *input signal* has been received (SENDER).

These variables can be used in *expressions* but cannot explicitly be assigned a *value*. For all *processes* present at *system initialization*, PARENT is given the same distinct *value*, which is different from the value of SELF for any *process*.

Signals to the *process* are *input signals* and *signals* from the *process* are *output signals*. An *input signal* is an entity intended to invoke the *process* and to communicate information to it. An *output signal* is intended to invoke another *process* and to communicate information to it.

The set of *signal identifiers* that appear attached to *input nodes* of the *process graph* denotes the set of *valid input signal identifiers* for this *process definition*. For each *state*, all *input signal identifiers* appear in either a *save-signal-set* or an input *node*.

The pair of *integers* contained in the *process definition* define the number of *instances* of the *process* which are created when the *system* is created and the maximum number of simultaneous *instances* of the same *process type*. When a *system* is created, the initial *processes* are created in a random order and no *actual parameters* are passed to the *process*.

When the *process* is created, it is given an empty *input port*, and *variables* are created with *undefined values*. Then the *process starts* by interpreting the *start node* in the *process graph.*

When a *valid input signal* arrives at the *process*, it is put into the *input port* of the *process*. Each *process* contains a single *input port*. The *input port* may *retain* any number of *input signals* so that several *input signals* are queued for the *process*. The set of *retained signals* are ordered in the queue according to their arrival time. If two or more *signals* arrive simultaneously, they are arbitrarily ordered. Attached to the *input port* is a possibly empty set of *timers*. Each *timer* contains a *value* of the *time type.*

The *process* is either waiting in a *state* or active performing a *transition*. For each *state*, there is a *save-signal-set*. When waiting in a *state*, the first *input signal* whose *identifier* is not in the *save-signal-set* is taken from the queue and received by the *process*. The *input port* will continuously compare NOW with the timer, if any exists, having the lowest *value* greater than zero. When the *value* of NOW is greater than or equal to the *value* of this timer, a *signal* with the same *name* as this timer is placed in the queue and then the timer is given the value of zero.

When a *process* is waiting, it is always in a unique *state* denoted by the corresponding *state node* in the *process graph*. When an *input signal* is received, the *process* becomes active, interprets the *input node* having the same *name* as the *input signal* and performs a *transition* leading to a new *state.*

2.3.6 *Process graph*

A *start node* is interpreted as a *start action*. The *start action* causes the *node* following the *start node* to be interpreted.

An *input node* is interpreted as an *input action* which receives and consumes the given *signal* and then the *node* following the *input node* is interpreted. The consumption of the *signal* makes the information conveyed by the *signal* available to the *process*. The *variables* in the *input node* are assigned the *values* of the corresponding *variables* in the *signal*. If there is no *variable* in the *input node* for a *variable* in the *signal*, the *value* of the *variable* is discarded. SENDER in the receiving *process* is given the *value* of the origin *process instance identifier* carried by the *signal.*

A *task node* is interpreted as a *task action*. The task *action* is the interpretation of a sequence of *statements* or *informal text*. When the *action* is complete, the *node* following the *task node* is interpreted.

A *set statement* will cause a *reset statement* on the *timer* given in the *statement* and will assign the given *time value* to the *timer.*

A *reset statement* will set the time *value* of the timer to zero. If any signals having the same *name* as the timer are present in the input queue, they will be removed from the input queue and will be discarded. All *signals* in the queue with the same *identifier* as the timer *signal identifier* are removed from the queue and discarded.

An *assignment statement* is informally interpreted as the *value* of the *variable* in the *assignment statement* takes on the *value* of the *expression* in the *assignment statement.*

An *output node* is interpreted as an *output action* which creates a *signal* and delivers it to the *block*. Then the *node* following the *output node* is interpreted. The *output signal* is an *instantiation* of a *signal type* defined by the *signal definition* indicated by the *signal identifier* in the *output node*. The *variables* in the *signal* are assigned the *values* of the *actual parameters* in the *output node*. If there is no *actual parameter* in the *output node* for a *variable* in the signal, the *variable* has *undefined value*. The origin *process instance identifier* carried by the *signal* is assigned the value of the SELF *variable*. The destination *process instance identifier* of the *signal* is assigned the *value* of the destination *expression* contained in the *output node.*

A *decision node* is interpreted as a *decision action* which answers a *question*. The *arc* which matches the *answer* to the *question* is chosen and the *node* following this *arc* is interpreted.

A *state node* is interpreted as the termination of a *transition* by giving the *process* a new *state* as denoted by the *name* contained in the *node*. The *input port* is informed that the *process* is now waiting in a *state* and is also presented the *save-signal-set* attached to the interpreted *state node*. After this, the *process* waits until given a new *input signal.*

A *stop node* is interpreted as the immediate termination of the *process*. This means that the *input signals* retained in the *input port* are discarded and that the *variables* created for the *process*, the *input port* and the *process* will cease to exist.

A *create request node* is interpreted as a *create request action*. The *create request action* causes the creation of a *process* in the same *block*. The *definition* of the *process* is in the same *block* and is identified by the *process identifier* in the *create request node*. As part of the *create request action*, the created *process*'s PARENT *variable* is given the *value* of the creating *process*'s SELF *variable*. The created *process*'s SELF *variable* and the creating *process*'s OFFSPRING *variable* are both assigned the same unique *process instance identifier value*. The *formal parameters* in the newly created *process* are assigned the *values* of the *actual parameters* contained in the *create request node*.

2.3.7 *Data type*

The predefined data types are used in the normal sense.

2.3.8 *Variable*

A *variable* can be assigned a *value*. The contents of the *value* assigned to a *variable* can be later retrieved from the *variable*. A *variable* also has a *data type* which restricts the class of *values* which can be assigned to the *variable*.

Within a *process*, there is a *variable* for each *variable definition* in the *process definition*. The *value* of a *variable* can only be modified by the *process* that defines the *variable*. The *value* of a *variable* is known only to the *process* that defines the *variable* unless the *variable* has the *reveal attribute*. The *reveal attribute* allows other *processes* in the *block* to *view* a *variable*. *Variables* possess the same lifetime as the declaring *process* (i.e. they are created when the declaring *process* is created and they cease to exist when the declaring *process* ceases to exist).

Within a *signal*, there is an anonymous *variable* for each occurrence of each *data type name* in the *signal definition*. The *value* of the *variable* can only be assigned in the *output node* that created the *signal* and can only be known in the *input node* that receives and consumes the signal. These *variables* have a lifetime bounded by the lifetime of the *signal*.

2.3.9 *Expression*

An *expression* is informally interpreted as producing a *value*.

2.3.10 *View definition*

A *view definition* defines a *variable name* which may only be used in a *viewing expression* to obtain the *value* of a *variable* owned by another *process*.

2.3.11 *Viewing expression*

The *value* of a *viewing expression* is the *value* of the *variable* identified by the *variable name* and *process instance identifier* in the *viewing expression*.

3 SDL/GR

A *system definition* is represented in SDL/GR syntax by:

– a *block interaction diagram* which contains the *system name* and the *channel definitions* and which identifies the *block definitions*. The *block interaction diagram* also identifies the *process definitions* which model each *block's* behaviour. The *block interaction diagram* may denote: 1) the lists of *signals* that pass between *processes* in the same *block*; 2) the lists of *signals* conveyed by *channels* between *blocks*, and 3) the creation of new *process instances* by other *process instances*;

– *process diagrams* which define the behaviour of each *process* and are the graphic representations of the *process definitions*. *Process diagrams* may contain *variable definitions*, *formal parameters* and *view definitions*;

- *signal lists* which name the *signals* conveyed by a *channel* or from one *process* to another inside the *block*. These may be incorporated in the *block interaction diagram*, using *signal list symbols*, or presented as separate lists in whatever form is felt suitable;
- *signal definitions* which give the *data types* and order of the *data values* which can be contained in a *signal*, for each *signal* named in the *signal lists*. These are specified using SDL/PR syntax.

In a *system definition*, *names* and *identifiers* may be used. *Names* are specified using SDL/PR syntax. *Identifiers* consist of a *name* together with *qualifiers*. A *qualifier* is the entity *type* with which the *name* is associated, and it represents the hierarchy *level* of the entity which is being identified. No two or more entities may have the same *identifier*. In SDL/GR, the qualifiers of *names* may be inferred from the context, but whenever they are used, they are specified using SDL/PR syntax.

3.1 *Block interaction diagrams*

The *block interaction diagram* for a *system* contains a *system name*, a set of *block symbols*, *environment symbols*, and a set of *channel symbols* and *signal list symbols*.

3.1.1 *Symbols*

The recommended *symbols* appear in Figure 2/Z.101 below.

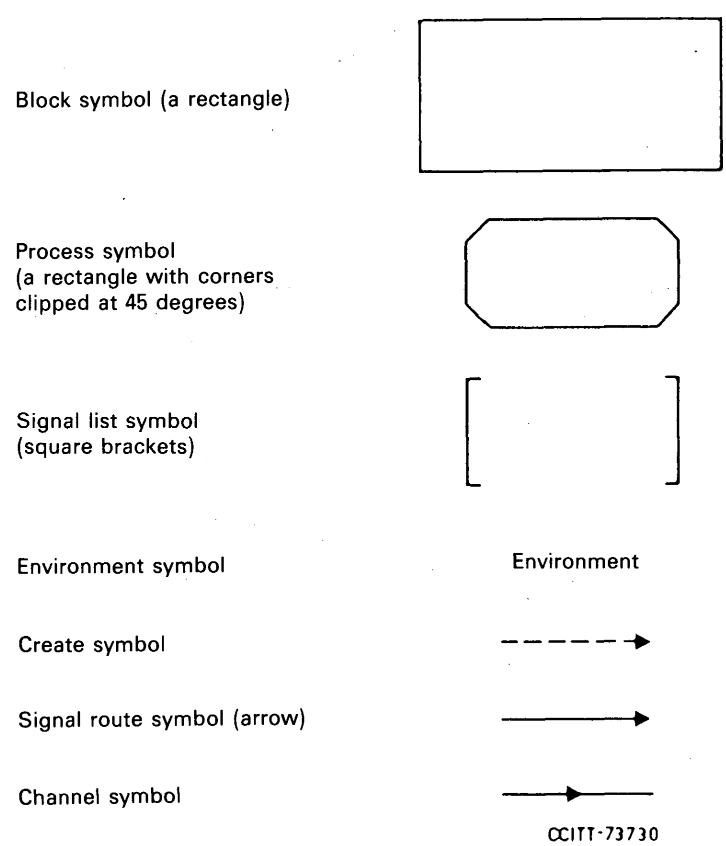


FIGURE 2/Z.101

Block interaction diagram symbols

3.1.2 *Relationship between SDL/GR block interaction diagrams and the abstract syntax and the use of symbols*

- The *environment symbol* represents the *system environment* and may appear a number of times.

- A *block symbol* contains a *name*, a nonempty set of *process symbols*, *signal route symbols* and may contain *create symbols* and *signal list symbols.*

- A *process symbol* contains a *process name* and may contain a *formal parameter list symbol.* The *process name* is the same as the *name* contained in the *process definition* which describes the behaviour of the *process.* The *formal parameter list symbol* contains a list of the *names* of the *formal parameters* which are initialized either when the *system* is created or when the *process instance* is dynamically created.

- A pair of *integer values* may be associated with a *process symbol*, the first *value* represents the number of *instances* of the *process* which exist when the *system* is created, the second *value* represents the maximum number of simultaneous *instances* of this *process type.* The two *values* are positioned in the top right-hand corner of the *process symbol.*

  The default for the first *value* is one. The default for the second *value* is infinity. This pair of *integers* is specified using the SDL/PR syntax. When one of the *integers* is not specified, the default *value* is chosen for it.

- A *channel symbol* has a *channel name* attached to it. The *channel symbol* has an origin end which is connected to a *block symbol* and a destination end which is connected to another *block symbol.* Alternatively, either the origin or the destination connection (but not both) may be connected to an *environment symbol* instead of a *block symbol.* A *signal list symbol* may be placed beside a *channel symbol* to identify the *signals* carried by the *channel.*

- A *signal route symbol* in a *block* is associated with a *signal list symbol.* A *signal route symbol* leads either from one *process* to another, or from a *process* to the origin end of a *channel* (at the *block* boundary), or from the destination end of a *channel* (at the *block* boundary) to a *process.*

- A *signal list* in a *block interaction diagram* is a list of *names*, the whole enclosed in the *signal list symbol* (i.e., in square brackets). The *signal list* may itself have a *name*, which is written above the *symbol.* Entries in the list (which are separated by commas and can be put in columns or rows) are the *names* of individual *signal definitions* and the *names* of other *signal lists.* Within the list, *signal list names* are distinguished from individual *signal definition names* by enclosing each *signal list name* in a further pair of square brackets. If it is decided to include *signal lists* on the *block interaction diagram* using *signal list symbols*, all *signal lists* should be included in the diagram. Figure 3/Z.101 shows examples of *signal lists.*

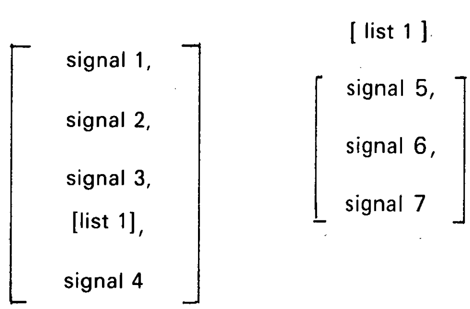


FIGURE 3/Z.101

Examples of signal lists in block interaction diagrams

– *Create symbols* lead from one *process symbol* to another *process symbol.* The former *process symbol* represents the "creating" *process.* The latter *process symbol* represents the "created" *process.*

3.1.3 *Drawing rules*

– *Channel symbols* are connected to the boundaries of *block symbols* with which they are associated. *Channel symbols* should join *block symbol* boundaries at 90°. If necessary, *channel symbols* may contain 90° bends. A *channel* symbol includes an arrowhead to show the direction of the flow of *signals* along the *channel.*

– *Signal route symbols* are connected to the boundaries of *block symbols* or *process symbols* with which they are associated. *Signal route symbols* join these boundaries preferably at 90°. If necessary, *signal route symbols* may contain 90° bends. Several *signal route symbols* may converge at the origin end of a *channel* (at the *block* boundary). Several *signal route symbols* may diverge from the destination end of a *channel* at the *block* boundary. A *signal route symbol* includes an arrowhead at one end to show the direction of the flow of *signals.*

– *Create symbols* are connected to *process symbol* boundaries by dotted lines which meet the *process symbol* boundaries at 90°. Arrowheads are used on these connecting lines so that the "creating" *process symbol* points to the "created" *process symbol.*

– The preferred orientation of *symbols* is shown in Figure 2/Z.101.

– The size of *symbols* may be chosen by the user.

– *Symbol* boundaries must not overlay or cross. An exception to this rule applies for the *channel symbols* and *signal route symbols* which may cross each other. There is no logical relationship between *channel symbols* or *signal route symbols* which do cross.

3.2 *Signal lists*

The union of all *signal names* in the *signal lists* associated with *signal route symbols* connected to a given *process definition* equals the set of *valid input signal names* for that *process.*

The union of all *signal names* in the *signal lists* associated with *signal route symbols* connected to the origin end of a *channel* equals the list of *names* in the *signal route list* associated with that *channel,* and equals the union of all *signal names* in the *signal route lists* associated with *signal symbols* (in the *channel* destination *block*) connected to the destination end of that *channel.*

3.3 *Process diagrams*

3.3.1 *Symbols*

The behaviour of a *process* is represented in graphical form by a *process diagram.* The *name* of the *process diagram* is the same as the *process name* in the *process definition* which it represents. Figure 4/Z.101 shows the *symbols* that are used in the *process diagrams* of SDL/GR form.

3.3.2 *Relationship between SDL/GR process diagrams and the SDL abstract syntax and the use of symbols*

Each *process diagram symbol* listed in Figure 4/Z.101 represents the equivalently *named node* of the *process graph* in the *abstract syntax.* Flow lines, which connect *symbols,* represent the directed *arcs* which connect *nodes.* Allowable connections of *symbols* by flow lines in an SDL-GR *process diagram* are shown in Figure 5/Z.101.

The *formal parameters, valid input signal* set, *variable definitions, view definitions, expressions* and *viewing expressions* are specified using SDL/PR syntax.

A *start symbol* represents a *start node* (see also § 2.2.3.3). The *start symbol* contains the *name* of the *process* it describes.

A *stop symbol* represents the *stop node.*

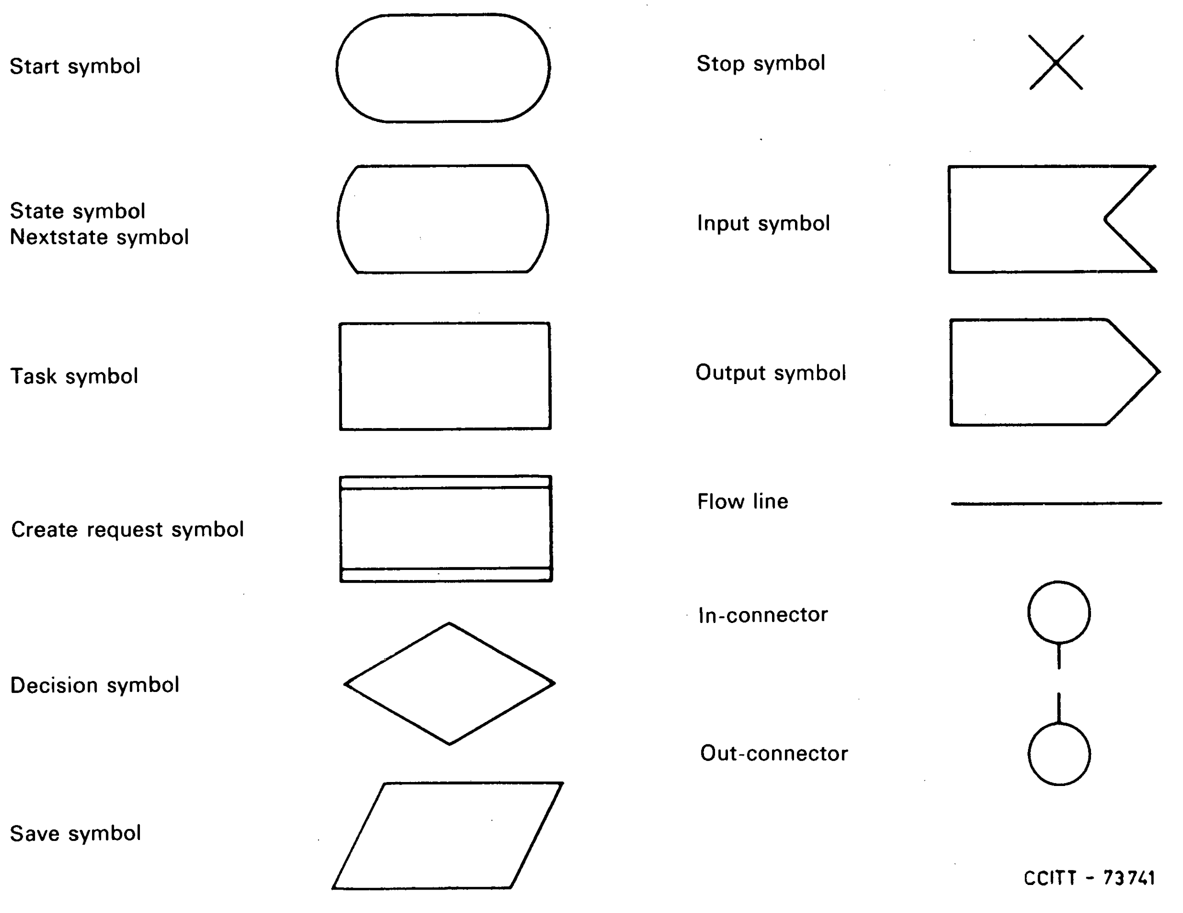


FIGURE 4/Z.101

SDL/GR process diagram symbols

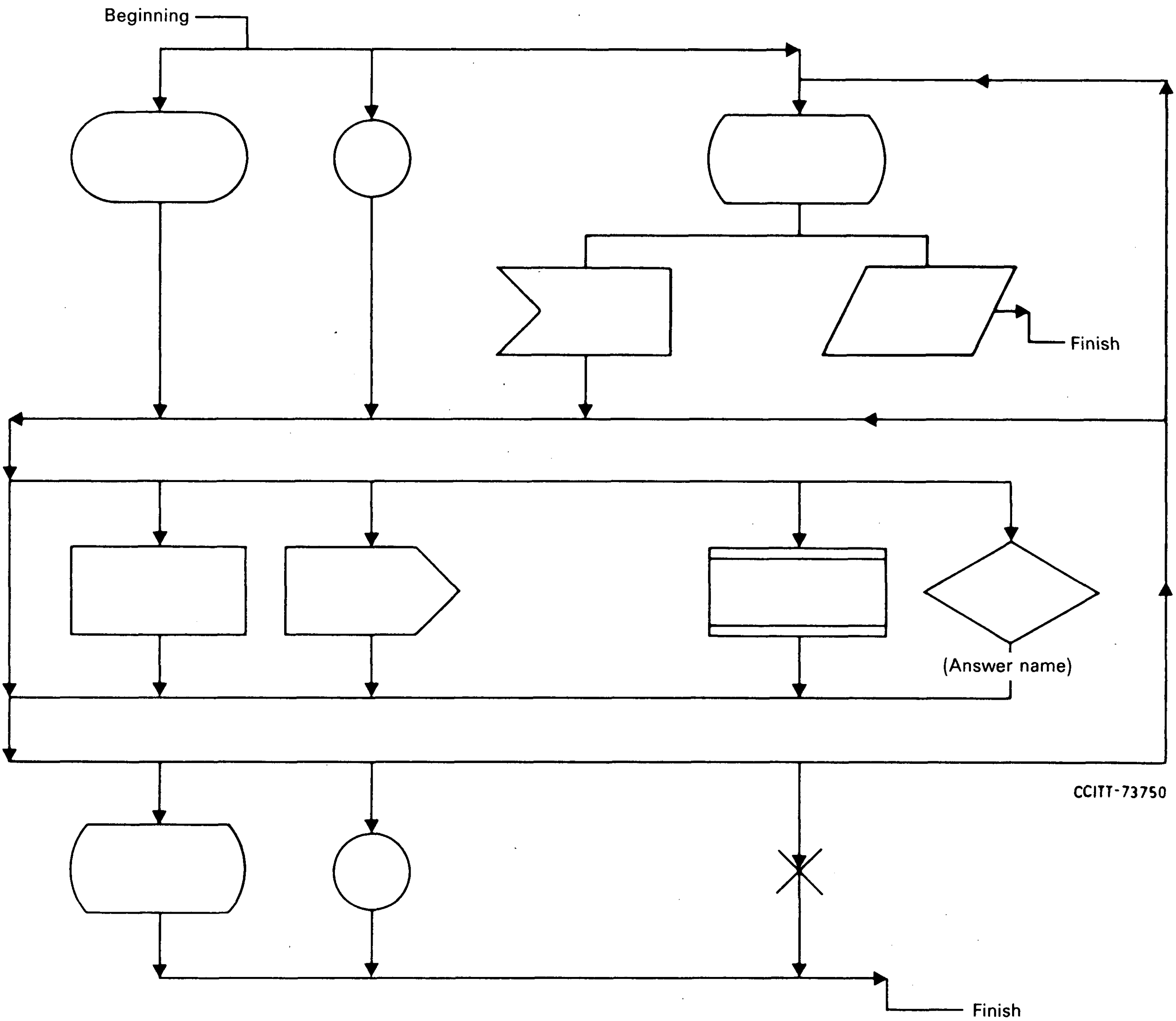


FIGURE 5/Z.101

Allowable connections of symbols by flowlines in an SDL/GR process diagram

A *state symbol* represents one or more *state nodes* and contains one or more *state names* separated by commas, or an asterisk, or an asterisk followed by a list of *state names* within brackets.

A *save symbol* represents the set of *saved signals* attached to a *state node.* It contains one or more *signal names*, separated by commas, or an asterisk.

An *input symbol* represents one or more *input nodes. Signal names* which are contained in the *input symbol* are separated by commas. Each of these *signal names* gives the *name* of one of the *input nodes* which this *input symbol* represents.

A *task symbol* represents a *task node.* The *task symbol* contains *task name* and may contain either a sequence of *statements* or informal text.

A *create request symbol* represents a *create request node.* It contains a *create request action* as specified in SDL/PR syntax.

A *decision symbol* represents a *decision node.* It contains a question and may contain a *decision name.* Two or more *flow lines* lead from the *decision symbol* to other *symbols*; each such *flow line* has attached to it (i.e., written alongside or inserted in a break in the *flow line*) its own *answer name.* The ELSE *answer* implies an *answer* which is for any *answer* that is not covered by any other *answer name.*

An *output symbol* represents one or more *output nodes. Signal names* which are contained in the *output symbol* are separated by commas. Each of these *signal names* gives the *name* of an *output node* which this *output symbol* represents. The destination *process instance identifier* can optionally be given in the *output symbol* using the SDL/PR syntax (that is, a TO keyword followed by an *expression* of the *type process instance identifier*, the TO keyword is after the list of *signal names*). Where the destination *process* of a *signal* cannot be uniquely determined because neither the *signal name* nor the context is sufficient to allow this, the *process instance identifier* is required.

A *nextstate symbol* represents an *arc* connecting the last *node* in a *transition string* to the following *state node.*

A *flow line* connecting two other symbols (representing *nodes*) represents the *arc* connecting the corresponding *nodes.*

A *comment symbol* is used to attach *informal text* to any other *symbol.*

An *in-connector* contains a *label* and represents the continuation of a *flow line* from a corresponding *out-connector* which contains the same *label.*

3.3.3 *Graphical conventions*

3.3.3.1 *Implicit transitions*

The *abstract syntax* of SDL requires that either a *save* or an *input* leading to a *transition* be specified for every signal in the *valid input signal* set of a *process* for every *state* of the *process.* SDL/GR provides implicit "null" *transitions* for any signals for which neither *transitions* nor *saves* are given explicitly. A null *transition* is equivalent to a connection from a *state symbol* to an *input symbol* which is then connected back to the same *state symbol.* For a given *state*, SDL/GR thus discards any *signals* which are not explicitly mentioned in conjunction with that *state.* Also the *data* contained by such *signals* is discarded.

If no *start symbol* appears in an SDL/GR *process diagram*, an implicit *start symbol* which is directly connected to the "starting" *state symbol* is assumed. The starting *state* may be identified by implication from its *name*, or by a *comment.*

3.3.3.2 *Flow lines and connectors*

Where two or more *symbols* are followed by a single *symbol*, the *flow lines* leading to that *symbol* converge. This *convergence* may appear as one *flow line* flowing into another or as more than one *out-connector* associated with a single *in-connector*, or as separate *flow lines* entering the same *symbol.*

Where a *symbol* is followed by two or more other *symbols*, a *flow line* leading from that *symbol* may diverge into two or more *flow lines.*

Arrowheads are required whenever two *flow lines* converge and whenever a *flow line* enters an *out-connector* or a *state symbol.* Arrowheads are prohibited on *flow lines* entering *input symbols.* In all other circumstances, the arrowheads are optional.

3.3.3.3 *Multiple appearance*

Whenever *state symbols*, or *connector symbols*, appear having the same *name*, they are considered to represent the same semantic entity. The resulting entity is considered to have the union of all incoming and outcoming *flow lines* from all its multiple representations. Whenever one or more *stop symbols* appear on the same *process diagram*, they all represent the *stop node*.

3.3.3.4 *Shorthand notation*

A shorthand notation is provided to allow reference to all, or all other *signals*, or *states*, in either the *input*, *save* or *state symbol*.

An input *symbol* attached to a *state* and containing an "*" (asterisk) indicates that the following *transition* applies to all incoming *signals* that do not appear otherwise in *input symbols* or *save symbols* attached to any appearance of that *state symbol*. Only one input *symbol* or one *save symbol* containing an "*" is allowed for any *state*.

A *save symbol* attached to a *state* and containing an "*" indicates that all signals that do not appear in *input symbols* attached to any appearance of that *state symbol* should be *saved*.

An "*" in a *state symbol* denotes all *states* in that *process* and indicates that the following *transitions* or *saves* should be interpreted in every *state*. An "*" followed by a list of *state names* in brackets indicates that the following *transitions* or *saves* should be interpreted in every *state* except for those listed. Such *state symbols* must not have any incoming *flow lines*.

An "–" in a *nextstate symbol* means that the following *state* is the same *state* from which the current *transition* was started. The "–" is not allowed in a *nextstate symbol* that follows the *start symbol*.

A *nextstate* symbol and a *state symbol* can be merged only if they represent the same *state symbol*.

3.3.3.5 *Miscellaneous*

All *symbols* of the same *type* shall preferably be of the same size within any one diagram.

The preferred orientation of *symbols* is horizontal and the preferred aspect ratio of *symbols* is 2:1.

Mirror images of *input* and *output symbols* are allowed.

Flow lines are horizontal or vertical and have sharp corners.

Flow lines that cross do not have a logical relationship.

The text associated with a *symbol* should be placed within that *symbol* where practical.

3.3.3.6 *SDL template*

A template suitable for hand drawing the basic set of SDL *symbols* is enclosed with the SDL user guidelines.

3.4 *Text extension symbol*

A *text extension symbol* may be attached to all SDL/GR *symbols*. The text contained in this *symbol* is to be regarded as contained in the *symbol* to which the *text extension symbol* is attached. The *text extension symbol* is shown in Figure 6/Z.101.

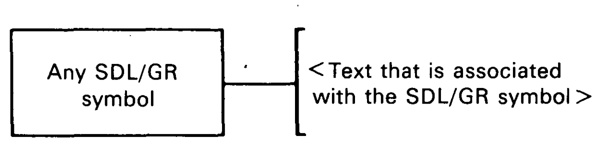


FIGURE 6/Z.101

Text extension symbol

3.5 *Comments in SDL/GR*

In all SDL/GR diagrams, *comments* may be inserted wherever the user finds it appropriate. The *comments* may be inserted by using either the SDL/GR *comment symbol* (see Figure 7/Z.101) or the SDL/PR syntax for *comments* (see § 4.3.2, lexical rule 7).

FIGURE 7/Z.101

Comment symbol

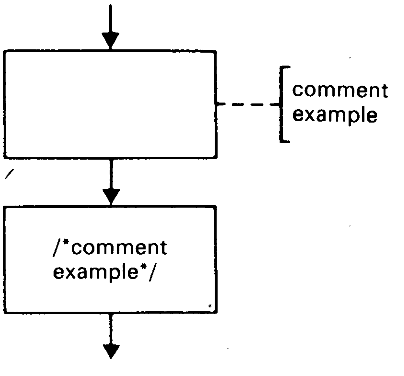


FIGURE 8/Z.101

Examples of comments in SDL/GR

4 Linear syntax

4.1 *General*

This section defines SDL/PR and relates it to the common language model (see § 2).

A *system definition* in the SDL/PR syntax is represented by a sequence of statements bounded by the words SYSTEM and ENDSYSTEM.

The detailed rules for the SDL/PR syntax are contained in the *syntax diagrams* (see § 4.3).

Reference to a named entity outside its definition is made by an identifier. The *identifier* is composed of a *name* and an optional *qualifying part.* The *qualifying part* must be used when the *name* alone will not uniquely determine the item being referred to.

4.2 *Keywords*

SDL/PR uses a number of keywords to express SDL concepts as defined in the abstract syntax. Some of the keywords are used in pairs to reflect the structuring of SDL into SDL/PR.

4.2.1 *Paired structuring keywords concerned with definitions*

The word "embrace" is used with these word pairs to indicate their role as delimiters.

SYSTEM ENDSYSTEM	embrace the concept of *system definition* (the SDL/PR representation of a system starts with the keyword SYSTEM and ends with the keyword ENDSYSTEM).
BLOCK ENDBLOCK	embrace the concept of a *block definition.*
PROCESS ENDPROCESS	embrace the concept of a *process definition.*

4.2.2 *Single keywords concerned with definitions*

The keywords in this paragraph are used to indicate that a definition follows.

DCL	introduces the representation of the *variable definition.* The keyword REVEALED is used within a DCL statement to identify revealed *variables.*
VIEWED	introduces the representation of the *view definition.*
SIGNAL	introduces the representation of the *signal definition.*
CHANNEL	introduces the representation of the *channel definition.* The keyword FROM is used within the CHANNEL definition to indicate the origin *block* of the *channel* and the keyword TO is used to indicate the destination *block.* The keyword ENV is used to refer to the *environment.* The keyword WITH is followed by the list of the *signals* carried by the *channels.*
FPAR	introduces the representation of the *formal parameter definition.*
DURATION TIME TIMER NATURAL INTEGER REAL CHARSTRING CHARACTER BOOLEAN PID	represent the *predefined data types.*
VIEW	introduces the representation of the *viewing expression.* It is used within an *expression* wherever a *variable* declared as VIEWED is used.
SET	introduces the representation of the *set statement.*
RESET	introduces the representation of the *reset statement.*
SYSTEM BLOCK PROCESS	introduce the *qualifying part* of an *identifier.*

4.2.3 *Keywords associated with nodes in a process graph*

The process graph in the *abstract syntax* consists of *nodes* connected by directed *arcs.*

Keywords are chosen to correspond to *nodes,* and the *arcs* which connect *nodes* in the *abstract syntax* are represented by the ordering in which the keywords occur (see § 4.2.5).

START	represents the *start node.* If this *keyword* is not present, the first STATE keyword, following PROCESS, represents the starting *state.*
STATE	introduces the representation of one or more *state nodes.* The *save-signal-set* attached to a *state node* is represented by the keyword SAVE followed by one or more *signal identifiers.*

INPUT	introduces the representation of one or more *input nodes.*
TASK	introduces the representation of a *task node.*
OUTPUT	introduces the representation of one or more *output nodes.* The destination *process instance identifier* can optionally be given by the keyword TO followed by an *expression* which yields a *process instance identifier value.* Where the *signal* destination cannot be uniquely determined, the keyword TO is required.
DECISION ENDDECISION	embrace the concepts of a *decision node.* The keyword ELSE is used to represent the *answer* for all cases not explicitly named.
CREATE	introduces the representation of a *create request node.*
STOP	represents a *stop node.*

4.2.4 *Keywords associated with arcs*

JOIN represents an *arc* between *nodes* that are not *state nodes.* The first *node* is generally represented by the keyword immediately before the keyword JOIN, the second *node* is always identified by having the same *label identifier* as the keyword JOIN. There are some exceptions to this general explanation as far as the first *node* is concerned (see § 4.2.5).

If the second *node* is another JOIN, the *arc* is connected with the *node* that this JOIN refers to.

If the keyword having the join *label* is NEXTSTATE, the second *node* is the *state* with the same *name* (the NEXTSTATE rules are valid).

NEXTSTATE represents an *arc.* The first *node* of the *arc* is represented by the keyword immediately before the keyword NEXTSTATE, the second *node* is the *state* with the same *name.*

4.2.5 *Representation of arcs in SDL/PR*

The representation rule of an *arc* in SDL/PR is given by the keywords ordering.

There are some exceptions to this general meaning in case of keywords such as JOIN and NEXTSTATE as it is said in the previous paragraph.

Moreover, when a keyword (associated with a *node* or an *arc*) immediately follows an *answer,* the first *node* of the *arc* is the preceding matching *decision.*

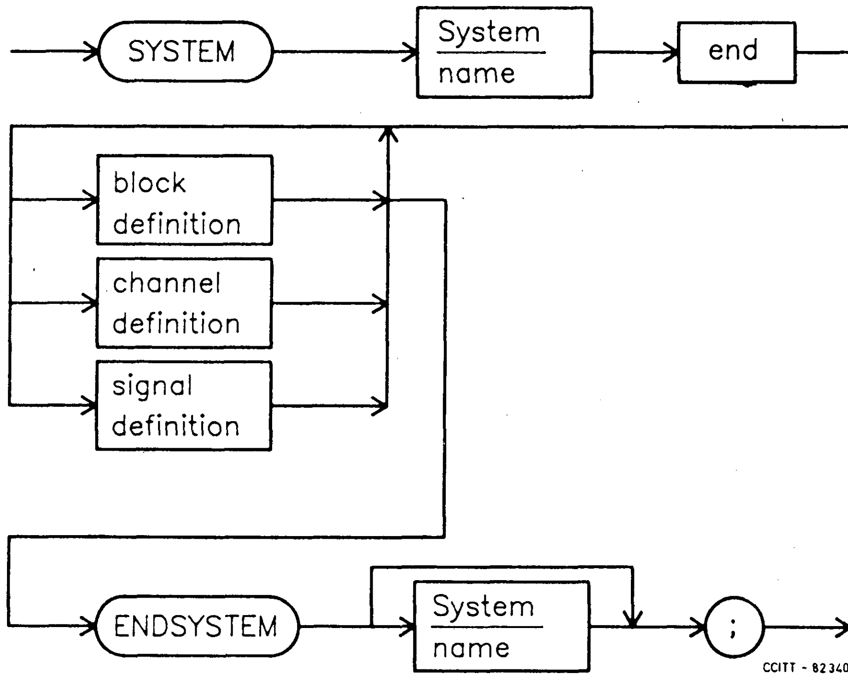
If the keyword immediately before a keyword associated with a node or an arc is ENDDECISION, the first *nodes* of the *arcs* are represented by the last keywords in all *transition strings* of the *decision* having no terminator statements.

The last keyword of a *decision* branch represents a *node* not connected with the keyword that follows the next result *name,* but connected with the keyword after the ENDDECISION. This rule is clearly not valid if the last keyword of a *decision* branch is a terminator statement.

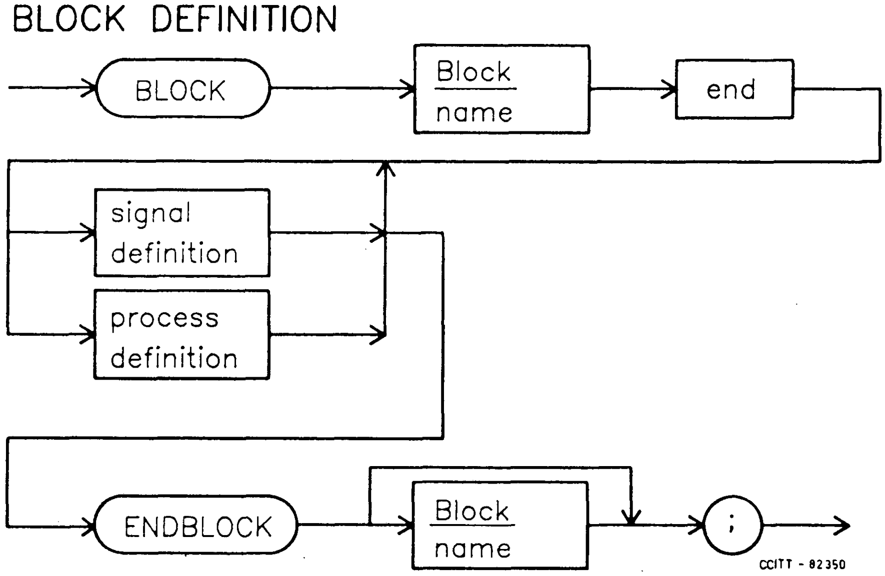
4.3 *Reserved words in SDL/PR*

Certain words are reserved in SDL/PR and may not be used as *names.* The list of reserved words is found in Annex C.2 to Recommendations Z.100 to Z.104.

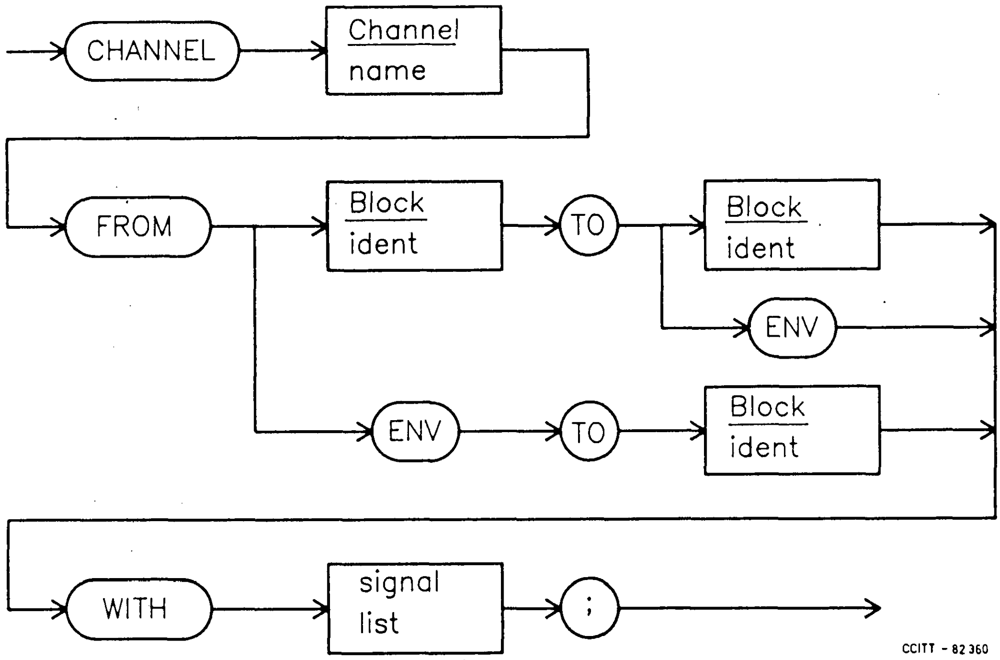
SYSTEM DEFINITION



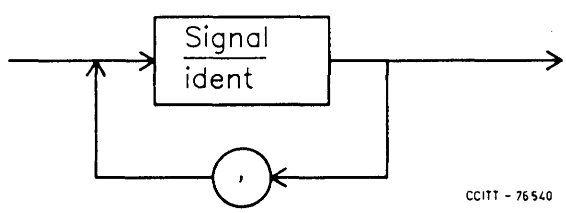
BLOCK DEFINITION



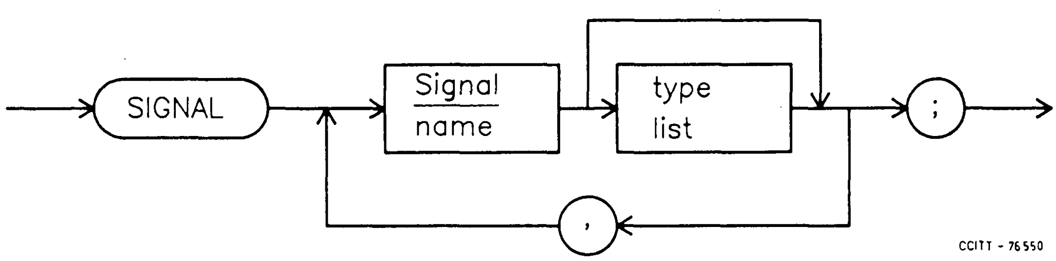
CHANNEL DEFINITION



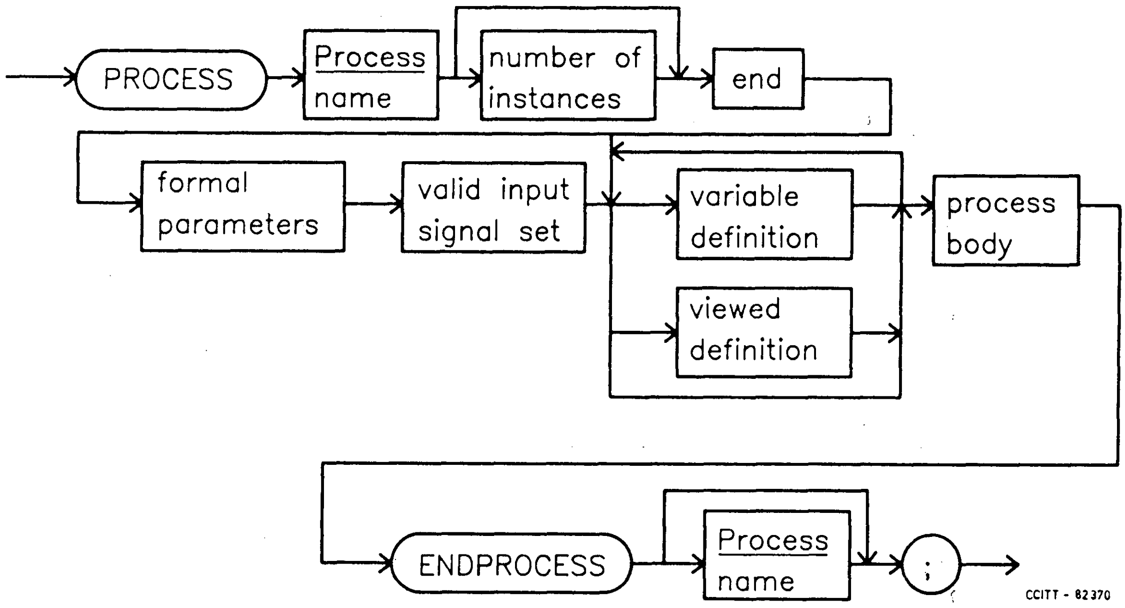
SIGNAL LIST



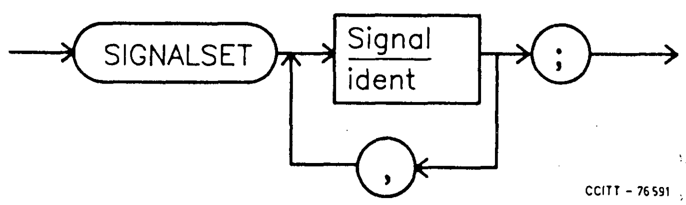
SIGNAL DEFINITION



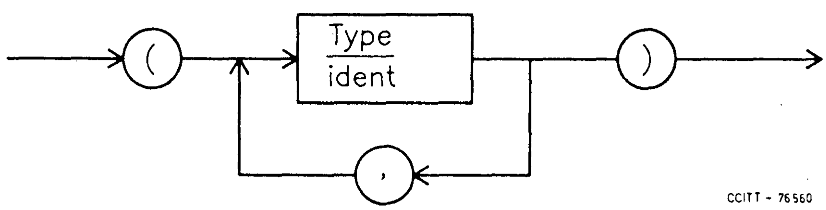
PROCESS DEFINITION



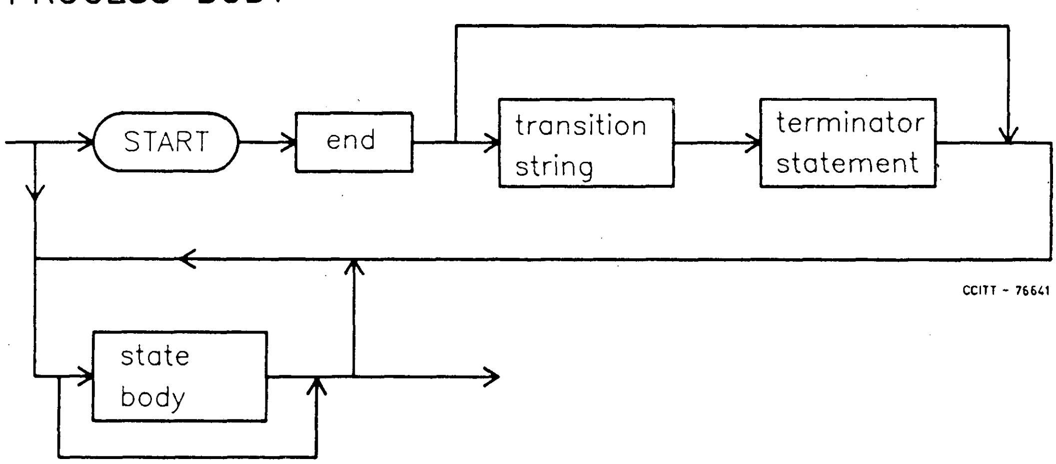
VALID INPUT SIGNAL SET



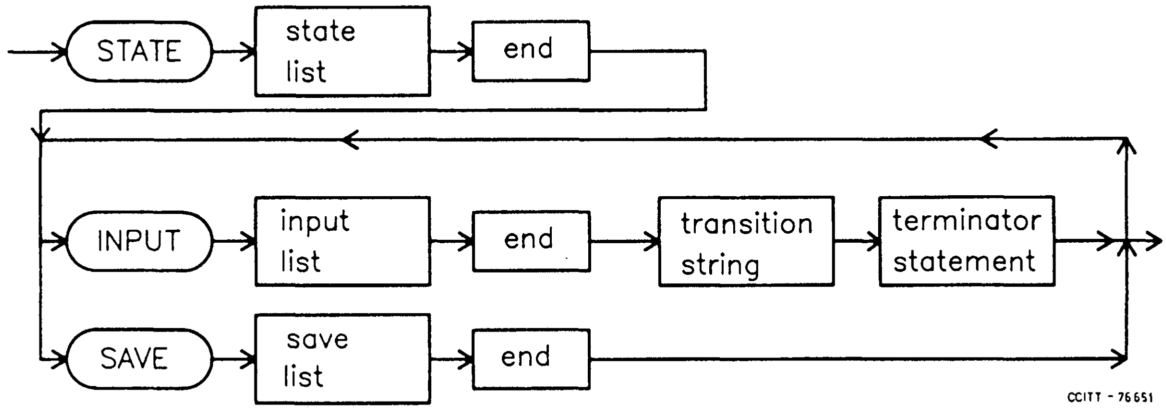
TYPE LIST



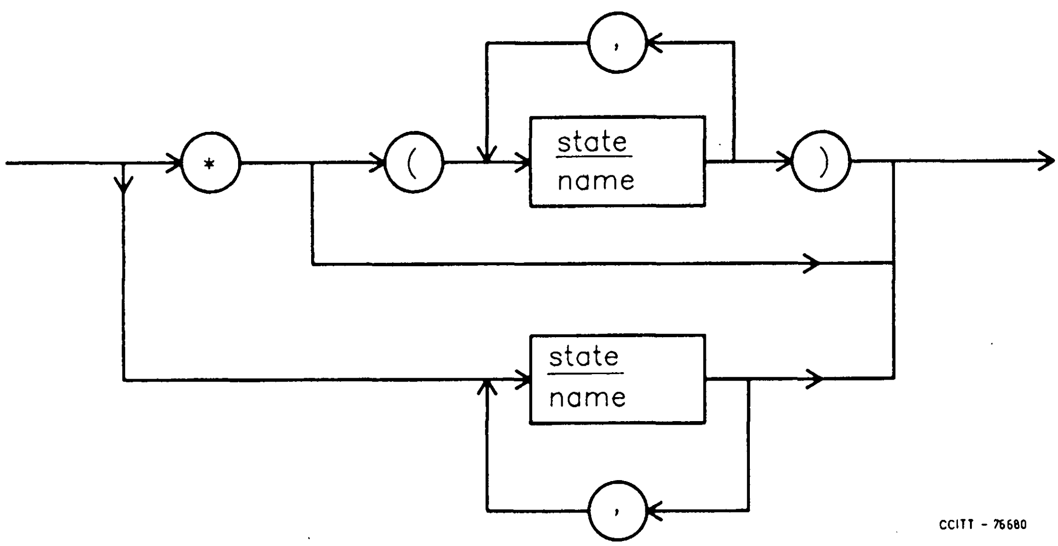
PROCESS BODY



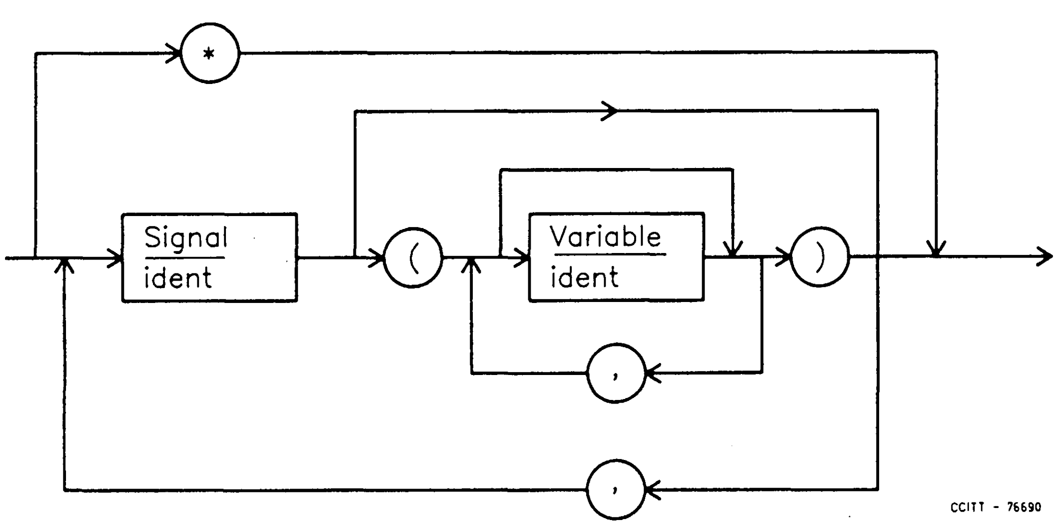
STATE BODY



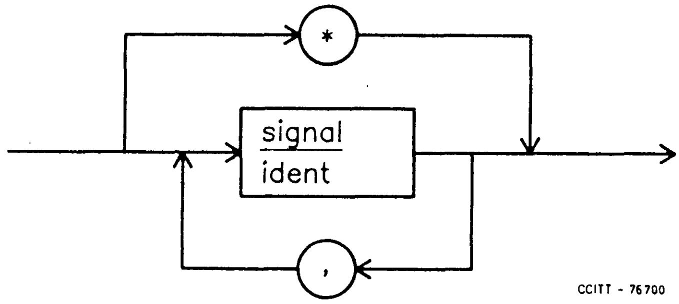
STATE LIST



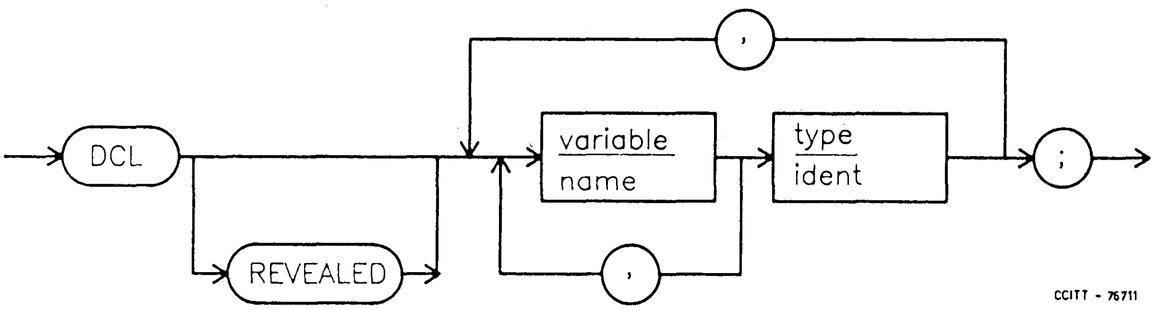
INPUT LIST



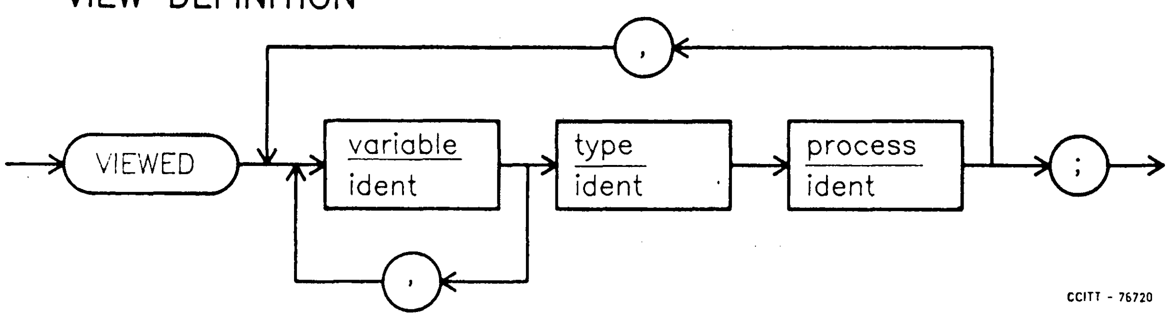
SAVE LIST



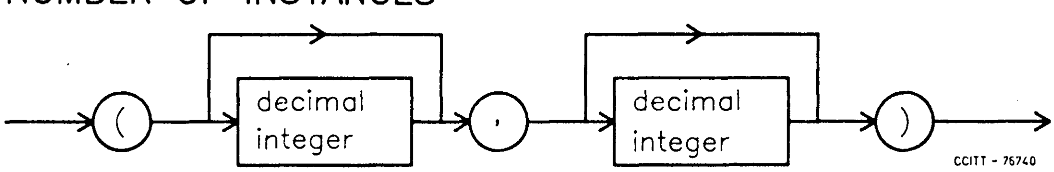
VARIABLE DEFINITION



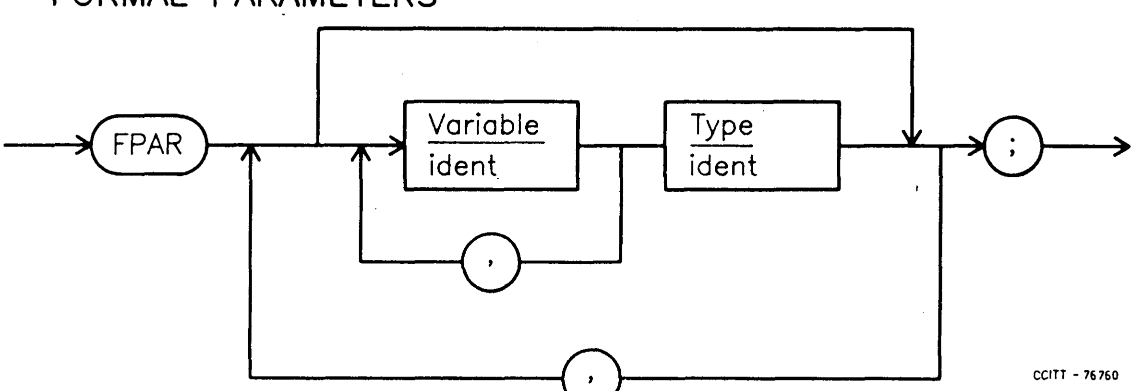
VIEW DEFINITION



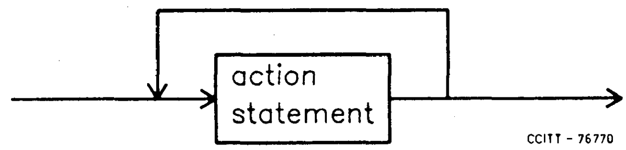
NUMBER OF INSTANCES



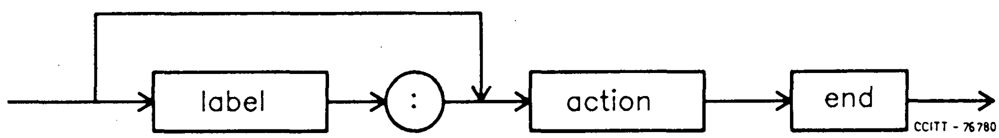
FORMAL PARAMETERS



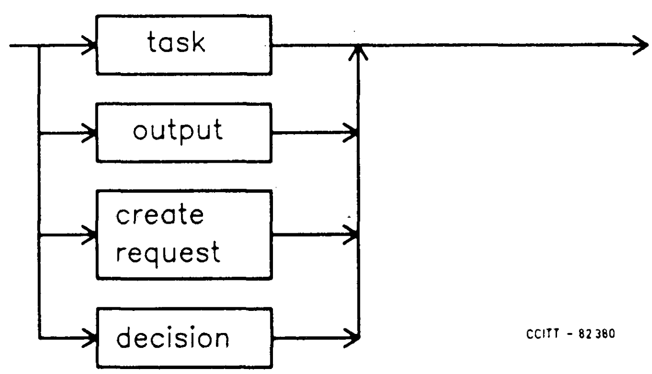
TRANSITION STRING



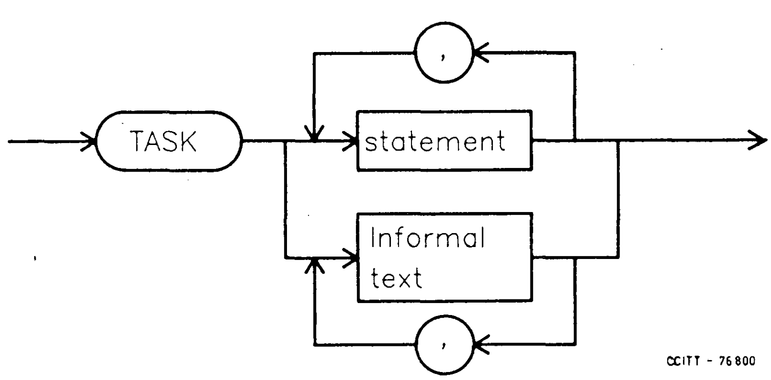
ACTION STATEMENT



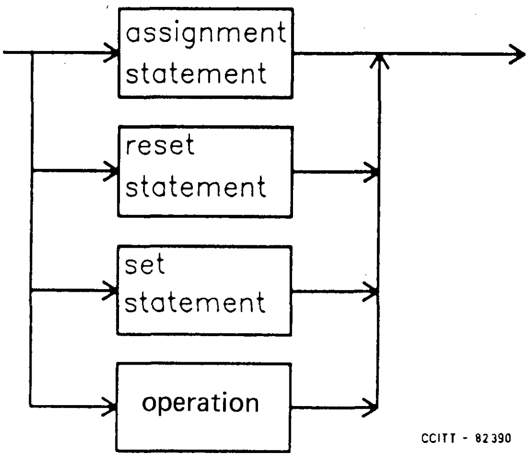
ACTION



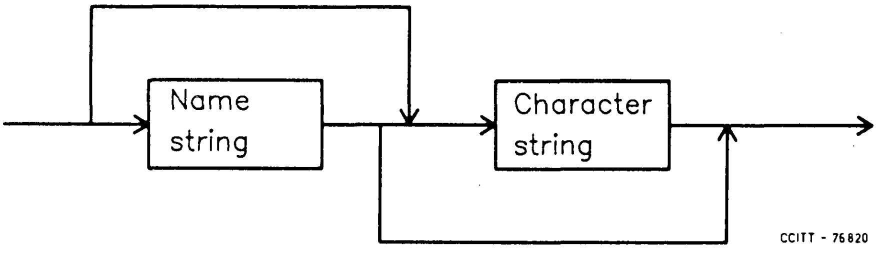
TASK



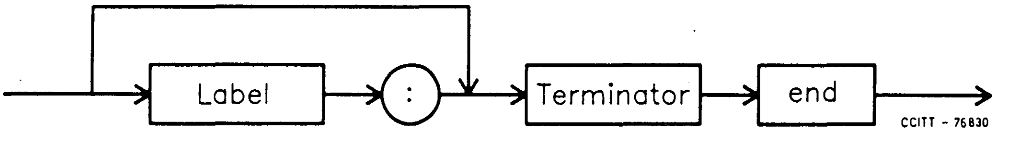
STATEMENT



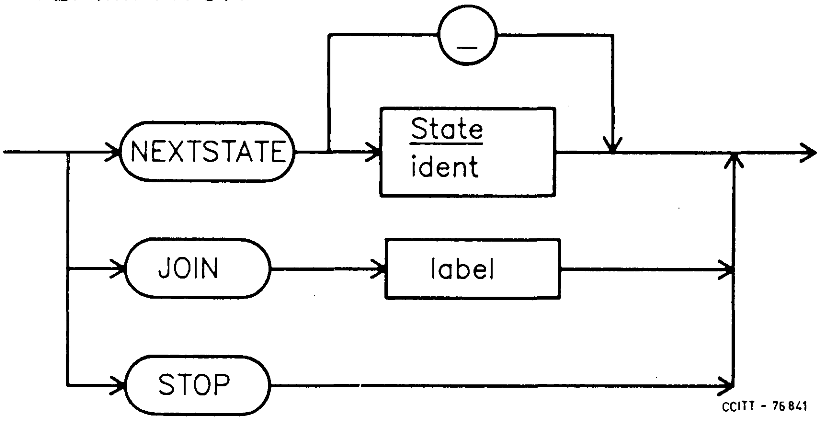
INFORMAL TEXT



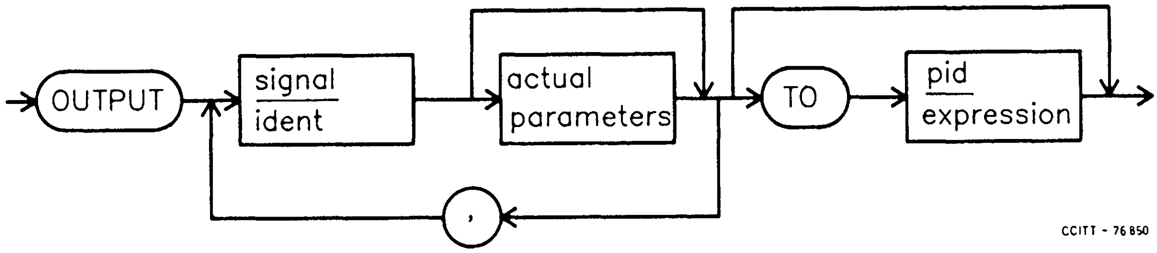
TERMINATOR STATEMENT



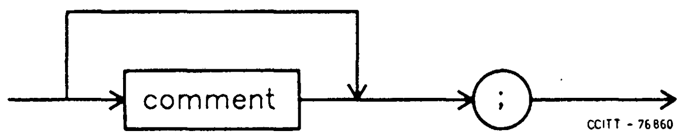
TERMINATOR



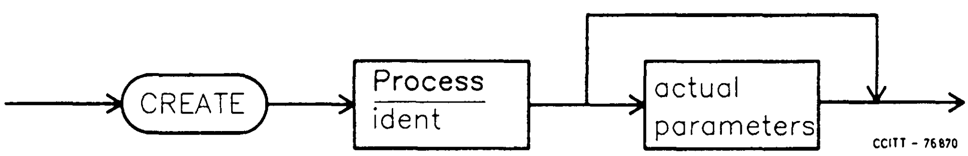
OUTPUT



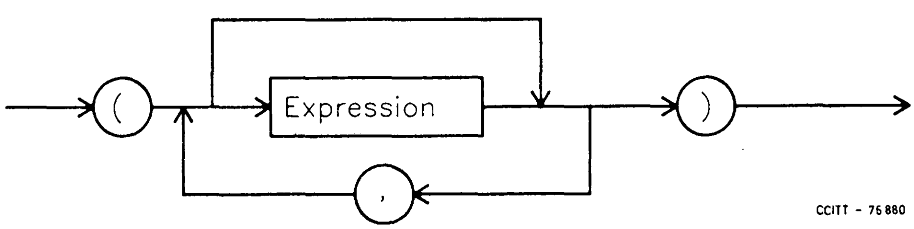
END



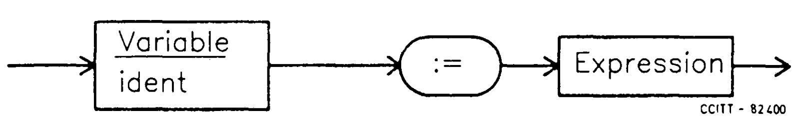
CREATE REQUEST



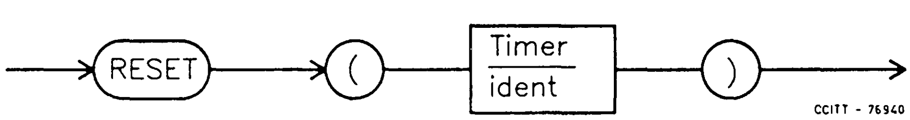
ACTUAL PARAMETERS



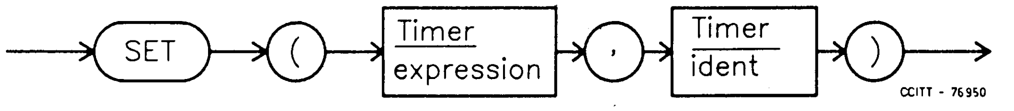
ASSIGNMENT STATEMENT



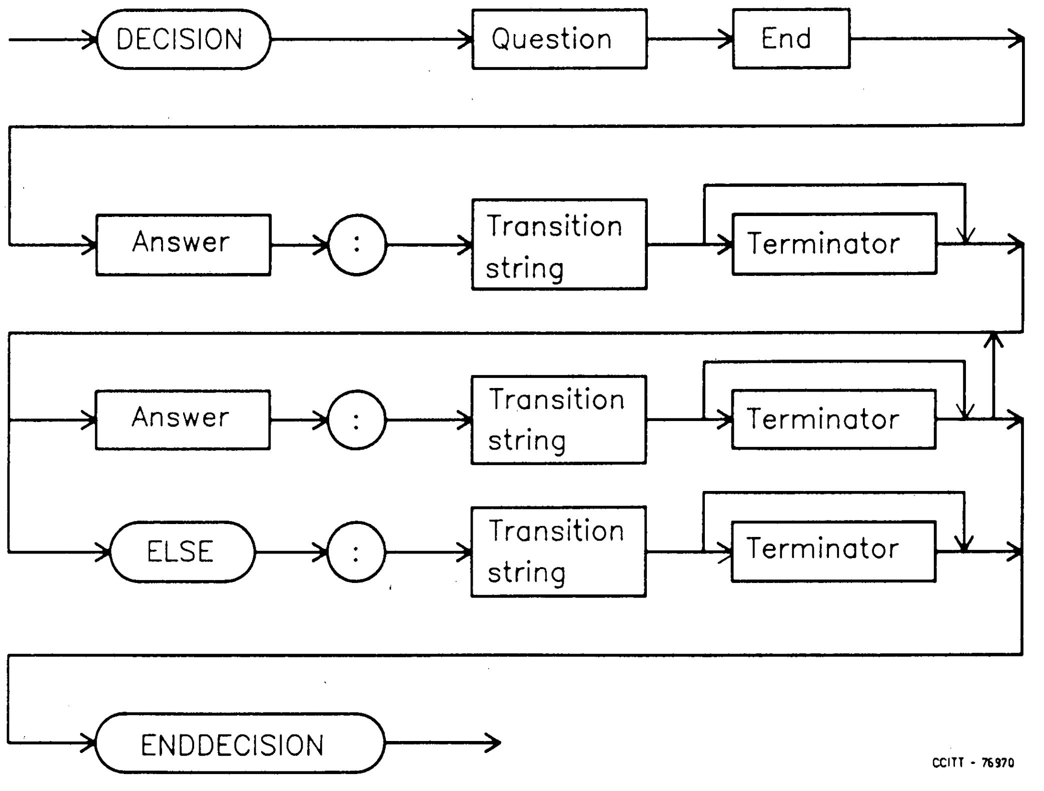
RESET STATEMENT



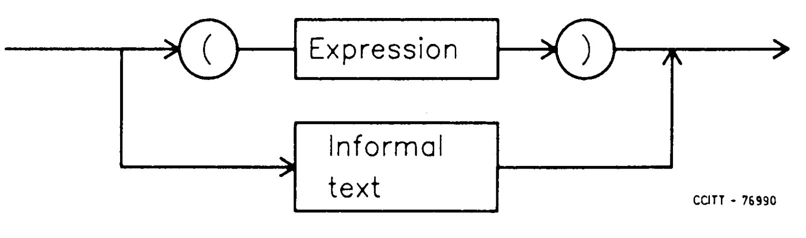
SET STATEMENT



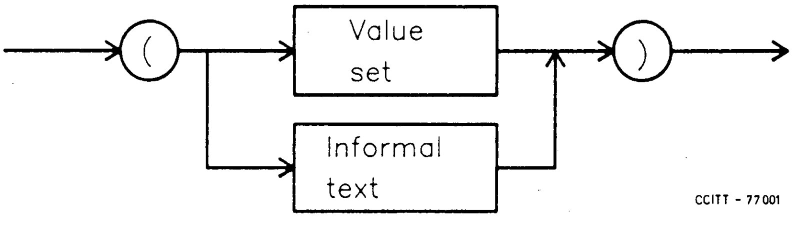
DECISION



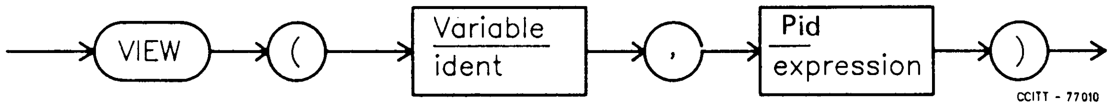
QUESTION



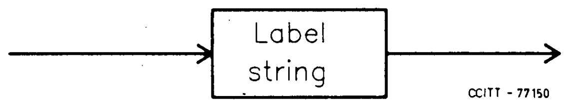
ANSWER



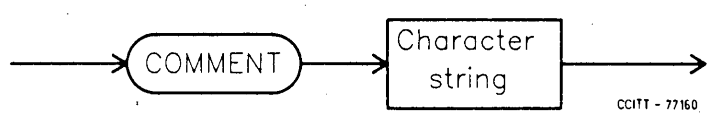
VIEWING OPERATOR



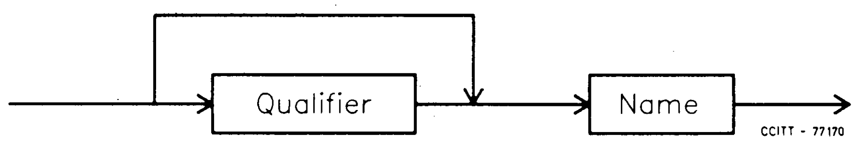
LABEL



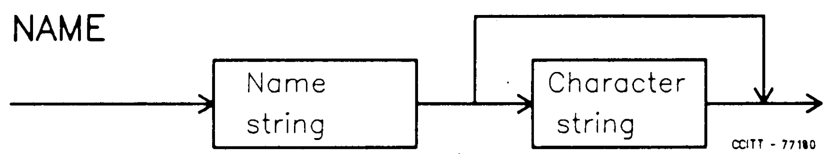
COMMENT



IDENT



NAME



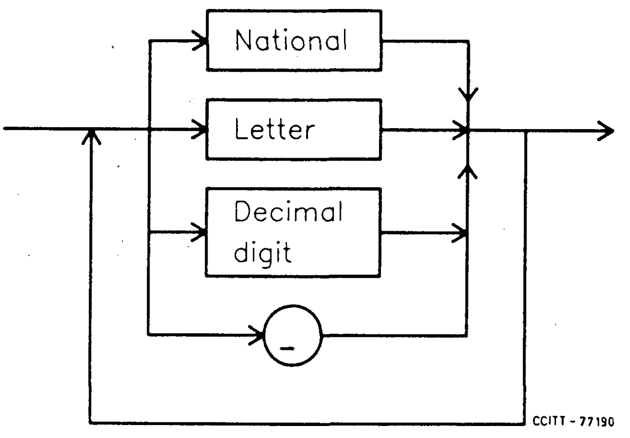
4.4.1 *Lexical units*

4.4.1.1 *Lexical rules*

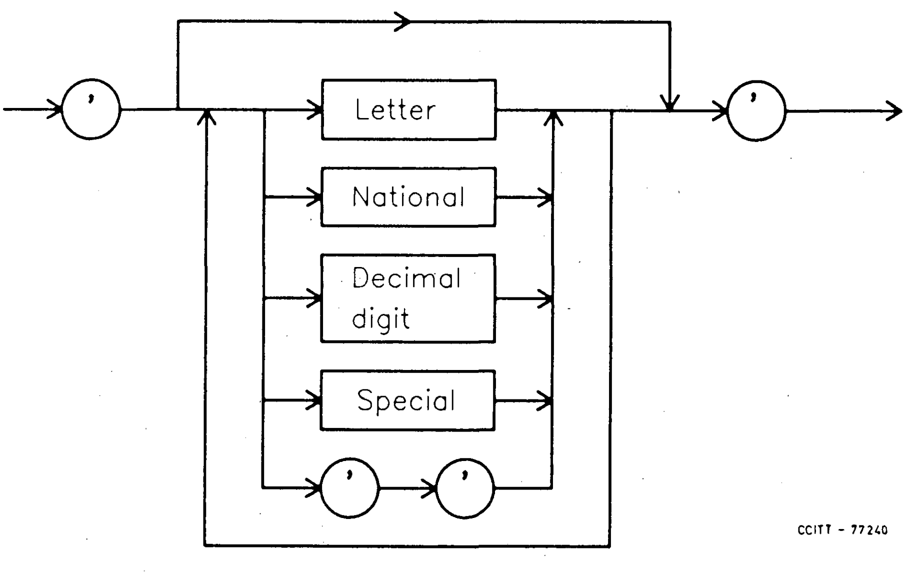
- All the punctuation marks [e.g. , . ; ' : ! = ( )] and operation symbols (e.g. +, −, *, <, > ...) are lexical units which may take the place of spaces.
- Two lexical units must be separated by one or more spaces.
- Keywords belong to the same lexical category as namestring, and they are reserved.
- Outside lexical units several spaces have the same "meaning" as one space.
- Tabulation characters (VT, HT, CR, BS ...) may be considered as spaces.
- All letters and nationals are always interpreted as if uppercase, except with a charstring.
- Wherever spaces may occur comments may be inserted delimited by ' /*' and '*/', these comments have the same meaning as one space. The comment must not contain the special sequence '*/'.

4.4.1.2 *Syntax diagrams*

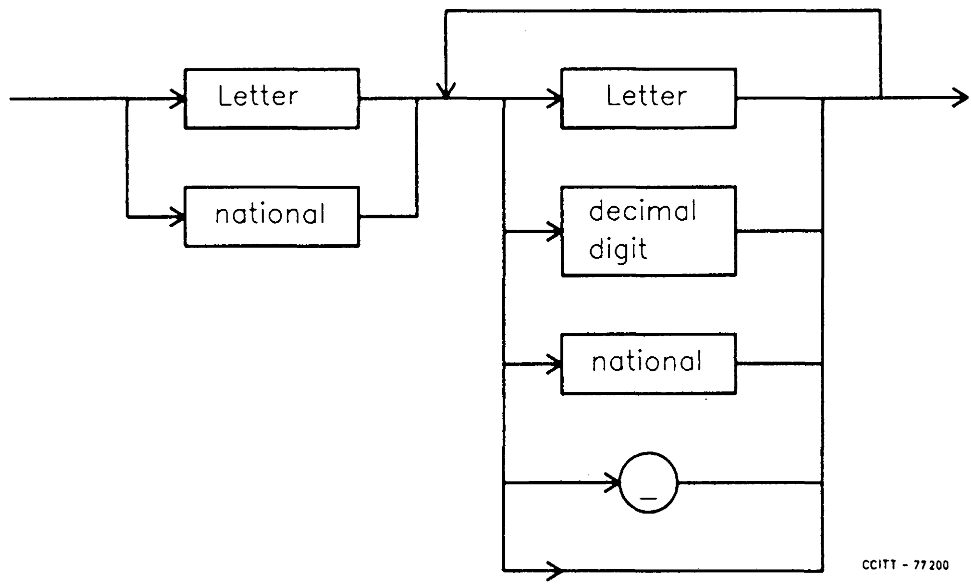
LABEL STRING



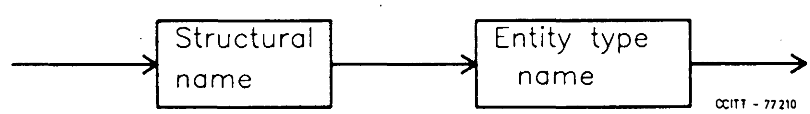
CHARACTER STRING



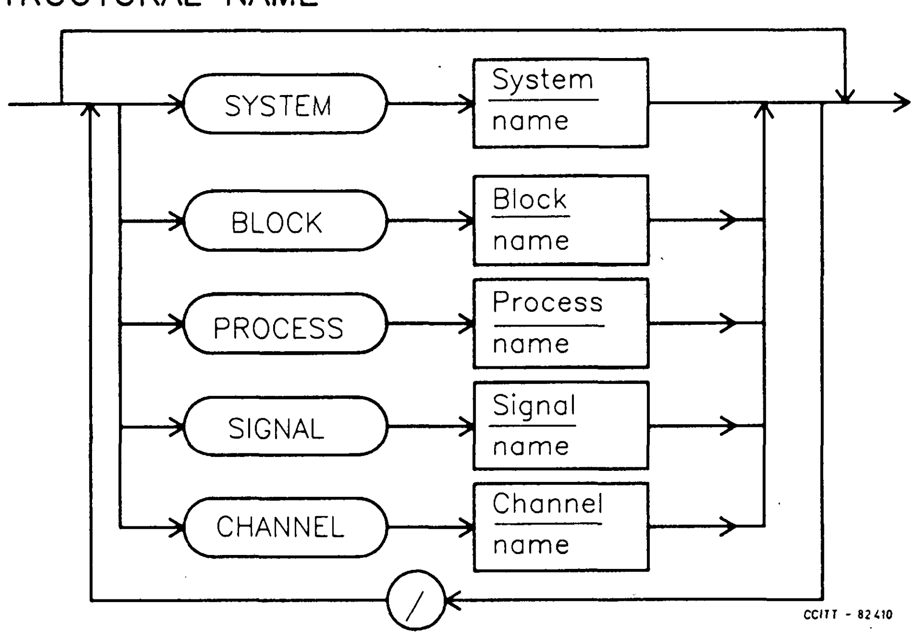
NAME STRING



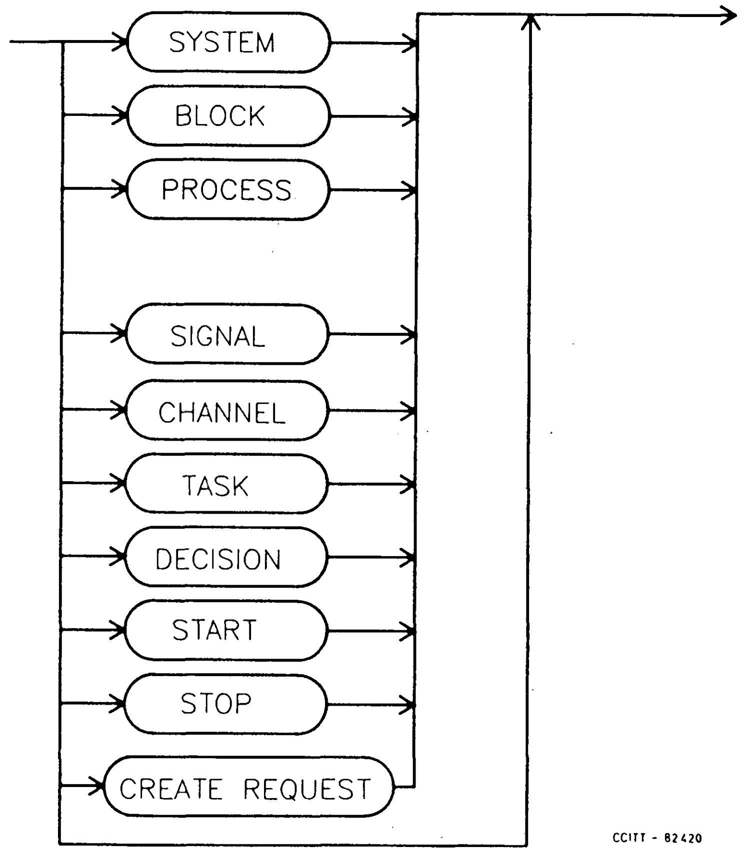
QUALIFIER



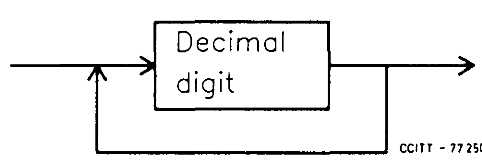
STRUCTURAL NAME



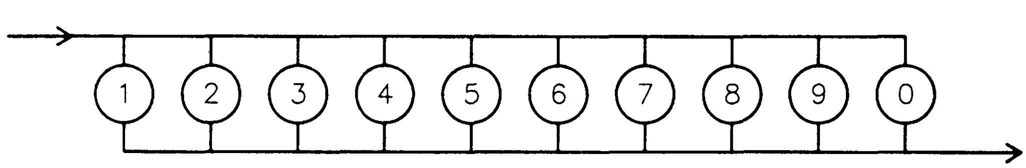
ENTITY TYPE NAME



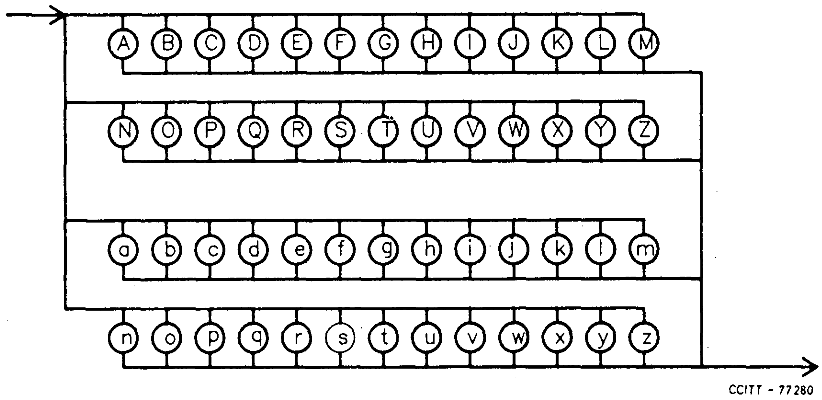
DECIMAL INTEGER



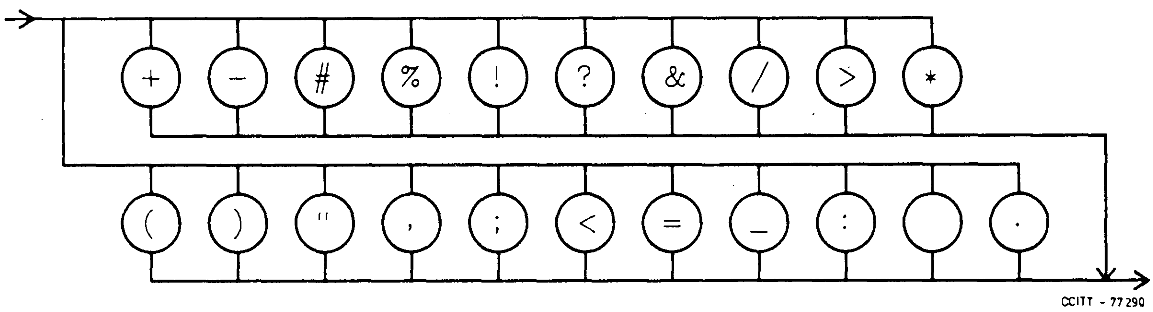
DECIMAL DIGIT



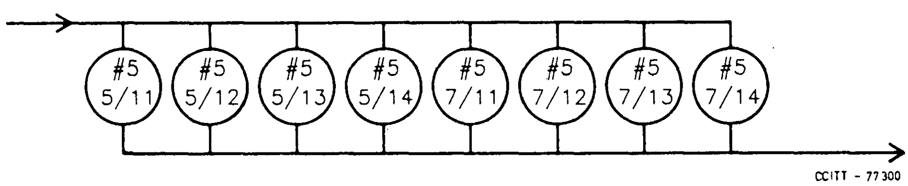
LETTER



SPECIAL

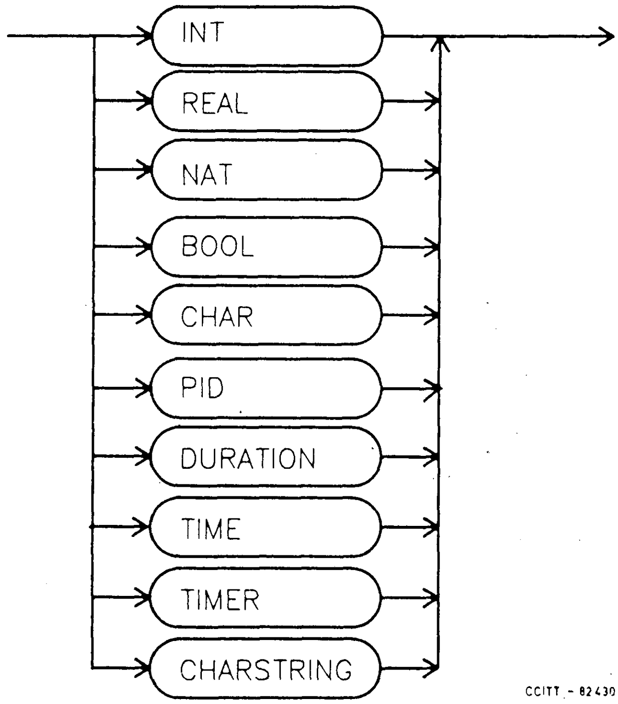


NATIONAL



The above referenced positions refer to the positions in the CCITT International Alphabet No. 5 reserved for national use

PREDEFINED DATA TYPE NAMES



CCITT - 82430

Recommendation Z.102

STRUCTURAL CONCEPTS IN SDL

1 Introduction

This Recommendation defines a number of concepts needed to handle hierarchical structures in SDL. The basis for these concepts is the SDL as defined in Recommendation Z.101 and the defined concepts are strict additions to those defined in Recommendation Z.101. There is no conflict between the definitions contained in this Recommendation and those contained in Recommendations Z.103 and Z.104.

The intention with the concepts introduced in this Recommendation is to provide the user of SDL with means to describe large and/or complex systems. The SDL as defined in Z.101 is suitable for specifying or describing relatively small systems which may be understood and handled at a single level of *blocks.* When a larger, or complex system should be represented there is a need to partition the system *specification* or *description* into manageable units, which may be handled and understood independently. It is often suitable to perform the partition in a number of steps, resulting in a hierarchical structure of units representing the system.

There is also a need to use structural concepts in order to specify or describe the required or actual structure of a system.

This Recommendation defines concepts for the *partitioning* of:

blocks into *sub-blocks, sub-channels* and new *channels,*

channels into *blocks* and *channels,*

processes into *sub-processes.*

The concepts are such that the resulting hierarchical structure, representing the system, will provide the reader with series of overviews from which he can gain a general appreciation before descending to a more detailed description. This means also that the concepts will support design technologies aiming at "stepwise refinement" by adding more detailed information in a number of steps.

2 Common language model

2.1 *General*

In Recommendation Z.101 a *system* is described as composed of a set of *blocks* connected to each other and to the *system boundary* by unidirectional *channels.* This Recommendation introduces concepts for describing the *partitioning* of *blocks, channels* and *processes* into sub-components.

Each *block,* in a *system,* may be partitioned into one or more *sub-blocks.* In this *partitioning,* new *channels* are introduced to connect the *sub-blocks* to each other; in addition, the *channels* terminating at and originating from the partitioned *block* may be split into *sub-channels.*

Block A is partitioned into:

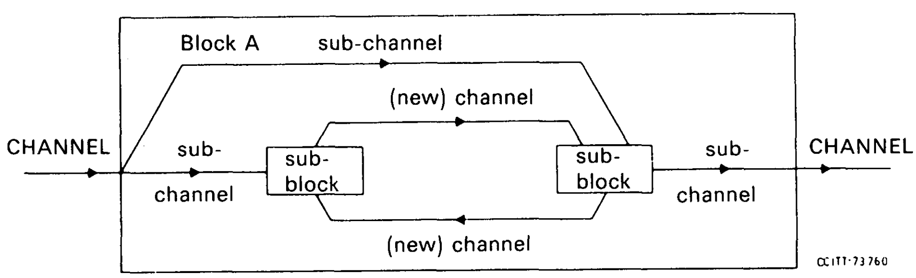


FIGURE 1/Z.102

The partitioning of a block

A *sub-block,* in turn is a *block* and may be partitioned. This *partitioning* may be repeated any number of times resulting in a hierarchical structure of *blocks* and their *sub-blocks.* The *sub-blocks* of a *block* are said to exist on the next lower *level* in the *block tree*:

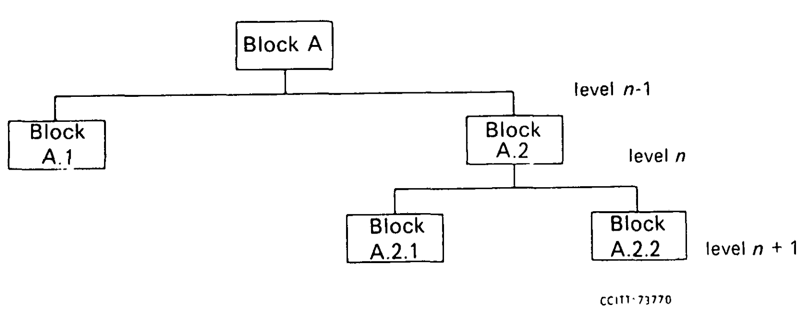


FIGURE 2/Z.102

A Block-tree

When a block is partitioned into *sub-blocks*, the rule from Recommendation Z.101 that a *block definition* should contain one or more *process definitions* is relaxed, and is only valid for a *block* which is not further partitioned (i.e. the "leaf"-*blocks* in the *block tree* describing the *system* must contain *processes*). However, if a *block definition* contains *process definitions*, the *sub-block definitions* must include at least the *sub-process definitions* resulting from the *partitioning* of the *processes*.

The *sub-channels* resulting from the *block partitioning* are *channels*.

Channels may also be partitioned. This results in a set of new *blocks*, new *channels*, one *incoming channel*, and one *outgoing channel*:

$\overrightarrow{C}\ [1_1]$

CHANNEL C is partitioned into:

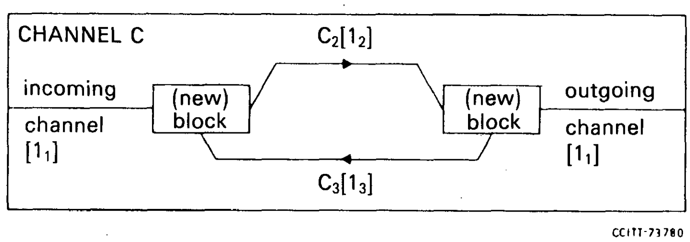


FIGURE 3/Z.102

The partitioning of a channel

Note that the *signal list* associated with the original *channel* C is associated with the *incoming* and *outgoing channels*.

Both the *partitioning* of *blocks* and the *partitioning* of *channels* leaves unchanged the interfaces of the partitioned objects.

A *process definition* may be partitioned into a set of *sub-process definitions*. These two descriptions of the behaviour are alternative in the sense that when the *system definition* is interpreted either the set of *sub-process definitions* or the *process definition* is interpreted. The *process definition* is to be considered as an alternative description with relation to the set of *sub-process definitions*.

A *sub-process* is a *process*, and may in turn be partitioned into *sub-processes*. The resulting hierarchy of *processes* is represented in a *process tree*.

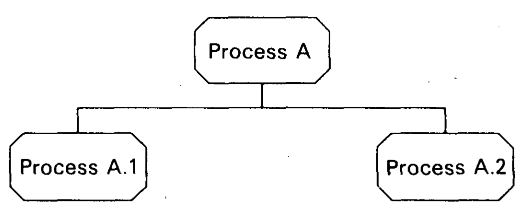


FIGURE 4/Z.102

Process – sub-process relation

As *sub-processes* are *processes*, the behaviour described by the partitioned *process* is partitioned into a set of concurrent "sub-behaviours". Methods to ensure that this *partitioning* is correct are necessary but not part of the SDL.

The *partitioning* of a *block* into *sub-blocks*, and its *processes* into *sub-processes* may be done at the same time. As a *process* has to be contained in a *block*, its *sub-processes* have to be contained in *sub-blocks* of the *block* containing the partitioned *process*. The partitioned *blocks* and *processes* may be regarded as separate structures related to each other:

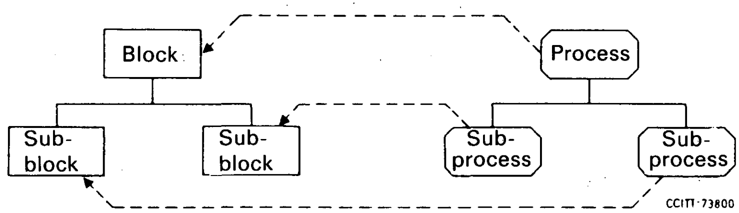


FIGURE 5/Z.102

Relation between block and process trees

If a *block* is partitioned, then any *process* it contains may also be partitioned, in which case all its sub-processes must appear in the *sub-blocks* of the *block*. If the *process* is not partitioned, then it must appear in one of the *sub-blocks*. In order to further describe the behaviour of the *sub-blocks* new *processes* and signals may also be included in the *sub-blocks* irrespective of whether there were *processes* in the *block*. However, the "leaf"-*blocks* must contain *processes*.

If, in the total *partitioning* representation of a *system*, *process definitions* appear on more than one *level*, then several consistent sub-sets of the representation may be found. A consistent sub-set is a selection of the *block definitions* and *process definitions* in the total representation such that:

a) It contains the highest *level* in the *block tree*;

b) If it contains a *block* it must contain its parent;

c) If it contains a *sub-block* of a *block*, it must also contain all other *sub-blocks* of that *block*;

d) All "leaf"-*blocks* in the resulting structure contain *processes*.

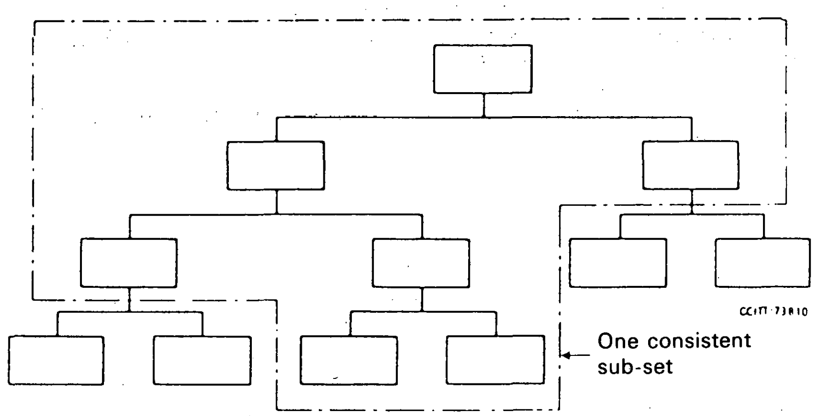


FIGURE 6/Z.102

A consistent sub-set of a system representation

If *processes* appear in a *system definition* at more than one level, then several consistent sub-sets of the representations may be found. These sub-sets represent alternative descriptions of the *system*, with various degrees of detail. They may be used to provide readers with both overviews and detailed descriptions. They may be chosen so they support the interpretation of the *system* on alternative levels of abstraction. If several alternative representations of the behaviour exist, all but the most detailed one are to be considered as overviews of the behaviour.

2.2 *Abstract syntax*

This *abstract syntax* is based on the *abstract syntax* given in Recommendation Z.101. Only the additions to the definitions in Recommendation Z.101 are given here.

Block definition

A *block definition* may also contain an *internal part block definition*, and if it does so it need not contain *process definitions.*

Internal part block definitions

The *internal part block definition* contains one *block substructure definition* and it contains one *process substructure definition* for each of the *process definitions* contained in the *block definition.* It may also contain *signal definitions* and *data type definitions.*

Block substructure definition

A *block substructure definition* may contain one or more *sub-block definitions* and one or more *channel definitions.*

For each of the terminating endpoints of *sub-channels* of the enclosing *block* there must be at least one *sub-channel definition* having that endpoint as originating endpoint, and the reverse must hold for all originating *sub-channel* endpoints of the enclosing *block.* The union of the *signal lists* of the *sub-channel definitions* having the same endpoint as a *subchannel* leading to or from the enclosing *block* must be identical to the *signal list* of that *block,* in addition to this *signal lists* of the *sub-channel definitions* originating from a terminating endpoint must be disjoint.

All *channel definitions* contained in the *block substructure definition* must either connect *sub-blocks* to *channel* endpoints of the enclosing *block* or *sub-blocks* to each other.

Sub-block definition

A *sub-block definition* is a *block definition.*

Channel definition

A *channel definition* may also contain a *channel substructure definition.*

Sub-channel definition

A *sub-channel definition* is a *channel definition.*

Channel substructure definition

A *channel substructure definition* contains two or more *channel definitions,* one or more *block definitions,* and may contain *signal definitions.*

All *channel definitions* contained in the *block substructure definition* must connect *sub-blocks* to each other, and all *sub-channel definitions* must connect *channel* endpoint of the enclosing *block* to *sub-blocks.*

Process substructure definition

A *process substructure definition* is associated with a *process name* and contains one or more *process names,* each associated with a *sub-block name.*

The associated *sub-process name* must be the name of a *process definition* contained in the enclosing *block definition.* The contained *process names* must be *names* of *sub-process definitions* contained in the *block definition* having the associated *sub-block name.*

Each *signal name* in the *valid input signal* set of the associated *process* must appear in exactly one of the *valid input signal* sets of the *sub-process definitions* having the contained *sub-process names.* Each *signal name* attached to the *output node* of the associated *process* must be attached to at least one *output node* of the *sub-process definitions* having the contained *sub-process names.*

Sub-process definition

A *sub-process definition* is a *process definition.*

2.3 *Interpretation*

The interpretation rules given below are defined as additions to the corresponding set of rules defined in Recommendation Z.101.

Channel

If a *channel definition* contains a *channel substructure definition* then either the *channel* may be interpreted, as defined in Recommendation Z.101 or the *channel substructure definition* may be interpreted.

If the *channel substructure definition* is interpreted, every *signal* delivered to the originating endpoint of the *channel* is given to the *channel substructure*, and every *signal* delivered by the *channel substructure* is given to the terminating endpoint of the *channel.*

Channel substructure

A *signal* delivered to the *channel substructure* is given to the *incoming channel* and a *signal* delivered by the *terminating endpoint* of the *outgoing channel* is delivered to the enclosing *channel.*

Block

If a *block definition* contains one or more *process definitions* and also contains an *internal part block definition* then either the *block* may be interpreted as defined in Recommendation Z.101, or the *internal part block* may be interpreted. If the *block* contains no *process definitions*, the *internal part block* must be interpreted.

If the *internal part block* is interpreted, all *signals* delivered to the *block* will be given to the *internal part block*, and all *signals* delivered by the *internal part block* will be further delivered to *channels* leading from the *block* in the same manner as if delivered from a *process* contained in the *block.*

Internal part block

Each *process* in the enclosing *block* is replaced by a *process substructure.*

If a *signal* given to the *internal part block* from the enclosing *block* is addressed to a *process*, then the *signal* is given to the *process substructure*, else it is given to the *block substructure.*

A *signal*, delivered by the *block substructure* will be given to the enclosing *block.* If that *signal* was sent from a *process* appearing as a *sub-process* of a *process substructure*, the sender-attribute of the *signal* will be modified to the *process instance identifier* of the replaced *process.*

Block substructure

A *block substructure* contains *blocks* and *channels*, according to the *block substructure definition*, these are interpreted according to the rules defined for *blocks* and *channels.*

Process substructure

A *process substructure instance* replaces one *process instance* of the referenced *process definition.* It also denotes a set of *sub-process instances*, one *instance* for each *sub-process name* in the *definition.* Each *sub-process* is allocated to the *block* with the associated *block name.*

Each *signal* given to the *process substructure* will be re-addressed to the *sub-process* which has the *signal name* in the valid *input signal* set and given to the *block substructure* of the enclosing *block.*

3 Graphic syntax

The following graphic syntax is an addition to the syntax defined in Recommendation Z.101. The additions cover the representation of the structure and *partitioning* of a *system.*

An overview of the structure of a *system* is given by the *Block Tree Diagram.* The *partitioning* of *blocks*, *processes* and *channels* into sub-components is represented in the *Block Interaction Diagram* and the *Channel Substructure Diagram.*

The set of documents and diagrams describing the *system* may be large. It is essential that the documents are related to each other by references and proper titles, however syntactic means for doing this do not form part of the SDL graphic syntax.

3.1 *Block tree diagram*

The *block tree diagram* is intended to give an overview of the structure of a *system*, i.e. the *partitioning* of the *system* into a hierarchical structure of *blocks.* Details on how the *blocks* are connected by *channels* are given in the *block interaction diagram.*

A part of the diagram may also be used to give an overview of how a *block* is *partitioned* into *sub-blocks.* In this case, the partitioned *block* is shown as the root box.

3.1.1 *Symbols*

The symbol used in a *block tree diagram* is a box which represents a *system* or a *block*. The *name* of the represented object should appear inside the box.

Each *block* (box) is connected downwards to its *sub-blocks* (boxes) to form a hierarchical tree, as the example given in Figure 7/Z.102 below:

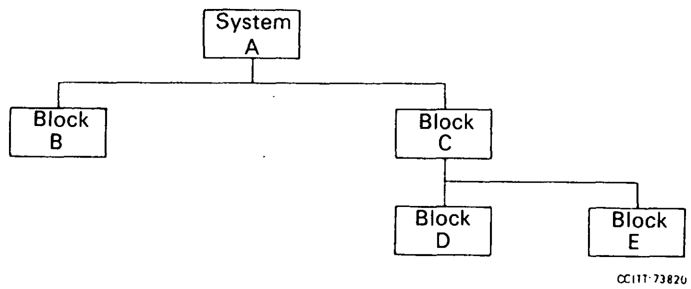


FIGURE 7/Z.102

Example of a block tree diagram

3.1.2 *Relationship to the SDL abstract syntax*

The structure shown has its equivalence in the *system definitions*, and in the *block substructure definitions* of the *blocks* in the *system.*

3.1.3 *Graphical conventions*

The tree should preferably be drafted so that the *blocks* at the same *level* appear beside each other in the representation.

As a *block tree diagram* of a large *system* will also be large, it may be suitable to split the diagram into several diagrams. This splitting should be such that the first diagram, having the *system* as the root, is chopped off so that a set of further partitioned *blocks* appears as not partitioned. In the following diagrams these *blocks* appear as roots. For example, in Figure 8/Z.102 the diagram from Figure 7/Z.102 is split in two diagrams.

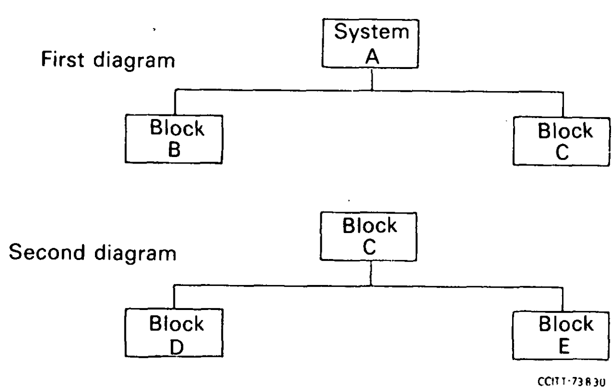


FIGURE 8/Z.102

Example of splitting a block tree diagram into several diagrams

3.2 *Block interaction diagram*

This diagram represents the *partitioning* of a *block* into *sub-blocks*, *sub-channels* and (new) *channels*. The diagram has basically the same form as the *block interaction diagram*, introduced in Recommendation Z.101, representing the partitioning of a *system* into *blocks* and *channels*.

3.2.1 *Symbols*

The symbols used to represent a *block interaction diagram* are shown in Figure 9/Z.102 below:

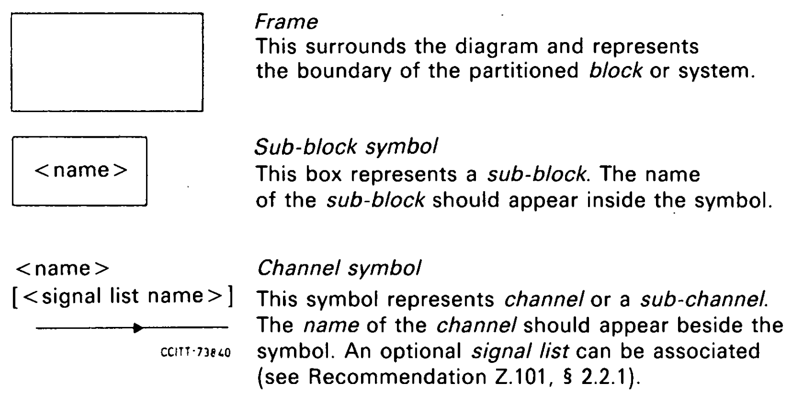


FIGURE 9/Z.102

Symbols used in a block interaction diagram

In addition to these symbols, *signal definitions* may appear in the diagram, using the SDL/PR syntax.

The rules for connecting the symbols are the same as for the *block interaction diagram* (see Recommendation Z.101), with the only exception that in the title of the diagram, it should be made clear that the diagram is a *block interaction diagram* for a *block*. A simple example of a *block interaction diagram* is given in Figure 10/Z.102 below:

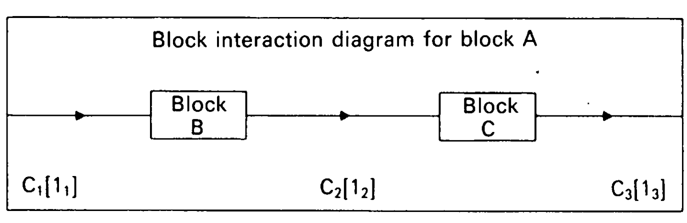


FIGURE 10/Z.102

Example of a block interaction diagram

3.2.2 *Relationship to the SDL abstract syntax*

A *block interaction diagram* represents a *block substructure definition*. The definitions contained in the *block substructure definitions* are represented by the *block* and *channel* symbols together with *signal definitions* given in *SDL/PR*.

3.2.3 *Graphical conventions*

The same graphical conventions described for the *interaction diagram*, defined in Recommendation Z.101 apply to the *block interaction diagram.*

In addition to this, it is often useful to describe several *levels* of *block partitioning* in one diagram. This is obtained by replacing a *block symbol,* in a diagram, by the *block interaction diagram* for that *block.* An example of this is given in Figure 11/Z.102 below:

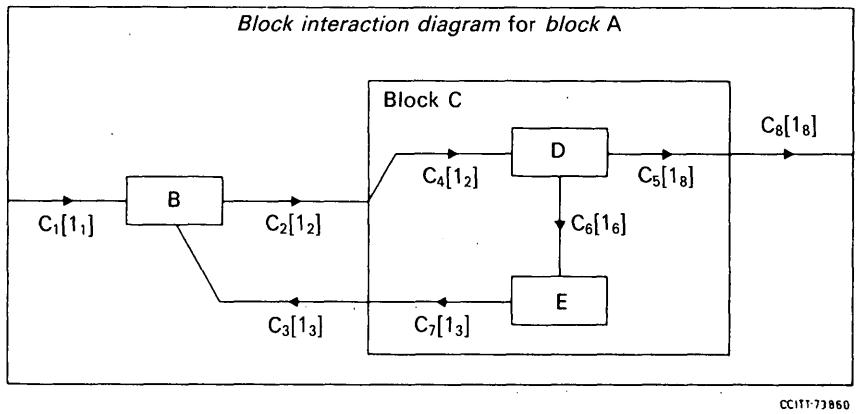


FIGURE 11/Z.102

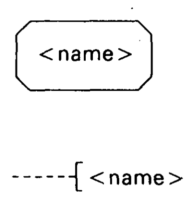
Example of a nested block interaction diagram

3.3 *Process tree diagram*

A *process tree diagram* describes the *partitioning* of a *process* into *sub-processes* and where these *sub-processes* are allocated.

3.3.1 *Symbols*

The symbols used to compose a *process tree diagram* are shown below in Figure 12/Z.102:



Process symbol
The symbol represents a *process*, whose *name* should appear inside the symbol.

Allocation symbol
The comment symbol is used to indicate where the *process*, or *sub-process*, is allocated. The *name* in the comment should be the name of the *block* to which the *process* is allocated.

FIGURE 12/Z.102

Symbols used in process tree diagram

Each process is connected downwards to its *sub-processes* to form a hierarchical tree, as in Figure 13/Z.102 below:

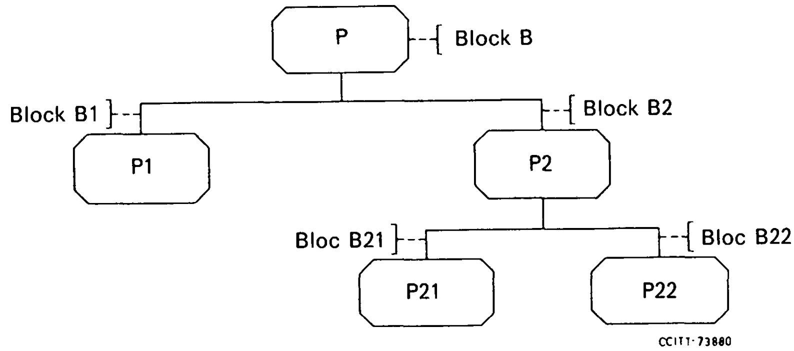


FIGURE 13/Z.102

Example of a process tree diagram

3.3.2 *Relationship to the SDL abstract syntax*

The diagram represents a *process substructure definition.* The *name* in the root symbol is the associated *process name*, and the *names* in the leaf symbols are the contained *sub-process names.* The *block names* in the allocation symbols are the associated *sub-block names.*

3.3.3 *Graphical conventions*

The tree should preferably be drafted so that *processes* at the same *partitioning level* appear beside each other in the diagram.

If a *processes tree diagram* is large, it may be suitable to split the diagram into several diagrams. This splitting should be such that the first diagram is chopped off so that a set of further partitioned *processes* appears as not partitioned. In the following diagrams these *processes* appear as roots. For example, in Figure 14/Z.102 the diagram from Figure 12/Z.102 is split into two diagrams.

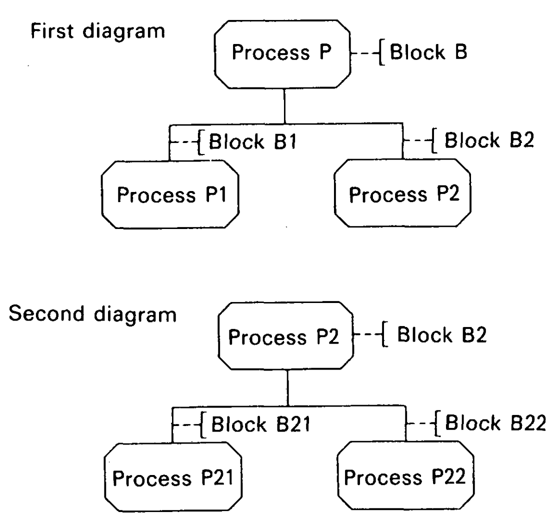


FIGURE 14/Z.102

Example of splitting a process tree diagram into several diagrams

3.4 *Channel substructure diagram*

This diagram represents the *partitioning* of a *channel* into sub-components. As the components are *blocks* and *channels* the diagram resembles a *block interaction diagram.*

3.4.1 *Symbols*

The symbols used to represent a *channel substructure diagram* are shown below in Figure 15/Z.102.

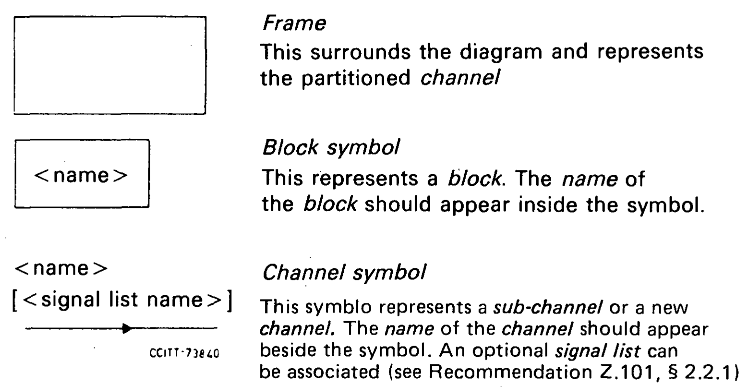


FIGURE 15/Z.102

Symbols used in the channel substructure diagram

In addition to these symbols, *signal definitions* may appear in the diagram, using the SDL/PR syntax.

The rules for connecting the diagram are the same for the *interaction diagram* (see Recommendation Z.101), with the only exception that the title of the diagram should make it clear that the diagram is a *channel substructure diagram.*

A simple example of a *channel substructure diagram* is given in Figure 16/Z.102 below:

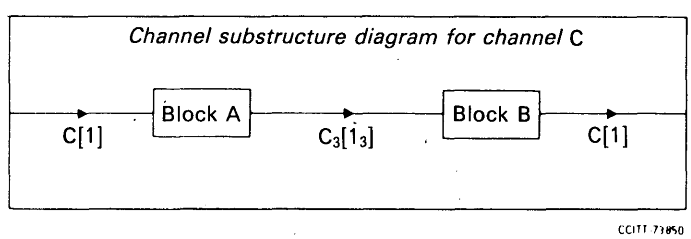


FIGURE 16/Z.102

Example of a channel substructure diagram

3.4.2 *Relationship to the SDL abstract syntax*

A *channel substructure diagram* represents a *channel substructure definition.* The definitions contained in the *channel substructure definition* are represented by the *block* and *channel* symbols together with the definitions in *SDL/PR* syntax.

The *channel* leading from the frame into the diagram represents the *incoming channel* and the *channel* terminating at the frame, represents the *outgoing channel.*

3.4.3 *Graphical conventions*

The same graphical conventions as described for the *interaction diagram* in Recommendation Z.101 apply to the *channel substructure diagram.*

4 SDL/PR

The following SDL/PR syntax is an addition to the syntax defined in Recommendation Z.101. The additions cover the representation of the structure and *partitioning* of a *system.*

Note that in the examples, *SDL/PR* keywords appear in capital letters.

4.1 *Block definition*

The *block definition* in *SDL/PR* is extended to optionally include the non-terminal "*Block substructure definition*".

4.1.1 *Syntax*

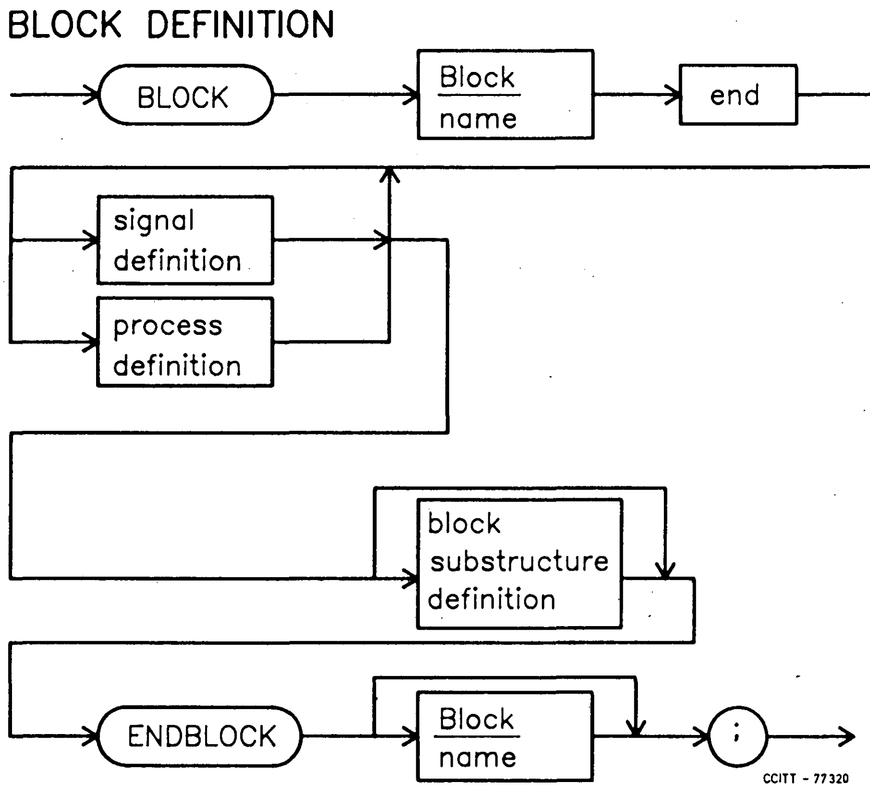


FIGURE 17/Z.102

Syntax diagram for block

4.2 *Block substructure definition*

The *block substructure definition* represents the *partitioning* of a *block* into *sub-blocks, sub-channels* and new *channels.*

4.2.1 *Syntax*

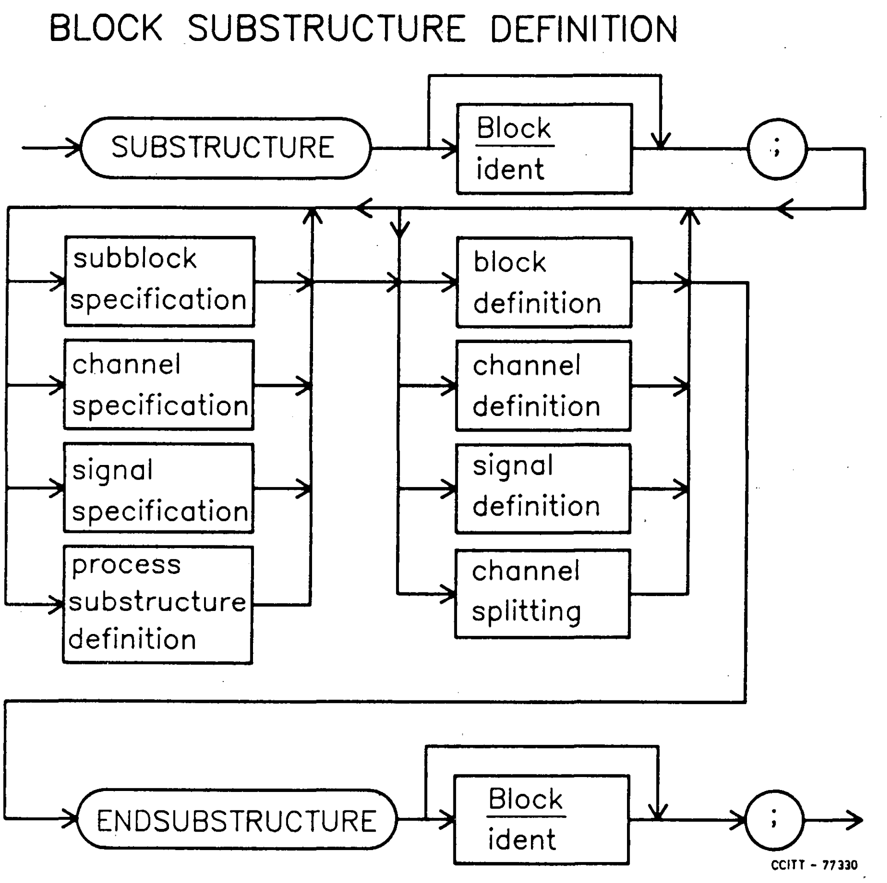
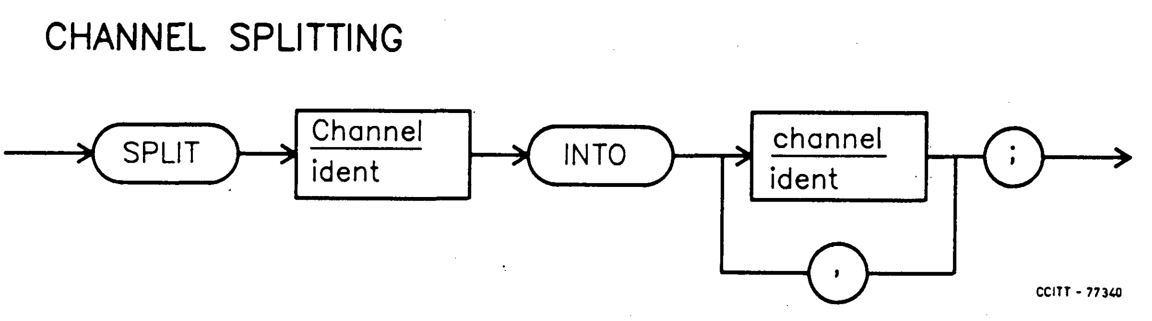
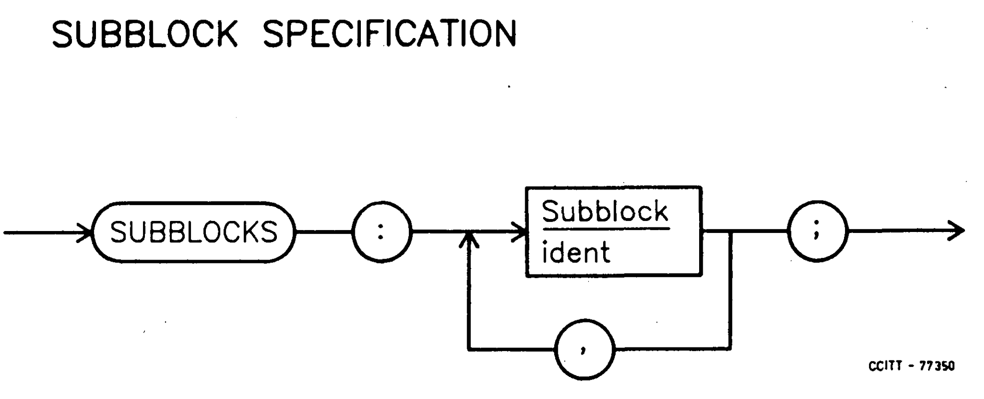


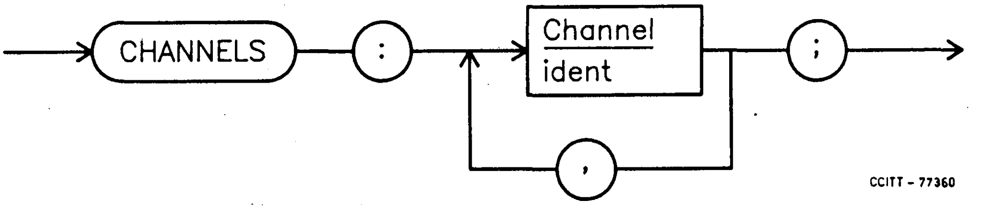
FIGURE 18/Z.102

Syntax diagram for block substructure diagram



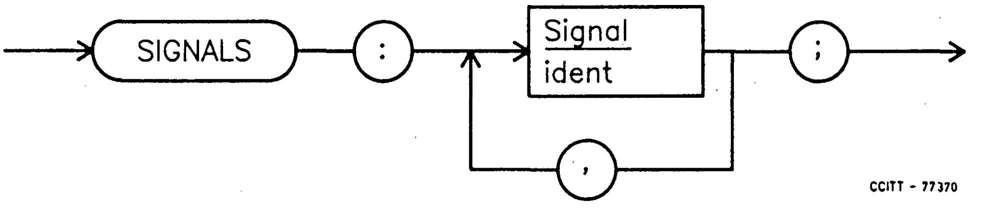


CHANNEL SPECIFICATION



CCITT - 77360

SIGNAL SPECIFICATION



CCITT - 77370

Example:

```
BLOCK b:

    ...
    SUBSTRUCTURE b;
        SUBBLOCKS  b1, b2, b3;
        CHANNELS   c1, c2, d1, d2, e;
        SIGNALS    s1, s2, s3;

    SPLIT c INTO c1, c2;
    SPLIT d INTO d1, d2;

    ...
        /* Channel definitions */
        /* Block definitions */
        /* Signal definitions */

    ENDSUBSTRUCTURE;

ENDBLOCK b;
```

4.2.2 *Relationship to the SDL abstract syntax*

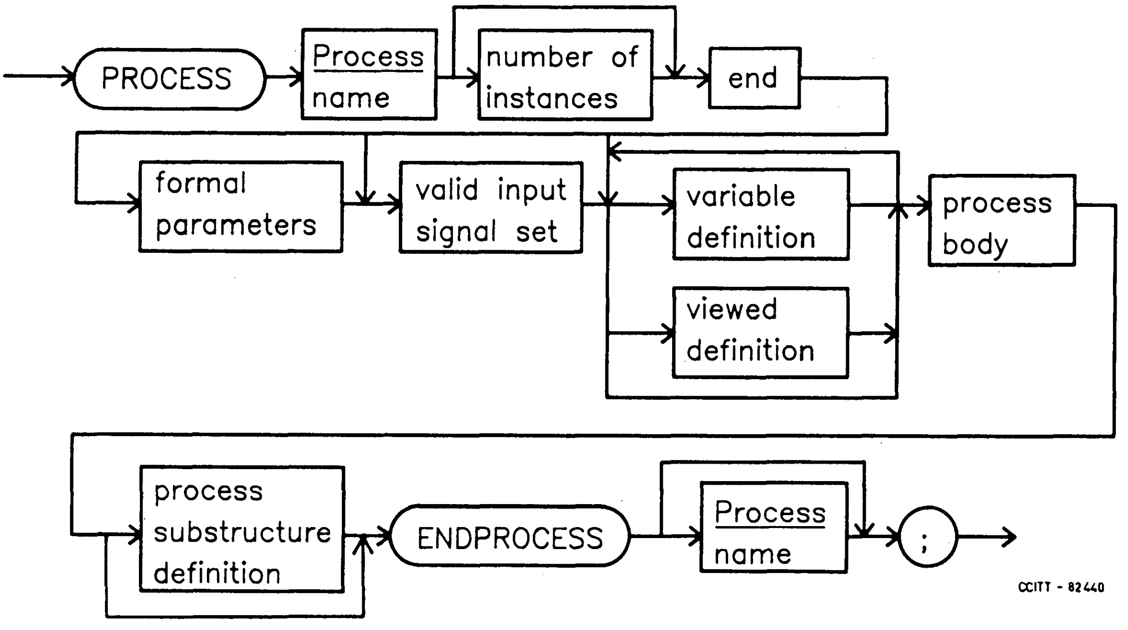
The syntax represents the *block substructure definition* and the internal part block definition in the *abstract syntax*. The *names* of *blocks*, *signals* and *channels* are references to the contained *block definitions*, *signal definitions* and *channel definitions*.

4.3 *Process substructure*

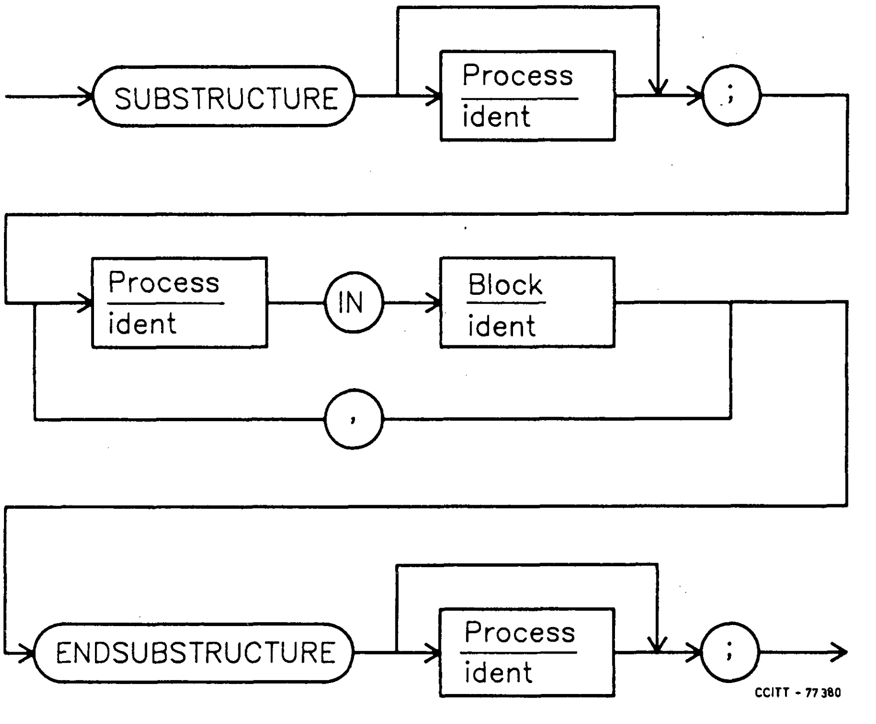
This syntax represents the *partitioning* of a *process* into *sub-processes*, and the allocation of these into *sub-blocks*. The *partitioning* may be shown both in the *process definition* and the *block substructure definition*.

4.3.1 *Syntax*

THE PROCESS DEFINITION IS EXTENDED AS FOLLOWS:



THE PROCESS SUBSTRUCTURE DEFINITION IS A FOLLOWS:



The following example shows the *process substructure* as a part of a *process definition*:

Example:

```
...
    SUBSTRUCTURE p1;
        p2 IN b2;
        p3 IN b1;
    ENDSUBSTRUCTURE;
...
ENDPROCESS p1;
```

The next example shows the *process substructure* as a part of a *block substructure definition*:

Example:

```
...
    SUBSTRUCTURE b;

        SUBSTRUCTURE p;
            p1 IN b2;
            p2 IN b1;
        ENDSUBSTRUCTURE

    ...
    ENDSUBSTRUCTURE;
ENDBLOCK b;
```

4.3.2 *Relationship to the SDL abstract syntax*

The syntax construction represents the *process substructure definition* in the abstract syntax. The associated *process name* is the *name* in the starting clause and the contained set of *process names*, each associated with a *block name*, are the *names* separated by the "IN" keyword.

4.4 *Channel substructure*

This syntax represents the *partitioning* of a *channel* into a set of *channels* and *blocks.*

4.4.1 *Syntax*

The syntax for a *channel* is extended to contain the non-terminal "*channel substructure*" as an optional part.

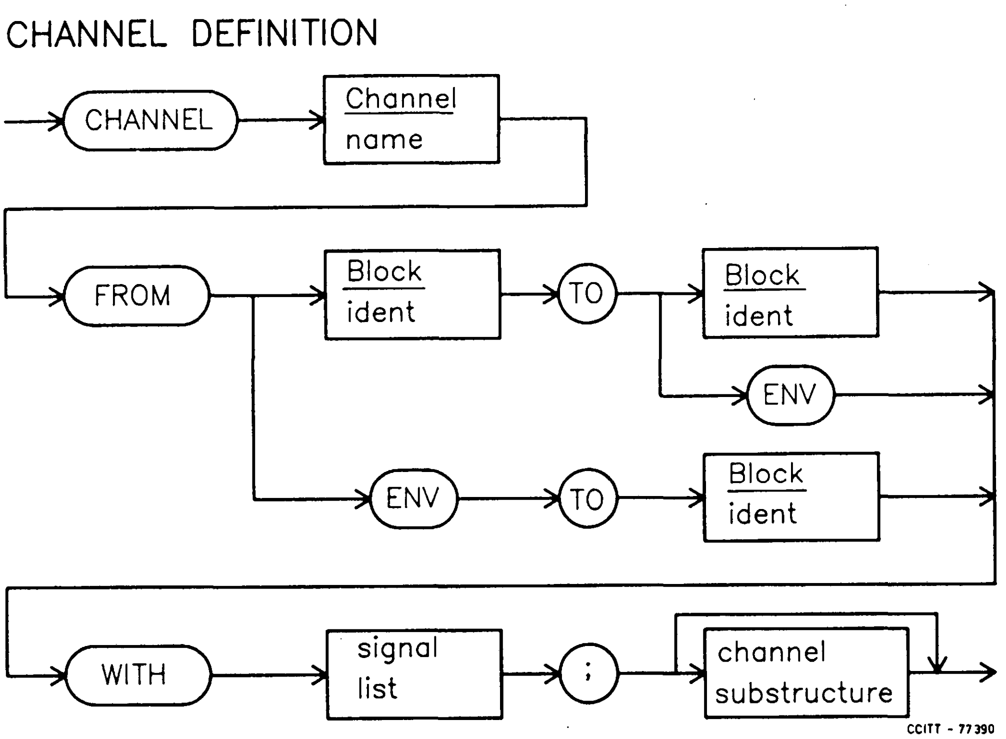


FIGURE 19/Z.102 (1 of 4)

Syntax diagrams for channel substructure

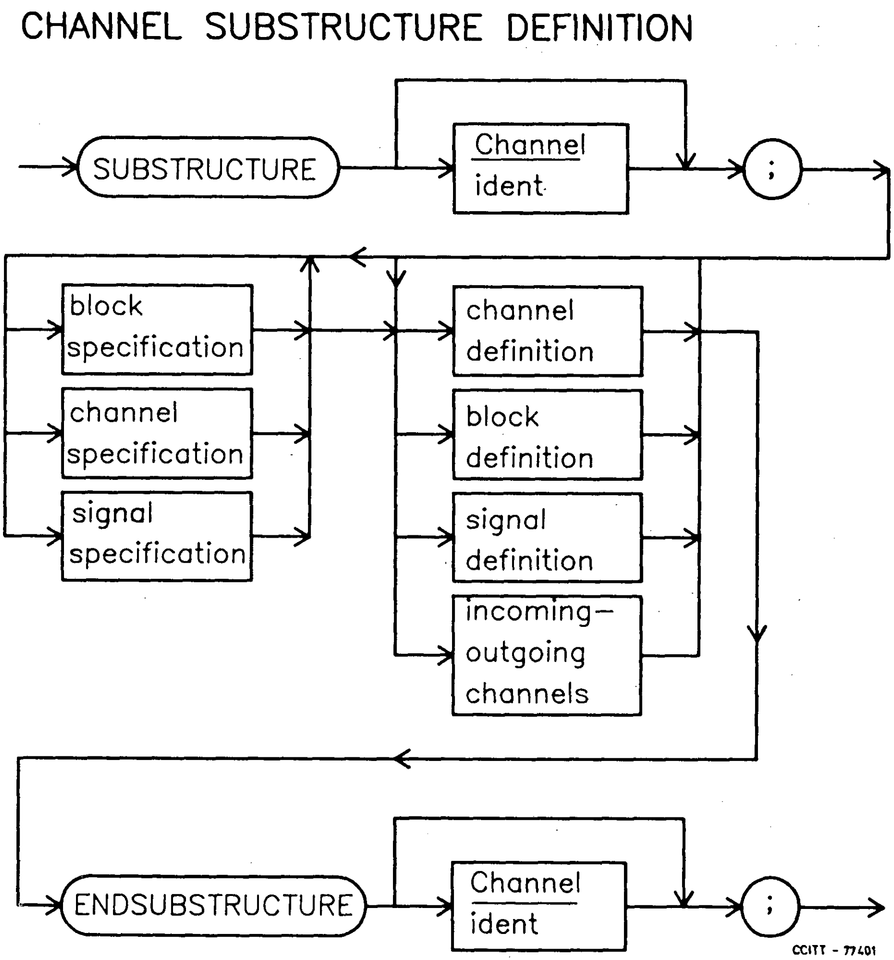


FIGURE 19/Z.102 (2 of 4)

Syntax diagrams for channel substructure

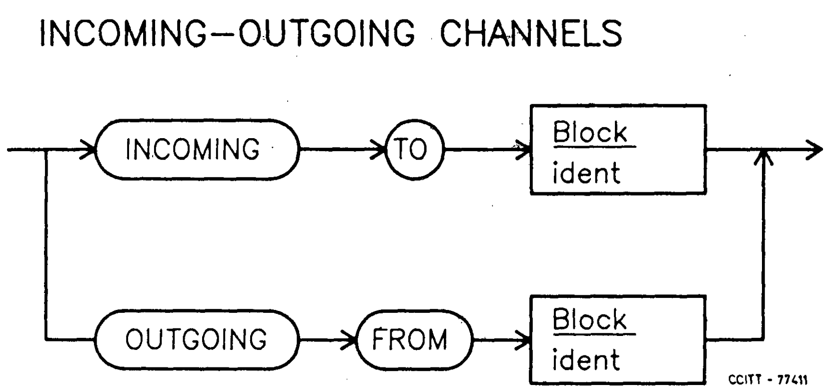


FIGURE 19/Z.102 (3 of 4)

Syntax diagrams for channel substructure

BLOCK SPECIFICATION

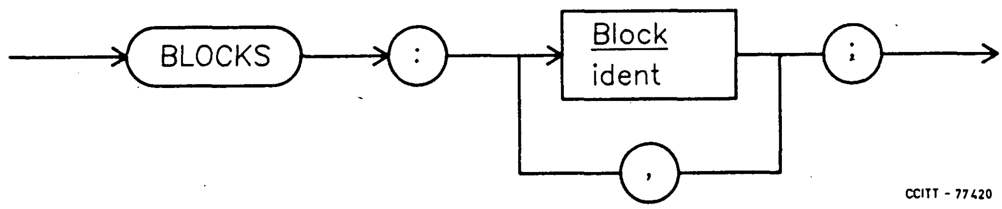


FIGURE 19/Z.102 (4 of 4)

Syntax diagrams for channel substructure

The *channel substructure* is a set of references to the components and additional definitions of the substructure.

Example:

```
CHANNEL c FROM b TO d;
    SUBSTRUCTURE
        INCOMING TO e;
        OUTGOING FROM f;
        BLOCKS: e,f;
        CHANNEL c1 FROM e TO f WITH s1,s2,s3;
        BLOCK e;
            ...
        ENDBLOCK e;
        BLOCK f;
            ...
        ENDBLOCK f;
    ENDSUBSTRUCTURE;
```

4.4.2 *Relationship to the SDL abstract syntax*

The syntax represents the *channel substructure definition* in the *abstract syntax.* The *block, channel, signal* and *data identifiers* given in the syntax are references to the contained definitions.

Recommendation Z.103

FUNCTIONAL EXTENSIONS TO SDL

1 Introduction

This Recommendation defines a number of additional concepts and shorthand notations in the SDL. The basis for these additions are the basic SDL, as defined in the Recommendation Z.101. The intention with these additions is to provide the users of SDL with convenient concepts and shorthand notations.

The following concepts and shorthand notations are defined in this Recommendation:

Procedures

These give a way to represent a portion of a *process graph* by one element, which may be referred to several times. The detailed *behaviour* of the *procedure* is defined elsewhere, inside or outside the *process graph. Procedures* may be used to support the use of structured-design methods with SDL, by permitting the decomposition of a *process graph* into a hierarchy of sections. The concept is similar to the procedure concept normally appearing in programming languages.

Import and Export of values

This is a shorthand notation for the *signal* interworking between *process instances* when they are to share a *value* of a *variable* owned by one of the *processes.*

Enabling condition

This is a shorthand notation to avoid «state explosion» when the reception or saving of a set of *signals* is conditional.

Continuous signal

This is a shorthand notation to represent the *signal* interwork when a continuous condition, external to the *process,* is observed.

Macro

This is a syntactic method for the user to define a shorthand notation. A *macro* is the definition of a composite syntax element in terms of other syntax elements already defined. A *macro* has no semantics of its own.

Option

This is a syntactic facility to represent several, alternative behaviours in one diagram, or linear text. It has to be decided, before the diagram or linear text is interpreted, which alternative offered by an *option* should be chosen.

This Recommendation also defines an extension graphic syntax of SDL, that is the use of pictorial elements in *state* symbols.

2 Procedures

A *procedure* is a means of giving a name to an assembly of items and representing this assembly by a single reference. The rules for *procedures* impose a discipline upon the way, in which the assembly of items is chosen, and limit the scope of the name of *data items* and *signals.*

Procedures are intended to:

a) permit the structuring of a *process graph* into several levels of detail;

b) maintaining the compactness of specifications by allowing a complex assembly of items which may be regarded in isolation to be represented by a single item;

c) allow commonly used assemblies of items to be pre-defined and used repeatedly.

Procedures are defined by means of *procedure definitions.* The *procedure* is invoked by means of a *procedure call* referencing to the *procedure definition.* Parameters are associated with a *procedure call:* these are used both to pass *values,* and also to control the scope of *data* and *signals* for the *procedure* execution. Which *signals* and *data items* are affected by the interpretation of a *procedure* is controlled by the parameter passing mechanism.

2.1 *Common Language Model*

The definitions following are to be considered strictly as additions to what is defined in the Recommendation Z.101.

2.1.1 *An Introduction to the Common Model*

A *procedure* is a section of a *process graph* which may be regarded in isolation. It has a single entry point and a single exit point. A *procedure call* may appear wherever a *task* may appear in a *process graph* or a *procedure graph.* A *procedure* may contain *states.* All *data items* used in a *procedure* must be defined within the *procedure definition.* If a *data item* is defined as a *formal parameter,* it uses *values* of *data items* in the calling environment and/or will give *values* to *data items* in the calling environment, otherwise the item is local to the *procedure,* and has to be defined in the *procedure definition.*

A *procedure* is interpreted only when a *process instance* calls it, and is interpreted by that *process instance.* When a *procedure call* is interpreted, the *procedure graph* must be interpreted before continuing the interpretation of the *transition* in which the *call* appears. This means that the calling *process instance's input port* and *valid input signal* set is used whilst interpreting the *procedure.* All *signals* referenced within a *procedure* must be *named* as *formal parameters* of the *procedure.* All *signals* in the *valid input signal* name set of the calling *environment* which are not referenced within the *procedure* must also be *named* as *parameters* of the *procedure*, i.e. the *additional-save-set*. This allows the *procedure* to automatically *save* any other *signals* which may arrive in the *input port*. If this were not done the introduction into a *transition* of a *procedure call* to a *procedure* containing a *state* would have hidden side effects of losing *signals* (by implicit «null transitions») which arrived while a *process instance* was interpreting a *procedure*.

A *procedure definition* is given by a *procedure graph*, and may appear wherever a *data type definition* may appear.

A *procedure graph* follows the same rules as a *process graph*, with the following exceptions:

- a *start node* is replaced by a *procedure start node*;
- a *stop node* is replaced by a *return node*;
- only the *data items, formal parameters* and *signal names* which are declared within the *procedure* are visible to the *process instance* while interpreting the *procedure*, i.e. a *procedure* cannot refer to any *data item* or *signal* which is outside the *procedure* unless it is declared as a *formal parameter* (*signals* necessarily belong to the calling environment of the *procedure*, and all *signals* which are *input, saved* or *output* in the *procedure* must be *declared*);
- the same *signal name actual parameter* may not be mapped onto more than one *signal name formal parameter*;
- in the *abstract syntax* the set of signals to be *saved* is passed as *parameters* of the *procedure call*. However, in the concrete syntax, all *state nodes* contained within a *procedure* have an implicit *save signal set* containing all *input signals* which are not declared as *formal parameters* in the *procedure*.

2.1.2 *Abstract Syntax*

To the *abstract syntax* defined in Z.101 the following additions are made:

System Definition

A *system definition* may also contain one or more *procedure definitions*.

Block Definition

A *block definition* may also contain one or more *procedure definitions*.

Process Definition

A *process definition* may also contain one or more *procedure definitions*.

Process Graph

A *process graph* may also contain a *procedure call node*.

A *transition string* may also be a *procedure call node* followed by a *transition string*.

A *procedure call node* contains a *procedure identifier* and an *actual parameter list*.

Procedure Definition

The *procedure definition* is associated with a *procedure name* and contains a *formal parameter list*, an *additional-save-set*, and a *procedure graph*, and may contain *procedure definitions* and *data definitions*.

The *formal parameter list* may be empty. Each *formal parameter* in the list must be attributed with one of the attributes *IN, IN/OUT* or *SIGNAL*.

The *names* of the *formal parameters* attributed with *IN* or *IN/OUT* may not be used in *variable definitions* contained in the *procedure definition*; and the set of *formal parameters names* attributed with *SIGNAL* must contain all the *signal names* referenced in the *procedure graph*.

Procedure Graph

A *procedure graph* is a graph whose *nodes* are connected by directed *arcs*. An *arc* entering a *node* is called an incoming *arc* and an *arc* exiting the *node* is called an outgoing *arc*. A *node* having an *arc* as incoming *arc* follows the *node* having the same *arc* as an outgoing *arc*.

The following types of *nodes* are allowed on a *procedure graph*:

State node
Input node
Task node
Output node

Decision node
Procedure Call node
Procedure Start node
Return node
Create Request node

Every *node* of the graph has a *name* and a *type. State nodes* within a *procedure graph* have different *names.*

The following rules define the connectivity of a *procedure graph* :

- Each *procedure graph* contains one and only one *procedure start node.* The *procedure start node* is followed by a *transition string.* The *start node* does not follow any other *node.*
- A *transition string* can be one of the following:
  a) null followed by either a *state node* or a *return node,*
  b) an action string followed by a *transition string,*
  c) a *decision node.*
- An action string can be one of the following:
  a) a *task node,*
  b) an *output node,*
  c) a *create request node.*
- A *decision node* is followed by two or more *decision arcs.*
- A *decision arc* is a named *arc* followed by a *transition string.*
- A *state node* is followed by one or more *input nodes.*
- An *input node* is followed by a *transition string.*
- The *return node* has no *nodes* following it.
- Each graph has at most one *return node.*
- Each *node* is reachable from the *procedure start.*

A *call node* contains a *procedure identifier,* an *additional-save-list* and an *actual parameter list.* In this list there must be one *actual parameter* for each of the *formal parameters* in the referenced *procedure definition.* Each *actual parameter* must match the *type* of the corresponding *formal parameter.* Any *signal actual parameter name* may not appear more than once in the *actual parameter list.*

Each *actual parameter* for which the corresponding *formal parameter* has the attribute *IN/OUT* must be a *variable name.*

The *additional-save-set* is a possibly empty set of *signal identifiers.*

2.1.3 *Interpretation*

The following additions are made to the interpretation rules of Recommendation Z.101:

Process

A *call node* is interpreted as the interpretation of the *procedure start node* of the *procedure definition* referenced by the *name* of the *node,* then the *node* following the *call node* is interpreted.

The *additional-save-set* is given the *value* of all *signal identifiers* in the *valid input signal set* of the calling *process,* which do not appear as *actual parameters* of the *node.*

Procedure

When a *process* interprets a *call node* referencing the *procedure definition* containing the *procedure graph,* the *procedure graph* is interpreted. The *nodes* of the *procedure graph* are interpreted in the same manner as the equivalent *nodes* of the *process graph,* with the following exceptions and additions:

- Each *formal parameter* attributed with *IN* denotes a typed *variable.* This *variable* is local to the *procedure* and is created when the *procedure start node* is interpreted and ceases to exist when the *return node* is interpreted.
- Each *formal parameter* attributed with *IN/OUT* denotes a *synonym name* for the *variable* which is given as *actual parameter.* This *synonym name* is used throughout the interpretation of the *procedure graph* when referring to the *value* of the *variable* or when assigning a new *value* to the *variable.*

Procedure start node

The interpretation of a *procedure start node* involves:

- A local *variable* is created for each *formal parameter* attributed with *IN*, having the *name* and *data type* of the *formal parameter*. The *variable* is assigned the *value* of the *actual parameter*, which may be undefined.
- Each *variable name* given in the *formal parameters* attributed with *IN/OUT* is used as a *synonym name* for the *variable name* given as *actual parameter*. The *variable* represented by the *synonym name* is evaluated once only at the *procedure start node*, and not at each use of the *formal parameter* in the *procedure*.
- Each *signal name* given in the *formal parameters* attributed with *SIGNAL* is used as a *synonym name* for the *signal identifier* given as *actual parameter*.
- The *node* following the *procedure start node* is interpreted.

State node

The *state node* is interpreted in the same way as in a *process graph* with the exception that the *save-signal-set* presented to the *input port* is union of the *additional-save-set* and the *save-signal-set* of the *process* or *procedure* which called the current *procedure*.

2.2 *SDL/GR*

The addition of the *procedure* concept into SDL causes the addition of one new symbol in the *process diagram*: the *call* symbol.

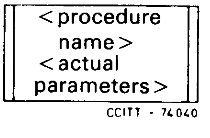
The *procedure definition*, is in the graphic syntax represented by a *procedure diagram*. This diagram is similar to the *process diagram*, except for the *procedure start symbol* and the *return symbol*.

In the following only the additions, required by introducing *procedure*, to the graphic syntax as defined in Recommendation Z.101, are given.

2.2.1 *Process diagram*

2.2.1.1 *Symbols*

The following symbol is used to represent the *procedure call*:



The symbol may appear in a *process diagram* wherever a *task* symbol is allowed.

The *procedure name* and the *actual parameters* are given using the SDL/PR syntax.

FIGURE 1/Z.103

The procedure call symbol

2.2.1.2 *Relationship to the SDL abstract syntax*

The *procedure call symbol* represents the *procedure call node* in a *process graph*.

The *actual parameters* given in the symbols represents the *actual parameters* attached to the *procedure call node*. Any *valid input signal* not given as *actual parameter* is a member of the *additional-save-set* for the *procedure call node*, denoting that that *signal* should be *saved* throughout the *procedure* execution.

Return node

The interpretation of the *return node* involves:

- all *variables* created by the interpretation of the *procedure start node* will cease to exist;
- all *synonym names* established by the interpretation of the *procedure start node* will cease to exist;
- interpreting the *return node* completes the interpretation of the *procedure start node.*

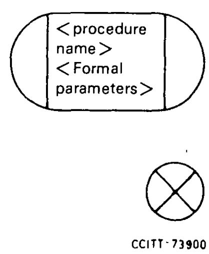
2.2.2 *Procedure diagram*

A *procedure diagram* is similar to a *process diagram*, the only differences being that the *start symbol* is replaced by the *procedure start symbol*, and the *stop symbol* is replaced by the *return symbol.*

In the following only the additional syntax is defined.

2.2.2.1 *Symbols*

In Figure 2/Z.103 below two additional symbols are defined:



Procedure start symbol
The *procedure name* and *formal parameters* are given using the *SDL/PR* syntax.

Return symbol

FIGURE 2/Z.103

Additional symbols for procedure diagram

2.2.2.2 *Relationship to the SDL abstract syntax*

The *procedure diagram* represents the *procedure definition* in the SDL *abstract syntax.*

The symbols common with the *process diagram* have the same relations to the *abstract syntax* as when appearing in a *process diagram.* The additional *procedure start symbol* and the *return symbol* represent respectively the *procedure start node* and the *return node.*

The *parameters* attached to the *procedure start symbol* represents the *formal parameters* attached to the *procedure start node.*

2.2.2.3 *Graphical conventions*

Generally the same graphic conventions as for the *process diagram* apply. The conventions for the *procedure start symbol* and for the *return symbol* are the same as for the *process start symbol* and the *stop symbol* respectively.

Wherever a *return symbol* appears it represents the *return node* (multiple appearance of a *return symbol*).

2.3 *SDL/PR*

SDL/PR syntax for a *procedure definition* is given below. The *procedure definition* may appear wherever a *data type definition* may appear in the syntax. The syntax is similar to that of the *process definition.*

Below is given only the additions to the syntax already defined in Recommendation Z.101.

2.3.1 *System*

2.3.1.1 *Syntax*

The syntax of *procedure definitions* is added to the syntax of *system*:

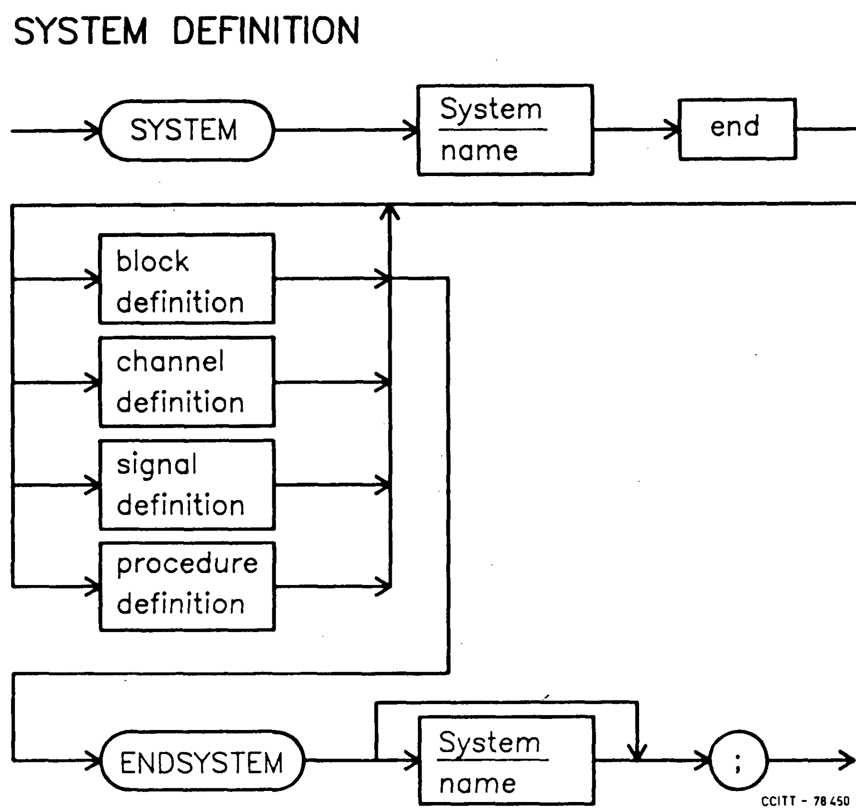


FIGURE 3/Z.103

Syntax diagram for system

2.3.2 *Block*

2.3.2.1 *Syntax*

The syntax of *procedure definitions* is added to the syntax of *block*:

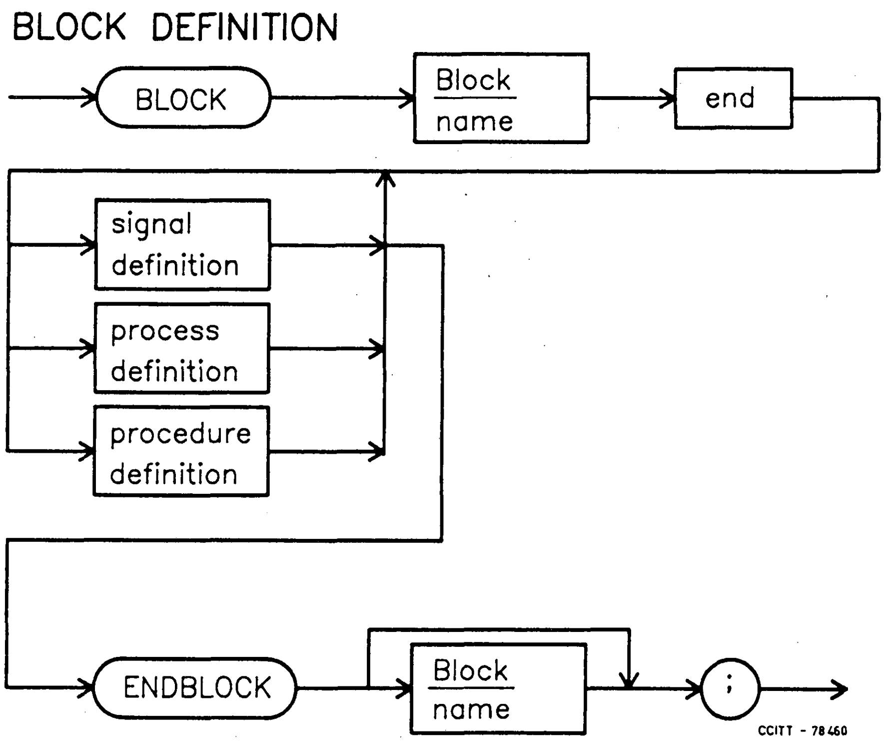


FIGURE 4/Z.103

Syntax diagram for block

2.3.3 *Process*

2.3.3.1 *Syntax*

The syntax of *procedure definition* is added to the syntax of *process*, and the syntax of *procedure call* is added to the syntax of a *transition string*:

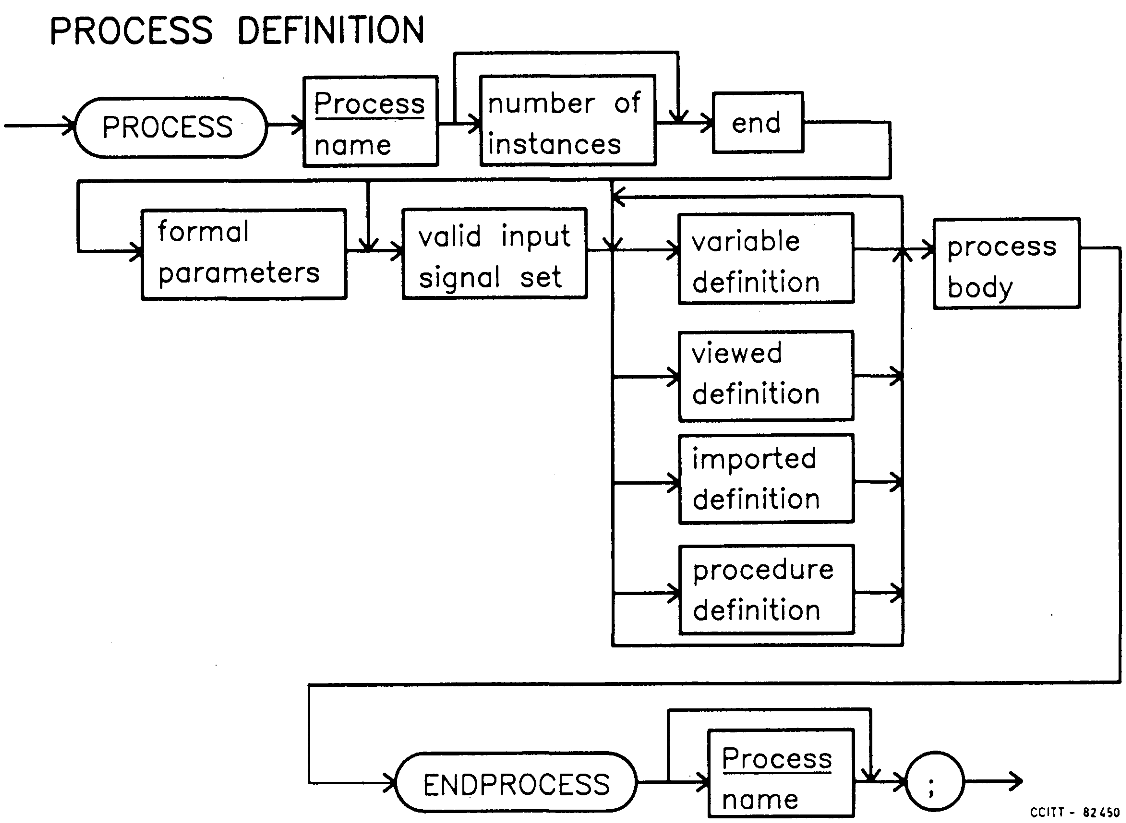


FIGURE 5/Z.103

Syntax diagram for process

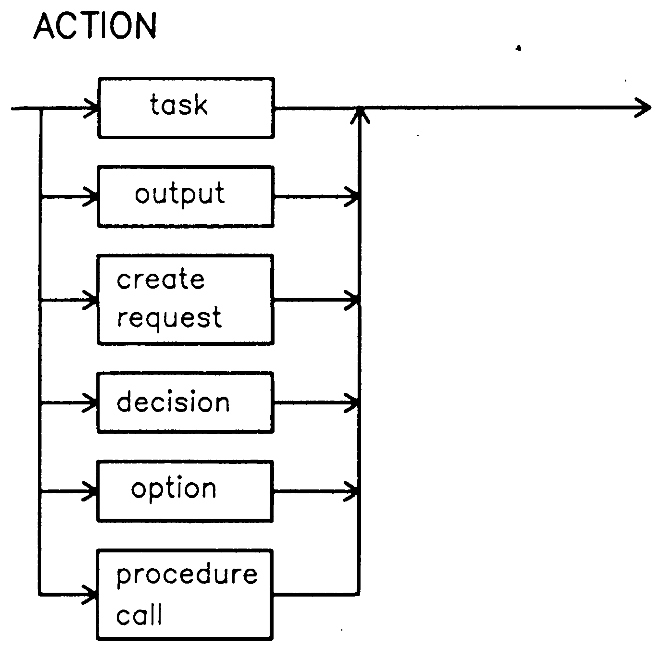


FIGURE 6/Z.103 (1 of 2)

Syntax diagram for action

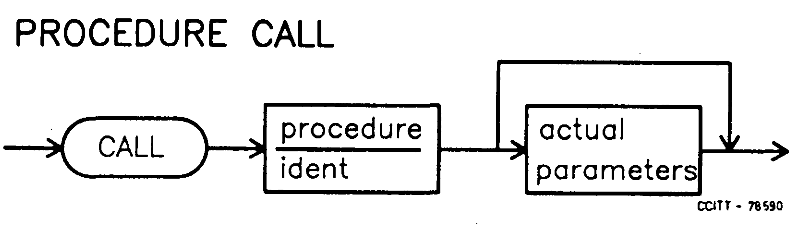


FIGURE 6/Z.103 (2 of 2)

Syntax diagram for procedure call

2.3.3.2 *Relationship to the SDL abstract syntax*

The *procedure call* in a *transition* represents the *call node* in the *abstract syntax*, and the *actual parameters* represent the *actual parameters* attached to the *call node.*

Each *actual parameter* given has to conform with the *data type* of the corresponding *formal parameter* in the referenced *procedure definition.*

2.3.4 *Procedure*

2.3.4.1 *Syntax*

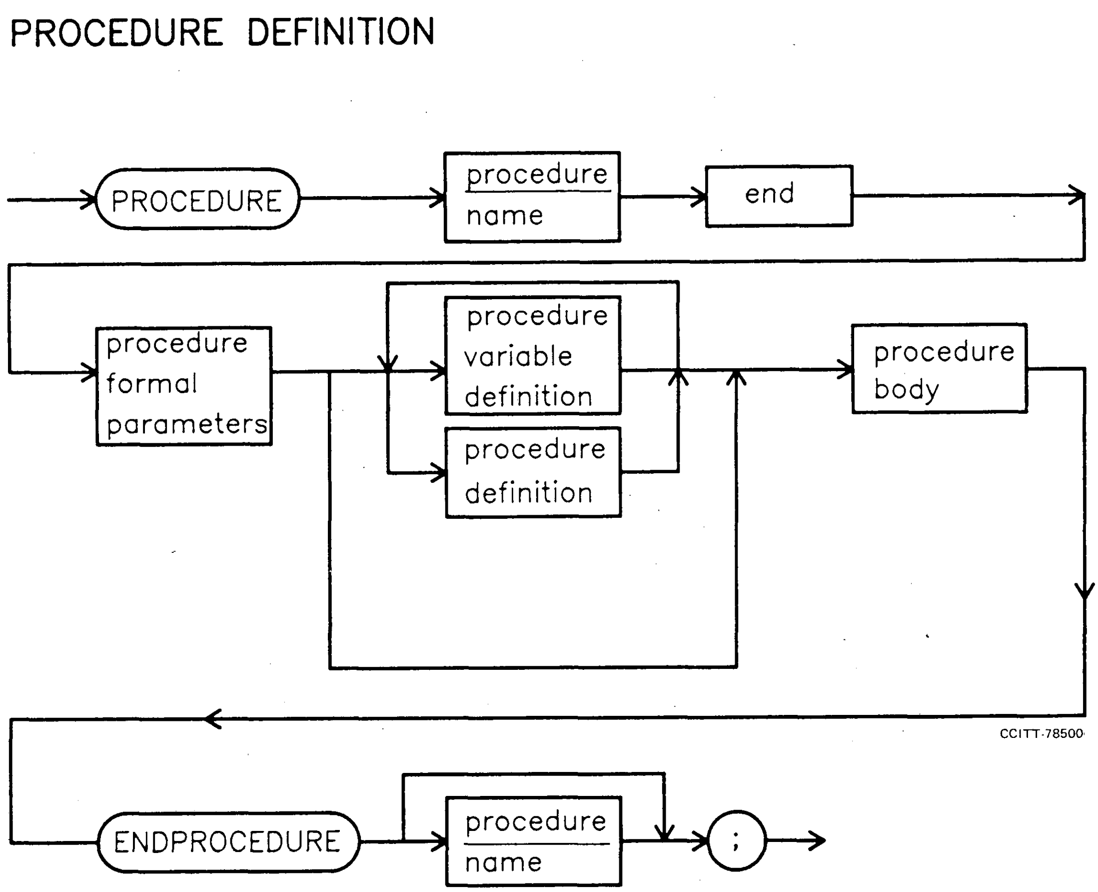


FIGURE 7/Z.103 (1 of 5)

Syntax diagrams for procedure

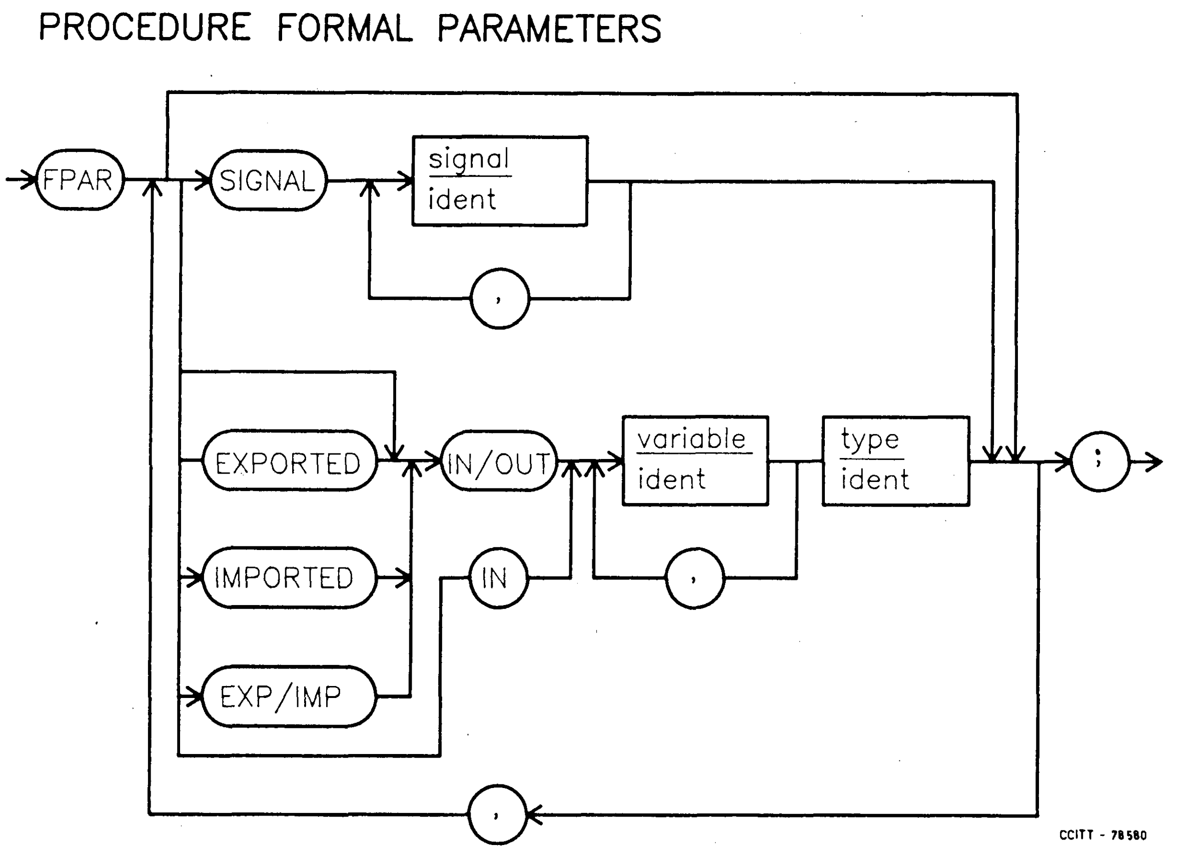


FIGURE 7/Z.103 (2 of 5)

Syntax diagrams for procedure

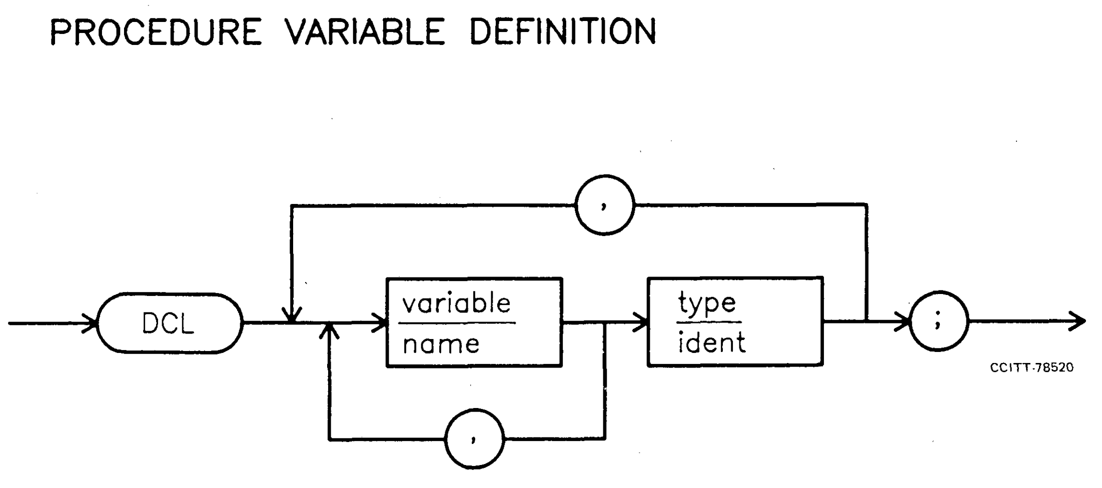


FIGURE 7/Z.103 (3 of 5)

Syntax diagrams for procedure

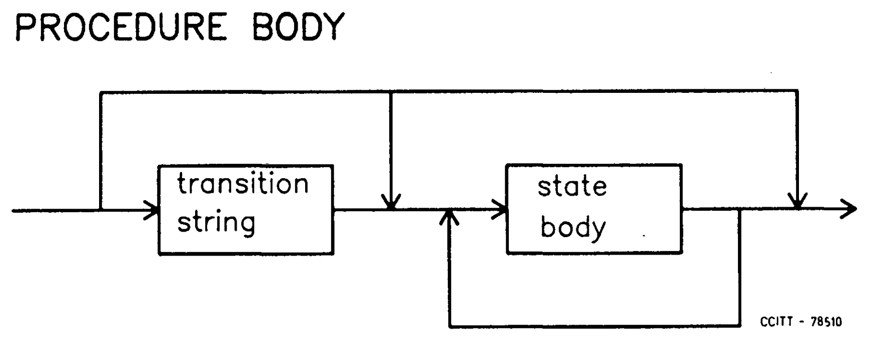


FIGURE 7/Z.103 (4 of 5)

Syntax diagrams for procedure

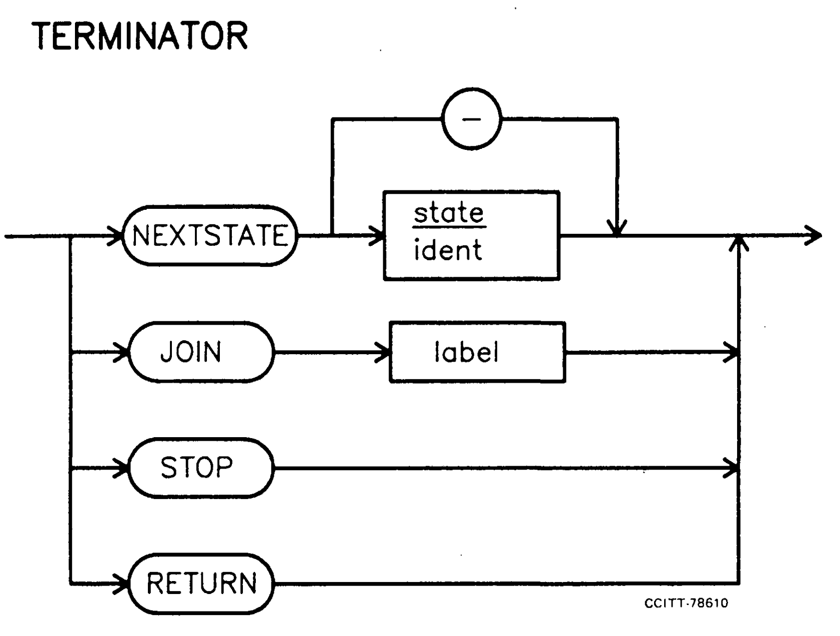


FIGURE 7/Z.103 (5 of 5)

Syntax diagrams for procedure

2.3.4.2 *Relationship to the SDL abstract syntax*

The *procedure* syntax diagram represents the *procedure definition* of the *abstract syntax*. The *formal parameters* in the syntax diagram represent the *formal parameters* attached to the *procedure start node*.

The syntactic constructs allowed within a *procedure* and common with those allowed in a *process* have the same relations to the *abstract syntax* as when they appear in a *process*. The *procedure start node* is implied and is followed by the first syntactic construct of the *procedure*, and the *return* statements, represent the *return node*.

3 Composite operations for communication between SDL processes

A *composite operation* is a standard shorthand notation for assemblies of SDL concepts. The *composite operations* are defined in terms of already present SDL concepts and do not add to the semantics of the language. Thus they do not change the *abstract syntax* on its interpretation. They are introduced for the convenience of the users of SDL.

The *composite operations* defined here provide some alternative methods of communication between SDL *process instances.* While they appear significantly different from the normal *signal* exchange mechanism of SDL to the user, they are in fact based upon the normal SDL semantics.

The *composite operations* provide:

a) sharing *values* of *data items* between *processes*, even if they are allocated in different *blocks* ;

b) the capacity to temporarily enable or disable the reception of particular *signals* without the need to show this exactly with separate *states*;

c) *continuous signals*, for which *transitions* are caused by a change of a *value* of the *signal.*

Each of the *composite operations* is given a syntax of its own and are defined in terms of "normal" SDL concepts. Thus the composite operations involve for the user implied or hidden *states, signals* and *procedures.* The *composite operations* are defined in §§ 3.1 to 3.3 using illustrative names for the normal SDL symbols. When any *composite operation* is used in an SDL representation, the implied SDL symbols have unique implied names. These implied names are so chosen that there are no clashes between different occurrences of the *composite operation* or other names in the SDL representation. The SDL user is thus free to employ these names for other purposes should he wish to do so.

As the *composite operations* are defined in other SDL terms they may in some situations lead to side effects, these are discussed in the SDL User Guidelines.

3.1 *Imported and exported values*

In SDL a *variable* is always owned by, and local to, a *process instance.* Normally the *variable* is visible only to the *process instance* which owns it, though it may be *declared* as a *shared value* (see Recommendation Z.101) which allows other *process instances* in the same *block* to *view* the *value* of the *variable.* If a *process instance* in another *block* needs to *view* a *variable,* a *signal* interchange with the *process instance* owning the *variable* is needed.

The following paragraphs describe a standard method, and a shorthand notation for doing this. The technique will be called *imported values,* since communication is by means of copies of the *value* of the *variable,* only the owning *process instance* having access to the *variable* itself. The technique may also be used to export *values* to other *process instances* within the same *block,* in which case it provides an alternative to the use of *shared values.* This alternative is necessary when either it is likely that the *block* will be decomposed such that the *process instances* concerned are in different *sub-blocks* or when the *values* are used in *enabling conditions* (see § 3.3).

The *process instance* which owns a *variable* whose *values* are made available to other *process instances* is called the *exporter* of the *variable.* Other *process instances* which use the *variable* are known as *importers* of the *variable.*

Access to the value of the *variable* is obtained by an exchange of *signals.* The *importer* sends a *signal* to the *exporter,* and waits for the reply. In response to this *signal* the *exporter* sends a *signal* back to the *importer* with the *value* the *variable* had when the last *export operation* was made.

The *variables* whose *values* are *imported* and *exported* are defined as such in their *variable definitions.* These *variables* are also identified by *definitions* in the *channels* carrying the implied *signal* interchange.

The definition of a *variable* as *exported* produces an implicit definition of a copy of it to be used in the *import* and *export* operations.

The *process instance* discloses the *value* of a *variable* defined as *exported* by means of the statement:

EXPORT (x) where x is the *name* of the *variable.*

The EXPORT (x) causes the storing of the current *value* of x into the implicit copy and the sending of *signals* to *process instances* awaiting an *enabling condition* (§ 3.2) or *continuous signals* (§ 3.3).

The EXPORT operation can be made in conjunction with the manipulation of the *variable* or independently of it, e.g:

EXPORT (x): = <expression>;

or

EXPORT (x);

The construct may only appear in a *task.*

The access from another *instance* may only be done by means of the syntactic construct:

IMPORT(x,Pid) where x is the *name* of the *variable*, and Pid is a reference to the owning *process instance.*

This construct may appear in a *task* and in a *decisions.*

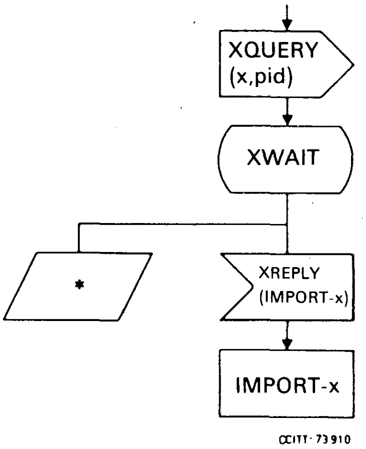
3.1.1 *Definition*

3.1.1.1 *Import operation by importer*

The *import* operation is modelled in the *importer's process graph* by the following sequence of *nodes* :

- an *output node*, by *name* XQUERY, carrying a reference to the *name* of the *variable* whose *value* should be accessed, and addressed to the *process instance* which owns the *variable* (the *exporter*);
- a *state node* with a *save signal set* including all *signals* except XREPLY;
- an *input node* for the *signal* XREPLY returning the *value* of the value of the requested *variable* ;
- an implicit *variable* of the same *type* as the *exported variable* is assigned the *value* associated to the XREPLY *signal.* This implicit *variable name* replaces the *import* operation. At each occurrence of the construct operation, a new implicit variable is used.

The sequence of operations defining the *import* operation is explained below using the graphic SDL syntax. The *IMPORT*(x,pid) statement will correspond to the following *transition string.*



Note – This diagram is explanatory. A user of the *import* operation would only write: IMPORT (x, pid).

FIGURE 8/Z.103

Import operation explained in SDL/GR

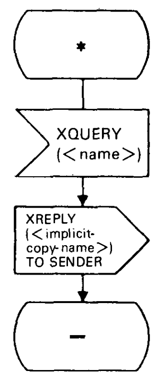
3.1.1.2 *Import operation by the exporter*

As the *import* operation implies actions performed by the *exporter* the following implied *transition* is added to every *state* of the *exporter* (including all implicit states):

- an *input node* for the *signal* XQUERY;
- an *output node* by the *name* XREPLY, returning the *value* implicit copy of the requested *variable* to the requesting *process instance* ;
- a return to the *state* from which the *transition* originated.

(It should be noted that the exporter is also modified with other *transitions* if the *exported values* are used in *enabling conditions* and/or *continuous signals*, see §§ 3.2 and 3.3.)

The implied *transition* using the graphical syntax of SDL is shown in Figure 9/Z.103.



CCITT-73920

Note – This diagram is explanatory only. Since the *transition* is implied the user does not write anything.

FIGURE 9/Z.103

Import operation by the exporter explained in SDL/GR

3.1.1.3 *Export operation*

The *export* operation is the means by which the owner of an *exported variable* disclose to importers the current *value* of the *variable.* As a result of the *export* operation, the implicit copy of the *variable* is given the current *value* of the *variable.* In the absence of *enabling conditions* or *continuous signals* the operation itself is modelled as a *task node,* assigning the *value* to the *variable.* However, the *export* operation interacts with the models for *enabling condition* and for *continuous signal,* see §§ 3.2 and 3.3.

3.1.2 *Rules for using imported values*

– All *imported values* must be defined as *IMPORTED* in the *importer* and as *EXPORTED* in the *exporter.* The export attribute in a *variable definition* makes the *variable identifier visible* throughout the system.

– The source and destination of the *import operation* is shown by including the *name* of the *imported value* in a *signal list* attached to a *channel* or to a *signal* path within a *block.* The source and destination may be in different *blocks* or in the same *block.*

– A *process instance* which *imports values* may also *export values,* but it must not *import values* of its own *variables.*

– An *imported value* <name> is *imported* by using the statement:

IMPORT(<name>,<pid>)

in a *task, decision* or *enabling condition.* The <name> is the *name* of the imported item and the <pid> is the *process instance identifier* of the owning *process instance.* All other references in the *process* to <name> will be interpreted as references to the local copy of the *value* of <name>.

– The disclosure of the *value* of an exported *variable* is made through the statement:

EXPORT(<name>):= <expression>

or

EXPORT (<name>)

contained in a *task.*

3.1.3 *Concrete syntax*

Imported values are referenced at three points in *process definition*. The syntax for these is the same in both the *SDL/PR* and the *SDL/GR*.

3.1.3.1 In *variable definitions* and *import definitions* the keywords *IMPORTED* and *EXPORTED* identify the use to be made of the declared *variable*.

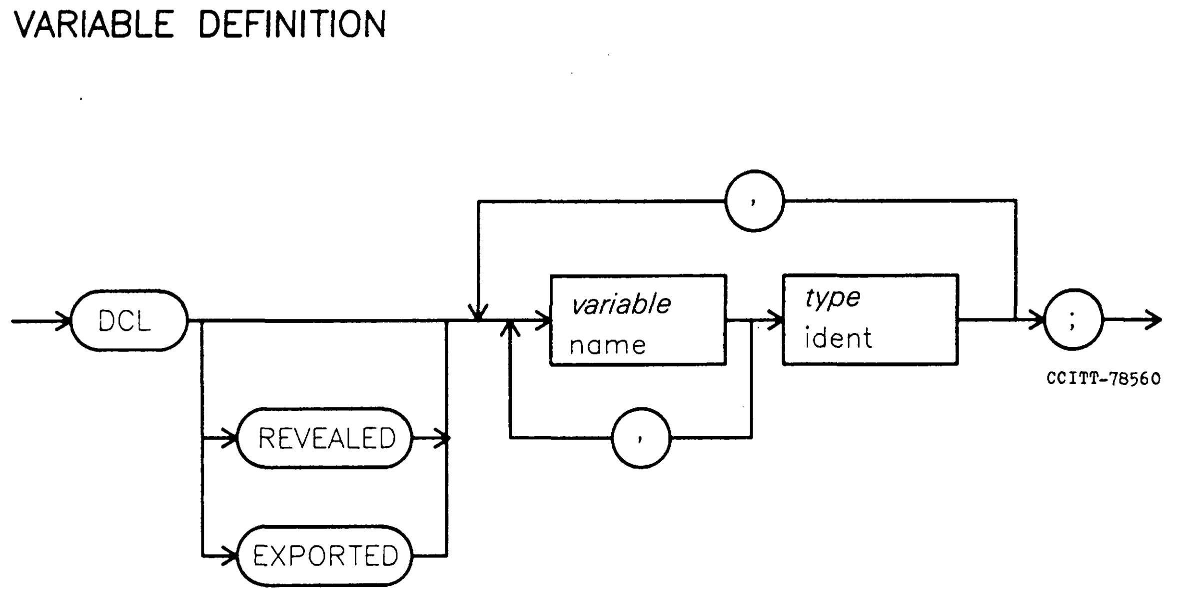


FIGURE 10/Z.103 (1 of 2)

Syntax diagram for variable definition and import definition

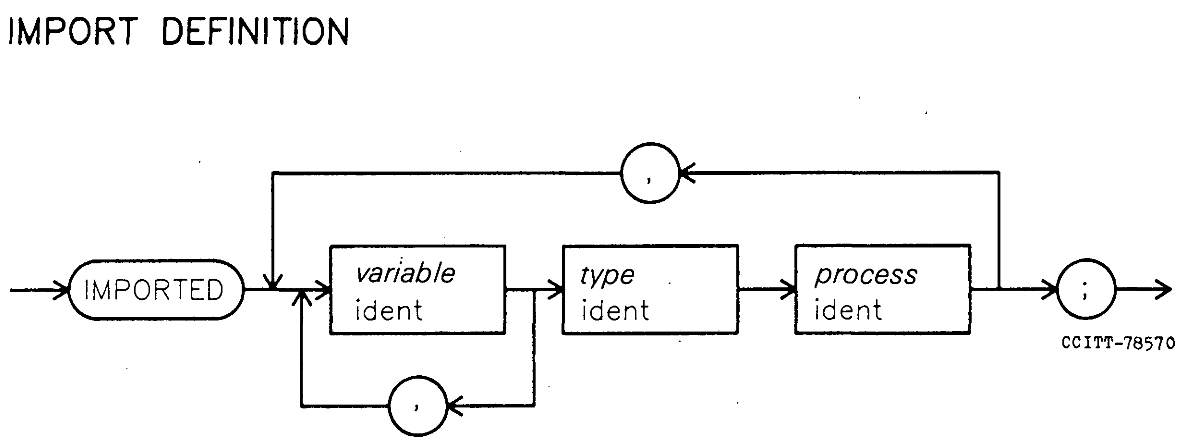


FIGURE 10/Z.103 (2 of 2)

Syntax diagram for variable definition and import definition

3.1.3.2 In *signal lists* in SDL/GR an *imported value* is distinguished from a *signal* by enclosing the *name* of the *imported value* in round brackets.

Signal-name-1
(Variable name)
Signal-name-2

The syntax in SDL/PR is:

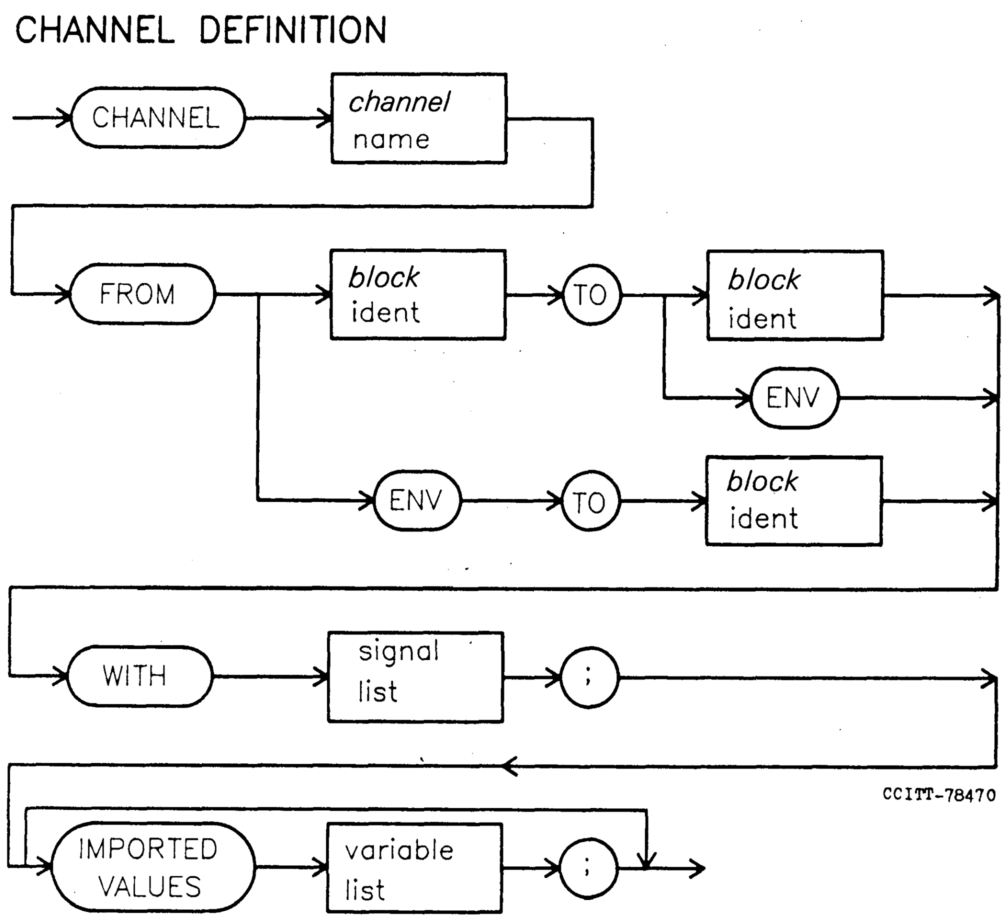


FIGURE 11/Z.103 (1 of 2)

Syntax diagram for channel definition and variable list

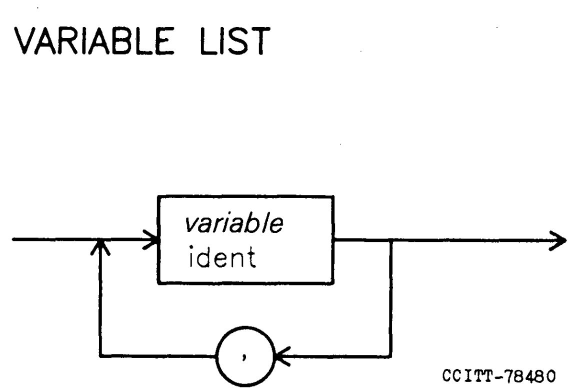


FIGURE 11/Z.103 (2 of 2)

Syntax diagram for channel definition and variable list

3.1.3.3 The statement *EXPORT*(<name>) may be used in a *task name*; similarly, the statement *IMPORT*(<name>,<pid>) may be used in *task, decision* and *enabling condition names.*

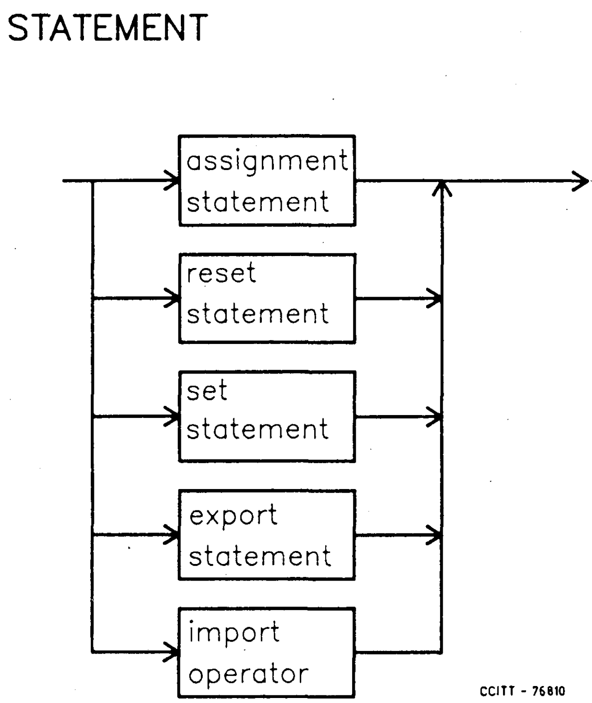


FIGURE 12/Z.103 (1 of 3)

Syntax diagram for export statement

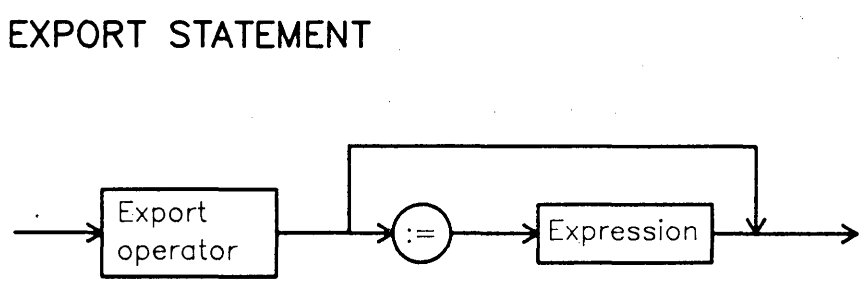


FIGURE 12/Z.103 (2 of 3)

Syntax diagram for export statement

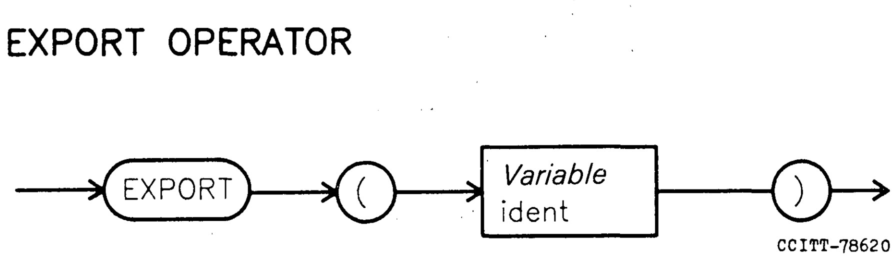


FIGURE 12/Z.103 (3 of 3)

Syntax diagram for export statement

IMPORT OPERATOR

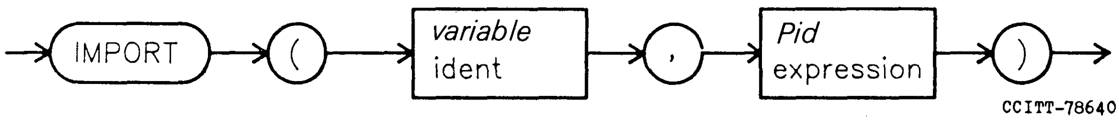


FIGURE 13/Z.103

Syntax diagram for import statement

3.2 *Enabling condition*

A reduction in the number of *states* in a *process graph* can often be achieved by having means of attaching conditions to the commencement of *transitions.* The conditions are assertions involving *variable*, and are known as *enabling conditions.* If the condition is true and the associated *input signal* is retained, the *transition* proceeds normally. If the *enabling condition* is not true, the *transition* is inhibited and the *input signal* is instead *saved.*

In normal SDL such conditional execution of *transitions* would need to be modelled with separate *states* for each *value* of the *enabling condition.* The concise notation provided by *enabling conditions* is useful in simplifying a *process graph.*

Enabling conditions may be expressed using both local and/or *imported values. Shared values* must not be used.

The *enabling condition* is described in terms of basic SDL concepts (see Recommendation Z.101). An *enabling condition* is used in a *process graph* and is attached to an *input symbol* or an *input* statement. It is an assertion involving *data values* which are either local or *imported values.* The condition must be a Boolean expression, hence returning the *values* TRUE or FALSE. Each *state* which is followed by *transitions* controlled by *enabling conditions* in fact represents a set of *states,* one for each possible combination of the *enabling condition.* The conditions are evaluated before one of the appropiate member of the set of *states* is selected as the next *state.*

The interpretation of an *enabling condition* depends on the *data values* used in the condition. If an *enabling condition* contains only *values* which are local to the *process* it needs only to be evaluated once. If it contains *imported values* (see § 3.1), the condition must be evaluated each time one of the *imported values* changes: the evaluation is invoked by implied *signal* interchange between the *importer* and *exporter(s)* of the *values.*

Enabling conditions which use *imported values* make use of further hidden *signals,* in addition to the XQUERY defined in § 3.1, which are XATTACH and ZDETACH. These are requests for the *exporter* to add or remove the *importer* from a list of those *process instances* which need to be informed of changes in the *exported value.* The *exporter* maintains such an implicit list, XLIST, for each *value* it *exports.* A further *signal,* XWAKE, is sent by the *exporter* to each member of the list whenever it makes an *export* operation on the corresponding *data item.*

3.2.1 *Definition*

3.2.1.1 *Single enabling condition based on local values*

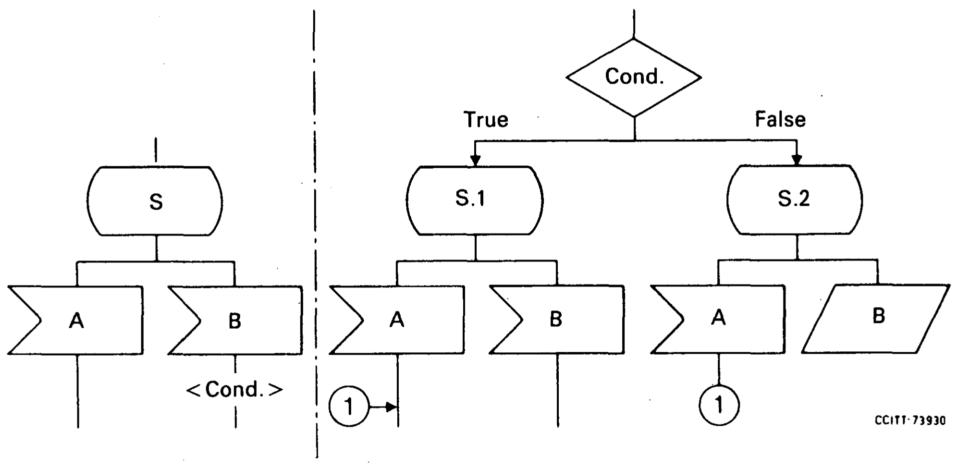
When only one of the *inputs* following a *state* has an attached *enabling condition,* and that condition contains only local *values,* the interpretation is as follows.

The *state,* the following *inputs* and *saves* are replaced by:

– a *decision node* which evaluates the *enabling condition* ;
– the "TRUE" branch of the *decision* is followed by a *state node* with the same *save-signal-set* as the original *state* and followed by the same set of *inputs* as the original *state* (the *input* to which an *enabling condition* was attached appears as an *input node*);
– the "FALSE" branch of the *decision* is followed by a *state node* having the *save-signal-set* of the original *state* extended with the *signal name* of the *input* having the *enabling condition* attached;
– each of the *input nodes* are followed by the same *transitions* as in the original graph.

In the following figure the definition of the Simple *Enabling Condition* based on local *values* only are shown using the graphic syntax of SDL:

Note – The user writes this:



Note — This diagram is explanatory only.

FIGURE 14/Z.103

Single enabling condition using local values only explained in SDL/GR

3.2.1.2 *Single enabling condition containing imported values*

When only one of the *inputs* following a *state* has an attached *enabling condition*, which contains at least one *imported value*, the interpretation will lead to an implied *signal interchange* with the owners of the *imported values*.

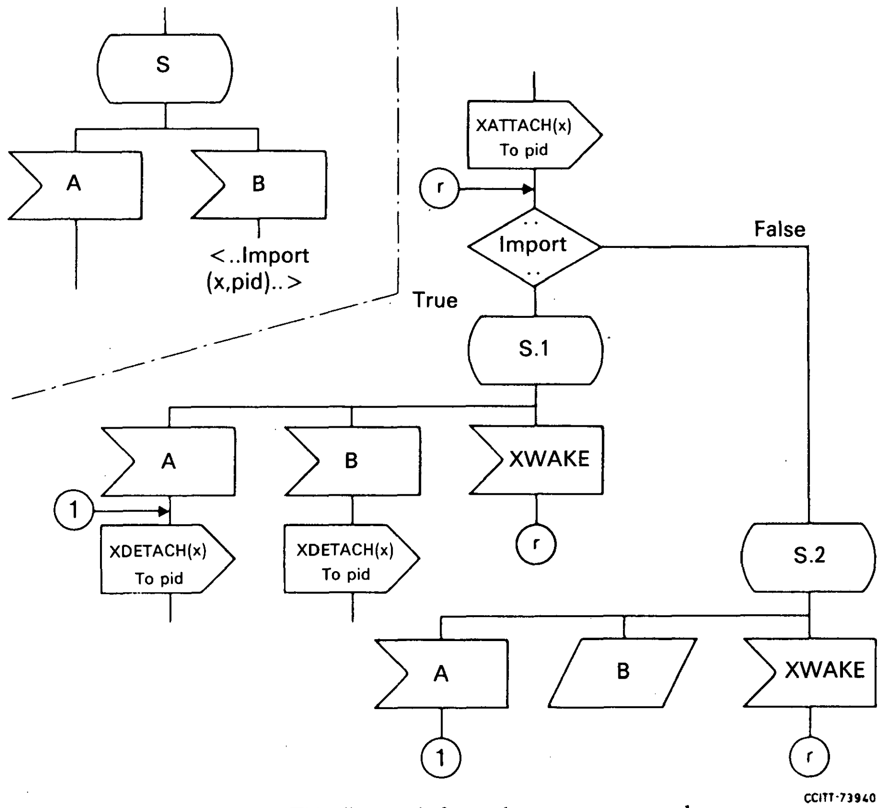
As the *variables* holding the *imported values* are handled by other *process instances*, working concurrently, the *value* of the *enabling condition* may change while the *process instance* is waiting in a *state*. Any such change should lead to a new evaluation of the condition. When entering an *enabling condition* construct the owners of the *imported values* are informed that this *instance* should be notified if any changes of the *value* occurs. Thus when an *imported value* is changed also a *signal* is sent to all *processes* that need informing, so that they evaluate the conditions again.

The definition is as follows:

The *valid input signal* set of the *process* which contains the *enabling condition* is extended by the implicit *signal* XWAKE. The *state* and the set of *inputs* and *saves* following it are replaced by:

- a sequence of *output nodes* for the *signal* XATTACH (one for each of the *imported values* used in the *enabling* condition);
- the attach sequence is followed by a *decision* and two *state nodes*, as in § 3.2.1.1;
- the *decision*, because it contains *imported values*, is itself expanded as in § 3.1.1.2, so that it is preceded by *import operations* for each *imported value* it uses;
- in addition to the expansion defined in § 3.2.1.1 the two *state nodes* are also followed by an *input node* for the *signal* XWAKE, with the following *transition* leading back to the *decision* evaluating the condition;
- also in addition to the expansion defined in § 3.2.1.1 each *input node*, except for the XWAKE, are followed by a sequence of *output nodes* for the *signal* XDETACH (one for each of the *imported values* used in the *enabling condition*) and the sequence is then followed by the same *transition* as in the original construct.

Note – The user draws this:



Note — This diagram is for explanatory purpose only.

FIGURE 15/Z.103

Single enabling condition using imported values explained using SDL/GR

3.2.1.3 *Actions by the exporter when imported values are used in enabling conditions*

When an *exported value* is used in an *enabling condition* the *exporter* has the following implied properties in addition to those defined in §§ 3.1.1.2 and 3.1.1.3.

To invoke the revaluation of *enabling conditions* using *imported values* the *exporter* implicitly maintains a list, XLIST, of *process instances* to be informed if the *values* change. *Process instances* will enter that list by sending the *signal* XATTACH, and will remove themselves from the list by sending the *signal* XDETACH.

Whenever a new *value* is *exported* by means of the *export* operation all *process instances* in the XLIST will be informed of this new *value* by the *signal* XWAKE.

The definitions of the addition are:

Each *process* which *exports values* is extended by:

a) adding the implicit *signals* XATTACH and XDETACH to the *valid input signal* set;

b) defining a list XLIST.<name> for each value which is *exported*. The lists will hold *process instance identifiers*.

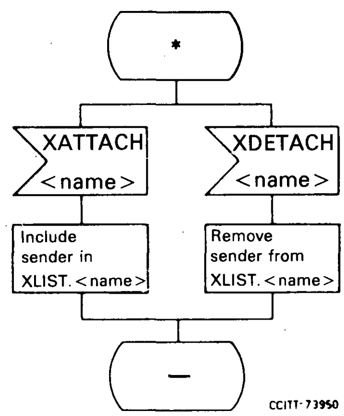
To each *state* of the *process*, including all implied *states*, is added two implied *transitions*, in addition to the implied *transition* defined in § 3.1.1.2. The first of these *transitions* consists of:

- an *input node* for the *signal* XATTACH;
- a *task* which adds the *process instance identifier* of the sender of the *signal* XATTACH to the XLIST corresponding to the *variable* nominated in the *signal*; whose *value* is *exported*;
- a return to the *state* from which the *transition* originated.

The second implied *transition* consists of:

- an *input node* for the *signal* XDETACH;
- a *task* which removes the *process instance identifier* of the sender of the *signal* XDETACH from the XLIST nominated in the *signal*;
- a return to the *state* from which the *transition* originated.

The additional implied *transitions* of each *state* of the *exporter* is shown below using the graphical syntax of SDL:



Note – This diagram is for explanatory purpose only.
All transitions are implied.

FIGURE 16/Z.103

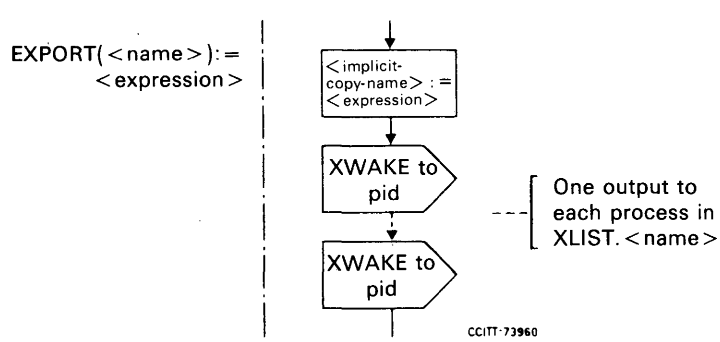
Implied transitions of the exporting process explained in SDL/GR

The following implied *transition* is added to the definition of the *export* operation, § 3.1.1.3:

- following a *task* including an *export* operation is a sequence of *output nodes* for the *signal* XWAKE conveying the disclosed *value*. There will be one XWAKE *signal* sent to each *process instance* currently included in the XLIST for the corresponding *variable* which was disclosed the *value*.

The addition to the *export operation* is shown below using the graphic syntax of SDL:

The user writes:



Note — This diagram is for explanatory purpose only.

FIGURE 17/Z.103

Implied actions by the export operation explained in SDL/GR

3.2.1.4 *Multiple enabling conditions*

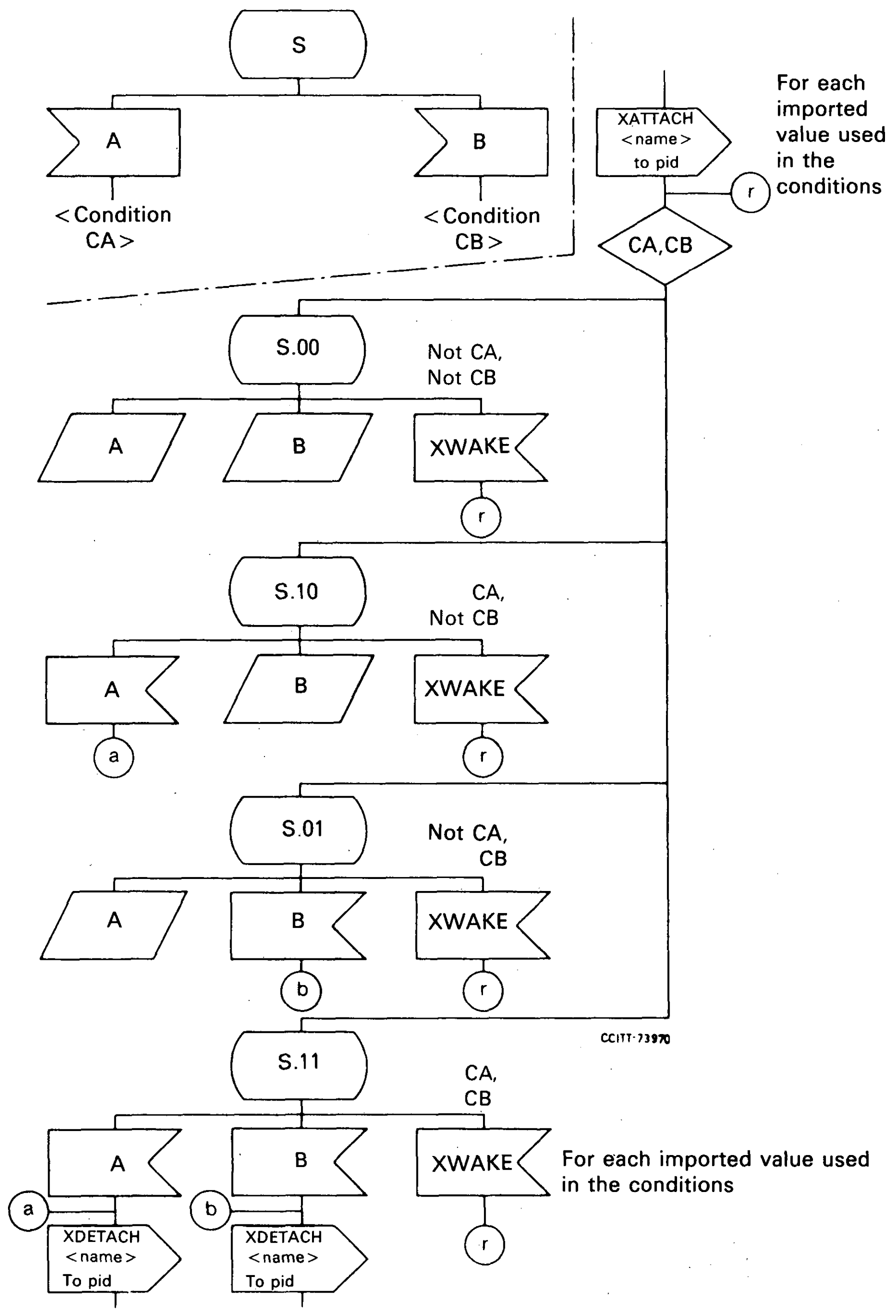
Having several *enabling conditions* following a *state* is known as multiple *enabling conditions.*

If n of the *inputs nodes* following a *state* have *enabling conditions* attached to them, then the definitions of single *enabling conditions* will be extended as:

a) the *state* and the following *inputs* and *saves* are represented by a *decision node* for a complex condition whose *value* is an n-tuple with elements the *value* of the elementary *enabling conditions.* This is followed by 2**n *state nodes,* each followed by the *transition* processing which is appropiate to the particular *values* of the *enabling conditions* which named the arc leading to that *state node*;

b) if any of the *enabling conditions* refer to *imported values*, the definitions in § 3.2.1.2 also applies.

The user writes this:



Note — This diagram is for explanatory purposes only.

FIGURE 18/Z.103

Multiple enabling conditions explained using the SDL/GR

3.2.2 *Rules for using enabling condition*

1) *Enabling conditions* may be attached to any *input signal.*

2) There may be only one *input* following a *state* containing a given *input signal,* regardless of whether or not the *input* has an *enabling condition* attached.

3.2.3 *Concrete syntax*

3.2.3.1 *SDL/GR*

Enabling conditions are shown in the *SDL/GR* by an *enabling conditions symbol* following an *input symbol.* The element is a pair of angle brackets embracing a *Boolean* condition. The graphical syntax is shown in Figure 19/Z.103:

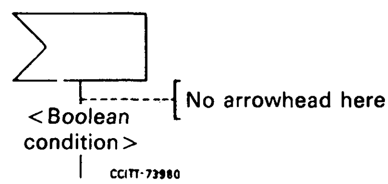


FIGURE 19/Z.103

SDL/GR syntax for the enabling condition

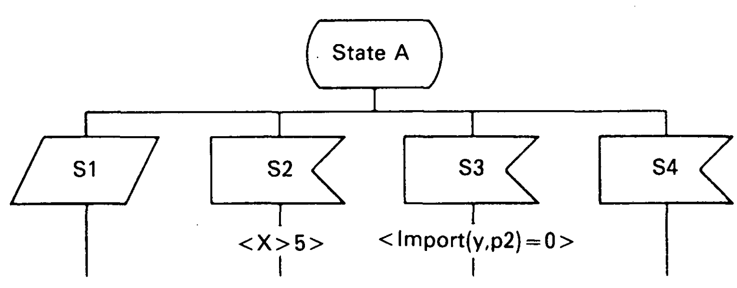


FIGURE 20/Z.103

Example of use of SDL/GR enabling condition syntax

3.2.3.2 *SDL/PR*

In *SDL/PR* the syntax of the *input* statement is modified by adding the *PROVIDED* phrase. The *PROVIDED* phrase contains a *Boolean* expression. The syntax diagram for the modified *input* statement is given in Figure 21/Z.103:

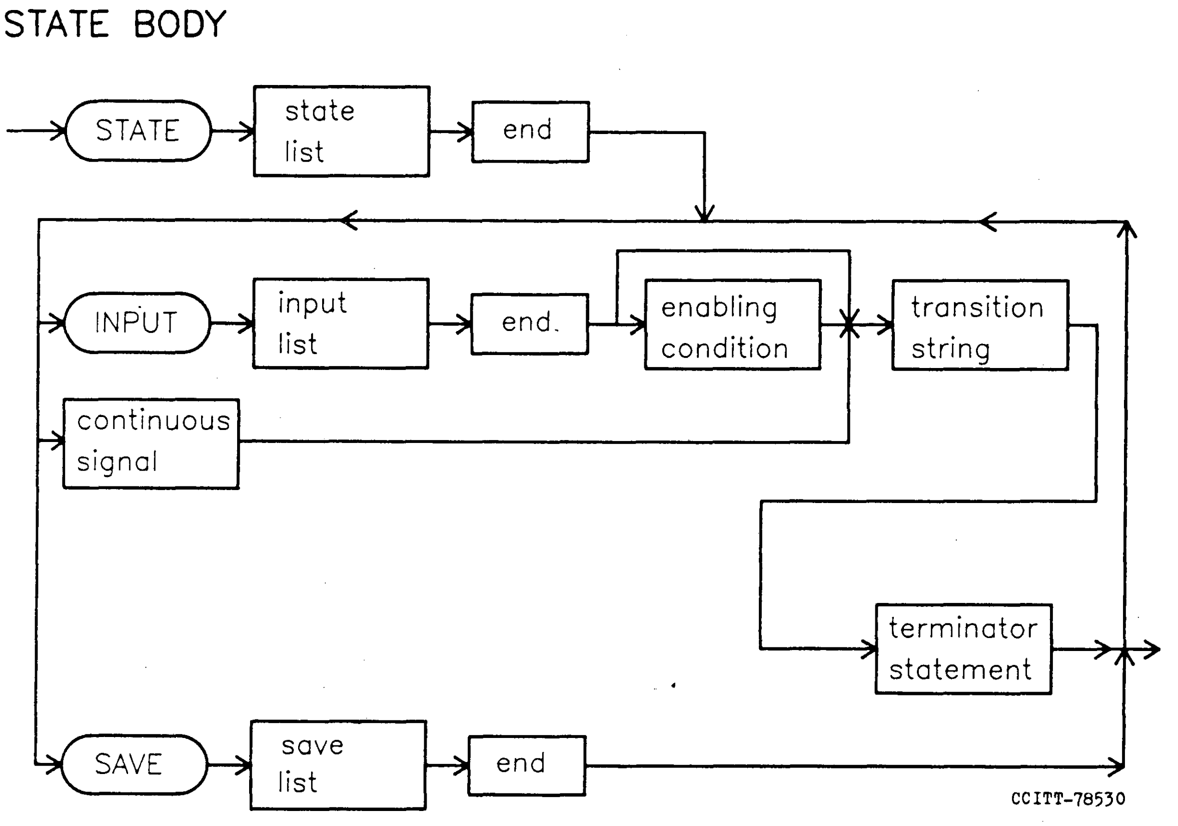


FIGURE 21/Z.103 (1 of 2)

Syntax diagram for the input statement including the PROVIDED phrase

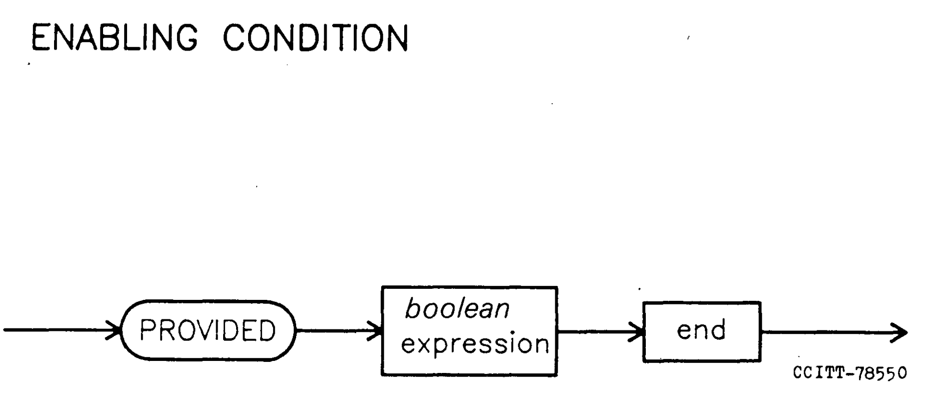


FIGURE 21/Z.103 (2 of 2)

Syntax diagram for the input statement including the PROVIDED phrase

3.3 *Continuous signal*

In describing *systems* with SDL, the situation frequently arises where a user would like to show a *transition* as being caused by a change in the *value* of a *variable* external to the *process.* The *value* might, as an example be a high or low voltage on a line or a number in a status register. The normal way to achieve this in SDL would be to arrange that a *signal* is generated when the change in the *value* occurs and to base the *transition* upon the reception of that *signal.* The necessity to explicitly *define*, generate and receive such *signals* may complicate the *process graphs.* The *composite operation* known as a *continuous signal* allows a change in the *value* of a condition to directly initiate a *transition.*

An *enabling condition* represents a *decision* before entry to a *state.* When *imported values* are used the *enabling conditions* provides exits from a *state* by means of the implicit XWAKE *signal.* It may also be used without an associated *input signal* when it becomes a *continuous signal.*In this case, the *enabling condition* does not represent an extra, implied *state* but instead defines a circumstance in which there is an escape from the *state* from which the *enabling condition* follows. This circumstance has a priority lower than the *retained signals.*

Several *continuous signals* may lead from the same *state* and these may be such that more than one is true at the same time. Each *continuous signal* is associated with a priority which determines the relative order in which *continuous signals* are tested.

A *continuous signal* which uses only local *values* gives a means of conditionally exiting from a *state* if no *signals* are waiting in the *input port* at the time of entry. A *continuous signal* which uses *imported values* adds to this the capacity to revaluate the condition when there is a change in one of the *imported values* used in the condition.

3.3.1 *Definition*

The following definition is based upon the definitions of *import* and *export* of *values*, see § 3.1, and *enabling conditions*, see § 3.2.

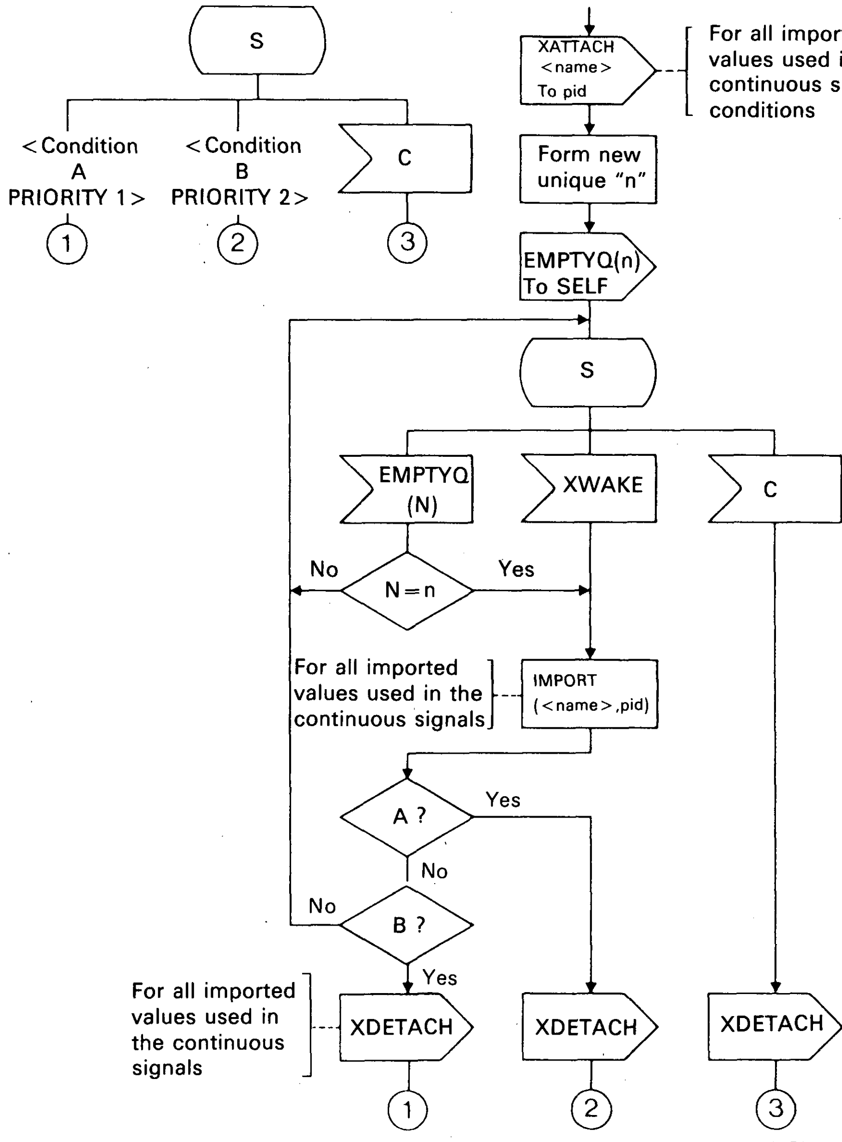
The interpretation of a *state* which is followed by *continuous signals* is presented as a general model in which there are several *continuous signals* which collectively reference several *imported values.* If there are no *imported values* used, the model is simplified by the elimination of XATTACH and XDETACH *outputs* and the XWAKE *input.*

The *state* and the set of *inputs* and *saves* following it, together with the *continuous signals* are replaced by:

1) a sequence of *output nodes* for the *signal* XATTACH, one for each of the *imported values* used in the *continuous signal* condition;
2) a *task* which creates a unique *value* to be used in the EMPTYQ *signal*, used in 3);
3) an *output node* for a *signal* EMPTYQ which is sent to the *process instance identity* of the sender, i.e. to its own *input port*;
4) a *state node* as in the original *process graph* or *procedure graph*, followed by a set of *input nodes* which includes the original set and two other *input nodes*;
5) each *input node* is followed by a sequence of *output nodes* for the *signal* XDETACH, one for each of the *imported values* used in the *continuous signal* conditions. This sequence is then followed by the *transition* processing which originally followed the *input node*;
6) the *state node* in 4) is also followed by an *input node* for the *signal* XWAKE, this initiates the *transition* in 7);
7) a sequence of *task nodes* to *import* each of the *imported values* used in the *continuous signal* conditions;
8) a sequence of *decision nodes* for each of the *continuous signal conditions*, the first *decision* evaluated being that for the highest priority *continuous signal* (the lowest number in the concrete syntax);
9) the FALSE branch of each *decision* leads to a *decision node* for the *continuous signal* condition of next lower *priority.* The FALSE branch of the lowest priority *continuous signal decision* leads back to the *state* in 4);
10) the TRUE branch of each *decision* leads to a sequence of *output nodes* for the *signal* XDETACH, one for each of the *imported values* used in the *continuous signal* conditions, followed by the *transition* processing corresponding to the *continuous signal* condition tested in the *decision*;
11) the *state node* in 4) is also followed by an *input node* for the *signal* EMPTYQ, in turn followed by a *decision node* which tests that the *signal* carries the *value* given to it in 2), i.e. that it is the same *signal* sent in 3), and not an earlier, unprocessed EMPTYQ *signal.* The TRUE branch of this *decision* leads to the *continuous signal* processing in 7), and the FALSE branch leads back to the *state node* of 4).

The *valid input signal* set of the *process* containing *continuous signals* using *imported values* is extended by the implicit *signal* XWAKE. The *valid input signal sets* of any *process* containing *continuous signals* are extended by the implicit *signal* EMPTYQ.

The user writes this:



Note – This diagram is for explanatory purposes only.

FIGURE 22/Z.103

Continuous signal explained in SDL/GR

3.3.2 *Rules for using continuous signals*

1) *Continuous signal* may follow any *state.*

2) *Continuous signal condition* may be based on local *values* and/or *imported values.*

3) No two *continuous signals* following the same *state* may have the same priority number.

3.3.3 *Concrete syntax*

3.3.3.1 *SDL/GR*

In *SDL/GR* a *continuous signal* is indicated by an *enabling condition symbol* which directly follows a *state symbol*, i.e. the *transition* is not headed by an *input symbol*. The symbol contains, as well as the *continuous signal* condition, the keyword PRIORITY followed by a priority number (natural number). The smaller the number, the higher is the priority of the *continuous signal*.

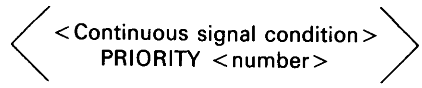


FIGURE 23/Z.103

SDL/GR symbol for the continuous signal

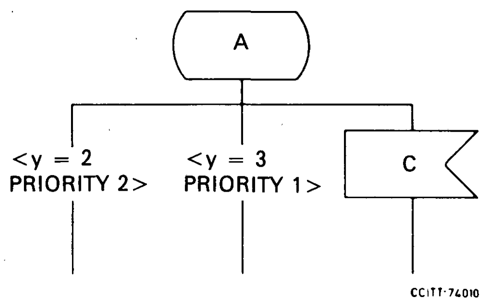


FIGURE 24/Z.103

Example of the use of SDL/GR continuous signal

If only one *continuous signal* is following a *state*, the PRIORITY clause may be omitted. If the clause is omitted the priority number "1" is implied.

3.3.3.2 *SDL/PR*

In *SDL/PR* a PROVIDED statement, followed by a transition string, represents the *continuous signal*. The statement contains a PRIORITY clause. The smaller the number, in the PRIORITY clause, the higher is the *priority* of the *continuous signal*.

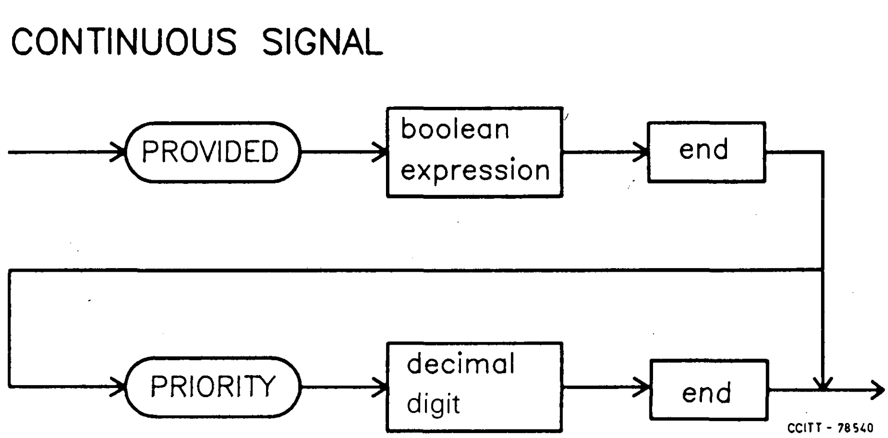


FIGURE 25/Z.103

Syntax diagram for continuous signal

4 Macros

A *Macro* is a shorthand notation, defined by the user, that can be included in one or more places in the concrete SDL representation of a *system.* It represents a reference to a definition in a document elsewhere. A *macro* is only a part of the concrete syntax, and has to be substituted by the body of its definition in order to interpret the SDL representation in which it appears.

4.1 *Definition*

A **macro** may represent any collection of syntactic items, however, it may not be recursive for obvious reasons (infinite expansion!).

The *macro* can have zero or more *inlets* and zero or more *outlets.* In case of more than one *inlet* there should be a *label* attached to each *inlet* corresponding to the *inlet label* in the *macro definition,* in case of a single *inlet* the *label* may be omitted. The same applies for the *outlets.*

There is no *scope* or *visibility* associated to the *macro* concept as such. The interpretation of a *macro* reference may only be obtained when the *macro* is substituted by its definition.

4.2 *Concrete syntax*

4.2.1 *SDL/GR*

4.2.1.1 *Syntax*

The reference to a *macro definition* is shown by the *macro symbol* in the *SDL/GR.*

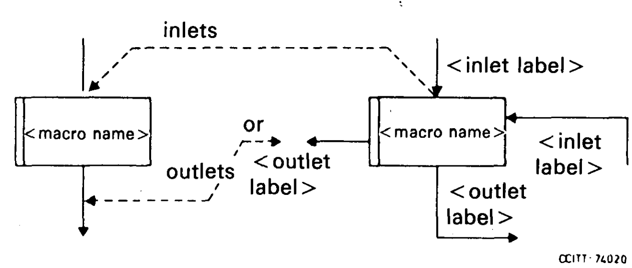


FIGURE 26/Z.103

The macro symbol in SDL/GR

The *inlets* to and *outlets* from the *macro* is represented by *flow lines* leading to/from the symbol. *Labels* may optionally be attached to the flow lines.

The *macro symbol* contains the *name* of the *macro definition* it refers to, and a *comment* may be attached to the symbol.

The *macro definition* is entitled:

< name > MACRO DEFINITION

where the <name> is the *name* of the *macro*, and which is used in a *macro symbol.*

The *macro definition* contains the graphical representation which replaces the *macro* symbol before interpretation takes place. The *inlets* to and the *outlets* from the definition are represented by flow lines leading to and from the symbols in the definition respectively. Both the *inlets* and the *outlets* may have *labels* attached to them.

4.2.1.2 *Symbols*

In Figure 27/Z.103 two additional symbols are defined:

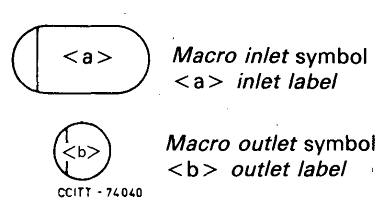


FIGURE 27/Z.103

Additional symbols for macro

4.2.1.3 *Rules for using macros in the SDL/GR*

The *macro symbol* may be inserted at any place in a diagram, and may represent any collection *SDL/GR* symbols.

There must be the same number of *inlets* in a *macro symbol* as there are *inlets* in the referenced *macro definition.* The set of *labels* on the *inlets* in the *macro symbol* must be the same as the set of *labels* on the *inlets* in the *macro definition.* The same rule applies for *outlets* and *outlet labels.*

Inlets/outlets may terminate/originate from any side of the symbol.

As the semantic of a *macro* is obtained by substituting the reference by the collection of symbols in the definition, all graphic conventions apply to the diagram in which all macro appearances have been substituted. This may lead to unexpected consequences due to rules as for example the multiple appearance of *states.* This is further discussed in the SDL User Guidelines.

4.2.2 *SDL/PR*

4.2.2.1 *Syntax*

The macro call can be put in any diagram using the syntax:

MACRO macro name;

The macro expansion is a piece of an SDL/PR program starting with:

MACRO EXPANSION macro name;

and ending with:

ENDMACRO macro name;

in the last statement, the macroname is not mandatory.

This definition must be placed immediately after the SYSTEM, BLOCK or PROCESS construct respectively depending on where the *macro* is referenced. The *macro definition* may contain any character and its correctness can only be evaluated after it has replaced the *macro* statement.

4.2.2.2 *Rules for using macros in the SDL/PR*

The same rules and concerns as when *macros* are used in the *SDL/GR* also applies for the *SDL/PR.*

5 Options

When several similar applications are specified or described using SDL often the same *process definition* can be used in the different *systems* if it is slightly modified. The *OPTION* facility in SDL provides means for defining *processes* generic for several applications by introducing alternative optional parts of the descriptions.

5.1 *Definition*

An *option* is the selection of alternative parts of a *process definition*, according to the evaluation of an *option expression.* The selection is made before the *process definition* is interpreted.

The *option* facility is only part of the concrete syntax, and should be considered as a shorthand notation providing one generic description for several applications rather than having one specific description for each application.

5.2 *Concrete syntax*

5.2.1 *SDL/GR*

5.2.1.1 *Symbols*

The following symbol is used in the *SDL/GR* to represent *option* :

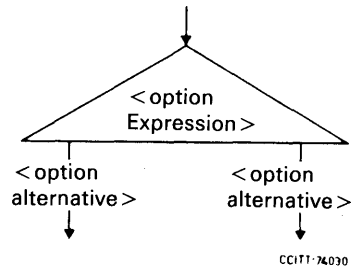


FIGURE 28/Z.103

The option symbol in SDL/GR

5.2.1.2 *Rules for using options in the SDL/GR*

The *option symbol* may follow a *task symbol*, a *decision symbol* outlet, an *output symbol* or a *procedure symbol* in a *process diagram*. The *option symbol* may be followed by a *state symbol*, a *task symbol*, a *decision symbol*, an *output symbol* or a *procedure symbol*.

The *<option expression>* contained in the symbol is an *expression* such that one of the *<option alternative>* following the symbol is uniquely chosen after evaluation, and such that it can be evaluated before interpreting the *process*. Each *option alternative* must be a *value* of the same *type* as the *option expression*. When the resulting *process* is interpreted the unreachable parts of the *process definition* should be considered as deleted.

5.2.2 *SDL/PR*

5.2.2.1 *Syntax*

The *option* is represented in the *SDL/PR* by the following syntax:

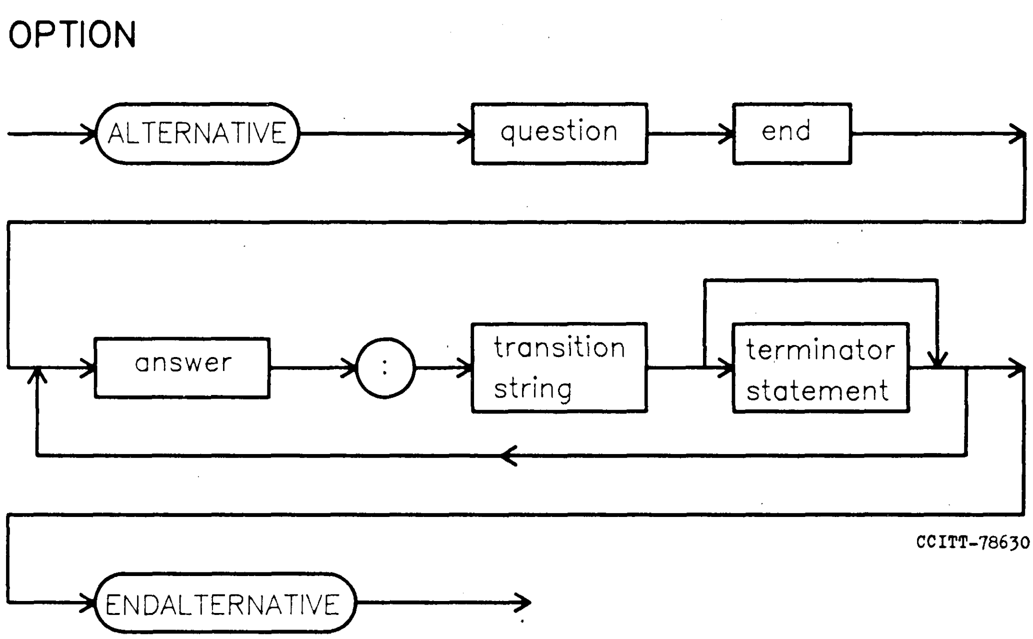


FIGURE 29/Z.103

Syntax diagram for option

5.2.2.2 *Rules for using options in the SDL/PR*

The *<option expression>* contained in the statement is an expression such that one of the *<option alternative>*s can be uniquely chosen after evaluation, and such that it is evaluated before interpreting of the *process.* Each *option alternative* must be a *value* of the same *type* as the *option expression.* When the resulting *process* is interpreted the unreachable parts of the *process definition* should be considered as deleted.

6 Pictorial elements in SDL/GR

When the graphical syntax of SDL is used to represent *process definitions* the use of *pictorial elements* to form a *state picture* within a *state symbol* is an optional part of the *SDL/GR.*

Such *state pictures* can provide advantages when applied to certain *system definitions,* resulting in more compact and less verbal *process diagrams.* The *state picture* describes, in terms of *pictorial elements* and qualifying text, the actual status of the *process* when in that state. Also the assumed status of the environment of the *process* may be described in the *state picture.* When using *state pictures* the actions to be performed in *transitions* between the *states* is implied, by the difference of the described status.

When using *pictorial elements*, the syntax and semantics as defined elsewhere in the Recommendation Z.101 apply. However, these semantics and syntax are extended as defined in the following.

6.1 *Semantics of state pictures*

When using *pictorial elements* a *state node* is represented by a *state symbol.* The *state symbol* is identified by its *name,* and contains a *state picture* consisting of *pictorial elements, values* of *variables, input variables* and qualifying text.

A *state picture* can represent:

1) using *pictorial elements* and qualifying text, the *values*, in that *state,* in a selected subset of the total set of *variables* associated with that *process.* The selected subset may include *variables* which serves purely as proxy for *variables* associated with other *processes.* These "proxy" *variables* carry the *value* of *variables* associated with other *processes.* These *values* are obtained either by *viewing* or *importing* the *value*;
2) using the *values* of *input variables*, the *input actions* with respect to the *valid input signals* for that *state.*

The repertoire of *pictorial elements* is in principle unlimited, since new *pictorial elements* can be invented to suit any new application of the SDL. However, in applications to telecommunications switching and signalling functions, the following repertoire of *pictorial elements* has been found to have considerable versatility:

- process boundary (left or right),
- terminal equipment (various),
- signalling receiver,
- signalling sender,
- combined signalling sender and receiver,
- timer supervision process,
- switching path (connected, reserved),
- switching module,
- charging in progress,
- control element,
- uncertainty symbol.

Standard symbols for these *pictorial elements* are recommended in § 6.3.

6.2 *Rules of interpretation*

1) *Input variables* are *Boolean variables* and each *input variable* corresponds to one and only one *signal* from the set of *valid input signals* for the *process.* A change in the *value* of an *input variable* between *state pictures* always represents the consumption of a *signal* by an *input* of a *process.* Therefore, *input variables* can be used to represent those conditions of a *process* which, if changed, will result in the *process* performing a *transition.* The *values* of an *input variable* can be associated with *pictorial elements.*

2) The presence of *pictorial elements* in a *state picture* indicates specific *values* for a subset of *variables* while that *process* is in that *state*. The *values* of additional *variables*, particularly ones associated with this initial subset can be indicated by the qualification of the *pictorial elements* by qualifying text. Qualifying text is not an *input variable*; changes in the qualifying text DO NOT represent the consumption of an *input signal* by an *input action* of the *process*.

3) Positioning:

   a) The positioning of any *pictorial elements* (other than a *process boundary*) relative to a *process boundary* determines whether the *pictorial elements* is "internal" or "external" to the *process*. An internal *pictorial elements* represents *variables* which are owned by the *process*. An external *pictorial element* represents *variables* which are owned by another process so that the *viewing* and *import* mechanisms must be used to access these *variables*.

   b) Rule a) also applies to the distinction between *internal* and external qualifying text, by substituting the term "qualifying text" for *pictorial elements* in this rule.

4) Cardinal rule:

   The total processing involved when going from one *state* to the following *state* is that required to effect the changes in the state *pictures*, together with the processing indicated in any *decisions*, *outputs* or *tasks* appearing in the *transition* between the *states*. Thus:

   a) The change from the appearance of an internal *pictorial element* in one *state* to the absence of that *pictorial element* in the following *state*, or vice versa, corresponds to a change in the *values* of some *variables* which can be equivalently represented by the use of a *task* in the *transition* between the *states*.

   b) The change from the appearance of an external *pictorial element* in one *state* to the absence of that *pictorial element* in the following *state*, or vice versa, corresponds to a change in *variables* owned by another *process*. This change can be equivalently represented either by an *output signal* to that other *process* or simply by the *input signal* from that *process*.

   c) Rules (a) and (b) also apply to the appearance or disappearance in the *state picture* of qualifying text, by substituting the term "qualifying text" for *pictorial elements* in those rules.

5) For a given *process diagram*, particular *pictorial elements* (or a particular combination of *pictorial elements* and qualifying text) is positioned uniquely within the *state picture* so that the presence or absence of this *pictorial element* (or combination) in a *state symbol* can be quickly determined by comparing the *state picture* with other *state pictures* in the *process diagram*.

6) When a *signalling sender* appears in a *state picture*, its qualifying text identifies a *signal* which has been *output* prior to, or (in the case of a *continuous signal* controlled by the *process*) prior to and during this *state*.

6.3 *Recommended symbols for pictorial elements*

When using *pictorial elements*, each *state* is represented by a *state symbol* containing a *state picture* with the format shown in Figure 30/Z.103:

A basic set of *pictorial elements* has been standardized for use in the *SDL/GR* with application to the *system description* of telecommunications call handling processes, including signalling protocols, network services and signalling interworking processes. Many of these *pictorial elements* are capable of being applied in applications of the *SDL/GR* beyond call handling processes, and their application to other processes in telecommunications, where appropriate, is encouraged.

The recommended symbols for the basic set of *pictorial elements* is shown in Figure 31/Z.103 below:

The choice of pictures for *pictorial elements* has been based upon the considerations and general selection criteria presented in Annex A to this Recommendation, which should be consulted before developing additional *pictorial element* symbols for wider applications of the *SDL/GR*.

The recommended proportions for *pictorial element* symbols are shown in Annex B to this Recommendation.

The template which is enclosed in the inside back cover of this fascicle and which is suitable for hand drawing the basic set of *SDL/GR* symbols, includes in this basic set the *pictorial element* symbols shown in Figure 31/Z.103.

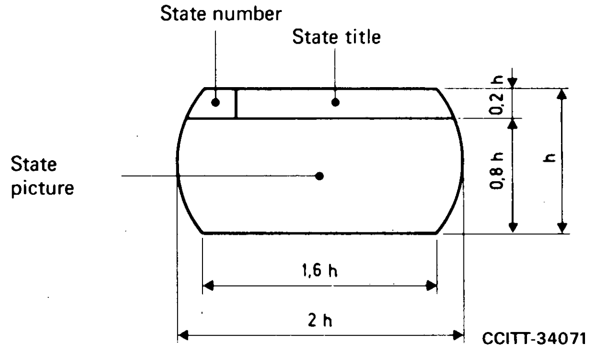


FIGURE 30/Z.103

Recommended format of a state symbol with state picture

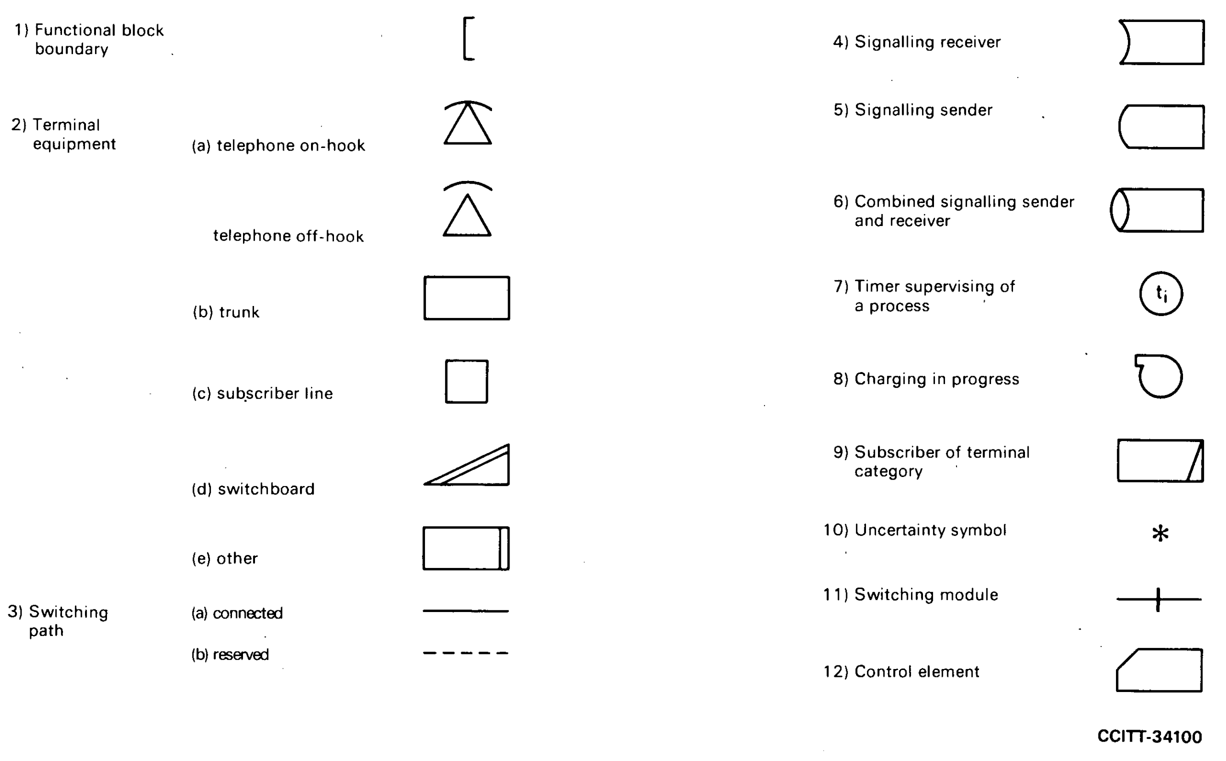


FIGURE 31/Z.103

Recommended symbols for the basic set of pictorial elements concepts

6.4 *Special conventions and interpretations used in the state oriented extension of SDL/GR*

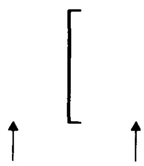
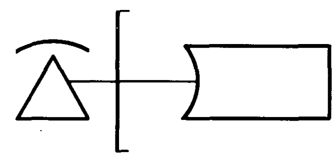
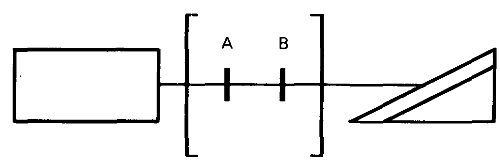
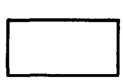
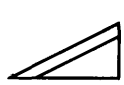
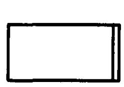
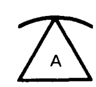
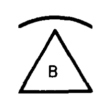
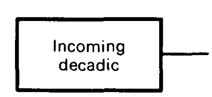
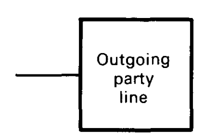
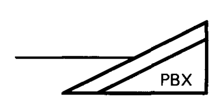
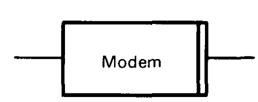
A number of special conventions and interpretations have been defined in this section with regard to the state oriented extensions of *SDL/GR*. These include:

- The special interpretation required of *process diagrams* according to the so-called *CARDINAL RULE* (see § 6.2, rule 4).
- The unique positioning of *pictorial elements* (or *pictorial elements* and qualifying text) within a *state picture* that is required when using *pictorial elements* (see § 6.2, rule 5).
- The special interpretation required for the *variables* represented by *external pictorial elements* and external *qualifying text*, as proxy *variables* for other *variables* associated with other *processes*.

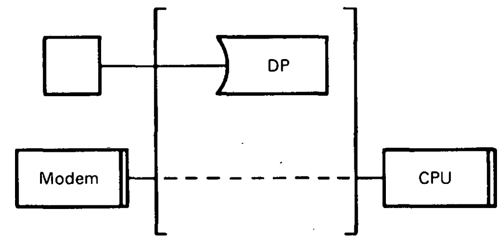
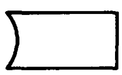
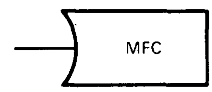
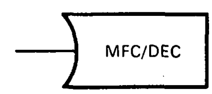
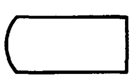
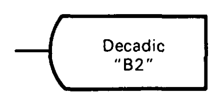
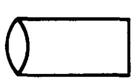
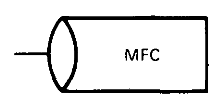
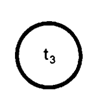
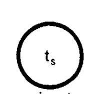
ANNEX A

(to Recommendation Z.103)

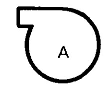
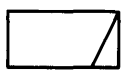
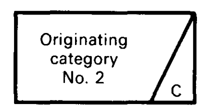
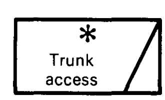
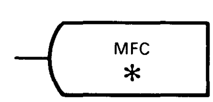
Examples of the use of the basic set of pictorial elements

No.	Pictorial element	Comment	Examples
1.	*Functional block* (FB) *boundary*  exterior interior	To distinguish elements inside and outside the FB boundary. Only the states of elements within the boundary can be changed directly by this process.	1.1 A handset outside the FB boundary connected to a digit receiver inside the FB boundary  1.2 A trunk outside the FB boundary connected via a two-stage switching unit to a switchboard outside the FB boundary 
2.	*Terminal equipment* a) Telephone set on-hook  off-hook  b) Trunk  c) Subscriber line [except a)]  d) Switchboard  e) Other 	It can be useful to show terminal equipment (e.g. telephone set and switchboard equipment) outside the FB boundary, to improve understanding of the processing work.	2.1 A on-hook $\bar{h}$  2.2 B off-hook h_B  2.3 Incoming decadic trunk junctor (from a space division switching exchange)  2.4 Outgoing subscriber line to a party line  2.5 PBX switchboard  2.6 Modem 

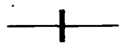
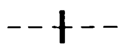
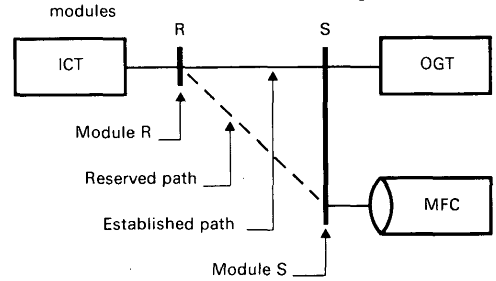
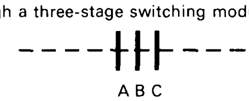
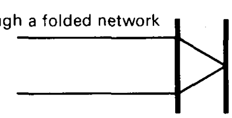
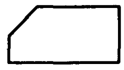
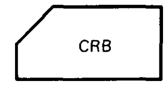
CCITT-20880

No.	Pictorial element	Comment	Examples
3.	*Switching path* a) connected b) reserved	To show connectivity between terminal equipment and/or signalling devices involved in the process.	3.1 Subscriber line connected to a dial-pulse digit receiver and a modem with a reserved path to a central processing unit (CPU) 
4.	*Signalling receiver* 	To specify a signal reception process, and to indicate the nature of the signals received, especially those crossing the functional block boundary.	4.1 Multi-frequency code signalling receiver  4.2 MFC/decadic signalling receiver 
5.	*Signalling sender* 	To specify a signal sending process, and to indicate the nature of the signals sent, especially those required to cross the functional block boundary.	5.1 Decadic signalling sender with a backward signal "B2" being sent 
6.	*Combined signalling sender and receiver* 	This conveniently combines the functions of a signalling sender and signalling receiver.	6.1 MFC sender-receiver 
7.	*Timer supervising a process* 	Timers affect the subsequent behaviour of the process. *Note* The related input symbol indicating timeout expiry, may be shown as $\bar{t}_i$.	7.1 Timer t_3 is running  7.2 Generic timer t_s is running  where s = 1, 2, ... n define different service tones.

CCITT-20890

No.	Pictorial element	Comment	Examples
8.	*Charging in progress (and which customer is being charged)* 	The charging policy is significant to the Administration, the manufacturer and the customer.	8.1 Subscriber A is currently being charged 
9.	*Subscriber or terminal category (and identity information)* 	Changes in the subscriber or terminal category, for each party in a multi-party call, can affect the behaviour of the process.	9.1 The A party has trunk access barred  9.2 The C party has originating category No. 2 
10.	*Uncertainty symbol* *	This substitutes for deliberately undefined information that is shown unambiguously in other state pictures. In certain cases, two or more states may be safely merged into one, with a net gain in the intelligibility of the diagram, by using the uncertainty symbol.	10.1 Handset either on-hook or off-hook  10.2 Subscriber category either "bar trunk access" or not, in this state of the process  10.3 An undefined MFC signal is being sent in this state 

CCITT-20900

No.	Pictorial element	Comment	Examples
11.	*Switching module*  or 	To show what switching modules are involved in the process. *Note* – The horizontal line is the pictorial element for a switching path, which may be connected or reserved. The vertical line can be used to represent either a complete switching module (when the internal structure of the module is not required) or else one of the switching stages within a switching module.	11.1 A path connected through one switching module LLN = Line link network  11.2 Paths connected and reserved through two switching modules  ICT – Incoming trunk OGT – Outgoing trunk MFC – Multi-frequency code *Note* – In this example, ICT is connected to OGT, but ICT is not connected to the MFC sender/receiver. 11.3 A path connected through a three-stage switching module RSN  11.4 A path reserved through a three-stage switching module ABC  11.5 A path connected through a folded network 
12.	*Control element (assigned to a process)* 	To show what control equipment is involved in the process (especially modules that must be dimensioned). This symbol can be used to indicate that particular software elements have been assigned to the process.	12.1 Call register buffer 

CCITT-20910

ANNEX B

(to Recommendation Z.103)

Selection criteria for pictorial elements

B.1 *General*

The choice of symbols for PEs has been based upon the following considerations and general selection criteria, which should be consulted before developing additional PE symbols for wider applications of the SDL.

B.2 *Typical readers*

It is expected that the SDL diagrams using PEs will be read by both technical and non-technical people in the following contexts: marketing new facilities; specifying new facilities; developing hardware and software from a specification; project management; operation and maintenance of an exchange; traffic engineering; education and training courses in telephony. It is expected that SDL diagrams will serve as common documentation:

a) between Administrations and manufacturers,

b) between different departments within these organizations,

c) as telephone exchange documentation, and

d) in training manuals and textbooks.

It is not expected that SDL diagrams (as diagrams) will be directly read by machines. Instead it is expected that the SDL/PR form of the SDL (including PE information) will be read by machines which will draw diagrams (see b) and c) of § B.3).

B.3 *Typical drawing methods*

It is expected that SDL diagrams using PEs will usually be drawn by technical people, including draftsmen, either:

a) by hand, using a template as a drawing aid, and/or

b) displaying the diagram electronically on a graphics visual display unit, and/or

c) by using an electronically controlled plotter.

B.4 *Methods of reproduction*

The typical methods of reproduction are expected to be:

a) the dye-line or blue-print methods, as in conventional drafting;

b) photocopying by office machines, including photo-reduction;

c) photo-printing in general.

B.5 *Ease of reproduction*

In order to permit convenient reproduction of SDL diagrams using the dye-line or blue-print methods of reproduction as well as photocopying and photo-printing, PE symbols should consist of clear lines without shading.

B.6 *Ease of drawing*

The following criteria reflect the assumption that the primary drawing technique will be to draw by hand using a template, and the secondary techniques will be displaying a diagram on an electronically controlled screen, and drawing the diagram with an electronically controlled pen:

a) each PE symbol should be easy to draw with pen or pencil, either free-hand or using a stencil;

b) all PE symbols should be drawn using the same thickness of lines;

c) PE symbols should be created by synthesis of very simple geometric lines and curves in order to permit easy electronic generation of PE symbols.

B.7 *Ease of comprehension*

This is the most important consideration of all, since it is characteristic of SDL documentation that the readers are much greater in number than the drawers (or authors). This requirement is expressed by the following criteria concerning PE symbols:

a) *Appropriateness* – The shape of each symbol should be appropriate to the concept that the symbol represents.

b) *Distinctiveness* – When choosing a basic set of symbols, care should be taken to permit each symbol to be readily distinguishable from others in the set.

c) *Affinity* – The shapes of PEs representing different but related functions, e.g. receivers and senders, should be related in some obvious way.

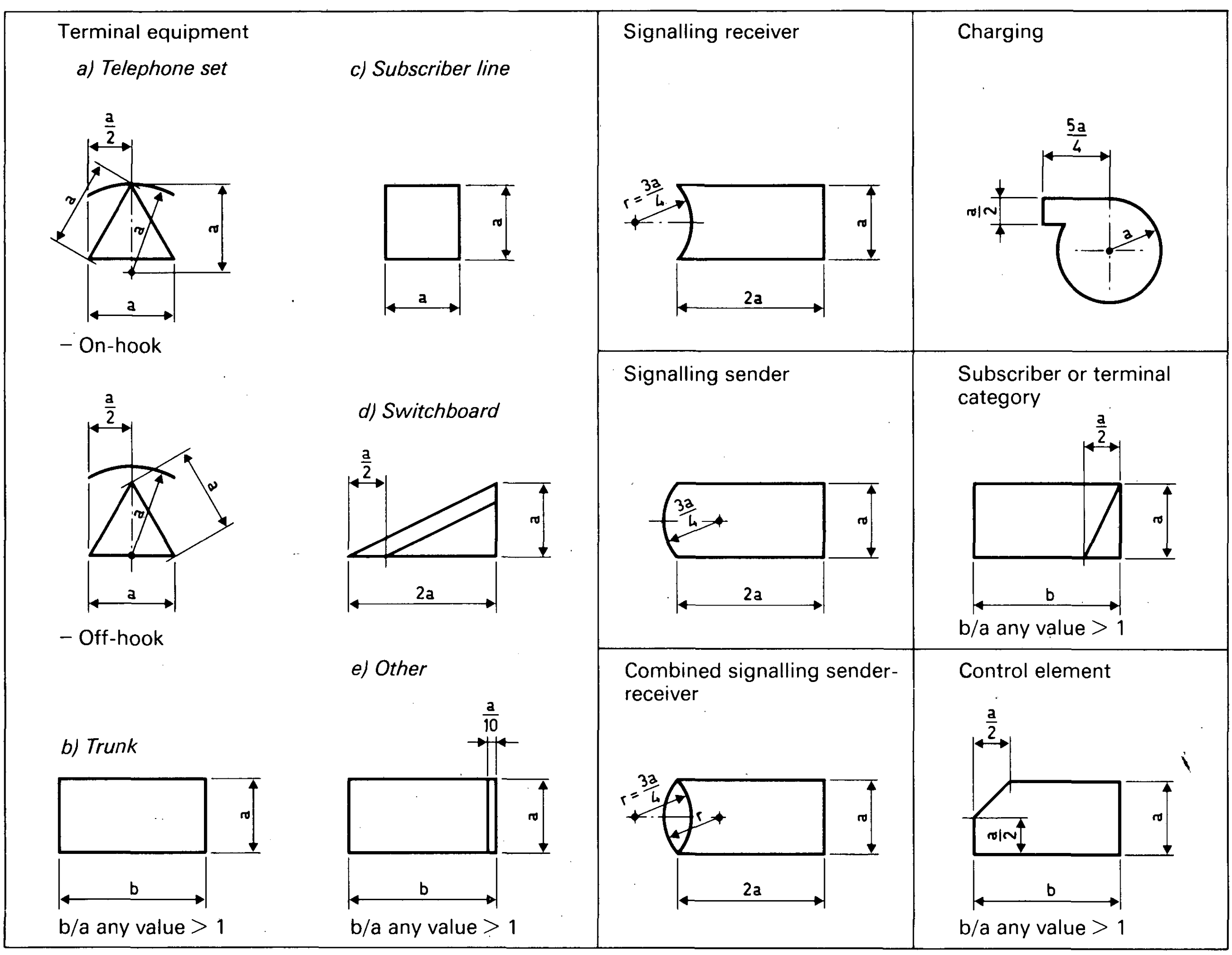
d) *Association of abbreviated text with symbols* – In some cases it is expected that abbreviated text will be associated with a PE in order to indicate the class of PE; e.g. the letters MFC associated with a receiver symbol to indicate that multi-frequency coded signals are to be received. In these cases, the PEs should incorporate enclosed space to permit the use of a very small number of alphanumerical characters.

e) *Limited set* – The total number of symbols should be kept to a minimum in order to permit easy learning of the pictorial method.

ANNEX C

(to Recommendation Z.103)

Recommended proportions for the basic sets of pictorial elements



CCITT-34110

DATA IN SDL

1 Introduction

This Recommendation defines the *data* concept in SDL; the SDL *data* terminology, the use of *pre-defined data types*, and the facility to define *new data types*.

The main occurrences of *data* in SDL are in *data type definitions*, *expressions*, the application of *operators*, *variables*, *values*, *constants* and *literals*.

Data type definitions

Data in SDL is principally concerned with *data types*. A *data type* defines a set of *values*, a set of *operators* which can be applied to these *values*, and a set of *axioms* defining the behaviour when these *operators* are applied to the *values*. The *values*, *operators* and *axioms* collectively define the *properties* of *data types*. These properties are defined by *data type definitions*.

SDL allows the definition of any needed *data type*, including *structuring* mechanisms (composite types), subject only to the requirement that such a definition can be formally specified. By contrast for programming languages there are implementation considerations which require that the set of available *data types* and, in particular, the structuring mechanisms provided (*array*, *structure*, etc.) be limited.

Expressions and operators

Expressions allow for the manipulation of *values* (by applying appropriate *operators*), to return new *values*.

Variables

Variables are objects which can be associated with a *value* by *explicit* or *implicit assignment*. When the *variable* is *accessed*, the *value* is returned.

Values, constants and literals

All *data types* have at least one *value*. For most *data types* there are *literal* (syntactic) forms to denote the *constant values* of the *data type* (for example for *Integers*). *Data types* for which there are *literals* to denote *values* are said to have *denotable constants*. There may be more than one *literal* to denote the same *constant value* (for example 12 and H'C both denote the same *Integer* constant), and the same literal denotation may be used for more than one data type. Some types do not have denotable constants. For example, values of the data type Stack can only be generated by the application of *operators* which return stack *values*.

In a specification language, it is essential to allow *data types* to be formally described in terms of their behaviour, rather than by composing them from provided primitives, as in a programming language. The latter approach invariably involves a particular implementation of the *data type*, and hence restricts the freedom available to the implementor to choose appropriate representations of the *data type*. The SDL approach allows any implementation providing that it is feasible and correct with respect to the SDL.

In SDL, all *data types* are *abstract data types*. Examples of these are given in § 5 where the *pre-defined data types* of the language are defined.

Although all *data types* are abstract, and the *pre-defined data types* may be re-defined by the user, some effort has been made in SDL to provide *pre-defined data types* which are familiar in both their behaviour and syntax. These are:

Array, Boolean, Character, Charstring, Duration, Integer, Natural, Powerset, PId, Real, String, Time and Timer.

Composite types can be formed by the use of *Struct data types*.

1.1 *The SDL data formalism*

In SDL, *data* is modelled by a *type algebra.* A *type algebra* is a set of *domains*, a *designated domain*, and a set of functions mapping between the *domains.* Each *domain* is the collection of all the possible *values* for a *data type.* The *designated domain* is the *data type* currently being described. The functions represent the *operations* of the *data type.*

The *domain* and *operations*, together with the behaviour (specified by the *axioms*) of the *data type*, form the *properties* of the *data type.*

Introduction of a *syntype* creates a subset of the *values* of an already defined *type.* Introduction of a *newtype* creates a distinct *new data type*, with *properties inherited* from the *parent*, but with different *identifiers* for these *properties.* In the concrete syntax, these *names* need not be distinct, and this ambiguity must be resolved by context.

A *generator type* is an incomplete *type description*; before it assumes the status of a *data type*, it must be *instantiated* by providing this missing information.

The functions which map between the domains of a data type are normally partitioned into two classes. The generator functions, which map onto the *designated domain*, and so produce (possibly new) *values* of the *data type.* All other functions are semantic functions and ascribe meaning to the *type* by mapping onto other defined *types.* The semantic functions include the predicates which map onto the *Boolean domain.*

2 Common language model

2.1 *General*

In Recommendation Z.104, the concepts of *variables*, *pre-defined data types*, *values*, *expressions* and informal *expressions* are used. This Recommendation rigorously defines *variables*, pre-defined data types, *values* and *expressions*, and also extensions to the Recommendations Z.101, Z.102 and Z.103 which allow the introduction of new *data types.*

2.2 *Abstract syntax*

This abstract syntax extends that defined by Recommendations Z.101, Z.102 and Z.103.

Data definition

A *data definition* is a *data type definition* or a *synonym definition.*

System definition

A *system definition* may contain *data definitions.*

Block definition

A *block definition* may contain *data definitions.*

Internal part block definition

An *internal part block definition* may contain *data type definitions.*

Channel substructure definition

A *channel substructure definition* may contain *data definitions.*

Process definition

A *process definition* may contain *data definitions* and *variable definitions.*

Procedure definition

A *procedure definition* may contain *data definitions* and *variable definitions.*

Input node

A *variable* mentioned in an *input node* must be defined in a *variable definition* and must have the same *data type* as the corresponding *data type* in the *signal definition.*

Data type definition

A *data type* definition contains a *type name* and either a *newtype* description or a *syntype* description.

Newtype description

A *newtype* description contains a possibly empty set of *value names*, a set of one or more *operator* introductions and a possibly empty set of *data type axioms.*

All *value names* in the *data type description* must be unique within the *data type.*

All *operator names* in the *data type description* must be mutually exclusive.

All *data type* names in the same context must be unique.

Operator introduction

An *operator* introduction either introduces one of the *universal operators* which it is permissible to introduce with any *data type*, or introduces an *operator name* together with the *operator typing.*

Operator

An *operator* is either a *universal operator* with a *data type identifier* or is a user-defined *operator identifier* with a *data type identifier.* In either case the *data type identifier* allows the *data type definition* which defined the *operator* to be established.

Operator typing

An *operator typing* contains the list of *data type identifiers* of the *parameters* to the *operator* and the *data type identifier* of the *result* of applying the *operator.*

At least one of the *data type identities* in the list, or the result, must be that of the *data type* being defined.

The result *type* must not be a *syntype.*

Axioms

Axioms are statements of truth which hold under all conditions for the *type* being defined, and thus specify the behaviour of types.

Assignment statement

An *assignment statement* contains an *assignment operator*, a *variable identity* and an *expression.* The *assignment operator* is either the *universal operator* for assign qualified with the *data type identity* of the *data type* of the *variable* or the *user defined insert operator.*

Syntype description

A *syntype* description contains a *syntype name*, the *parent data type identity* and the set of *value identities* of the *parent data type* which are valid for the syn *data type.*

Expression

An *expression* is either a *primary*, or an *operation.*

Operation

An *operation* contains an *operator* and a list of one or more *expressions.* There are as many *expressions* contained in the *operation* as there are *data types* defined for the *parameters* of the *operator.*

Primary

A primary is one of the following:

– a *synonym identity*,
– a *value identity*,
– a *variable identity*, or
– a *conditional expression.*

Conditional expression

A *conditional expression* is a *Boolean expression*, and a list of two *expressions* of the same *data type.*

Variable definition

A *variable definition* consists of a *variable name* list and a *data type identifier.*

Universal operation

A *universal operator* is either a *variable operator* or a *comparator.*

A *variable operator* is *declare, assignment* or *access. Declare* is used to *declare variables*; *assignment* is used for *assigning* to *variables* and *access* is used whenever a *variable identity* is *interpreted* as a *value.*

A *comparator* is either one of the *ordering operators* or an *equality operator.* An *ordering operator* is either less than or greater than.

All the *universal operators* include the *data type* to which they are relevant as a *qualifier* so that different *data types* introduce different *operators.* For example, the two *data types* square and cube introduce two *operator identities* for *assign*, square!assign and cube!assign.

For a *data type* D, the *operator typing* for comparators is:

D, D – > Boolean

For a *data type* D, the *assign operator* requires a *variable identity* of *data type* D and *value* of *data type* D.

For a *data type* D, the *access operator* requires a *variable identifier* of *data type* D and delivers a *value* of *data type* D.

For a *data type* D, the *declare operator* requires a *variable identifier.*

Synonym definition

A *synonym* definition contains a *synonym name* and a *constant expression.*

Constant expression

A *constant expression* is either a *constant value* or an *operation* all of whose parameters are *constant expressions.*

Constant value

A *constant value* is either a *value identifier* or a *synonym identifier.* A *synonym* may not be recursively defined.

2.3 *Interpretation rules*

2.3.1 *Process*

Instantiation of *process* takes place before the *start node* is *interpreted*, and causes a *declare operation* to be applied to every *variable name* which appears in a *variable definition* in the *process definition.* As a result of the *declare operator*, the *variables* declared become associations from *variable identities* to an *initial value* (which will be *undefined value* unless otherwise specified in the *axioms* for the *data types* of the *variables*). The *variable identities* declared when instantiating a *process* contain the *variable name*, the *process instance identity* and the *data type identity* from the *variable definition.*

2.3.2 *Procedure start node*

Calling a *procedure* causes *variables* defined within the *procedure* to be *created* for a *procedure* in a similar way to *instantiating* a *process.*

2.3.3 *Process graph*

The *interpretation* of an *output node* causes each of the *expressions* in the *output node* to be *interpreted* in the sequence specified, and the resulting values to be *assigned* to *anonymous implicit variables* which are associated with the *signal.* These *variables* are considered to be *declared* when the *output node* is *interpreted.* Each of these *variables* has the *data type* of *syn data type* associated with the corresponding position in the *signal definitions.* The *values* which are *assigned* to these *variables* are the *values* conveyed by the *signal.*

The *interpretation* of a *decision node* causes the *expression* contained in the *decision node* to be *interpreted* followed by the choosing of the *arc* which is associated with the *value* delivered by the *expression.*

Create request

The *interpretation* of a *create request node* causes each of the *expressions* in the *create request node* to be *interpreted* in the sequence specified.

The *instantiation* of the *process* being *created* then takes place together with the *declaration* of the *formal parameter* of the *process* and the *assignment* of the corresponding resulting *value* of each *expression* in the *create request node* to each *formal parameter.* The *created process* then executes separately from (but concurrently with) other *processes.*

Call node

A *call node* causes *expressions* used as *formal parameters* attributed with *IN* to be *interpreted* before the *interpretation* of the *start node* of the *procedure.* Each of the *expressions assigns* its value to the corresponding *actual parameter.*

2.3.4 *Procedure*

A *formal parameter* attributed with *IN/OUT* is *interpreted* as the *variable identifier* of the corresponding *actual parameter* in the context of the *procedure call.* A formal parameter attributed with *SIGNAL* is *interpreted* as the *signal identifier* of the corresponding *actual parameter* in the context of the *procedure call.*

2.3.5 *Assignment statement*

The *assignment statement* is *interpreted* as combining the old *value* of the *variable* with the *value* of an *expression* and binding the *variable identity* to this new *value.*

The *assignment operator* determines the rules for combining the old *value* of the *variable* with the value of the *expression.* These rules are determined by the use of the *assignment operator* in *data type axioms* of the *data type.* If the *assignment operator* is the *universal operator* for *assignment,* then the *variable* is *bound* to the *value* of the *expression.*

The *value* of the *expression* must be one of the *values* of the *data type* of the *variable* of the *assignment operator.* For the *universal operator* for *assignment* the *data type* of the *parameter* is that of the *variable.*

2.3.6 *Expression and primary*

An *expression* is *interpreted* as the *primary* which forms the *expression.* The *primary* is either an *operation,* a *synonym,* a *value identifier,* or a *variable identifier,* and is so *interpreted.*

2.3.6.1 *Operation*

An *operation* is *interpreted* as an application of the *operator* to the list of *values* obtained by *interpreting* the list of *expressions.* The *interpretation* of the *operation* is determined by the use of the *operator* in the *data type axioms* of the *data type.*

2.3.6.2 *Synonym identifier*

A *synonym identifier* is *interpreted* as the *constant expression* defined in the *synonym definition.* The *constant expression* is *interpreted* in the same way as an *expression.*

2.3.6.3 *Value identifier*

A *value identifier* is *interpreted* as the *value* it denotes.

The semantics of the *value* which a *value identifier* denotes is determined by the use of the *value identifier* in the *data type axioms* of the *data type.*

2.3.6.4 *Variable identifier*

A *variable identifier* is *interpreted* in one of the two ways depending on context. Within an *expression* a *variable identifier* is *interpreted* as an *access.* In the context of an *assignment statement* a *variable identifier* is *interpreted* as a *variable,* which is the *binding* of the *variable identifier* to a *value. Access* to a *variable* is *intepreted* as the *value* to which the *variable* is *bound,* except that *access* to the *undefined value* is *interpreted* and an *error.*

2.3.6.5 *Conditional expression*

A *conditional expression* is *interpreted* as the first or second *expression* in the list, depending on whether interpretation of the *Boolean expression* yields *true* or *false.*

2.3.7 *Data type definitions*

These are not *interpreted.*

3 SDL/GR

The standard SDL specification of the behaviour of *tasks, decisions,* etc. (i.e. the internal structure of these *nodes*) is the SDL/PR form. Thus there is no specific *graphic syntax* for data.

Where the *data definitions* (for *data types, variables* or *synonyms*) are included, they should be defined with or referenced by the diagram which includes them.

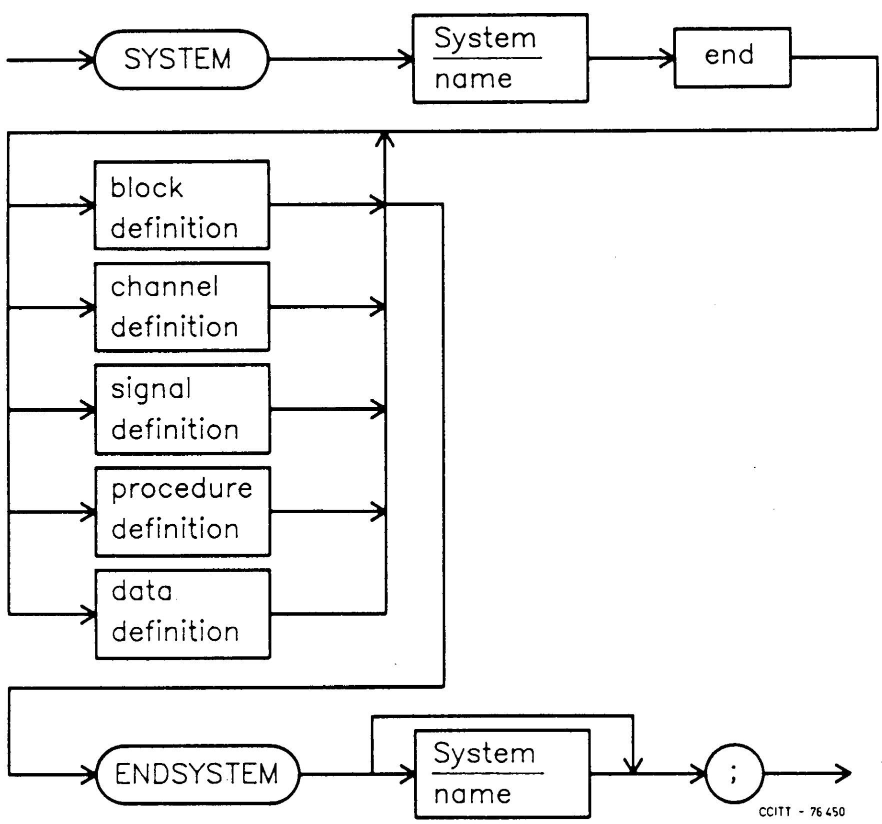
4 SDL/PR

4.1 *Addition to syntax*

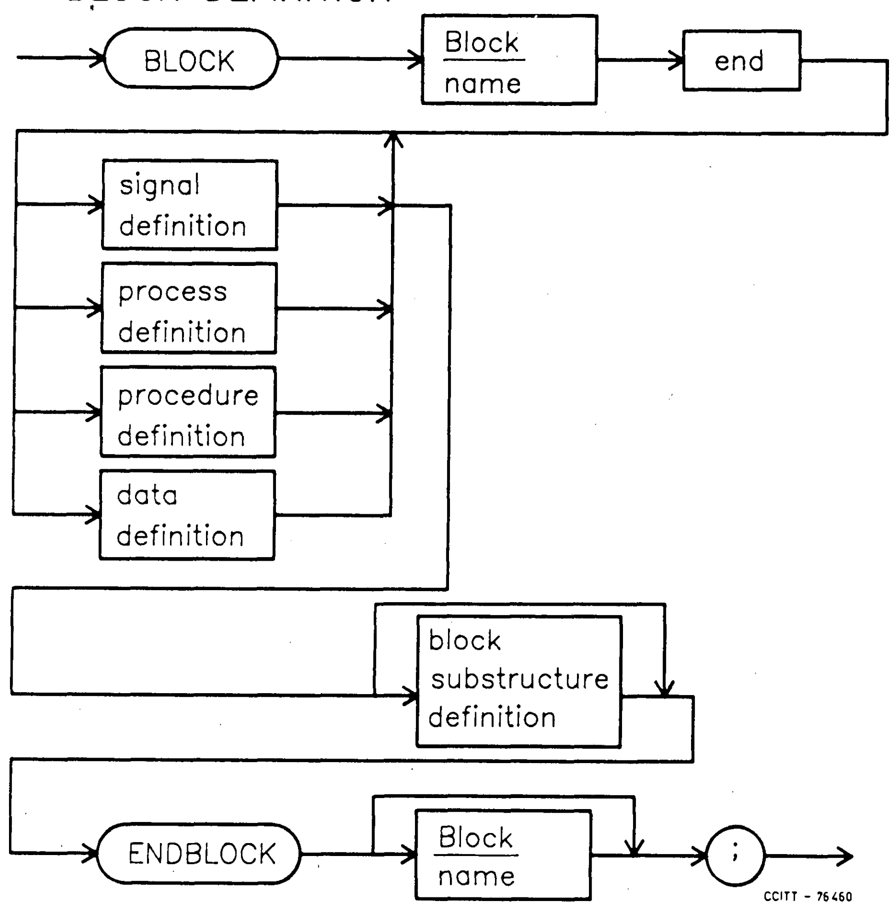
Data definitions are added to the syntax as defined in Recommendations Z.101, Z.102 and Z.103.

Data definitions may appear where a *signal definition* appears in a *system definition, block definition, block substructure definition* or a *channel substructure definition.* A *data definition* may also appear where a *variable definition* may appear in a *process definition* where a *procedure variable definition* may appear in a *procedure definition.*

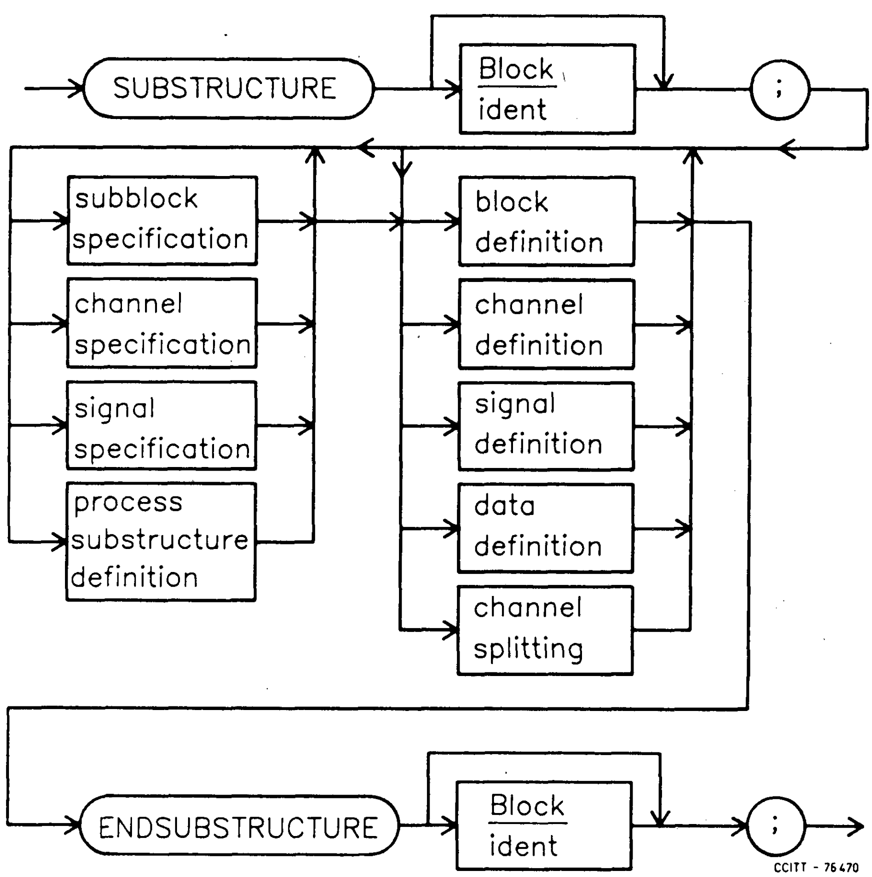
SYSTEM DEFINITION



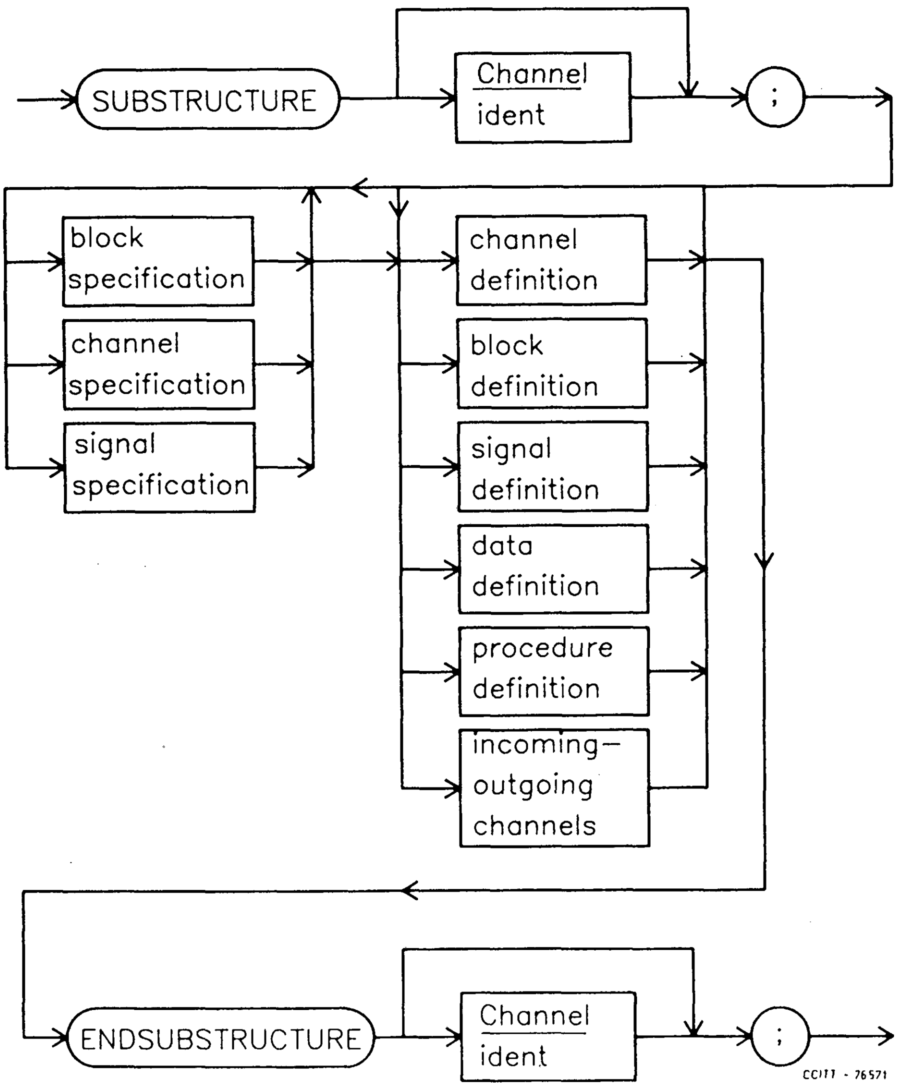
BLOCK DEFINITION

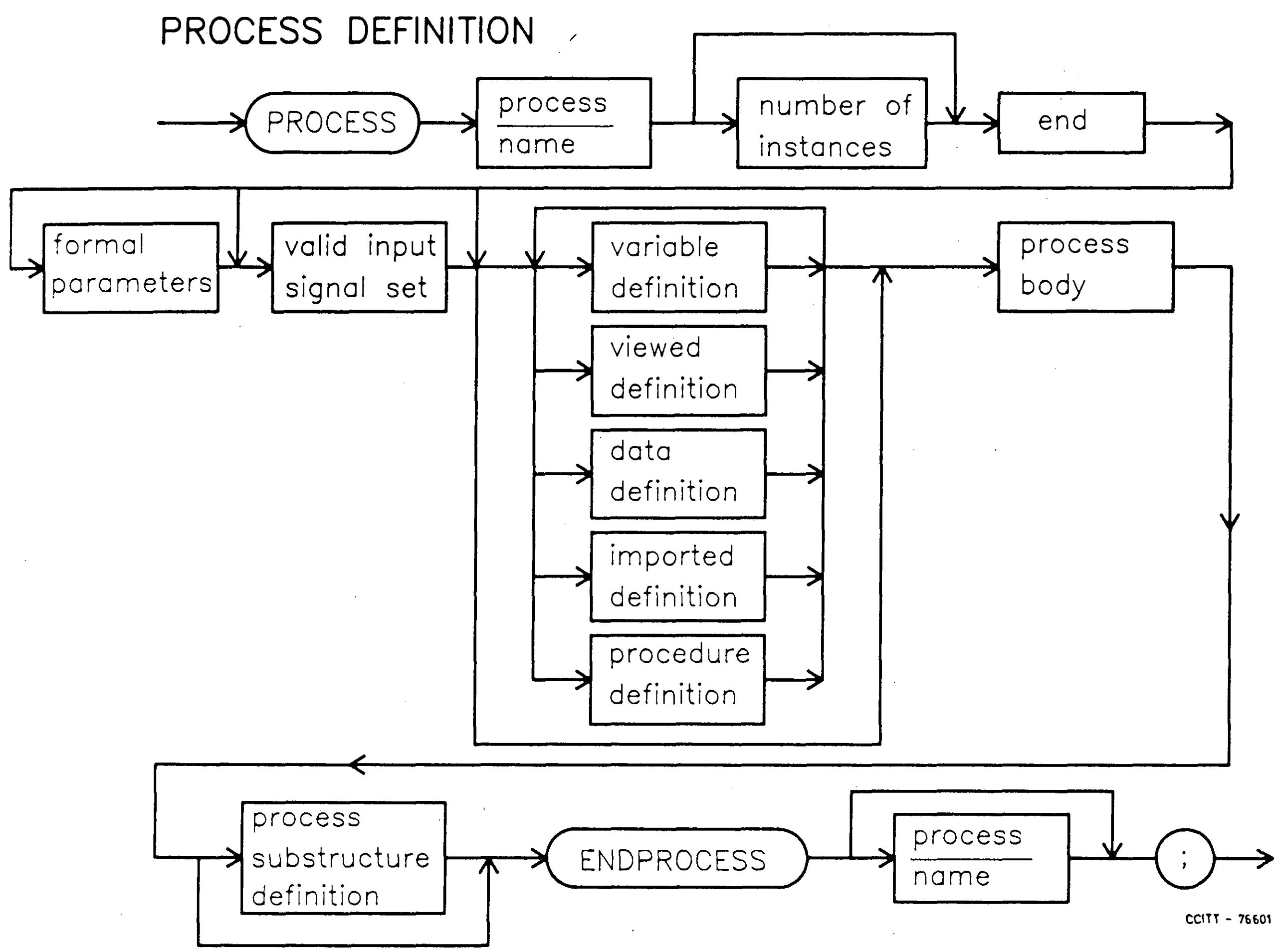


BLOCK SUBSTRUCTURE DEFINITION



CHANNEL SUBSTRUCTURE DEFINITION



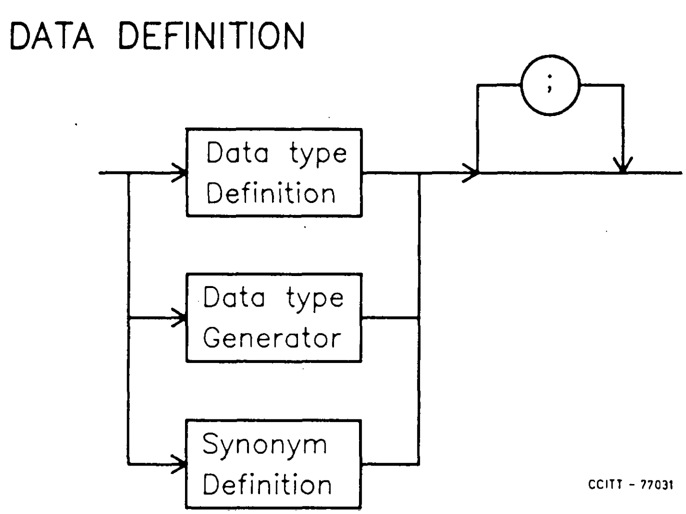


The syntax of *data type* is extended to include user defined *data type identifiers.*

The syntax for *assignment statements* and *expressions* is given. (Note expressions and assignment statements are mentioned in Recommendation Z.101 but not defined.)

4.2 *Data definition*

4.2.1 *Syntax*



4.2.2 *Semantics*

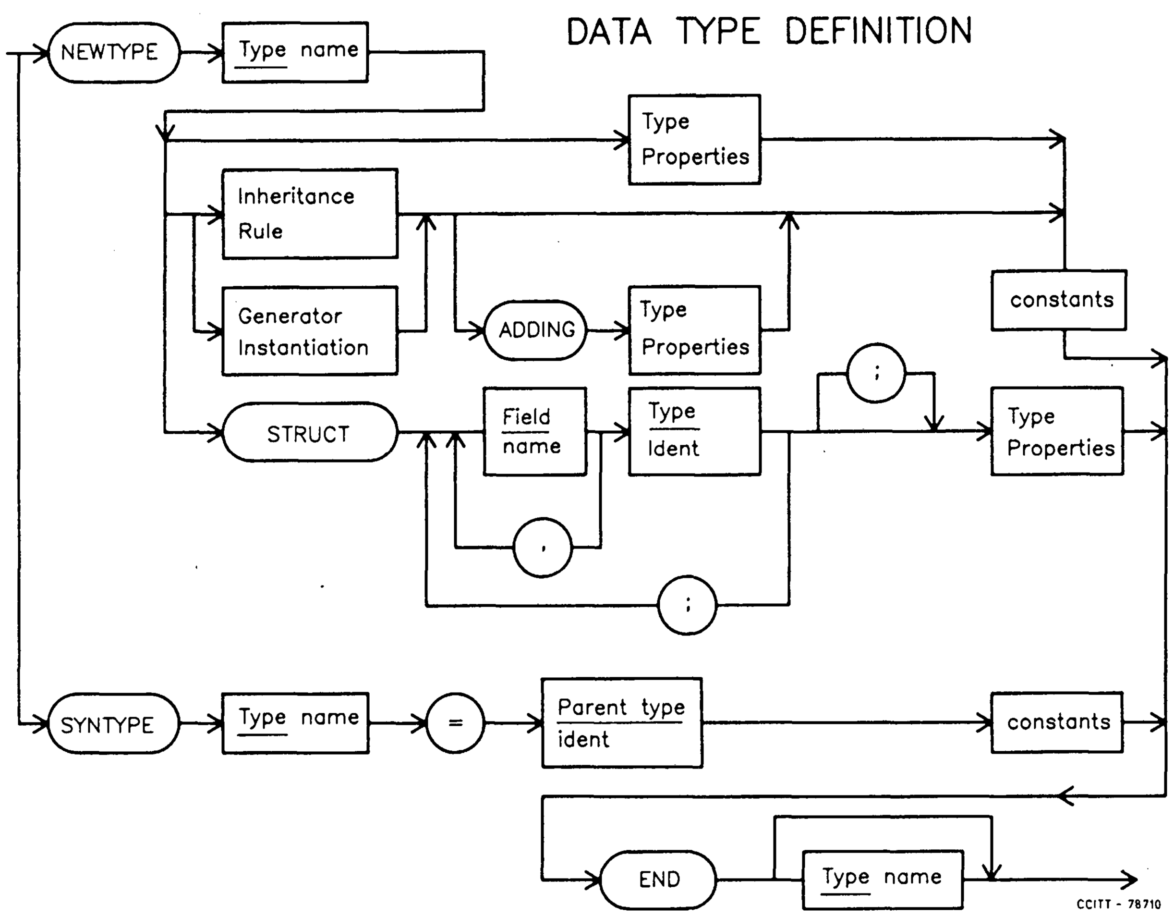
A *data definition* is used to introduce the *names* and *properties* of *data types, data type generators* or *synonyms.*

4.2.3 *Relationship to SDL Abstract Syntax*

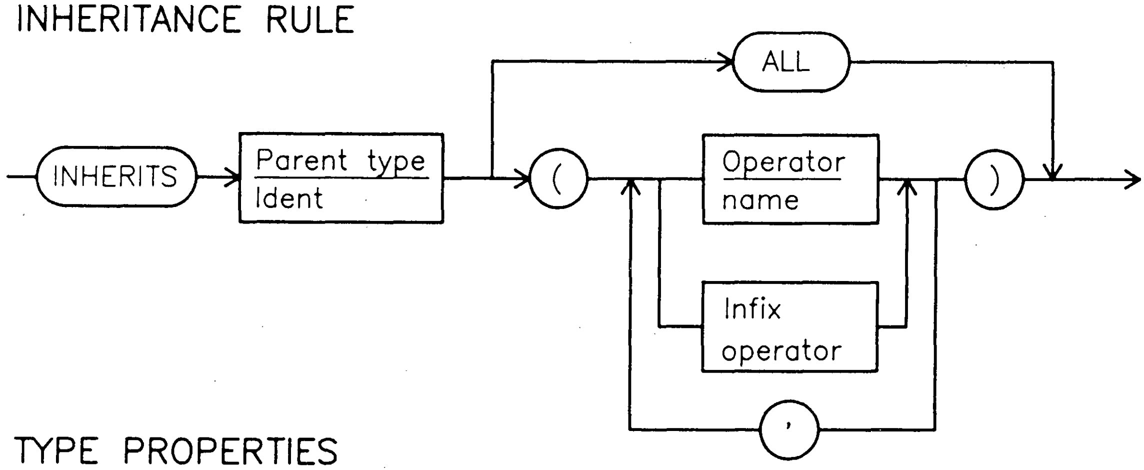
A *data definition* in SDL/PR represents a *data definition* in the *abstract syntax.* If the *data definition* is a *data type generator* then there is no direct correspondence with *abstract syntax,* as the *data type generator* serves only to define text which is considered to be textually expanded on *generator instantiation.*

4.3 *Data type definition*

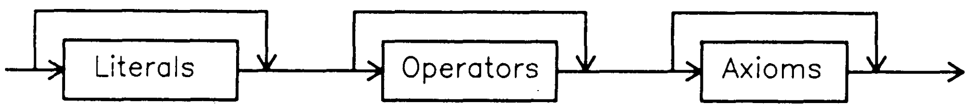
4.3.1 *Syntax*



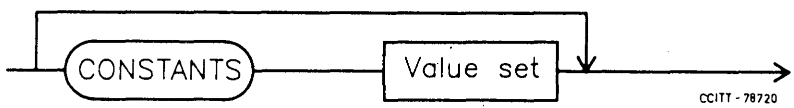
INHERITANCE RULE



TYPE PROPERTIES

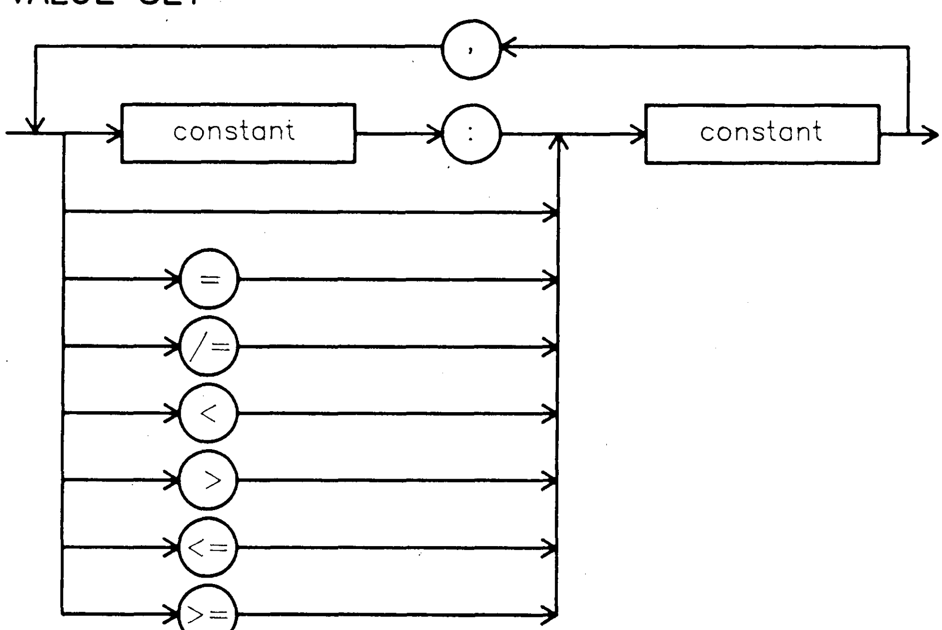


CONSTANTS



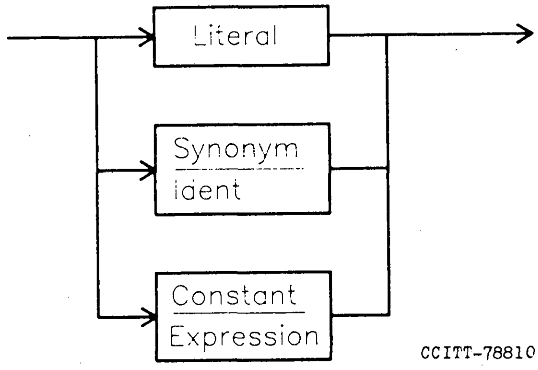
CCITT - 78720

VALUE SET



CCITT-78730

CONSTANT



4.3.2 *Semantics*

The name given in a *data type definition* is a *data type name.*

4.3.2.1 *NEWTYPE*

A *NEWTYPE data type definition* introduces a *new data type.*

If no *inheritance rule* is specified then the *new data type* is not based on any other *type.* The *type properties* are used to introduce any *literals* for that *type*, the *operators* applicable to the *type* and (optionally) the *properties* of the *type* by stating *axioms* which hold true for the *type.*

A *new type* may be based on another *type* by using *NEWTYPE* in combination with *inheritance rules* In this case, the *value* set of the *new data type* is disjoint from the *value set* of the *parent type.* Although the *values* and *operators* of the new *data type* are distinct from those of the parent data type, the *literals* and *names* of the *operators* for the *new type* will be *overloaded*, that is they will be the same *literals* and *names* as for the *parent type* and whether a *literal* or *name* is appropriate to the *new type* or *parent data type* will have to be decided either by *qualification* or by *context.* If the *binding* of a *literal* or *operator name* to a *type* cannot be determined then the SDL specification is *ambiguous* and hence *invalid.* When *ALL* is given for an *inheritance rule* then all the *operator names* are *overloaded* for the *new data type.* Otherwise the *operator names* specified in the *inheritance rule* must be *operator names* of the *parent type* and these *names* are defined for the *new type.* The *axiom set* and *literal set* of the *parent data type* is *inherited.*

As well as the *inherited literals*, *operator names* and *axioms*, a new *data type* may have additional *literals*, *operators* and *axioms* specified as *type properties* after the keyword *ADDING.* These *literals*, *operator names* and *axioms* must not conflict with those *inherited.*

A *data definition* of the form:

```
NEWTYPE X/* details */
    CONSTANTS /* constant list */
    END X;
```

is equivalent to

```
NEWTYPE Anon /* details */
    END Anon;
```

followed by

```
SYNTYPE X = Anon
    CONSTANTS /* constant list */
    END X;
```

The use of a *constant restriction* on a *NEWTYPE implicitly declares* an *anonymous NEWTYPE* (Anon above) without that restriction, which is then used as the *parent* of a *SYNTYPE* with the *constant restriction.* To enforce the *anonymity*, the *parent name* is stated to be distinct from all other *names* denoted in the SDL specification outside of the particular *implicit declaration.*

4.3.2.2 *SYNTYPE*

A *data type* may also be defined to have a subset of the *values* of the *parent data type* by using *SYNTYPE.* In this case the *value set* is either specified after the *keyword CONSTANTS,* or all the *values* of the *parent data type* have corresponding *values* in the *SYNTYPE. Variables declared* with a *SYNTYPE* may only be *assigned* the *values* specified.

Accessing a *variable* with a *SYNTYPE* yields a *value* of the *parent data type.* These *operations* of *declare, assignment* and *access* are the only *operations* allowed for *SYNTYPES.*

The *values* specified for a *SYNTYPE data type* must all be *values* of the *parent data type.* The *parent* of a *Syntype* is the second *type* nominated in the *Syntype* definition provided that *Type* is a *Newtype.* Otherwise the *parent* is the *parent* of the nominated *type.*

4.3.2.3 *Generator instantiation*

A *generator instantiation* is equivalent to the text of the *generator* with the *formal parameters* textually replaced by the *actual parameters.* Wherever the *generator name* is used in the text of the *generator,* it is replaced by the *name* of the *data type* or *generator* calling the *generator instantiation.* The equivalent text must complete a valid *NEWTYPE data type definition.* This *data type definition* formed by the textual expansion then defines the *properties* of the *data type name.*

4.3.2.4 *Struct*

A *data type definition* which includes a *Struct* implies *data types* for each *field name.* For a given *structure type* S, for each *field name* F_i and corresponding *type identifier* T_i, the following *axioms* are implicitly introduced (subject to *strengthening*; see below):

- the single *axiom* : extract!(insert!(S,Fi,I)', Fi) = I
- the *axiom set* : extract!(insert!(S,Fi,I)',Fj)
  = Extract(S,Fj);
  /* for all distinct Fi, Fj */

Where there are N *field names* in a *structure,* there will be N * N *axioms* of this form *implicitly introduced.* To guarantee resolution of *ambiguity,* SDL requires that *field names* within a given *structure* be unique.

Associated with the *structure* S is a set of *types,* one for each *field,* with *type name* S!Fi, the single *literal* Fi, and no other *properties.* This *type* becomes the *carrier* for the *field name* used in *Insert*! and *Extract*! operations.

The effect of a *Struct* definition is to create a (programming-language like) structure or record, although the definition may include additional *axioms* to *strengthen* the *behaviour* of the *type.* Where additional *axioms* explicitly introduced by the user conflict with the implicit default *axiom set* for that *Struct,* the inconsistency is resolved by discarding implicit *axioms.* Introduction of explicit *axioms* into a *Struct* requires great care.

4.3.3 *Relationship to abstract syntax*

A *data type definition* represents a *data type definition* in the *abstract syntax.* The *type name* represents the *type name* in the *abstract syntax.*

4.3.3.1 *NEWTYPE*

The *keywords NEWTYPE* and *END* embrace the *abstract syntax* concept of a *data type description.* The *value name set, operator introductions* and *type axiom set* in the *abstract syntax* are represented as follows:

a) *Value name set*

   The set of *literals* given by the *literals* in the *type properties* combined with the set of *literals* for the *parent data type* if *INHERITS* is specified.

b) *Operator introduction set*

   The set of *operators* given by the *operators* in the *type properties* combined with the set of *operators* of the *parent data type* if *INHERITS* is specified.

c) *Type axiom set*

   The set of *axioms* (if any) given by the *axioms* in the *data type properties* combined with the set of *axioms* of the *parent data type* if *INHERITS* is specified. If the *axiom set* is omitted, or is incomplete, then at least some operations can only be *interpreted informally.*

4.3.3.2 *SYNTYPE*

The *keywords SYNTYPE* and *END* embrace the *abstract syntax* concept of a *syntype description.* The *parent type identifier* represents the *parent data type identifier* in the *abstract syntax.* The *value set* represents the *value set* in the *abstract syntax.*

4.3.3.3 *Generator instantiation*

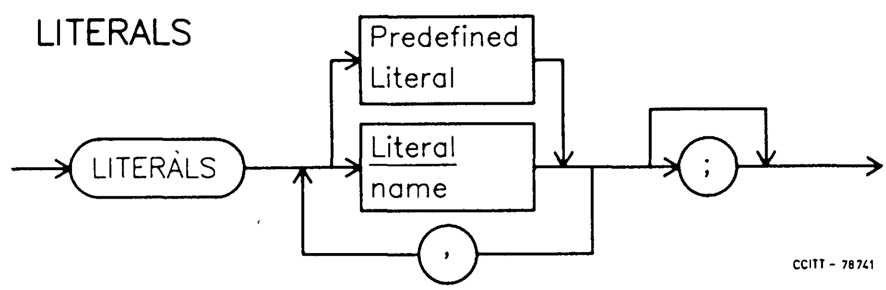
A *generator instantiation* denotes the text which would be obtained by textually expanding the *generator* as in § 4.3.2.3 so that it has the same relationship with the *abstract syntax* as the equivalent text.

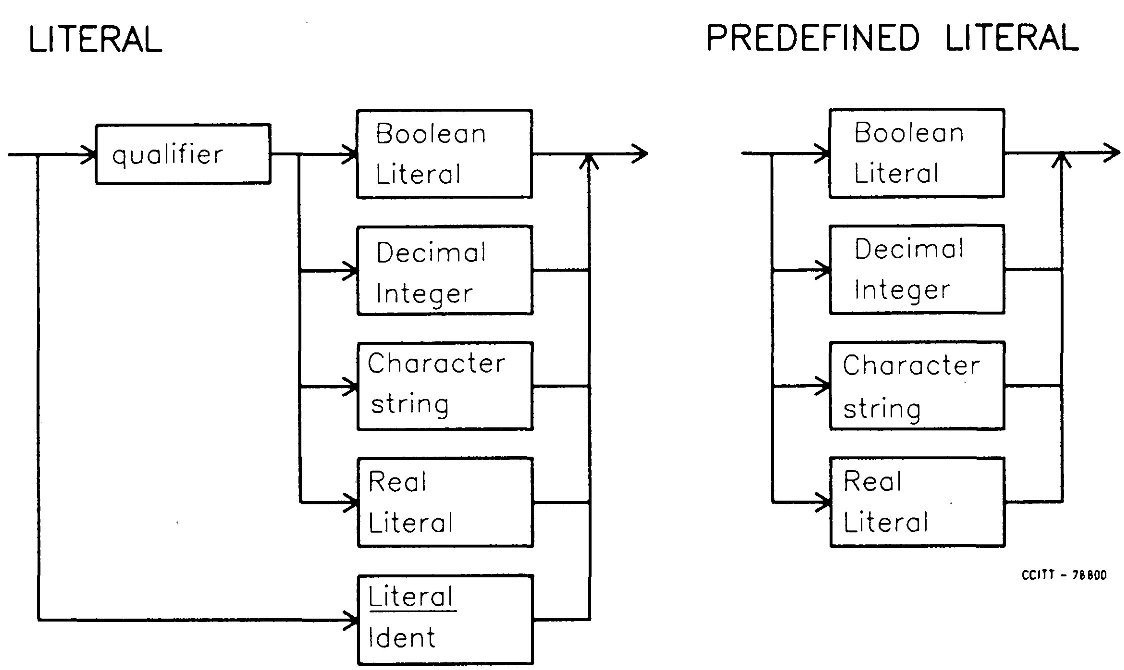
4.3.3.4 *STRUCT*

A *Struct* denotes the text obtained by explicitly nominating all relevant *properties* and thus (as for a generator) has the same relationship to the *abstract syntax* as the text so denoted.

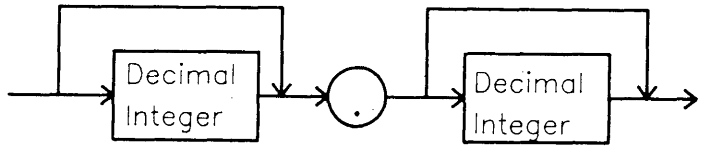
4.4 *LITERALS*

4.4.1 *Syntax*

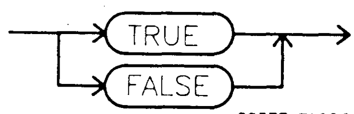




REAL LITERAL



BOOLEAN LITERAL



CCITT-78820

4.4.2 *Semantics*

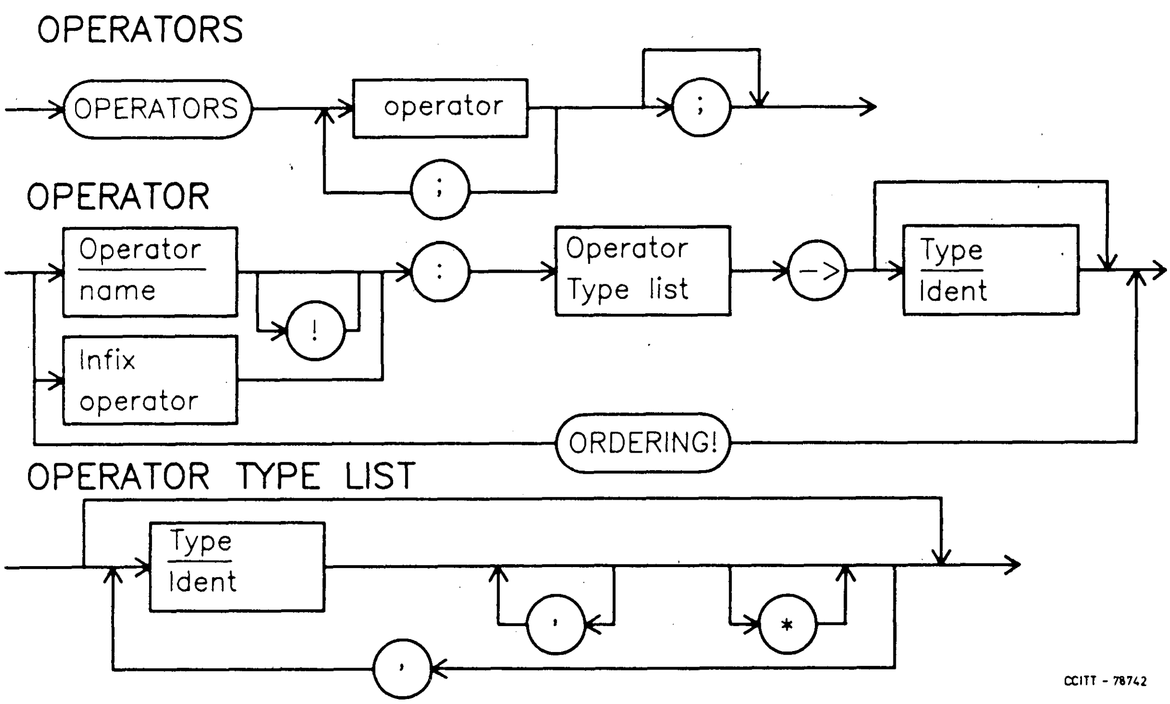
The *literals* which are used to denote the *values* of a *data type* are either *predefined* (for *predefined data types* or *data types* based on *predefined data types*) or are introduced by the list of denotations for the *literals* of a *data type* after the *keyword LITERALS.* Where a *type* includes the *Ordering! operations*, the literals should be conventionally nominated in ascending order.

4.4.3 *Relationship with abstract syntax*

The *literal names* introduced by the *literals* part of *data type properties* represents the *value names* of a *data type description* in the *abstract syntax.*

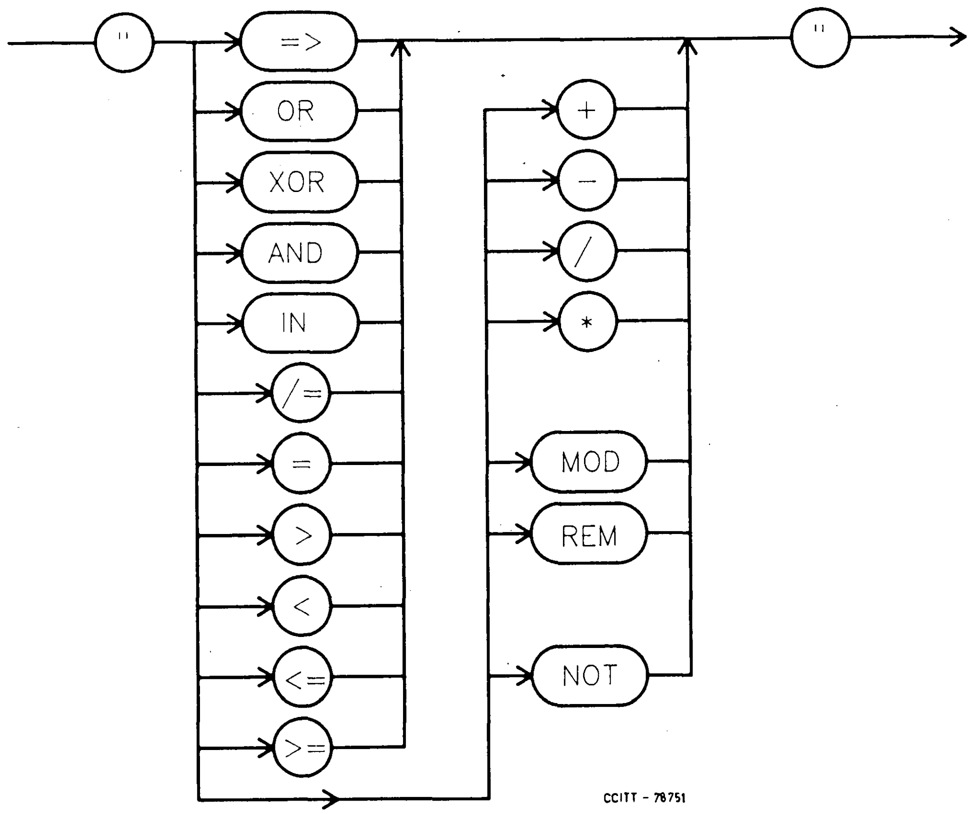
4.5 *Operators*

4.5.1 *Syntax*



CCITT - 78742

INFIX OPERATOR



4.5.2 *Semantics*

The *operators* of *data type properties* introduce the *names* for *operators* and the *parameterization* of these *operators.* The *parameterization* determines the number of *parameters* required and the *data type* of each *parameter* and also the *data type* of any *values* returned.

4.5.2.1 *Operator names*

The *ordering operators* are specified by including *Ordering*! in the *operators.* This is shorthand notation for introducing the following *operators* for a data type D.

“ < ” : D, D – > Boolean
“ > ” : D, D – > Boolean
“ <=” : D, D – > Boolean
“ >=” : D, D – > Boolean

The names of *infix operators* e.g. +, AND, OR, are enclosed in quotes in the *operators.* They may be used as *prefix operators* by using this quoted form, that is:

“+” (a,2)

is equivalent to a + 2.

An *operator name* may be optionally followed by an exclamation mark, which denotes that the *operator identity* may only be referred to directly in *data type definitions.* The exclamation mark forms part of the *name* of the *operator* and so must always be given when the *operator* is used.

The *names Assign!*, *Declare!*, *Access!*, "=" and "/=" are defined implicitly as *operators* for all *data types* with the *implied typing* for a *data type* D.

Assign! : D'*, D– >
Declare! : D'* – >;
Access! : D' – >;
"=" : D, D – > Boolean;
"/=" : D', D – > Boolean;

The *Assign* operator is the infix operator, ":=". There is no *Declare* operator as its application is implied from *declarations.* The *Access* operator is implied whenever a *variable* is mentioned in a *context* which requires a *value.* The equals and not equals *operators* are the *infix operators* "=" and "/=" respectively.

4.5.2.2 *Operator typing*

The list of *data type identifiers* after the colon and before the symbol (– >) is called an *operator type list.* This *operator type list* specifies the *data types* of values which the *operator* requires. If one or more *data type identifiers* has a *prime attribute,* then the *operator* is an *active operator* otherwise it is a *passive operator.* An *active operator* can change the *values* associated with *variables,* whereas a *passive operator* is purely functional and so cannot change the *values* associated with *variables.*

For a *passive operator* all *type identities* in the *type list* specify that *actual parameters* of the *operator* have to be interpreted as *expressions.* Each of those *expressions* must yield a *value,* which is a member of the *set of values* of the *data type* of the corresponding position in the *data type list.*

For a *passive operator* there must be a *data type identifier* after the symbol (– >). This *data type identifier* specifies that the *operator* when applied, will yield a value of this *data type.* For an *active operator* some of the *data type identities* in the *operator type list* are followed by *primes* (').

A *prime* specifies that the *operator* requires a *variable* of the given *data type* as a *parameter,* and the *value* associated with the *variable* may be changed. No two *data type identifiers* may be followed by the same number of *primes* in the input *data type list.*

The number of *primes* distinguishes one *primed parameter* from another when an *active operation* is used with *priming* to denote the *value* associated with a *variable* given as a *parameter* (this is permissible only in *axioms*). For example:

```
OPERATORS SwapAndAdd: Int', Int'' – > int
        /* fragment of data type definition */
AXIOM SwapAndAdd (a,b)'= b;
        /* axiom stating the first parameter receives the value of the second parameter */
    SwapAndAdd (a,b)'' = a;
        /* axiom for second parameter */
    SwapAndAdd (a,b) = a+b;
        /* result axiom */
```

If a *primed parameter* is followed by an asterisk, then the *initial value* of the *variable* is not *accessed* when the *operation* is used. (Note that the *axioms* must be consistent with this, otherwise the SDL specification is *invalid.*)

If the *data type identifier* after the symbol (– >) is omitted for an *active operator,* the *operator* may not be used within an *expression.* There may be both *primed* and *unprimed parameters* in an input *data type list.*

There is syntactic ambiguity between a *data type name* followed by a quoted character string in a name string, and a primed *data type identifier* in a *data type list.* These ambiguities arise when a prime after a *type name* in a *type list* is followed by a *prime,* a comma or a minus sign (part of – >). In all cases the *prime* is taken as *priming* of the *data type identifier* and not as starting a quoted character string.

4.5.2.3 *Insert! and Extract! operators*

To allow the *axiomatic definition* of *arrays* and *structures,* there are two *predefined operator names,* which have a special denotation outside *data type definitions.* These *operators* are *Insert*! and *Extract*!. *Insert*! is an *active operator* and must be defined with the *data type identities* such that the first *data type* is *primed* and the other *types* are *unprimed.*

To apply *Insert*! outside *data type definitions*, the first *parameter* (which must be a *variable*) is written, followed by an open parenthesis, all the remaining *parameters* except the last, then a closing parenthesis and ":=", and finally the last parameter.

Thus with A a *variable* the *data type* of which has *Insert*! defined and i1, i2 and e *expressions* approriate to *Insert*! for this *data type.*

Insert!(A, i1, i2, e)

is written as

A(i1, i2) :=e

Since *Insert*! can be defined for more than one *data type*, the appropriate *Insert*! is determined from the *type* of the *variable. Insert*! must be defined with at least two *parameters* and must return the same *data type* as the first *parameter.*

Extract! is a *passive operator* which requires a *variable* as the first *parameter* for semantic reasons. To apply *Extract*! the first *parameter* is written followed by all the other *parameters* in parenthesis.

Thus,

Extract!(A, i1, i2)

is written as

A(i1, i2)

The application of *Extract*! is determined by the *type* of the *variable.*

4.5.3 *Relationship to abstract syntax*

For a *passive operator* an *operator name* or *infix operator* represents an *operator name* in the *abstract syntax.* For a *passive operator* the input *type list* represents the list of type *identities* of *parameters* in the *abstract syntax.* The *data type identifier* after the symbol (– >) represents the *data type identity* of the result of applying the operator.

An *active operator* is related to the *abstract syntax* by re-writing into *passive operators.* This also defines the ordering behaviour of the *operation.* For each *primed parameter* there is an implicit *passive operator* which returns the *value* required by the *axiom set.* For each *primed parameter* without an asterisk there is an *implicit variable* in each *process instance* using the *operator* with the same *data type* as the *parameter* which receives the *initial value* of the *parameter.*

The list of *data type identities* of *parameters* in the *abstract syntax* for each *implicit passive operator* is represented by the input *data type list* ignoring any asterisked parameters. There are as many *implicit passive operators* as there are *primed parameters* in the input *data type list* and each *primed parameter data type* is used as the *data type identity* of the result of applying one of these *operators.* For example:

```
OPERATORS
    complex: Integer', Integer", Integer, Integer''' * – > Bool
        /* swaps first two parameters, puts the sum of the first three parameters in the fourth parameter
        and returns true if the second and third parameters were equal */
AXIOMS
    complex (a,b,c,d)' = b;
    complex (a,b,c,d)" = a;
    complex (a,b,c,d)''' = a+b+c;
    complex (a,b,c,d)· = (b+c);
        /*see § 4.9.2 for the use of primes in axioms */.
```

The implicit operators are:

implied1!: Integer, Integer, Integer – > Integer
implied2!: Integer, Integer, Integer – > Integer
implied3!: Integer, Integer, Integer – > Integer
implied4!: Integer, Integer, Integer – > Boolean

If the *implied variables* are V1, and V2, an application of complex in a statement is equivalent to:

V1 := a; V2 := b;

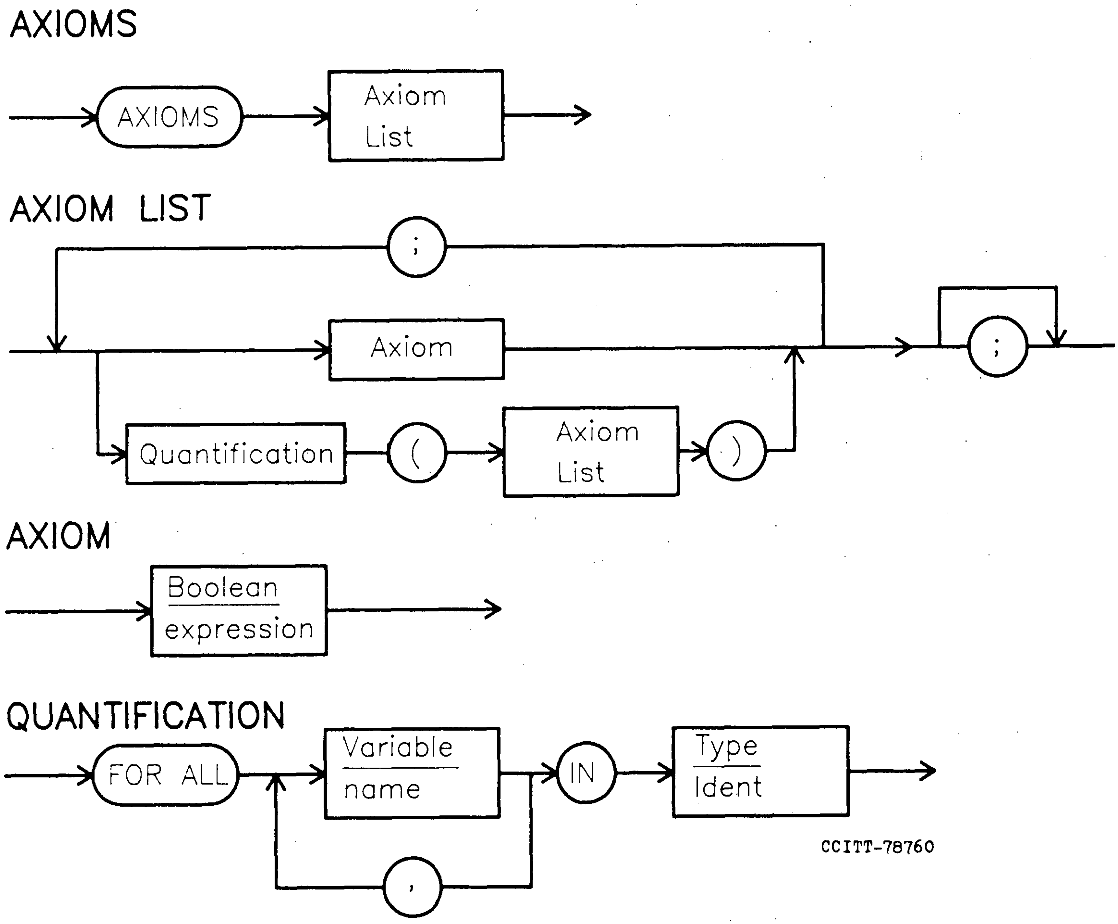
followed by the statement, but with V1 substituted for a, V2 substituted for b, and implied4! substituted for complex, followed by

a:=implied1! (V1,V2,c);
b:=implied2! (V1,V2,c);
d:=implied3! (V1,V2,c);

Where more than one *active operator* is present, in a statement, they are substituted in the order in which they would be *interpreted.*

4.6 *Axioms*

4.6.1 *Syntax*



4.6.2 *Semantics*

The *axioms* are a set of *Boolean expressions* which hold true for all *values* of the *variables* in the *axioms.*

Within an *axiom* a "*variable*" is never the name for a *process* or *procedure variable*, which has a *value* associated with it, but is used to denote that all *values* of a specific *type* may be substituted for the *variable* and the *axiom* is still true. When considering this substitution a given *variable name* always represents the same *value* in one *axiom.* For example in:

OPERATORS even: Integer -> Boolean

AXIOMS even (0) = True; /* axiom 1 */
even (1) = NOT even (i+1); /* axiom 2*/

The *axiom* 2 must hold for i=2, i=3, i=4 etc., that is,

even (2) = NOT even (2+1)
even (3) = NOT even (3+1)
even (4) = NOT even (4+1)

Usually the *data type* of a *variable* in an *axiom* can be determined by *context* e.g., in the above example it is required to be of *data type* int by the *operator syntax.*

Sometimes because of *overloading* of *names* and symbols (such as "+") in SDL, it is not possible to determine the *data type* of an *axiom variable* by context, so that the *optional quantification* is needed. *Quantification* forces a *variable* to be of particular *type.* It the use of a *variable* within one *axiom* is *ambiguous* or *inconsistent* then the SDL/PR is *invalid.*

Quantification also allows a *variable name* to represent the same substitution in more than one *axiom.*

The *variable names* chosen for *variables* in *axioms* must be distinct from *literals* appropriate to the *context* where the *variable* is used.

Since the *operators* "=" and "/=" are *implied* for all *data types,* the following *axioms* are always *implied*:

"/=" (a,b) = NOT ("=" (a,b))
"=" (a,a);
"=" (a,b) AND "=" (b,c) => "=" (a,c);
"=" (a,b) => "=" (b,a);

Whenever *Ordering*! is specified, the following *axioms* are *implied*:

"<" (a,b) => NOT ">" (a,b);
">" (a,b) => NOT "<" (a,b);
"<" (a,b) AND "<" (b,c) => "<" (a,c);
NOT "<" (a,a);
"<=" (a,b) => "<" (a,b) OR "=" (a,b)
">=" (a,b) => ">" (a,b) OR "=" (a,b)

4.6.3 *Relationship to abstract syntax*

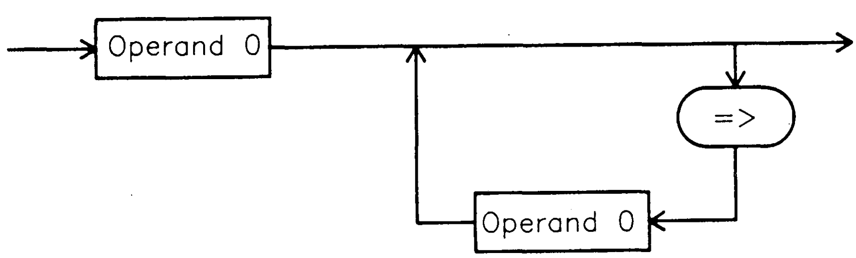
The *axioms* represent the *data type* axioms in the *abstract syntax.* Each *axiom* represents an *axiom* in the *abstract syntax.*

Quantification represents *quantification* in the *abstract syntax,* except that there is *implies quantification* in the *concrete syntax* for all *axiom variables* whose *data type* is determined by *context.*

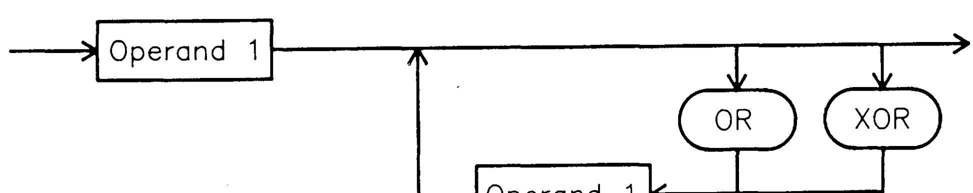
4.7 *Expressions*

4.7.1 *Syntax*

EXPRESSION

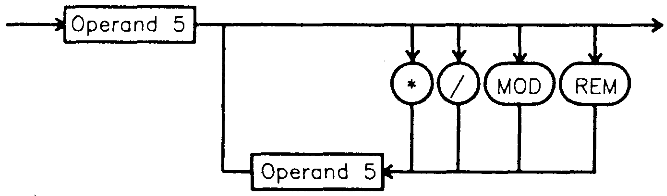


OPERAND 0

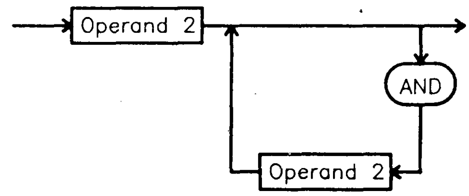


CCITT-78650

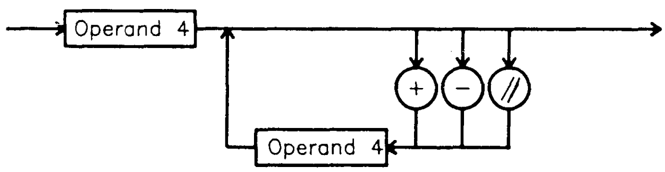
OPERAND 4



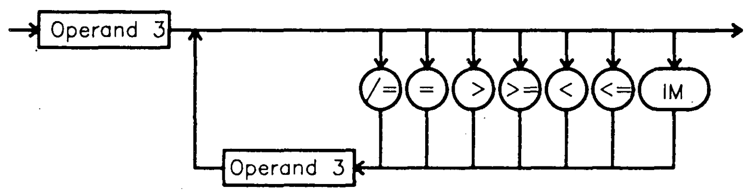
OPERAND 1



OPERAND 3

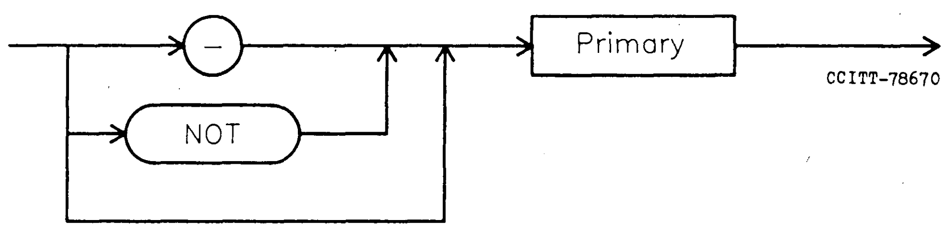


OPERAND 2



CCITT-78660

OPERAND 5



CCITT-78670

4.7.2 *Semantics*

An *expression* is either a *primary* or is the application of a number of "*infix*" operators.

The order of application of *operators* is determined by their appearance in the *syntax* in a similar way to programming languages such as CHILL (see Recommendation Z.200). However, SDL also allows the *Boolean implication operator* (= >), which has lower *precedence* than any other *operator.* This has the value *FALSE* for *Boolean operands* if the left hand *operand* is *TRUE* and the right hand *operand* is *FALSE.* Otherwise for *Boolean operands* the *implies operation* has the value *TRUE.*

Normally all *operators* will have the same *properties* and validity as defined in programming languages, but it should be noted that in SDL the user may define new meanings for these *operators* by including them in *data type definitions.* Nevertheless, the *precedence* of "*infix*" *operators* may not be changed.

The *value* yielded by a valid *operation* is determined by the *axioms* in *data type definitions.*

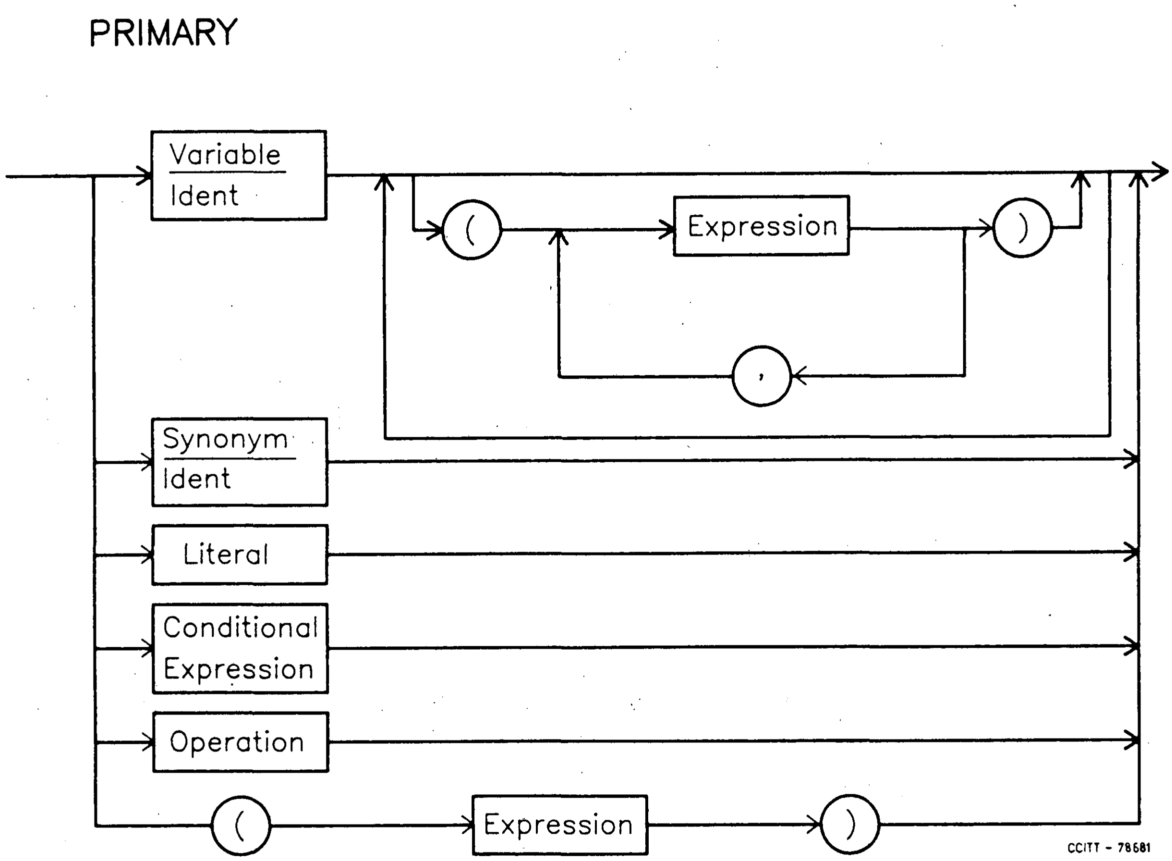
4.7.3 *Relationship to abstract syntax*

An *expression* represents an *expression* in the *abstract syntax*.

An "*infix*" *operator* can be *overloaded* and may represent any one of a number of *operators* in the *abstract syntax*. The *overloading* is resolved in two ways: either by the number and *type* of *parameters* or, in the case where the *parameters* themselves are *overloaded*, by the *data type* required in the *context* in which the *operation* is used.

4.8 *Primary*

4.8.1 *Syntax*



4.8.2 *Semantics*

A *primary* is a *variable identity* or a *literal* or a *synonym identity* or a *conditional expression* or an *operation* or a *bracketed expression*.

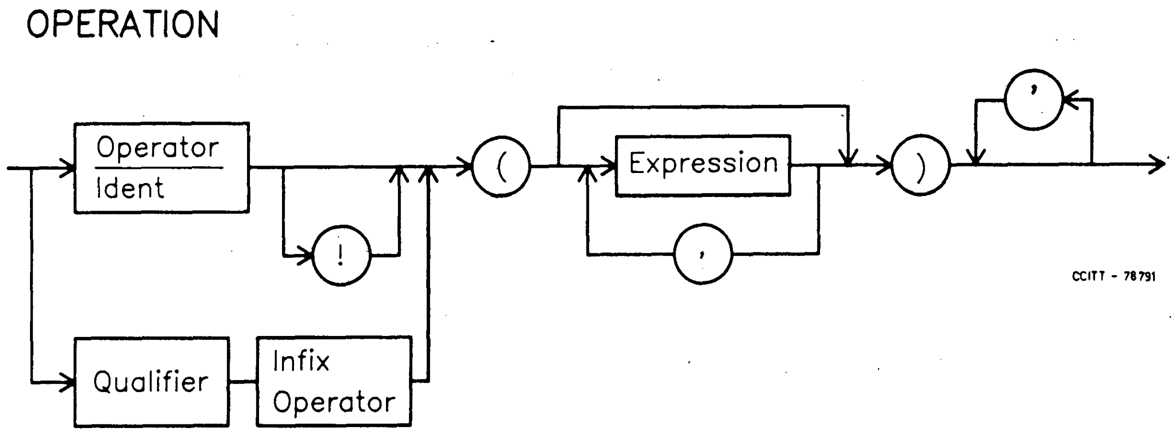
4.8.3 *Relationship to abstract syntax*

A *variable identity* or *literal* or *synonym identity* or *conditional expression* or *bracketed expression* represent a *variable identity access, value identity, synonym identity, conditional expression* or *expression* respectively in the *abstract syntax*.

When a *variable identity* is referred to in an *axiom* it represents an *axiomatic variable* rather than a reference to an *access* to a *variable* declared for a *process* or *procedure*. The *data type* of an *axiomatic variable* is determined by *context*.

4.9 *Operation*

4.9.1 *Syntax*



4.9.2 *Semantics*

An *operation* is the application of an *operator* defined in a *type definition.* The number and *type* of the *expressions* used as *actual parameters* must be consistent with the definition of the *operator identity.* These *parameters* may be used to determine which *operator* is being applied if the *operator name* is *overloaded.*

If the *operator* is being applied in an *axiom* then if the *name* is defined with an exclamation mark then this exclamation mark must be repeated in the *axioms.*

An *operator* defined with an exclamation mark may not be used outside a *type definition.*

The *primes* after the closing bracket of the *operator* may only be used in an *axiom* and denote that the *operation* has the resultant *value* of the *parameter* defined with that number of *primes.* For example:

```
OPERATORS
    example: ti', t2" ->

AXIOMS
    example: (v1,v2)" = v1
    /*value of v1 put in v2 */
```

When the *typing* of a *parameter* of an *operator* is specified with *primes,* the actual parameter must be a *variable,* except in the *context* of an *axiom.*

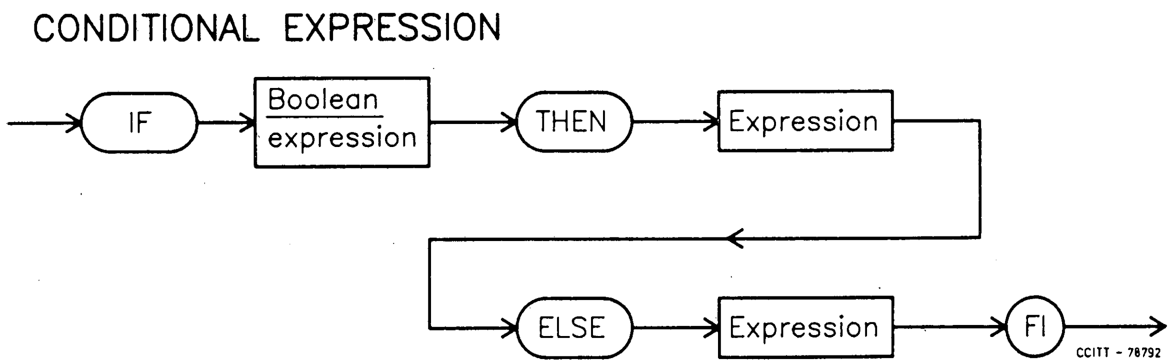
4.9.3 *Relationship to abstract syntax*

A *passive operator identifier* represents an *operator* in the *abstract syntax.* For a *passive operation,* the *expression* list represents the *expression* list in the *abstract syntax* for that *operation.*

An *active operation* represents *assignment* to *implicit variables* which are then used as *arguments* to *implicit passive operations,* whose *values* are *assigned* back into *variables* given as *actual parameters* (see § 4.5.3). Within an *axiom* an *active operation* represents an application of the appropriate *implicit passive operation* determined by the number of *primes* appended to the *operation.*

4.10 *Conditional expression*

4.10.1 *Syntax*



4.10.2 *Semantics*

The *conditional expression* is *interpreted* by *interpreting* the *expression* after *THEN* if the *Boolean expression* is *true*, and yields the *value* of the *expression* after *ELSE* otherwise.

Within an *axiom* each branch of the *conditional expression* need only be valid for the condition's under which it is selected. For example since log(x) is not defined for negative number, in

IF r > 0 THEN log(r) ELSE 0.0 FI

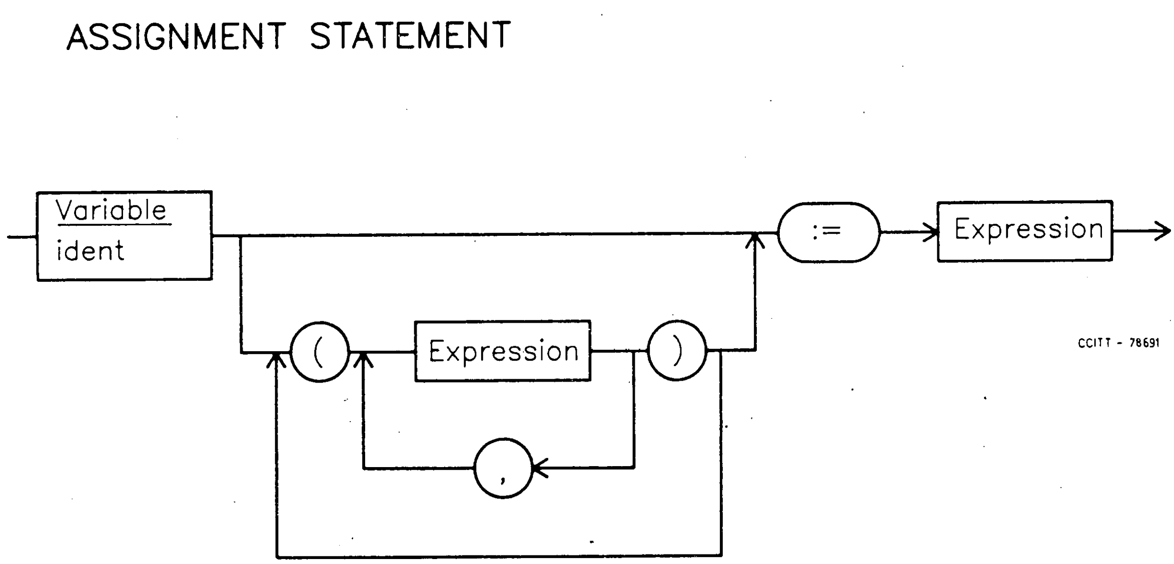
if does not matter that for r <= 0, log(r) is undefined.

4.10.3 *Relationships to abstract syntax*

A *conditional expression* represents a *conditional expression* in the *abstract syntax.*

4.11 *Assignment statement*

4.11.1 *Syntax*



4.11.2 *Semantics*

An *assignment statement* allows a *value* to be associated with a *variable.*

The *value* yielded by the *expression* on the right hand side is *assigned* to the *variable* or an element of the *variable* on the left hand side.

4.11.3 *Relationship to abstract syntax*

An *assignment statement* represents the application of either an *Assign!* operator or as an *Insert!* operator.

There is a mapping between the *syntax* of an *assignment* and the application of the appropriate *operator.*

Where no brackets are used on the left hand side of an *assignment* then the *assignment* represents the use of *Assign!*, such that

V : = e represents Assign!(V,e)

When a single pair of brackets is used, such an assignment represents *Insert!* such that

a(i) : = e represents Insert! (a,i,e)

and

a(i,j) : = e represents Insert! (a,i,j,e)

When multiple brackets are used an *assignment* is represented be recursive substitution such that

a(i)(j) : = e

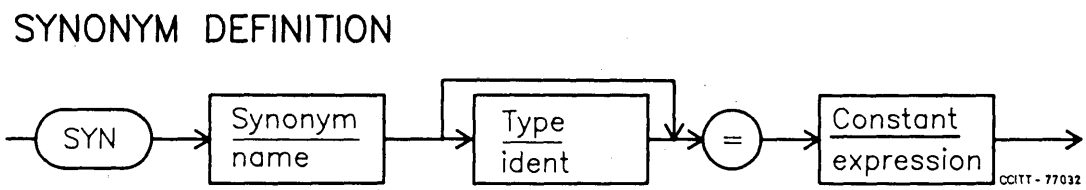
represents

vi: = a(i);
Insert!(vi,j,e);
a(i): = vi;

where vi is an *implicit variable* whose *type* is the same as a(i). The implied *Insert!* operations are *active operations*, which are represented in the *abstract syntax* in the normal way for *active operators* (see § 4.9.3).

4.12 *Synonym definition*

4.12.1 *Syntax*



4.12.2 *Semantics*

The *synonym* is equivalent to the *constant expression.* If the *type* of the *expression* cannot be determined by either the *constant* or the *context* of the *synonym definition* then a *type* must be specified otherwise the *value* and *type* of the *constant expression* and hence the value of the *synonym*), is determined by the *context* in which the *synonym definition* appears.

4.12.3 *Relationship to abstract syntax*

A *synonym* represents a *synonym definition* in the *abstract syntax.* If the *data type* is omitted, then it is implied by *context.*

4.13 *Data type generator*

4.13.1 *Syntax*

4.13.2 *Semantics*

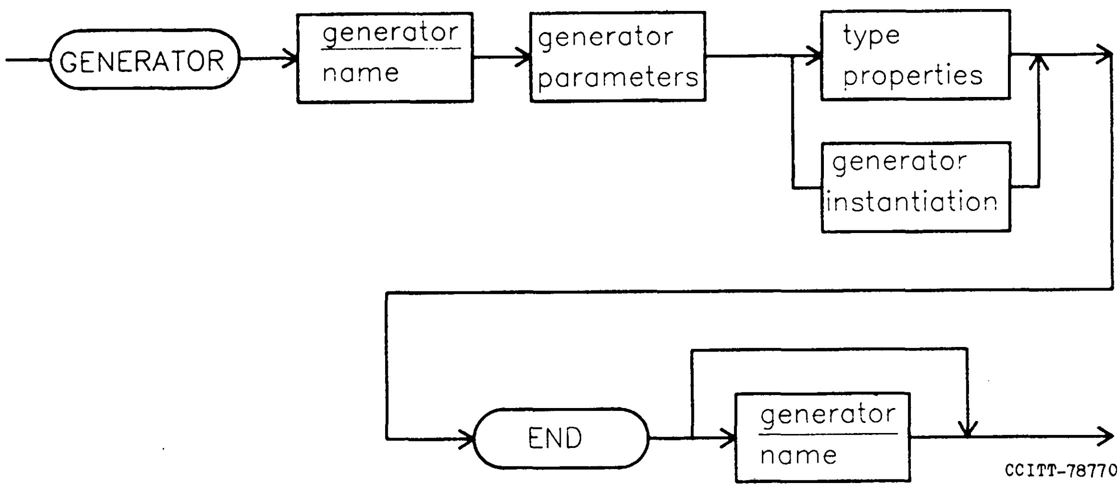
The *name* given in a *Data Type Generator* is a *Data Type Generator name.* The *properties* supplied in a *generator* form a partial specification of a *data type.* When a *generator* is used in a *Type Definition* or another *Data Type Generator*, the *parameters* to the *generator* are textually substituted in the *generator definition* and (together with any properties added in a *Type Definition*) then must form a complete *Type Definition* or another *Data Type Generator.* Where a *Data Type Generator* is defined in terms of a *generator instantiation,* the *generator parameters* may have the same *name* as those supplied to the *instantiation.* Such a *generator* is a *Partial Instantiation.* For example:

```
GENERATOR Stack (TYPE Component, CONSTANT Maxsize)
    /* details */
    END Stack;
GENERATOR IStack(CONSTANT Max) Stack(Integer, Max)
    END IStack;
    /* A stack of integers, maximum size not specified */
```

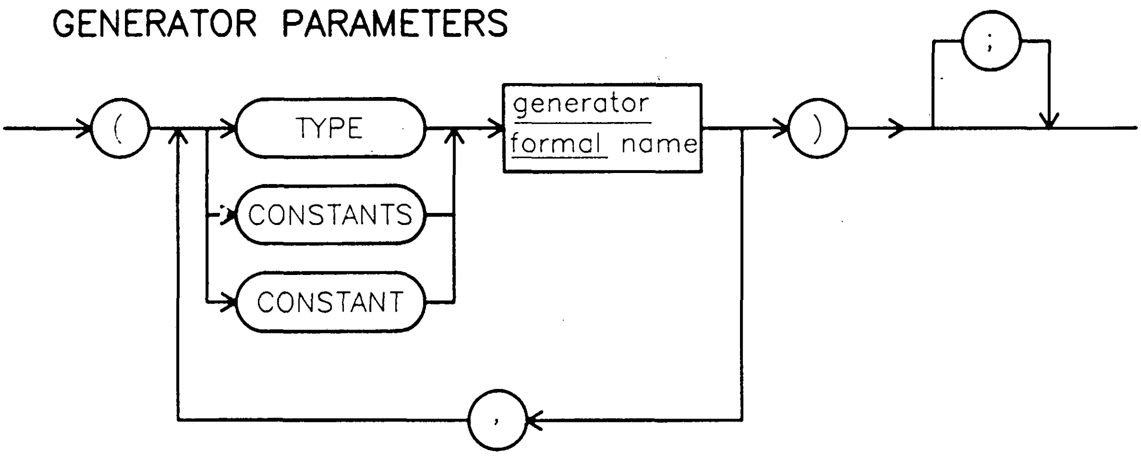
4.13.3 *Relationship to abstract syntax*

The Data Type Generator has no counterpart in the abstract syntax. Usage of a generator Instantiation in a Data Type Definition will denote the text formed by parametric substitution.

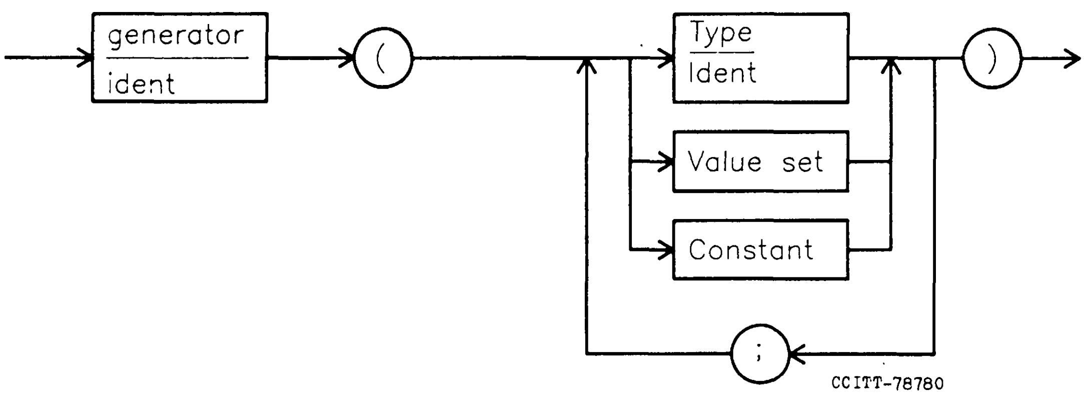
DATA TYPE GENERATOR



GENERATOR PARAMETERS



GENERATOR INSTANTIATION



5 Pre-defined data types

5.1 *Integer*

```
NEWTYPE Integer
        /* Literals according to integer literal syntax */

    OPERATORS
        "+": Integer, Integer -> Integer;
        "-": Integer, Integer -> Integer;
        "-": Integer, Integer -> Integer;
        "/" Integer, Integer -> Integer;
        Float: Integer -> Real;
        Fix: Real -> Integer;

    AXIOMS
        /* inherited from mathematical integers, adding: */
        Fix(Float(r)) = r;
        r - 1 < Float(Fix(r)) < = r;
        i/j = Fix(Float(i)/Float(j))

    END Integer;
```

5.2 *Real*

```
NEWTYPE Real
        /* Literals of the form specified in 'real literal' */

    OPERATORS
        "-": Real, Real -> Real;
        "+" Real, Real -> Real;
        "*": Real, Real -> Real;
        "/": Real, Real -> Real;
        "-": Real -> Real;

        /* Axioms are inherited from the mathematical reals; this work needs further study to be more
        formally specified here */

    END Real;
```

5.3 *Array*

```
GENERATOR Array(TypeIndex, TYPEItem);

    OPERATORS
        Insert!: Array', Index, Item ->;
        Extract!: Array, Index -> Item;

    AXIOMS
        Extract! (Declare!(V)', ]) = Error!;
        Extract!(Insert!(A, IPos, It), EPos =
          If Epos = Ipos Then It Else Extract(A, EPos) FI;

    End Array;
```

5.4 *Boolean*

```
NEWTYPE Boolean;

    LITERALS True, False;

    OPERATORS
        "NOT": Boolean -' Boolean;
        "AND": Boolean, Boolean -> Boolean;
        "OR": Boolean, Boolean -> Boolean;
        "=": Boolean, Boolean -> Boolean;
```

```
AXIOMS
    "NOT"(True) = False;
    "NOT"(False) = True;
    "AND" (A, B) = If A = False Then False Else B;
    "OR"(A, B) = If A Then True Else B;
    "=>"(A, B) = If A = True And B = False Then False Else True;

END Boolean;
```

5.5 *Character*

```
NEWTYPE Character

    LITERALS/* character strings of length 1, where the characters are those of the CCITT alphabet
     number 5 */;

    OPERATORS
        Ordering!;

    END Character;
```

5.6 *Natural*

```
SYNTYPE Natural Integer Constants >= 0 END Natural;
```

5.7 *Powerset*

```
GENERATOR Powerset(TYPE Item);

    LITERALS Empty;

    OPERATORS
        "IN": Item, Powerset -> Boolean;
        Incl: Item, Powerset' ->;
        Del: Item, Powerset' ->;
        Ordering!;

    AXIOMS;
        Declare(v)' = Empty;
        I IN Empty = False;
        I IN Incl(I2, S) = IF I = I2 THEN True ELSE I IN S FI;
        Del(I, Incl(I2, S)) = IF I = I2 THEN Del(I,S) ELSE Incl(I2, Del(I,S)) FI;
        For all S1, S2 in powerset (for all I in Item (S1 < S2 => (I IN S1 => I IN S2)));
        NOT(I in S) => Del(I, S) = S;

    END Powerset;
```

5.8 *PId*

```
NEWTYPE PId

    LITERALS Null;

    OPERATORS
        Create! : -> PId;

    AXIOMS;
        Declare!(v)' = Null;
        Create! /= Create!;
        /* A weak way of stating that all Creates yield unique values */
        Create! /= Null;

        /* PId values are returned by the interpretation of a create request node (they are denoted as
        being generated by the Create! function above). Every create request interpretation generates a
        unique PId value which is not Null. Every process instance implicitly declares three PId
        variables, named "Parent", "Self" and "Offspring".

        Interpretation of a Create request node generates a new and unique PId value, and assigns it to
        the Offspring for the creating task. The Self identifier of the created task is assigned this same
        value, while Parent of the created task is assigned the value of self of the creator. The length of
        Offspring in the created task is set to zero.

    */ END PId;
```

5.9 *String*

```
GENERATOR String (TYPE Item);
        /* Literals specified by Charstring literals, */
        /* only if generator is instantiated with Character */

    OPERATORS
        Declare!(v)'  : -> String;
        "//"          : String, String -> String;
        Length        : String -> Natural;
        First         : String -> Item;
        Last          : String -> Item;
        Extract!      : String, Natural -> Item;
        Insert        : String', Natural, String -> ;
        Insert!       : String', Natural, Item -> ;

    AXIOMS
        Length(Declare!(v)')=0;
        Length (S1 // S2) = Length(S1) + Length(S2);
        First(S1) = Extract!(S1, 1);
        Last(S1) = Extract!(S1, Length(S1));
        Extract!(Insert(S1,I,S2),j) =
          IF j< 1 THEN Error! ELSE
            IF j<= I THEN Extract!(S1,j) ELSE
              IF j<= Length(S2) + I THEN Extract!(S2,j-I)) ELSE
                IF j<= Length (S1) + Length(S2) THEN
                        Extract!(S1,J-Length(S2)) ELSE Error!
                FI
              FI
            FI
          FI;
        S1 // S2 = Insert(S1, Length(S1), S2);
        Extract!(Insert!(S1, I, It)', J) =
          IF I = J Then It Else Extract!(S1, J) FI;

    END String;
```

5.10 *Time*

```
NEWTYPE Time Inherits Real

ADDING
    Operators
        Now: -> Time /* the 'real' time */;

END Time;
```

5.11 *Duration*

```
NEWTYPE Duration Inherits Real ("+", "-", "*", "/")

ADDING
    Operators
        "+": Time, Duration -> Time;
        "+": Duration, Time -> Time;
        "-": Time, Duration -> Time;

CONSTANTS > = 0.0

END Duration;
```

5.12 *Timer*

```
NEWTYPE Timer

    OPERATORS
        Set: Time, Timer' * - > ; .
        Reset: Timer'  - > ;
        Active: Timer  - > Boolean;

    AXIOMS
        Active ( Set(Tm, Tmr)') = Tm > Now ();
        Active( Reset(Tmr) ) = False;
        /* Note that an active timer sends a signal to the process when it becomes inactive. This signal
        is named with the same name as the timer variable in the Set call */

    END Timer;
```

5.13 *Charstring*

```
NEWTYPE Charstring String (Character)
    Adding LITERALS /* character string literals */
    END Charstring;
```

Printed in Switzerland — ISBN 92-61-02231-6