INTERNATIONAL TELECOMMUNICATION UNION

CCITT

THE INTERNATIONAL
TELEGRAPH AND TELEPHONE
CONSULTATIVE COMMITTEE

RED BOOK

VOLUME VI – FASCICLE VI.11

FUNCTIONAL SPECIFICATION AND DESCRIPTION LANGUAGE (SDL)

ANNEXES TO RECOMMENDATIONS Z.100-Z.104

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

Geneva 1985

A NOTE FROM ITU LIBRARY & ARCHIVES

Due to technical restrictions, the template of SDL symbols has not been included in the scanned version of this document.

****

En raison de contraintes techniques, le gabarit de symboles du LDS n'a pas été inclus dans la version scannée de ce document.

****

Debido a restricciones de técnicas, la plantilla de símbolos del LED no se ha incluido en la versión escaneada de este documento.

INTERNATIONAL TELECOMMUNICATION UNION

CCITT

THE INTERNATIONAL
TELEGRAPH AND TELEPHONE
CONSULTATIVE COMMITTEE

RED BOOK

VOLUME VI – FASCICLE VI.11

FUNCTIONAL SPECIFICATION AND DESCRIPTION LANGUAGE (SDL)

ANNEXES TO RECOMMENDATIONS Z.100-Z.104

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

Geneva 1985

ISBN 92-61-02241-3

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

CONTENTS OF FASCICLE VI.11 OF THE RED BOOK

Annexes to Recommendations Z.100 to Z.104

Functional specification and description language (SDL)

Rec. No. Page

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.

Annexes to Recommendations Z.100 to Z.104

FUNCTIONAL SPECIFICATION AND DESCRIPTION LANGUAGE (SDL)

PAGE INTENTIONALLY LEFT BLANK

PAGE LAISSEE EN BLANC INTENTIONNELLEMENT

ANNEX A

(to Recommendations Z.100 to Z.104)

SDL glossary

Any term appearing in the SDL Recommendations Z.100 to Z.104 which is italicized *and* appears in this glossary is used strictly with the sense defined in this glossary.

If an italicized phrase, for example *procedure identifier*, is not in the glossary, then it may be the concatenation of two terms, in this case the term *procedure* followed by the term *identifier.* When a word is italicized but cannot be located in the glossary if may be a derivative of a glossary term. For example *exported* being the past tense of *export.*

Except where a term is a synonym for another term, after the definition of the term there is a main reference to the use of the term in the Z.100 series Recommendations. These references are shown in square brackets [ ] after definitions. For example [Recommendation Z.100 § 3.2] indicates that the main reference is in Recommendation Z.100 § 3.2. Since there are often no defining occurrences of terms, these references are only a guideline.

Annex C contains the graphical representation of symbols and diagrams, the keywords for the textual phrase representation and the use of these keywords, therefore this information is not repeated in the glossary.

abstract data type

An *abstract data type* is a *type* defined by algebraic relationship between values of the *type* and the application of operators on these *values.* [Recommendation Z.100 § 2.3, Recommendation Z.104, § 1.]

abstract syntax

The *abstract syntax* of SDL is given in Recommendations Z.101, Z.102, Z.103 and Z.104 and summarized in Appendix B to describe the conceptual structure of an *SDL* definition as compared with the *concrete syntaxes* which exist for each form of *SDL,* that is *SDL/GR, SDL/PR* and *SDL/PE.* [Recommendation Z.100, § 3.1.]

access

The predefined *operator* implicit with all data types which is applied wherever a variable yields the value associated with it. [Recommendation Z.104, § 4.5.]

active operator

An *operator* which requires one or more *variables* as parameters since it may change the *values* associated with these *variables.* [Recommendation Z.104, § 4.5.]

actual parameter

An *actual parameter* is a *value* handed to either a *process* or a *procedure* for the corresponding *formal parameter* when the *process* (or *procedure*) is *created* (or *called* respectively). [Recommendation Z.101, § 2.3, Recommendation Z.103, § 2.1.]

actual parameter list

An *actual parameter list* is the list of *values* associated with a *create request* or a *procedure call.* [Recommendation Z.101, § 2.3; Recommendation Z.103, § 2.1.]

additional-save-set

An *additional-save-set* is the set of additional signals saved in a *procedure.* [Recommendation Z.103, § 2.1.]

allocation symbol

The *symbol* in a *process tree diagram* of *SDL/GR* attached to a *process symbol.* The *process* or *sub-process* associated with the *process symbol* is allocated to the *block* whose *name* is contained in the *allocation symbol.* [Recommendation Z.102, § 2.3.]

annotation

An *annotation* in *SDL/GR* is a *comment.* An *annotation* in *SDL/PR* is a *comment* or a *note. Annotations* do not change the meaning of SDL. [Recommendation Z.101, § 3.3, § 4.3.]

arc

An *arc* is a directed connection between the *nodes* of a *process graph.* [Recommendation Z.101, §§ 2.2.]

array

The *predefined generator* used to introduce the concept of arrays. [Recommendation Z.104, § 5.3.]

assignment statement

An *assignment statement* is an *action* which associates a *value* with a *variable.* [Recommendation Z.101, § 2.2, Recommendation Z.104, § 4.11.]

axiom

A *Boolean expression* which holds true for all possible values of the variables used in the *axiom.* [Recommendation Z.104, § 4.6.]

B'

The *SDL/PR* keyword introducing a binary denotation of a numerical *value.* [Recommendation Z.104, § 4.8.]

basic SDL

The minimum subset of *SDL* defined by Recommendation Z.101.

behaviour

The *behaviour* or *functional behaviour* of a system in SDL is described as discrete responses to discrete stimuli. [Recommendation Z.101, § 1.]

block

A *block* is a synonym for a *block instance.*

block definition

A *block definition* defines the structural and connection properties of a named *block* type. [Recommendation Z.101, § 2.2.]

block interaction diagram

The *block interaction diagram* of *SDL/GR* represents the structure of *blocks, channels, signal lists, data lists, processes* and *signal* flow between *processes.* A *block interaction diagram* may also show the partitioning of a *system into blocks, sub-blocks, channels* and *sub-channels.* [Recommendation Z.101, § 3.1, Recommendation Z.102, § 3.2.]

block substructure

A *block substructure* for a *block* is the *partitioning* of the *block* into *sub-blocks* and new *channels* at lower *levels of abstraction.* [Recommendation Z.102, § 2.2.]

block substructure definition

A *block substructure definition* is optionally part of a *block definition* and defines a *block substructure.* [Recommendation Z.102, § 2.2.]

block symbol

The *block symbol* represents the concept of a *functional block* in a *block interaction diagram* of *SDL/GR.* [Recommendation Z.101, § 3.1.]

block tree diagram

A *block tree diagram* is the *SDL/GR* representation of the partitioning of a *system* into *blocks* at *lower levels of abstraction* by means of an inverted tree diagram. [Recommendation Z.102, § 3.1.]

Boolean

The logical *data type* having the values TRUE and FALSE, and the normal set of logical *operators.* [Recommendation Z.104, § 5.4.]

call node

The term *call node* is a synonym for *procedure call node.*

call symbol

The term *call symbol* is a synonym for *procedure call symbol.*

channel

A *channel* is the class of entity which conveys *signals* from one *block* to another *block. Channels* also convey *signals* to and from the *environment. Channels* are unidirectional, that is they only convey signals in one direction. [Recommendation Z.101, § 2.1.]

channel definition

A *Channel definition* defines the properties of a named *channel.* These properties are the origin *block,* the destination *block,* the set of *signals* which the *channel* may convey, and optionally a *channel substructure definition.* The destination *block* may be the *environment of. the system.* [Recommendation Z.101, § 2.2, Recommendation Z.102, § 2.2.]

channel substructure

A *channel substructure* is a *partitioning* of a *channel* into a set of *channels* and *blocks* at a lower *level of abstraction.* [Recommendation Z.102, § 2.2.]

channel substructure definition

A *channel substructure definition* is an optional part of a *channel definition* which defines the *channel substructure.* [Recommendation Z.102, § 2.2.]

channel substructure diagram

A *channel substructure diagram* is the *SDL/GR* representation of a *channel substructure.* [Recommendation Z.102, § 3.4.]

Channel symbol

The *symbol* on a *block interaction diagram* of *SDL/GR* representing a *channel* or *sub-channel.* The arrowhead of the symbol points to the destination *block* and the other end of the symbol identifies the origin *block.* The set of *signals* conveyed by the *channel* may be represented by an associated *signal list symbol.* [Recommendation Z.101, § 3.1.]

Character

The *predefined data type* for which the *literals* are *charstring literals* of length one, and the operators are equal, not equal, and concatenation to form a *charstring value.* [Recommendation Z.104, § 5.5.]

charging in progress PE

A *pictorial element* indicating that charging is currently *taking place.* [Recommendation Z.103, § 6.1.]

Charstring

The *predefined data type* for which the *literals* are strings of CCITT No. 5 alphabet characters, and the *operators* are those of the *string predefined generator* instantiated for *characters*. [Recommendation Z.104, § 5.13.]

combined signalling sender and receiver PE

A *pictorial element* corresponding to a combined signalling sender and signalling receiver. [Recommendation Z.103, § 6.1.]

comment

Information which is in addition to or clarifies the *SDL* graph. In *SDL/GR comments* may be attached by a single square bracket connected by a dashed line to a *flow line* of a *process diagram* or any symbol. In *SDL/PR comments* are introduced by the keyword *COMMENT*. [Recommendation Z.101, § 3.3.]

composite operations

A shorthand notation representing a standard combination of primitive *SDL* concepts. A *composite operation* can be mapped to concepts in the *SDL abstract syntax* in a systematic way. [Recommendation Z.103, § 3.]

concrete syntactical form

SDL/GR, *SDL/PR* and *SDL/PE* are the *concrete syntactical forms* of *SDL* which can be mapped one to another by considering the conceptual underlying mathematical *SDL* graph. [Recommendation Z.100, § 3.1.]

concrete syntax

The *concrete syntax* for the various representations of *SDL* is the actual symbols used to represent *SDL* in the *SDL/GR*, the *SDL/PR* or the *SDL/PE* forms. [Recommendation Z.100, § 3.1.]

conditional expression

An *expression* with a *Boolean expression* after IF which controls whether the consequence *expression* after THEN or the alternative *expression* after ELSE is interpreted. [Recommendation Z.104, § 4.10.]

connected switching path PE

A *pictorial element* indicating connectivity between terminal equipment and/or signalling devices. [Recommendation Z.103, § 6.1.]

connector

A *connector* is a *symbol* used in *SDL/GR*. A *connector* (a circle) is either an *in-connector* or an *out-connector*. A *flow line* may be broken by a pair of *associated connectors*, with the flow assumed to be from the *out-connector* to its associated *in-connector*. [Recommendation Z.101, § 3.1.]

constant value

Either a *literal* or a *synonym.* [Recommendation Z.104, § 2.2.]

constant expression

A *constant value* or an *operator* with only *constant expressions* as parameters. [Recommendation Z.104, § 2.2.]

continuous signal

A *continuous signal* is a *composite operation* containing a *condition.* When the *condition* becomes true, the *state* which has the *continuous signal* attached is left. [Recommendation Z.103, § 3.3.]

convergence

In a *process diagram* of *SDL/GR* when two or more symbols are followed by a single symbol the *flow lines* converge on the symbol. 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. [Recommendation Z.101, § 3.3.]

create request action

An *action* which causes the creation and starting of a new *process instance* from a specified *process definition.* [Recommendation Z.101, § 2.3.]

create request node

A *node* in a *transition* in a *process graph* or a *procedure graph* where a *create request action* takes place. [Recommendation Z.101, § 2.2.]

create symbol

The *symbol* on a *block interaction diagram* of *SDL/GR* connecting the *process symbol* of the *creating process* with the *process symbol* of the *created process.* The arrowhead identifies the *offspring process* and other end of the create request symbol identifies the *parent process.* [Recommendation Z.101, § 3.1.]

create request symbol

The *symbol* on a *process diagram* of *SDL/GR* representing a *create request.* [Recommendation Z.101, § 3.3.]

D′

The *SDL/PR* keyword introducing a decimal denotation of a numerical *value.* Since this is the default denotation *D′* is optional. [Recommendation Z.104, § 4.8.]

data

The term *data* is a synonym for *data item(s).*

data item

A *data item* is either a *variable* or *value.*

data type

A *data type* of *data items* determine the *range*, the meaning of *values* in that *range* and the valid set of *operators* which may be used with items of that *data type.* (See also *predefined data type.*) [Recommendation Z.104, § 1.]

data type definition

Defines the *name*, *data values* and *operators* for a *data type.* [Recommendation Z.104, § 2.2.]

decision

A *decision* is an *action* at a *decision node* within a *transition* which asks a question to which the answer can be obtained at that instant and chooses one of the several outgoing *arcs* from the *node* to continue the *transition.* [Recommendation Z.101, § 2.3.]

decision node

A *decision node* is a *node* in a *transition* in a *process graph* or a *procedure graph* where a *decision* takes place. [Recommendation Z.101, § 2.2.]

decision name

A *decision name* is the *name* associated with a *decision.* The name of a *decision* must be a question which is well understood by the interpreter. The *names* of the *decision arcs* leading from a *decision* must constitute all possible results from the question. [Recommendation Z.101, § 2.3.]

decision symbol

A *symbol* representing the SDL concept of a decision in a *process diagram* of *SDL/GR.* [Recommendation Z.101, § 3.2.]

declare!

The predefined *operator* implicit with all data types associated with the instantiation of *variable instances.* [Recommendation Z.104, § 4.5.]

definition

The term *definition* is a synonym for *type definition.*

description

The implementation of the requirements of a system is described in a description of the system. Descriptions consist of *general parameters* of the system as implemented and the *functional description* (FD) of its actual behaviour. [Recommendation Z.100, § 1.1.]

divergence

In *SDL/GR* where a *symbol* is followed by two or more *symbols*, a *flow line* leading to that symbol may *diverge* into two or more *flow lines*. [Recommendation Z.101, § 3.3.]

Duration

The *predefined data type* representing the interval between two time instants. [Recommendation Z.104, § 5.11.]

enabling condition

An *enabling condition* is a *composite operation* for a method of conditionally accepting for *input* or alternatively *saving* a *signal* depending on a *condition*. [Recommendation Z.103, § 3.2.]

enabling condition symbol

The *symbol* in a *process diagram* or *procedure diagram* of *SDL/GR* representing an *enabling condition* (when it follows an *input symbol*) or a *continuous signal* (when it follows a *state symbol*). [Recommendation Z.103, § 3.2.]

environment

The term *environment* is a synonym for the *environment of a system*. [Recommendation Z.101, § 2.1.]

environment symbol

The *symbol* in a *process diagram* of *SDL/GR* representing the *environment of a system*. [Recommendation Z.101, §§ 3.1.1, 3.1.2.]

environment of a system

The *environment of a system* is a part of the system whose behaviour is not shown in SDL but interacts with the rest of the system by sending *signal instances* to the *system*. [Recommendation Z.101, § 2.1.]

equivalent behaviour

The term *equivalent behaviour* is a synonym for *equivalent functional behaviour*.

equivalent functional behaviour

Two *systems* (or *blocks* or *processes*) have *equivalent functional behaviour* if they have the same response to a given sequence of *signal instances* as seen from outside the *systems* (or *blocks* or *processes* respectively). [Recommendation Z.100, § 1.1.]

EXPORT

The syntactic construct EXPORT (variable name) is used both in *SDL/GR* and *SDL/PR* to represent the *export* of the *value* of a *variable*. [Recommendation Z.103, § 3.1.]

export

The term *export* is a synonym for *export operation.*

EXPORTED

The *SDL/PR* keyword indicating in a *variable definition* that the *variable* is *exported.* [Recommendation Z.103, § 3.1.]

exporter

An *exporter* of a *variable* is the *process instance* which owns the *variable* whose *value* is *exported.* [Recommendation Z.103, § 3.1.]

export operation

An *export operation* is the *composite operation* which allows one process to *export* the *values* of *data items* so that another process can access the *values.* [Recommendation Z.103, § 3.1.]

expression

An *expression* is either a *value name*, a *synonym name*, a *variable name*, a *conditional expression* or an *operator* applied to one or more *expressions.* [Recommendation Z.104, § 4.7.]

extract!

The *operator* which is implied outside *axioms* when a *variable* is *immediately* followed by bracketed *expressions* (except if this is followed by := in which case *insert!* is implied). [Recommendation Z.104, § 4.5.]

flow line

A *flow line* connects every *symbol* to the *symbol*(s) it follows on a *process diagram* of *SDL/GR.* [Recommendation Z.101, § 3.3.]

formal parameter

A *formal parameter* is a *variable name* contained in a *process definition* or *procedure definition* for which *actual parameters* are assigned to the *variables* when the *process* is created or *procedure* is called respectively. [Recommendation Z.101, § 2.2, Recommendation Z.103, § 2.1.]

formal parameter list

A list of *formal parameters* in a *process definition* or *procedure definition*. The *actual parameters* are matched by position in their respective lists. [Recommendation Z.101, § 2.2, Recommendation Z.103, § 2.1.]

frame (1)

A frame in a block interaction diagram of SDL/GR represents the *block* whose *partitioning* is defined by the *block interaction diagram.* [Recommendation Z.102, § 3.2.]

frame (2)

A frame in a *channel substructure diagram* of *SDL/GR* represents the *channel* whose *partitioning* is defined by the *channel substructure diagram.* [Recommendation Z.102, § 3.4.]

functional behaviour

(See behaviour.)

functional block

A *functional block* is an object of manageable size and relevant internal relationship, which has a name, a set of *channels* connecting it with other *blocks* (or the *environment*), and either internal *processes* or *block definition.* [Recommendation Z.100, § 2.1, Z.102, § 2.1.]

general parameters

The *general parameters* in both a *specification* and a *description* of a system relate to such matters as temperature limits, construction, exchange capacity, grade of service, etc. [Recommendation Z.100, § 1.1.]

generator

The term *generator* is a synonym for *data type generator.*

graphic syntax

(See *SDL/GR.*)

H'

The *SDL/PR* keyword introducing a hexadecimal denotation of a numerical *value.* [Recommendation Z.104, § 4.8.]

identifier

An *identifier* is a unique *name* for a *type* or an *instance* of a *type*, and is formed from a *qualifying part* and a *name part.* [Recommendation Z.100, § 2.1.]

implicit transition

In a *SDL/GR* process diagram, if neither explicit *input symbols* nor explicit *save symbols* are shown attached to a *state symbol* for one of the *valid input signals*, then there is an *implicit transition* which is an *input node* connected directly back to the same *state.* Thus these *signals* are discarded. [Recommendation Z.101, § 3.3.]

IMPORT

The syntactic construct IMPORT (variable name, process instance) is used both in *SDL/GR* and *SDL/PR* to represent the *import* of the *value* from a *process instance.* [Recommendation Z.103, § 3.1.]

import

The term *import* is a synonym for *import operation.*

IMPORTED

The *SDL/GR* keyword indicating in a *variable definition* that the *variable* is used to contain *values* which are *imported.* [Recommendation Z.103, § 3.1.]

importer

An *importer* of an *imported value* is the *process instance* which *imports* the *value.* [Recommendation Z.103, § 3.1.]

Import operation

An *import operation* is the *composite operation* which allows one process to *import* and access the *values* of *variables* owned by another *process.* [Recommendation Z.103, § 3.1.]

imported value

The *value* seen by a *process* for a *data item* which is *imported.* [Recommendation Z.103, § 3.1.]

IN

The *formal parameter* attribute denoting the case when a *value* is passed to a *procedure.* [Recommendation Z.103, § 2.3.]

IN/OUT

The *formal parameter* attribute denoting the case when a *formal parameter name* is used as a *synonym* for the *variable.* [Recommendation Z.103, § 2.3.]

in-connector

A *flow line* may be broken by a pair of *associated connectors,* with the flow assumed to be from the *out-connector* to its associated *in-connector.* [Recommendation Z.101, § 3.3.]

incoming channel

An *incoming channel* is a new *channel* formed when a *channel* is *partitioned.* An *incoming channel* conveys to the new *blocks* formed by *channel partitioning* all the *signals* which the *partitioned channel* conveys. [Recommendation Z.102, § 2.1.]

infix operator

One of the predefined dyadic *operators of SDL/PR* (=> OR XOR AND IN/= = > < <= >= + – // # / MOD REM) which are placed between the two parameters rather than before bracketed parameters, or one of the monadic prefix *operators* (+ – NOT). [Recommendation Z.104, § 4.5.]

inlet

An *inlet* of a *macro* is the point where a line enters the *macro.* [Recommendation Z.103, § 4.1.]

input

The term *input* on its own is a synonym for *input action.*

input action

An *input action* is an *action* which receives and consumes an *input signal* corresponding to a *signal name* and allows the *process instance* interpreting the *input action* to access the information contained in the *input signal.* [Recommendation Z.101, § 2.3.]

input node

An *input node* is a *node* in a *process graph* or a *procedure graph* where an *input action* takes place and has the same *name* as the *signal* which the *input action* consumes. [Recommendation Z.101, § 2.2.]

input port

The *input port* of a *process* receives and retains *signals* in the order of arrival until the *signals* are consumed by an *input action.* [Recommendation Z.101, § 2.3.]

input signal

An *input signal* of a *process* is one of the set of named *signals* that the *process* can receive at any of its *state nodes.* The set of valid *input signals* a *process* can receive is the set of all *signal names* which appear in any *input node* of the process. [Recommendation Z.101, § 2.3.]

input symbol

The symbols in a *process diagram* in *SDL/GR* representing the SDL concept of an *input.* [Recommendation Z.101, § 3.3.]

insert!

The *operator* which is implied outside *axioms* when a *variable* is immediately followed by bracketed *expressions* and then :=. [Recommendation Z.104, § 4.5.]

instance

An *instance* of a *type* is a value which has all the properties of the *type* and can be distinguished from other *instances* of the same type by an *identifier.* [Recommendation Z.100, § 2.1.]

instantiation

Instantiation is the creation of *instance* of an entity from a *type.* [Recommendation Z.100, § 2.1.]

Integer

The *integer type* is defined by the set of values −(infinity), . . ., −2, −1, 0, +1, +2, . . ., +(infinity) and their normal mathematical operations, i.e. +, −, multiply, divide, etc. [Recommendation Z.104, § 5.1.]

label

A *label* is an optional *name* attached to either an *inlet* or *outlet* from a *macro.* [Recommendation Z.103, § 4.2.]

level

The term *level* is a synonym for *level of abstraction.*

level of abstraction

A *level of abstraction* is one of the levels of a *block tree diagram.* A description of a *system* is one *block* at the highest *level of abstraction* and is shown as a single *block* at the top of a *block tree* diagram. [Recommendation Z.102, § 3.1.]

literal

A *literal* denotes a *value identity.* The *literals* 12, B'1100, 0'14 and H'C all denote the same *Integer value identity.* [Recommendation Z.104, § 4.4.]

macro

A *macro* is a named collection of syntactic items defined by the SDL user, which replace the use of the *macro name* BEFORE the meaning of the SDL representation is considered. [Recommendation Z.103, § 4.]

macro definition

A *macro definition* in *SDL/GR* is a named part of any *SDL/GR* diagram with *inlets* and *outlets* represented by lines indicated as flowing to and from (respectively) the part diagram. These lines may be *labelled.* [Recommendation Z.103, § 4.2.]

macro symbol

The *symbol* used in *SDL/GR* to denote a reference to a *macro definition* by name. [Recommendation Z.103, § 4.2.]

name

The term *name* is a synonym for the *name part* of an *identifier.*

name part

The *name part* of an *identifier* is a meaningful phrase in natural language which can be used in combination with a *qualifying part* of the *identifier* to identify a *type* or *instance* of an entity. [Recommendation Z.100, § 2.1.]

Natural

The *natural number type* is a *range type* of the *integer type* with the *values* 0, 1, 2, ... up to infinity. [Recommendation Z.104, § 5.6.]

newtype

A *newtype* introduces sets of *literals* and *operators* which are distinct from any other *literals* and *operators* (even though they may have the same names so that the different identities have to be resolved by qualification). The properties of a *newtype* are defined by the use of the *literals* and *operators* in *axioms.* [Recommendation Z.104, § 4.3.]

node

A *node* is a named place on a *process graph* which is joined to other *nodes* by *arcs.* The classes of *nodes* in SDL are *state nodes, input nodes, task nodes, output nodes, decision nodes, start nodes, stop nodes, procedure start nodes, procedure call nodes, procedure return nodes,* and *create request nodes.* [Recommendation Z.101, §§ 2.2.]

note

An *annotation* in *SDL/PR* which is relevant only to the *SDL/PR* representation. A *note* is a text string enclosed by /* and */. [Recommendation Z.101, § 4.3.]

O′

The *SDL/PR* keyword introducing an octal denotation of a numerical *value.* [Recommendation Z.104, § 4.8.]

OFFSPRING

The *OFFSPRING* of a creating *process* is a *data item* which has the same *value* as the *SELF data item* of the *process* most recently created by this creating *process.* If a *process* has not created any *processes* then its *OFFSPRING data item* is *undefined.* [Recommendation Z.101, § 2.3.]

operator

An *operator* when applied to one or more *values, delivers a value* determined by the use of the *operator* in *axioms.* The symbols + − * / are arithmetic *operators.* [Recommendation Z.104, § 4.5.]

operator typing

Defines the *data types* of the *data items* to which the operator is applied and the *data type* of the resultant *value* (if any). [Recommendation Z.104, § 4.5.]

option

An *option* is a *concrete syntax* construct in a *process definition* allowing different *behaviours* to be chosen BEFORE the *process graph* is interpreted. [Recommendation Z.103, § 5.1.]

option expression

An *expression* contained in an *option* which is evaluated to determine which *behaviour* is to be chosen. [Recommendation Z.103, § 5.2.]

option symbol

The *symbol* on a *process diagram* or *procedure diagram* of *SDL/GR* representing an *option.* [Recommendation Z.103, § 5.2.]

out-connector

A *flow line* may be broken by a pair of *associated connectors,* with the flow assumed to be from the *out-connector* to its associated *in-connector.* [Recommendation Z.101, § 3.3.]

outgoing channel

An *outgoing channel* is a new *channel* formed when a *channel* is *partitioned.* An *outgoing channel* conveys from the new *blocks* formed by *channel partitioning,* all the *signals* which the *partitioned channel* conveys. [Recommendation Z.102, § 2.1.]

outlet

An *outlet* of a *macro* is the point where a line leaves the *macro.* [Recommendation Z.103, § 4.2.]

output

The term *output* on its own is a synonym for *output action.*

output action

An *output* is an *action* within a *transition* which generates a *signal* which in turn acts as an *input signal* elsewhere. [Recommendation Z.101, § 2.3.]

output node

A *node* on a *process graph* where an *output action* takes place and has the same name as the *signal* it generates. [Recommendation Z.101, § 2.2.]

output symbol

The symbol in a *process diagram* of *SDL/GR* representing the SDL concept of an *output.* [Recommendation Z.101, § 3.3.]

PARENT

The *PARENT data item* of a *process* is the *SELF data item* of its parent *process,* that is the *process* which interpreted the *create request action* which started the *process.* [Recommendation Z.101, § 2.3.]

partitioning

Partitioning is the elaboration of behaviour of complex and/or large systems into sub-systems to provide a logical partitioning of the bahaviour of the system and different abstract viewpoints of the same system. [Recommendation Z.102, § 2.1.]

passive operator

An *operator* which requires only *values* as parameters and produces a *value* as a result. A *passive operator* cannot *change* the values associated with *variables.* [Recommendation Z.104, § 4.5.]

pictorial element (PE)

One of a number of standardized graphical forms used with *state pictures* in *SDL/PE* to represent switching system concepts. [Recommendation Z.103, § 6.]

PId

The *predefined data type* used to identify *process instances.* [Recommendation Z.104, § 5.8.]

predefined data type

For simplicity of description the term *predefined data type* is applied to both predefined *names* for *data types* and predefined *names* for *data type generators. Boolean, Character, Charstring, Duration, Integer, Natural, PId, Real, Time* and *Timer* are *data type names* which are predefined. *Array, Powerset,* and *String* are *data type generator* names which are predefined. [Recommendation Z.104, § 5.]

Powerset

The *predefined data type generator* generating *data types* with *values* which are mathematical ordered sets of *values.* Each *powerset value* is a set of *values* of the *data type* used to parameterize *powerset generator.* [Recommendation Z.104, § 5.7.]

procedure

A section of a *process graph* which can be regarded in isolation. A *procedure* is defined in one place but may be referred to several times, even in different *processes.* The *signals* and *variables* affected by interpretation of a *procedure* are controlled by parameter passing. [Recommendation Z.103, § 2.]

procedure call

A *procedure call* is the means of invoking a *named procedure* for interpretation and passing parameters to the *procedure.* [Recommendation Z.103, § 2.1.]

procedure call node

A *procedure call node* is a *node* in a *process graph* or a *procedure graph* where a *procedure call* takes place. [Recommendation Z.103, § 2.1.]

procedure call symbol

The *symbol* of *SDL/GR* representing a *procedure call.* [Recommendation Z.103, § 2.2.]

procedure definition

A *procedure definition* defines a section of a *process graph.* The definition associates the *procedure graph* with a *procedure name*, a *formal parameter list*, an *additional-save-set* and optionally further *procedure definitions* and *data definitions.* [Recommendation Z.103, § 2.1.]

procedure diagram

A *procedure diagram* is the *SDL/GR* representation of a *procedure graph.* [Recommendation Z.103, § 2.2.]

procedure graph

A *procedure graph* is a graph connected by directed *arcs* to describe the *behaviour* of a *procedure* and can form a section of a *process graph.* [Recommendation Z.103, § 2.1.]

procedure return

(See *return.*)

procedure start node

The *procedure start node* is a *node* and is a *procedure graph* where interpretation of the *procedure* is commenced by a call of the *procedure.* [Recommendation Z.103, § 2.1.]

procedure start symbol

The symbol in a *procedure diagram* of *SDL/GR* representing a *procedure start node.* [Recommendation Z.103, § 2.2.]

process

A *process* performs a function which requires various items of information to perform sub-functions. The sub-functions performed depend on the order in time that the information becomes available to the *process.* In the context of SDL a process is a class of entity, instances of which are either in a *state* awaiting an *input* or in a *transition.* The term *process* on its own is a synonym for *process instance.* [Recommendation Z.101, § 2.1.]

process definition

The behaviour of a *type* of the class *process* is described in a *process definition*, in terms of a closed directed graph of *inputs*, *saves*, *states*, *transitions*, *decisions*, *tasks* and *outputs.* [Recommendation Z.101, § 2.1.]

process diagram

A *process diagram* is the *SDL/GR* representation of a *process graph.* [Recommendation Z.101, § 3.2.]

process graph

A *process graph* is a graph connected by directed *arcs* to describe the bahaviour of a *process.* [Recommendation Z.101, § 2.2.]

process instance

A *process instance* is a dynamically created *instance* of a *process.* [Recommendation Z.101, § 2.3.]

process substructure

The *process substructure* of a *process* is the *partitioning* of the *process* into *sub-processes* and the allocation of these *processes* into *sub-blocks.* [Recommendation Z.102, § 2.2.]

process substructure definition

A *process substructure definition* is an optional part of a *process definition* which defines the *process substructure* of a *process.* [Recommendation Z.102, § 2.2.]

process symbol

The *symbol* on a *block interaction diagram* or a *process tree diagram* of *SDL/GR* representing zero or more *process instances.* The *process symbol* contains the *process name*, which identifies the *process definition*, and a *formal parameters list.* [Recommendation Z.101, § 3.1, Recommendation Z.102, § 3.3.]

process tree diagram

A *process tree diagram* of *SDL/GR* represents the *partitioning* of a *process* in a *block* into *sub-processes.* The allocation of the *sub-processes* to *sub-blocks* is shown by *allocation symbols* on the *process tree diagram.* The diagram is in the form of an inverted tree. [Recommendation Z.102, § 3.3.]

qualifiers

The term *qualifiers* is a synonym for the *qualifying part* of an *identifier.*

qualifying part

The *qualifying part* of an *identifier* is the information which has to be added to the *name part* of the *identifier* to form a unique *name.* The *qualifying part* of an *identifier* may be derived from the context of the use of the *name part.* [Recommendation Z.100, § 2.1.]

Real

The *real type* is defined by the set of ALL values between −(infinity) and +(infinity) and the normal mathematical operations, i.e. +, −, multiply, raise to power, etc. [Recommendation Z.104, § 5.2.]

reserved switching path PE

A *pictorial element* representing a reserved connection between terminal equipment and/or signalling devices. [Recommendation Z.103, § 6.1.]

RESET

The *operator* defined for the *timer data type* which allows timers to be cleared. [Recommendation Z.104, § 5.12.]

retained signal

When a *signal* arrives at a *process*, it is considered to be *received* and *retained* for that *process* (it is outside the *process* hence it is not yet *consumed* by it). [Recommendation Z.101, § 2.1.]

return

The *return* of a *procedure* is the destruction of the *variables* and *synonyms* created at the *procedure start* followed by the completion of the interpretation of the *procedure start.* [Recommendation Z.103, § 2.1.]

return node

A *return node* is the *node* in a *procedure graph* where the *return* from the *procedure* takes place. [Recommendation Z.103, § 2.1.]

return symbol

The symbol in a *procedure diagram* representing a *return* from the *procedure.* [Recommendation Z.103, § 2.2.]

reveal attribute

A *variable* owned by a *process* may have a *reveal attribute,* in which case another *process* in the same *block* can *view* the *value* associated with the *variable.* [Recommendation Z.101, § 2.3.]

save

A *save* is the postponement of *recognition of a signal* when a *process* is in a particular *state* in which *input* of that *signal* is not allowed. [Recommendation Z.101, § 2.2.]

save-signal-set

The *save-signal-set* of a *state* of a *process* is the set of *saved signal names* for that *state.* [Recommendation Z.100, § 2.2.]

save symbol

A symbol in a *process diagram* of *SDL/GR* representing the SDL concept of a save. [Recommendation Z.101, § 3.3.]

SDL/GR

The graphical representation of SDL. [Recommendation Z.100, § 3.1.]

SDL/PR

The textual phrase representation of SDL. [Recommendation Z.100, § 3.1.]

SDL/PE

The pictorial element form of SDL, which is a state oriented extension of *SDL/GR.* [Recommendation Z.100, § 3.5.]

SELF

The *SELF data item* of a *process* is the unique *process instance* value which distinguishes it from any other *process instance.* [Recommendation Z.101, § 2.3.]

SENDER

The *SENDER* data item of a *process* is equal to the origin process data item of the most recently consumed *signal.* [Recommendation Z.101, § 2.3.]

SET

The *operator* defined for the *Timer data type* which allows timers to be set. [Recommendation Z.104, § 5.12.]

shared value

A *shared value* is the *value* associated with a *variable* which has the *reveal* attribute in one *process* and is *viewed by another.* [Recommendation Z.101, § 2.3.]

SIGNAL

The *formal parameter* attribute for a *signal* parameter of a *procedure.* [Recommendation Z.103, § 2.3.]

signal

The term *signal* on its own is a synonym for *signal instance.*

signal definition

A *signal definition* defines a *name* as being a *signal name* and associates a list of zero or more *data types* with the *signal name.* [Recommendation Z.101, § 2.2.]

signal instance

A *Signal instance* is an *instance* of a *signal* communicating information to a *process instance* from either the *output action* of another *process instance* or the *environment.* Alternatively, a *signal instance* is used to communicate information from the *output action* of a *process* to the *environment.* [Recommendation Z.101, § 2.3.]

signalling receiver PE

A *pictorial element* representing a signalling receiver. [Recommendation Z.103, § 6.1.]

signalling sender PE

A *pictorial element* representing a signalling sender. [Recommendation Z.103, § 6.1.]

signal list

The list of *names* of all the *signals* which can be conveyed by a *channel* or internally in a *block* from one *process* to another. [Recommendation Z.101, § 2.2.]

signal list symbol

The symbol on a block *interaction diagram* of *SDL/GR* representing a *signal list* associated with a *channel* or *signal route symbol.* [Recommendation Z.101, § 3.1.]

signal route symbol

The *symbol* on an *interaction diagram* of *SDL/GR* indicating the flow of *signals* between a *process* and either another *process* in the same *block* or the *channels* connected to the *block.* [Recommendation Z.101, § 3.1.]

specification

The requirements of a system are defined in a *specification* of that system. A *specification* consists of *general parameters* required of the system and the *functional specification* (FS) of its required behaviour. [Recommendation Z.100, § 1.1.]

specification and description language (SDL)

The CCITT language used in the presentation of the *functional specification* and *functional description* of the internal logic processes in stored programmed control (SPC) switching systems. [Recommendation Z.100, § 1.1.]

start action

The *start action* of a *process* is interpreted before any other *action.* The *start action* initializes the *formal parameters* of the process. [Recommendation Z.101, § 2.3.]

start node

The *start node* in a *process graph* is the only node which does not follow another node. There is only one *start node* in a *process graph* and interpretation of a *process* starts at this node. The *start node* is where an *input action* takes place. [Recommendation Z.101, § 2.2.]

start symbol

The *symbol* in a *process diagram* of *SDL/GR* representing a *start node.* [Recommendation Z.101, § 3.3.]

state

A *state* is a condition in which the action of a *process* is *suspended* awaiting an *input signal.* [Recommendation Z.101, § 2.1.]

state node

A *node* in a *process graph* or a *procedure graph* where the *process* enters a *state.* [Recommendation Z.101, § 2.2.]

state picture

A *state picture* is a *state symbol* incorporating *pictorial elements* used to extend *SDL/GR* to *SDL/PE.* [Recommendation Z.103, § 6.]

state symbol

A *symbol* in a *process diagram* of *SDL/GR* representing the SDL concept of one or more states. [Recommendation Z.101, § 3.3.]

stop

An *action* which terminates a *process instance.* [Recommendation Z.101, § 2.3.]

stop node

A *node* in a *process graph* where the *stop* takes place. [Recommendation Z.101, § 2.2.]

stop symbol

The *symbol* on a *process diagram* of *SDL/GR* representing a *stop.* [Recommendation Z.101, § 3.1.]

String

The *predefined data type generator* generating *data types* with *values* which are lists of items of the *data type* used to parameterize the *string generator.* [Recommendation Z.104, § 5.9.]

Struct

A *data type definition* with *Struct* implicitly introduces *data types* for field names and implicit *axioms* which define the use of field names with *extract!* and *insert!* for partial values of structures. [Recommendation Z.104, § 4.3.]

sub-block

A *sub-block* is a *block* contained within another *block. Sub-blocks* are formed when a *block* is *partitioned.* [Recommendation Z.102, § 2.1.]

sub-block definition

A *sub-block definition* defines a *block* and forms part of a *block substructure definition.* [Recommendation Z.102, § 2.2.]

sub-block symbol

The symbol in a *block interaction diagram* or *block tree diagram* of *SDL/GR* which represents a *sub-block* and is identical to a *block symbol.* [Recommendation Z.102, § 3.2.]

sub-channel

A *sub-channel* is a *channel* formed when a *block* is *partitioned.* [Recommendation Z.102, § 2.1.]

sub-process

A *sub-process* is a *process* formed when a *process* is *partitioned.* [Recommendation Z.102, § 2.3.]

subscriber line PE

A *pictorial element* representing a subscriber line. [Recommendation Z.103, § 6.1.]

switchboard PE

A *pictorial element* representing a switchboard piece of terminal equipment. [Recommendation Z.103, § 6.1.]

switching module PE

A *pictorial element* representing a switching module associated with a connected or reserved switching path. [Recommendation Z.103, § 6.1.]

synonym definition

A *definition* of a *name* for a *data value.* [Recommendation Z.104, § 4.12.]

syntax diagram

Syntax diagrams are diagrams used to define the *SDL/PR* concrete syntax. [Recommendation Z.100, § 3.4.]

syntype

A *syntype* introduces a set of *values* which corresponds to a subset of the *values* of the parent *newtype. Access!, declare!* and *assign!* are the only operators for *syntypes* since *values* prior to assignment and after extraction from *syntype variables* are always *values* of the parent *newtype.* [Recommendation Z.104, § 4.3.]

system

A *system* is a set of *blocks* connected to each other and the *environment* by *channels.* The term *system* on its own is a synonym for *system instance.* [Recommendation Z.101, § 2.1.]

system boundary

The *system boundary* is the boundary between the *blocks* defined in terms of SDL and the *environment.* [Recommendation Z.101, § 2.3.]

system definition

A *system definition* defines the properties of a *system.* The properties of a *system* are the *blocks, channels, signals* associated with the *channels, data types* and *synonyms* of the *system.* [Recommendation Z.101, § 2.2.]

task

A *task* is an action within a *transition* containing either a sequence of *assignment statements, set statements* or *reset statements* or informal text. The interpretation of a *task* depends on and may act on information held by the system. [Recommendation Z.101, § 2.2.]

task node

A *node* in a *process graph* or a *procedure graph* where a *task* takes place. [Recommendation Z.101, § 2.2.]

task symbol

A symbol in a *process diagram* of *SDL/GR* representing the SDL concept of a task. [Recommendation Z.101, § 3.3.]

terminal equipment PE

One of a possible six *pictorial elements* representing the following types of terminal equipment: telephone on-hook, telephone off-hook, trunk, subscriber line, switchboard, or other. [Recommendation Z.103, § 6.1.]

text extension symbol

The text associated with this symbol is regarded as belonging to the symbol in *SDL/GR* to which the *text extension symbol* is attached. [Recommendation Z.101, § 3.]

Time

The *predefined data type* representing absolute time. [Recommendation Z.104, § 5.10.]

Timer

The *predefined data type* used for timers and which defines the *operators SET* and *RESET* for timers. [Recommendation Z.104, § 5.12.]

time supervision of a process PE

A *pictorial element* representing the running of a supervisory timer. [Recommendation Z.103, § 6.1.]

transition

A *transition* is an *action* sequence which occurs when a *process instance* changes from one *state* to another in response to an *input.* [Recommendation Z.101, § 2.2.]

transition string

A *transition string* is a sequence of zero or more *actions* between an *input node* and the following *state node, stop node* or *procedure return node.* [Recommendation Z.101, § 2.2.]

trunk PE

A *pictorial element* representing a trunk line interface. [Recommendation Z.103, § 6.1.]

type

A *type* is a set of properties for entities. The classes of *types* in SDL are *blocks, channels, data items, procedures, processes, signals* and *systems.* A *type* may be composed from entities of the same class, in which case these entities are sub-types (e.g. a *block* composed from *sub-blocks*). [Recommendation Z.100, § 2.1.]

type definition

A *type definition* defines the properties of a *type.* [Recommendation Z.100, § 2.1.]

valid input signal

At each *state* in a *process* there is a set of *signal names* for *signals* which can either be *input.* A *valid input signal* is a *signal name* which is a member of any of these sets. [Recommendation Z.101, § 2.3.]

value

A data *value* of a *data type* is one of the *values* which are associated with a *variable* of that *data type,* and which can be used with an *operator* requiring a *value* of that *data type.* [Recommendation Z.104, § 1.]

variable

A *variable* is an entity owned by a *process* which can be assigned a *value.* A *variable* yields the last value which was assigned to it when it is accessed. [Recommendation Z.101, § 2.3.]

variable definition

A *variable definition* defines an *instance* of a *data type.* [Recommendation Z.101, § 2.2.]

view

A *variable* can be *viewed* if the *value* associated with the *variable* is *revealed* by the *process* owning the *variable* and another *process* can access the *value.* [Recommendation Z.101, § 2.3.]

view definition

A *view definition* is a part of a *process definition* which defines and which *variables* are owned by another *process* and are only *viewed* by the process containing the *view definition.* [Recommendation Z.101, § 2.2.]

viewing expression

A *viewing expression is used within an expression* to indicate that the latest *value* of a *viewed variable* is obtained. [Recommendation Z.101, § 2.3.]

ANNEX B

(to Recommendations Z.100 to Z.104)

Abstract syntax summary

This summary contains all abstract syntax and associated wellformedness rules that are defined in Recommendations Z.101, Z.102, Z.103 and Z.104. The syntax is explained using an extended BNF[1] form as defined in Recommendation Z.200 and the rules are given in English. As this is a summary, the Recommendations should be consulted for the exact definitions.

B.1 *System*

(1) <System Definition> ::=

(Z.101) *System*— name

(Z.101) [<Block Definition>]+

(Z.101) [<Channel Definition>]*

(Z.101) [<Signal Definition>]*

(Z.104) [<Data Definition>]*

(Z.103) [<Procedure Definition>]*

Wellformedness

The structure definition must contain a signal definition for each signal name in the signal lists in each of the also contained channel definitions (Recommendation Z.101).

[1] BNF = Backus Naur Form.

B.2 *Block*

(2) <Block Definition> ::=

(Z.101) *Block*-name

(Z.101) [<Process Definition>]*

(Z.101) [<Signal Definition>]*

(Z.104) [<Data Definition>]*

(Z.103) [<Procedure Definition>]*

(Z.102) [<Block Substructure Definition> !]

Wellformedness

For each of the signal names associated with the input and output operations in the contained processes, either the block contains a signal definition or is a signal definition contained in the enclosing structure.

If a block definition contains block substructure definition, it need not contain process definitions (Recommendation Z.102).

B.3 *Block Sub-structure*

(3) <Block Sub-structure Definition> ::=

(Z.102) [<Sub-block Definition>]+

(Z.102) [<Sub-channel Definition>]

(Z.102) [<Channel Definition>]+

(Z.102) [<Process Sub-structure Definition>]*

(Z.102) [<Signal Definition>]*

(Z.104) [<Data Definition>]*

(Z.103) [<Procedure Definition>]*

(3.1) <Sub-block Definition> ::=

(Z.102) <block definition>

(3.2) <Sub-channel Definition> ::=

(Z.102) <Channel Definition>

Wellformedness

For each of the process definitions contained in the enclosing block, the block substructure definition must contain a process sub-structure definition.

To each of the terminating endpoints of channels to the enclosing block, there must at least be one sub-channel definition having that endpoint as the originating endpoint, and the reverse must hold for all originating channel endpoints to the enclosing block. The union of the signal lists of the sub-channel definitions having the same endpoint as a channel leading to or from the enclosing block must be identical to the signal list of that channel. In addition to this, the signal lists of the channel definitions originating from a terminating endpoint must be disjointed (Recommendation Z.102).

A channel definition contained in the sub-structure must connect sub-blocks to each other. All sub-channels must connect channel endpoints of the enclosing block to sub-blocks.

B.4 *Channel*

(4) <Channel Definition> ::=

(Z.101) *Channel*-name

(Z.101) [*Origin Block*-identifier! ENVIRONMENT]

(Z.101) [*Destination Block*-identifier! ENVIRONMENT]

(Z.101) <Signal List>

(Z.102) [<Channel Sub-structure Definition> !]

(4.1) <Signal List> ::=

(Z.101) [*Signal*-name]+

Wellformedness

Only one of the origin block or destination block identifiers may be replaced with a reference to the environment (Recommendation Z.101).

The origin block identifier and the destination block identifier associated with a channel must be different and each must be the identifier of a block in the system or sub-block in the block, or must be the ENVIRONMENT (Recommendation Z.101).

B.5 *Channel sub-structure*

(5) <Channel sub-structure Definition> ::=

(Z.102) <Incoming Channel Definition>

(Z.102) <Outgoing Channel Definition>

(Z.102) [<Channel Definition>]*

(Z.102) [<Block Definition>]+

(Z.102) [<Signal Definition>]*

(Z.104) [<Data Definition>]*

(Z.103) [<Procedure Definition>]*

(5.1) <Incoming Channel Definition> ::=

(Z.102) <Channel Definition>

(5.2) <Outgoing Channel Definition> ::=

(Z.102) <Channel Definition>

Wellformedness

The incoming channel definition has the same origin endpoint as the enclosing channel definition. The outgoing channel has the same terminating endpoint as the enclosing channel definition. Both the incoming and the outgoing channel definitions have signal lists identical to the signal list of the enclosing channel definition (Recommendation Z.102).

B.6 *Signal*

(6) <Signal Definition> ::=

(Z.101) *Signal*-name

(Z.101) [*Data-Type*-identifier]*

Wellformedness

The used data type identifiers must either be predefined or names of data type definitions in the enclosing structural entities.

B.7 *Process*

(7) <Process Definition> ::=

(Z.101) *Process*-name

(Z.101) <number of instances>

(Z.101) [<Formal Parameter>]*

(Z.104) [<Data Definition>]*

(Z.101) [<Viewing Definition>]*

(Z.103) [<Procedure Definition>]*

(Z.101) <Process Graph>

(7.1) <number of instances> ::=

<*integer value* identifier> <*integer value* identifier>

(7.2) <Viewing Definition> ::=

(Z.101) <*variable* identifier>

<*type* identifier>

<*process definition* identifier>

Wellformedness

All type names referred to must either be predefined, defined in the process definition or defined in the enclosing structural concepts.

B.8 *Process sub-structure definition*

(8) <Process Sub-structure Definition> ::=

(Z.102) *Process*-name

(Z.102) [*Sub-process*-name *sub-block*-name]+

Wellformedness

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

Each signal name in the valid input signal set of the process must appear in exactly one of the valid input signal sets of the sub-processes. Each signal name attached to an output node of the process must be attached to at least one output node in one of the sub-processes.

B.9 *Process Graph*

(9) <Process Graph> ::=

(Z.101) <*Process* Start Node> <Process Transition>

(9.1) <Process Transition> ::=

<Transition String> <State Node>

!<Transition String> <Stop Node>

B.10 *Transition String*

(10) <Transition String> ::=

(Z.101) <Task Node> <Transition String>

(Z.101) !<Output Node> <Transition String>

(Z.101) !<Decision Node>

(Z.101) !<Create Request Node> <Transition String>

(Z.103) !<Procedure Call Node> <Transition String>

(Z.101) !

B.11 *Start node*

(11) <Start Node> ::=

B.12 *State node*

(12) <State Node> ::=

(Z.101) *State*-name

(Z.101) <Save Signal Set>

(Z.101) [<Input Node>]+

(12.1) <Save Set> ::=

(Z.101) [*Signal*-identifier]*

B.13 *Stop Node*

(13) <Stop Node> ::=

B.14 *Task Node*

(14) <Task Node> ::=

(Z.101) [[<Statement>]+ ! [<informal test>]*]

B.14.1 *Statement*

(14.1) <Statement> ::=

(Z.101) <Set statement>

(Z.101) ! <reset statement>

(Z.101) ! <assignment statement>

(14.1.1) <Set statement> ::=

(Z.101) <*time* expression> *timer*-identifier

(14.1.2) <Reset statement> ::=

(Z.101) *timer*-identifier

(14.1.3) <Assignment statement> ::=

(Z.101) <*Assignment* operator>

(Z.101) <*variable* identifier>

(Z.101) <Expression>

B.15 *Output node*

(15) <Output Node> ::=

(Z.101) <*Signal*-identifier>

(Z.101) [<Expression>]*

(Z.101) <Destination>

(15.1) <Destination> ::=

(Z.101) <*PId*-Expression>

B.16 *Input Node*

(16) <Input Node> ::=

(Z.101) *Input Signal*-identifier <Process Transition>

(Z.103) ! *Input Signal*-identifier <Procedure Transition>

Wellformedness

A process transition may only appear in a process definition and procedure transition may only appear in a procedure definition.

B.17 *Decision node*

(17) <Decision Node> ::=

(Z.101) <Question>

(Z.101) <answer> [<answer>]+

(17.1) <Question> ::=

(Z.101) <Expression>

!<informal text>

(17.2) <Answer> ::=

(Z.101) [< *value*-identifier>]+

[<procedure transition> ! <process transition>]

Wellformedness

A decision node must be followed by two or more answers (Recommendation Z.101).

The expressions in the decision name and the answers must be of the same type.

B.18 *Procedure call node*

(18) <Procedure Call Node> ::=

(Z.103) *Procedure*-identifier

(Z.103) [<Expression>]*

(Z.103) [*Signal*-identifier]*

(Z.103) <additional save set>

(18.1) <additional save set> ::=

[<Signal name>]*

Wellformedness

Each expression must be of the same type as the corresponding formal parameter in the procedure definition having the procedure name.

There must be one expression for each formal parameter, of not type signal, in the procedure definition, and one signal identfier for each formal parameter of type signal.

B.19 *Create request node*

(19) <Create Request Node> ::=

(Z.101) *Process*-identifier

(Z.101) [<Expression>]*

Wellformedness

There must be an expression for each formal parameter in the/by the process identifier referenced process definition, and each expression must agree in type with the associated formal parameter (Recommendation Z.101).

B.20 *Procedure*

(20) <Procedure Definition> ::=

(Z.103) *Procedure*-name

(Z.103) [<Data Definition>]*

(Z.104) [<variable definition>]*

(Z.103) [<Procedure Definition>]*

(Z.103) [<Formal Parameter>]*

(Z.103) <Procedure Graph>

Wellformedness

Each signal identifier appearing in the procedure graph must also appear as a formal parameter of type signal.

B.21 *Procedure graph*

(21) <Procedure graph> ::=
(Z.103) <Procedure Start Node> <Procedure Transition>

(21.1) <Procedure Transition> ::=
(Z.103) <Transition String> <State Node>
(Z.103) !<Transition String> <Return Node>

B.22 *Procedure Start Node*

(22) <Procedure Start Node> ::=

B.23 *Return node*

(23) <Return Node> ::=

B.24 *Data definition*

(24) <Data Definition> ::=
(Z.104) <Data Type Definition>
(Z.104) !<Synonym Definition>

B.25 *Variable definition*

(25) <Variable Definition> ::=
(Z.101) *Variable*-name
(Z.101) *Type*-identifier
(Z.101) [<reveal attribute> !]

B.26 *Identifier*

(26) <Identifier> ::=
(Z.100) <Qualifier> Name

B.27 *Qualifier*

(27) <Qualifier> ::=
(Z.100) <Structural Name> <Entity Type Name>

B.28 *Structural name*

(28) <Structural name> ::=
(Z.100) *System*-name
(Z.100) [*Block*-name]* [*Channel*-name]*
(Z.100) [*Signal*-name]* [*Process*-name]*
(Z.100) [*Procedure*-name]* [*Type*-name]*

B.29 *Entity type name*

(29) <Entity Type Name> ::=
(Z.100) System !Block !Process
(Z.100) !Procedure !Signal !Channel
(Z.100) !Task !Decision !Start
(Z.100) !Stop !Create Request
(Z.100) !Return !Procedure Start
(Z.100) !Call !Variable !Data type
(Z.100) !Operator !Value

B.30 *Data type definition*

(30) <Data Type Definition> ::=
(Z.104) *Data Type*-name <Data Type Description>
(Z.104) !*Data Type*-name (Syn Type Description>

B.31 *Data type description*

(31) <Data Type Description> ::=
(Z.104) [*Value*-name]**
(Z.104) [<Operator Introduction>]+
(Z.104) [Type-Axiom]*

Note 1 – The notation [ ]** means list of zero or more and possibly infinite number of items. Other than the property that the list may be infinite, it is the same as [ ]*.

Note 2 – Type-Axiom is not further defined in the abstract syntax. There is, however, a concrete syntax for axioms.

Note 3 – A plus (+) indicates that the group must be present and can be further repeated any number.of times. If syntactic elements are grouped using square brackets ([and]), then the group is optional.

B.32 *Operator introduction*

(32) <Operator Introduction> ::=
(Z.104) <Universal Operator>
(Z.104) !*Operator*-name <Operator Typing>

B.33 *Universal operator*

(33) <Universal Operator> ::=
(Z.104) <Variable Operator>
(Z.104) !<Comparator>

B.34 *Variable operator*

(34) <Variable Operator> ::=
(Z.104) Assign
(Z.104) !Access
(Z.104) !Declare

B.35 *Comparator*

(35) <Comparator> ::=
(Z.104) Less than
(Z.104) !Greater than
(Z.104) !Equal

B.36 *Operator typing*

(36) <Operator Typing> ::=
(Z.104) <Data Type List> <*Result Type*-identifier>

B.37 *Data type list*

(37) <Data Type List> ::=
(Z.104) [< *Data-Type*-identifier >]+

Wellformedness (for 30-37)

All *value names* must be mutually exclusive in a *type description.*

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

For each *operator typing,* one of the *data type identifiers* in the *data type list* must be the *data type identifier* of the type being defined.

B.38 *Syn type abstract syntax*

(38) <Syn Type description> ::=
(Z.104) <*Parent Type* Identifier>
[< *Value*-identifier >]*

Wellformedness

The *value identifiers* must all be members of the set of *value identifiers* of the parent *data type identifier.*

B.39 *Informal text*

(39) <Informal Text> ::=
(Z.104) *Well-understood*-name

B.40 *Expression*

(40) <Expression> ::=
(Z.104) <Primary>
(Z.104) !<Operation>
!<Conditional expression>

B.40.1 *Operation*

(40.1) <Operation> ::=
<operator> [<expression>]*

B.40.2 *Conditional expression*

(Z.104) <Conditional expression> ::=
(Z.104) <*Boolean* expression>
(Z.104) <Expression> <Expression>

B.41 *Primary*

(41) <Primary> ::=
(Z.104) *Synonym*-identifier
(Z.104) ! *Value*-identifier
(Z.104) ! *Variable*-identifier

B.42 *Operator*

(42) <Operator> ::=

(Z.104) *Operator*-identifier

(Z.104) !<Universal Operator> *Type*-identifier

B.43 *Synonym definition*

(43) <Synonym Definition> ::=

(Z.104) *Synonym*-name <Constant Expression>

B.44 *Constant expression*

(44) <Constant Expression> ::=

(Z.104) <Constant Value>

(Z.104) !<Operator> [<Constant Expression>]+

B.45 *Constant value*

(45) <Constant Value> ::=

(Z.104) *Value*-identifier

(Z.104) !*Synonym*-identifier

ANNEX C1

(to Recommendations Z.100-Z.104)

SDL/GR summary

In SDL/GR, a system consists of:

- Block interaction diagram (BID),
- Channel sub-structure diagram,
- Block tree,
- Process Tree,
- Process diagram,
- State overview diagram.

Some of these documents are essential to provide a system specification/description whilst others are auxiliary documents that may help in an easier understanding of the specification/description.

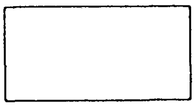
C1.1 *Block interaction diagram (BID)*

C1.1.1 *Symbols*

The BID contains a system name, a set of block symbols, environment symbols, a set of channel symbols and it may contain process symbols.

C1.1.1.1 *Frame line*

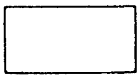
It surrounds the diagram and represents the boundary of the partitioned blocks.



CCITT-74040

C1.1.1.2 *Block symbol*

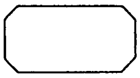
It contains the block name (see Recommendation Z.101, § 3.1.1 and U.G. § D.4.3.2).



CCITT-74040

C1.1.1.3 *Process symbol*

It contains the process name (see Recommendation Z.101, § 3.1.1 and U.G. § D.4.3.4) and may contain a couple of parenthesis containing the list of the formal parameters.



CCITT-74040

C1.1.1.4 *Environment symbol*

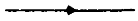
It is represented only by a keyword (see Recommendation Z.101, § 3.1.1).

ENVIRONMENT

C1.1.1.5 *Channel symbol*

It contains a channel name. The channel symbol has an origin end connected to a block symbol and a destination end connected to another block symbol. Alternatively, either the origin or the destination end points (but not both of them) may be connected to an environment symbol instead of a block symbol.

A channel symbol includes an arrowhead in the middle of the line to show the direction of the flow of signals.



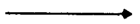
CCITT-74040

C1.1.1.6 *Signal route symbol*

A signal route symbol in a block may be associated with a signal list symbol.

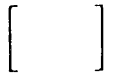
A signal route symbol leads either from one process to another, or from a process to the origin end of a channel (at the block boundary), or from the destination end of a channel (at the block boundary) to a process.

Signal route symbols may converge at the origin end of a channel (at the block boundary) and 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 symbols.



CCITT-74040

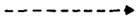
C1.1.1.7 *Signal list symbol*



CCITT-74040

The signal list symbol contains a list of names. The signal list itself may have a name. which is written above the symbol. Entries in the list, which are separated by commas and put in columns or rows, are the names of individual signals or names of other signal lists. Within the list, signal list names are distinguished from individual signal names by enclosing each signal list name in a further pair of square brackets.

C1.1.1.8 *Create symbol*



CCITT-74040

It is used between process symbols to indicate that a process instance at the arrowhead end is created by a process instance at the origin end of the dashed line. The create symbol may connect the same process to indicate the creation of an instance of that process by another instance of the same process.

C1.1.2 *Rules*

Each block has a certain number of directed lines to interconnect it with other blocks and/or with the environment.

If processes are represented in the interaction diagram, the signal lines between processes have to be only between processes of the same block that communicate with each other.

C1.1.3 *Conventions*

Channel symbols and signal lines should respectively join block symbol boundaries and process symbol boundaries preferably at 90 degrees. If necessary channel symbols may contain 90 degree bends.

Crossing of lines has no meaning.

C1.2 *Channel sub-structure diagram*

As the components are blocks and channels the diagram resembles a block sub-structure diagram.

C1.2.1 *Symbols*

The symbols used in this diagram are the same as those used in block sub-structure diagram.

C1.2.2 *Rules*

The rules for connecting the diagram are the same as for the block sub-structure diagram.

C1.2.3 *Conventions*

The same graphical conventions, as described for FBID, also apply for the channel sub-structure diagram.

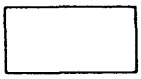
C1.3 *Block tree*

The block tree diagram contains boxes and "partitioned into" lines.

C1.3.1 *Symbols*

C1.3.1.1 *Box*

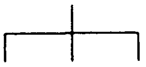
A box represents a system or a block. The name of the represented object should appear inside the symbol.



CCITT-74040

C1.3.1.2 *"Partitioned into" lines*

They represent the relation that the object above is partitioned into the sub-blocks below.



CCITT-74040

C1.3.2 *Rules*

The symbols are connected to form a hierarchical tree. The rules for the connection of the diagram are:

- There is exactly one root box (top box).
- Any other box must follow one branch of a "partitioned into" line.
- Any box may be followed by one "partitioned into" line.
- A "partitioned into" line must be followed by boxes at all its branches, and must follow a box.

C1.3.3 *Conventions*

The tree should be so drafted as one partitioning level appears as a consistent level in the representations, i.e. the "partitioned into" lines should have equal length downwards in the diagram.

C1.4 *Process tree*

The process tree contains process symbols and "partitioned into" lines.

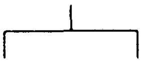
C1.4.1 *Symbols*

C1.4.1.1 *Process symbol*

As defined in Recommendation Z.101 (§ 3.1.2), the name of the referred process should appear inside the symbol. The comment syntax is used to show in which block the process is allocated.

C1.4.1.2 *"Partitioned into" lines*

They represent the relation that the process above is partitioned into the sub-processes below, and may be replaced by these.



CCITT-74040

C1.4.2 *Rules*

The symbols are connected to form a hierarchical tree. The rules for the connection of the diagram are:

- There is exactly one root process symbol (top process), which is followed by one "partitioned into" line.
- The "partitioned into" line is followed, at each of its branches by a process symbol.

C1.4.3 *Conventions*

If a process 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.

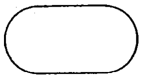
C1.5 *Process diagram*

A process diagram is a set of symbols connected by flow lines.

C1.5.1 *Symbols*

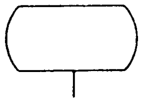
C1.5.1.1 *Start symbol*

It contains the process name, of the process it describes (see Recommendation Z.101, § 3.3.1 and U.G. §§ D.4.3 and D.6.3).


CCITT-74040

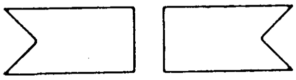
C1.5.1.2 *State symbol*

It contains the state name or an asterisk or an asterisk followed by states names within square brackets (see Recommendation Z.101, § 3.3.1 and U.G. §§ D.4.3 and D.6.3).


CCITT-74040

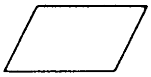
C1.5.1.3 *Input symbol*

It contains the signal names separated by commas or an asterisk (see Recommendation Z.101, § 3.3.1 and U.G. §§ D.4.3 and D.6.3).


CCITT-74040

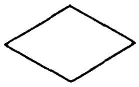
C1.5.1.4 *Save symbol*

It contains the signal names separated by commas or a star (see Recommendation Z.101, § 3.3.1 and U.G. §§ D.4.3 and D.6.3).


CCITT-74040

C1.5.1.5 *Decision symbol*

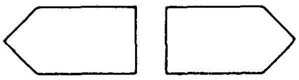
It contains the decision name (optional) and a formal or informal text phrase (see Recommendation Z.101, § 3.3.2 and U.G. §§ D.4.3 and D.6.3).



CCITT-74040

C1.5.1.6 *Output symbol*

It contains the signal names separated by commas (see Recommendation Z.101, § 3.3.1 and U.G. §§ D.4.3 and D.6.3).



CCITT-74040

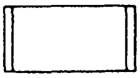
C1.5.1.7 *Task symbol*

It contains the task name (optional and a formal or informal text-phrase (see Recommendation Z.101, § 3.3.1 and U.G. §§ D.4.3 and D.6.3).

CCITT-74040

C1.5.1.8 *Procedure call symbol*

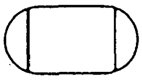
It contains the procedure name and the actual parameters within round brackets (see Recommendation Z.103, § 2.2 and U.G. §§ D.4.3 and D.6.3).



CCITT-74040

C1.5.1.9 *Procedure start symbol*

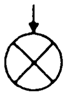
It contains the procedure name and the formal parameters within round brackets (see Recommendation Z.103, § 2.2 and U.G. §§ D.4.3 and D.6.3).



CCITT-74040

C1.5.1.10 *Return symbol*

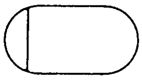
It is a circle with a cross inside (see Recommendation Z.103, § 2.2 and U.G. §§ D.4.3 and D.6.3).



CCITT-74040

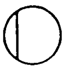
C1.5.1.11 *Macro inlet symbol*

It contains the macro name (see Recommendation Z.103, § 4.2 and U.G. §§ D.4.3 and D.6.3).


CCITT-74040

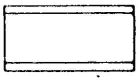
C1.5.1.12 *Macro outlet symbol*

It is a circle with a bar inside (see Recommendation Z.103, § 4.2 and U.G. §§ D.4.3 and D.6.3).


CCITT-74040

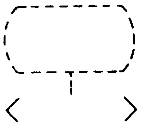
C1.5.1.13 *Create symbol*

It contains the process name and the actual parameters within brackets (see Recommendation Z.101, § 3.3.1 and U.G. §§ D.4.3 and D.6.3).


CCITT-74040

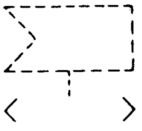
C1.5.1.14 *Continuous signal symbol*

It contains the condition text and then the keyword PRIORITY followed by the priority number associated (see Recommendation Z.103, § 3.3.3 and U.G. §§ D.4).


CCITT-74040

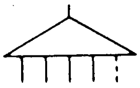
C1.5.1.15 *Enabling condition symbol*

It contains the condition text (see U.G. §§ D.4).


CCITT-74040

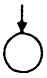
C1.5.1.16 *Alternative symbol*

It contains the alternative text (see Recommendation Z.103, § 5.2 and U.G. § D.4).


CCITT-74040

C1.5.1.17 *Join or connector symbol*

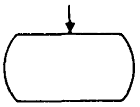
It contains the join name (see Recommendation Z.101, § 3.3.1 and U.G. § D.4).



CCITT-74040

C1.5.1.18 *Nextstate symbol*

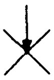
It contains the state name or a hyphen (see Recommendation Z.101, § 3.3.1 and U.G. § D.4).



CCITT-74040

C1.5.1.19 *Stop symbol*

It is a cross (see Recommendation Z.101, § 3.3.1 and U.G. § D.4).

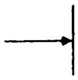


CCITT-74040

C1.5.2 *Rules*

- A state has to be connected with one or more input symbols or saves symbols (any connection with other symbols is wrong).
- A call procedure node can follow any node which is neither a state node, a stop node nor a return node.
- A call node can be followed by any node except a start node, a procedure start node, an input node, a return node or a save node.
- Enabling conditions may be attached to any input symbols.
- A continuous signal is attached to a flowline from a state symbol but without an input symbol.
- The macro call symbol may be inserted in any diagram (process diagram, BID, state overview diagram), and in any place of the diagram. It can have one or more inlets. In the case where more inlets are present, a label should be associated to each inlet. It can have none, one or more outlets; in the latter case a label should be associated to each outlet. Inlets are represented with arrows pointing to the macro symbol; outlets are represented with arrows outgoing from the macro symbol. The macro symbol inlets/outlets are connected to other symbols through flowlines of appropriate type, according to their meaning.
- Neither the stop symbol nor the return symbol of a procedure and of a macro are followed by any symbol.
- The create symbol can be put where there can be a task.
- The alternative symbol can be inserted directly in a transition only if the alternative behaviours do not include states and if they terminate at some point.
- A solid flow line may be broken by a pair of associated connectors, with the flow assumed to be from the out-connector to its associated in-connector (join symbol).
- 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 more than one out-connector associated with a single in-connector or as separate flow lines entering the same symbol.

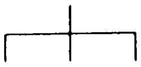
Example: convergence



CCITT-74040

- An out-connector is not followed by any other symbol, and no flow line starts from it.
- 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.

Example: divergence



CCITT-74040

- Arrowheads are required whenever two flowlines converge and whenever a flow line enters an out-connector or a state symbol. Arrowheads are prohibited on flow lines entering input symbols.
- The connect symbol is connected to a symbol or a solid flow line.

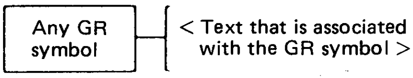
C1.5.3 *Conventions*

- All the symbols of the same type shall preferably be of the same size within any one diagram.
- The preferred orientation of the symbols is horizontal and the preferred ratio of symbols is 2:1.

C1.5.4 *General symbols*

C1.5.4.1 *Text extension symbol*

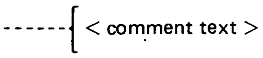
It can be attached to all SDL/GR symbols. The text contained in this symbol is to be regarded as contained in the symbol to which the text extension symbol is attached.



CCITT-74040

C1.5.4.2 *Comment symbol*

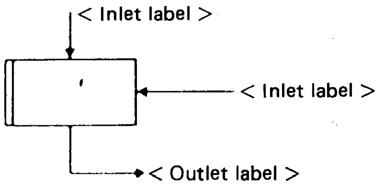
The comment symbol contains the comment text.



CCITT-74040

C1.5.4.3 *Macro, inlet and outlet symbols*

The macro symbol is the reference to the macro definition. The inlets to and the outlets from the macro are represented by flow lines leading to and from the symbol. Labels may be optionally attached to the flow lines. The macro symbol contains the name of the macro definition.



CCITT-74040

C1.6 *State overview diagram*

This diagram represents the sequence of states in the process. It is possible to see which states a certain state can be reached from (see U.G. §§ D.4.2 and D.4.3).

C1.6.1 *Symbols*

C1.6.1.1 *State symbol*

In this diagram the state symbol is a circle that contains the state name. (See U.G. §§ D.4.2 and D.4.3.)

C1.6.2 *Rules*

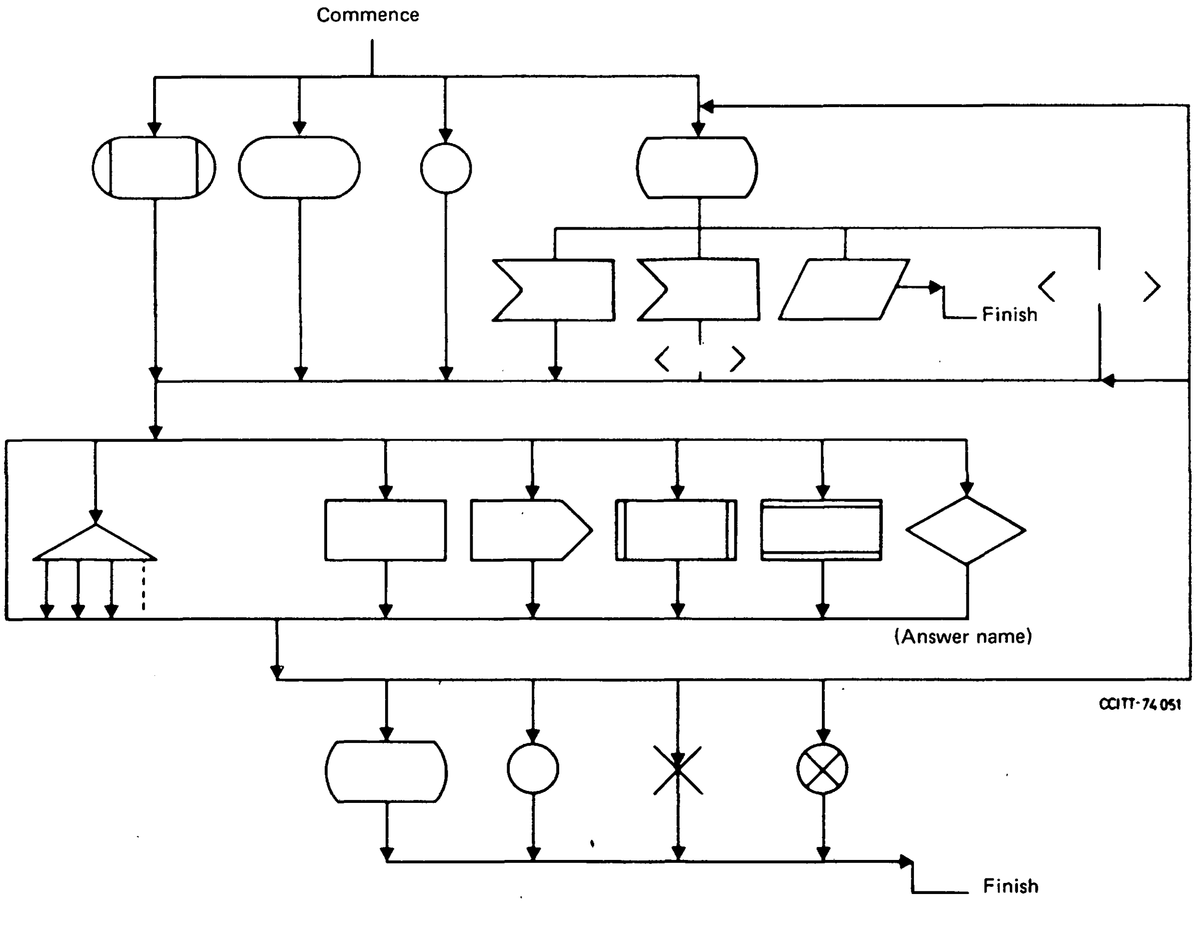
- The states are connected by arcs. The input signal names can be optionally written on the arcs.

C1.7 *General conventions*

In the graphic syntax there are some drawing conventions valid for all the types of diagrams:

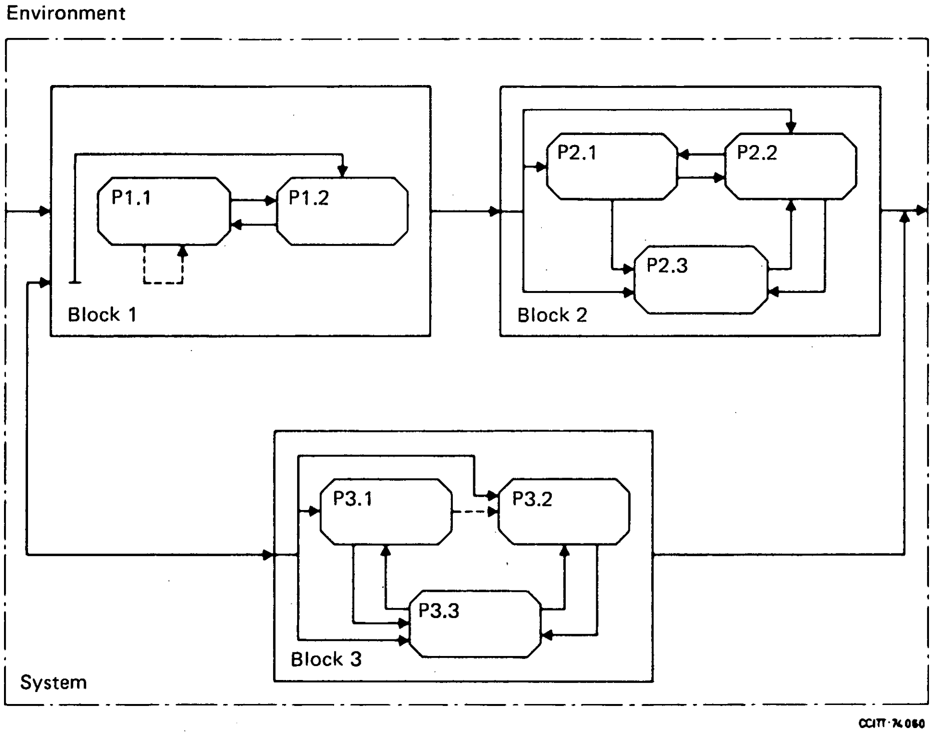
- the aspect ratio and size of symbols are variable at the user's discretion;
- symbol boundaries must not overlay or cross. An exception to this rule applies for the channel and signal flow symbols which may cross each other. There is no logical relationship between channel or signal flow symbols which do cross.

C1.8 *Allowable connections of symbols by flowline in an SDL/GR process diagram*

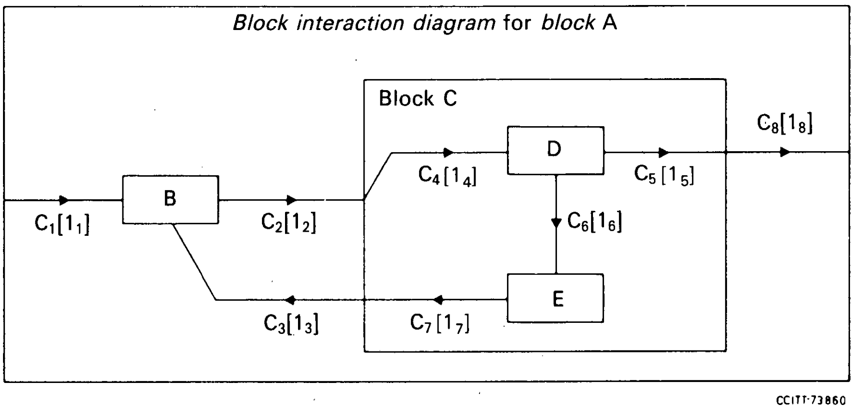


C1.9 *Examples*

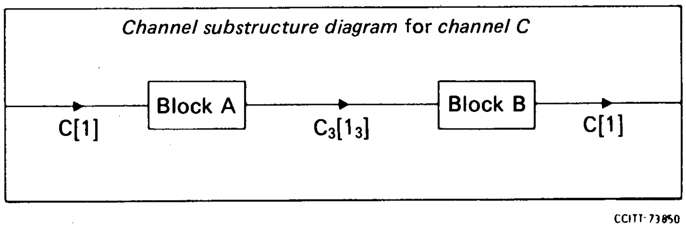
C1.9.1 *Block interaction diagram*



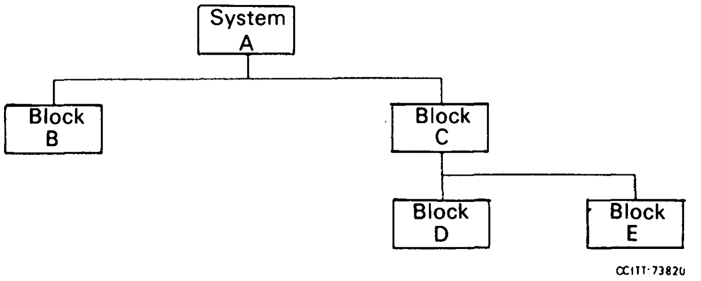
C1.9.2 *Block interaction diagram*



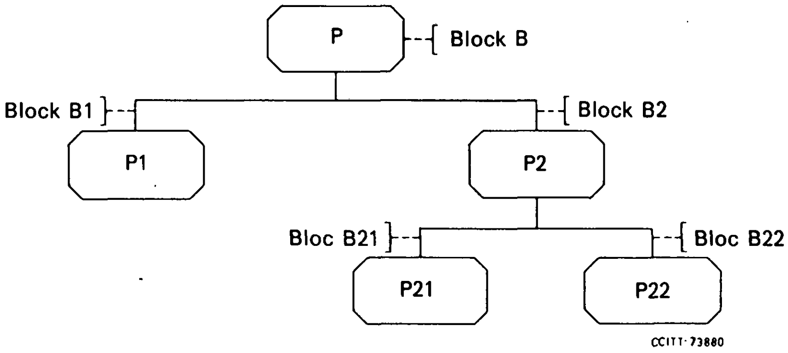
C1.9.3 *Channel sub-structure diagram*



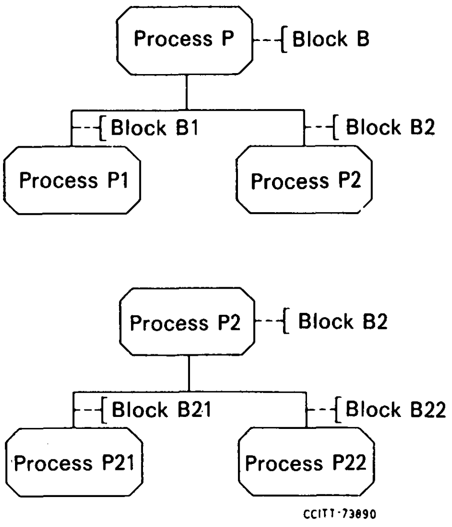
C1.9.4 *Block tree*



C1.9.5 *Process tree*



Another representation of the same process tree (representation useful when the process tree is large.



ANNEX C2

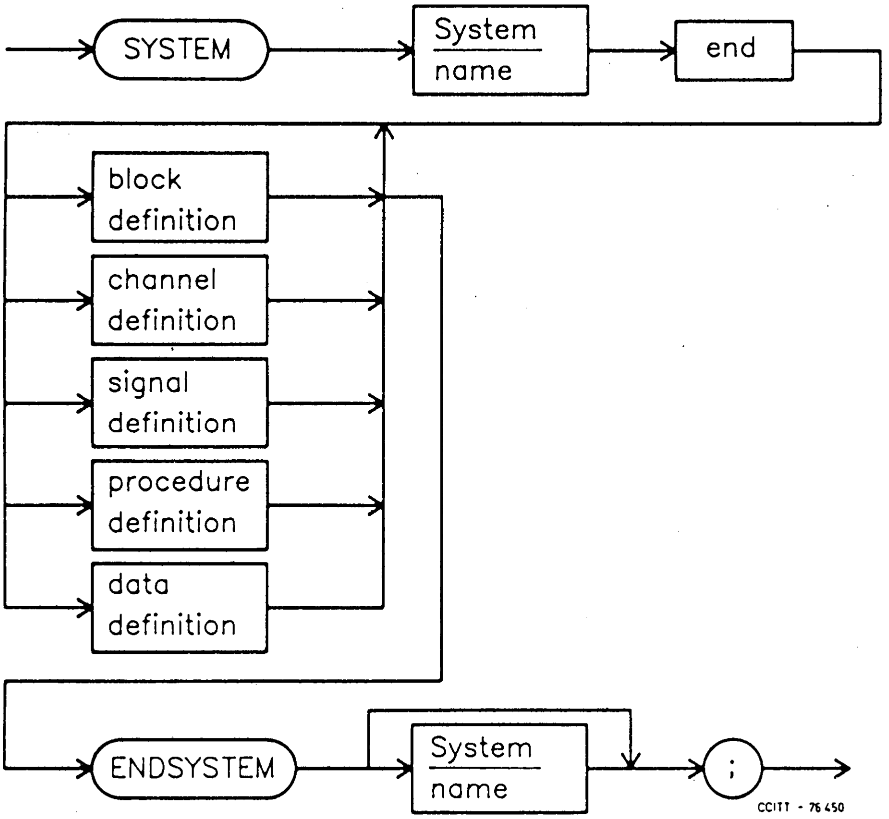
(to Recommendations Z.100 to Z.104)

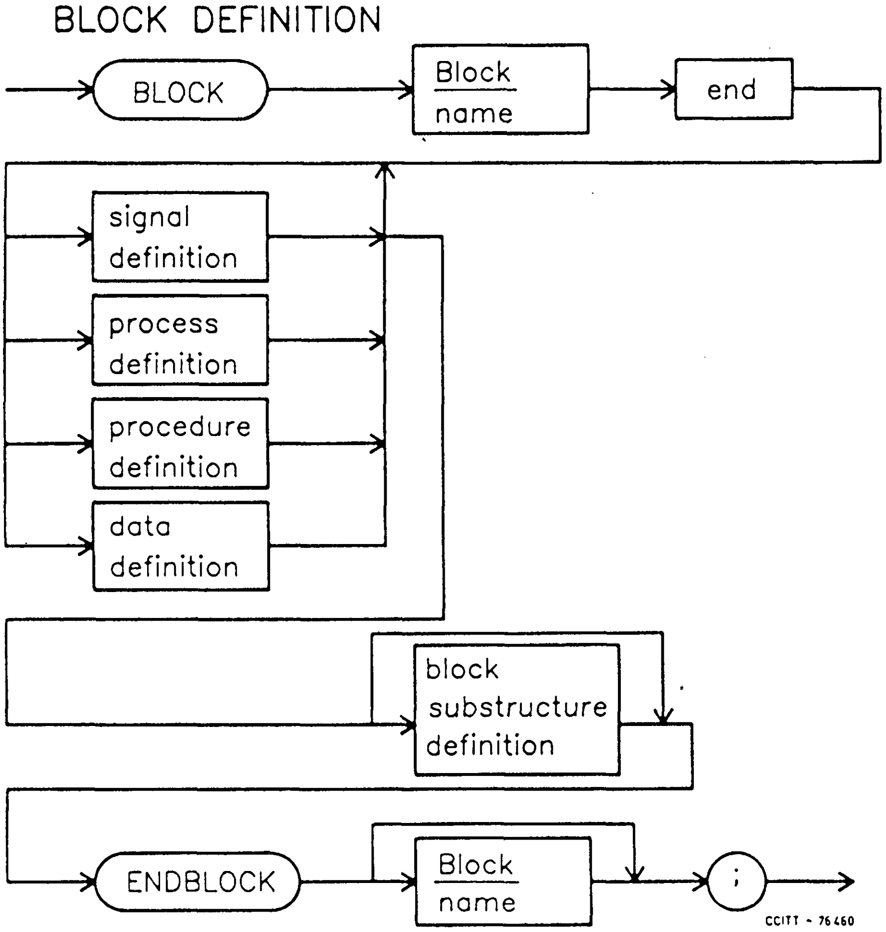
SDL/PR summary

C2.1 *Syntax*

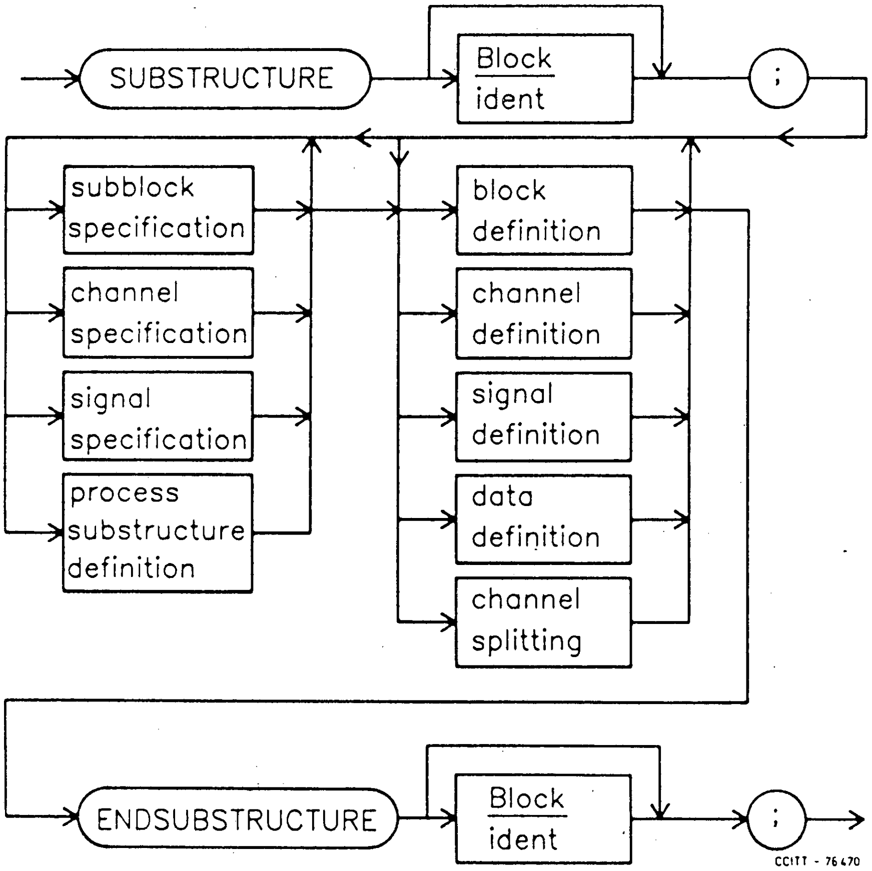
Syntax diagrams

SYSTEM DEFINITION

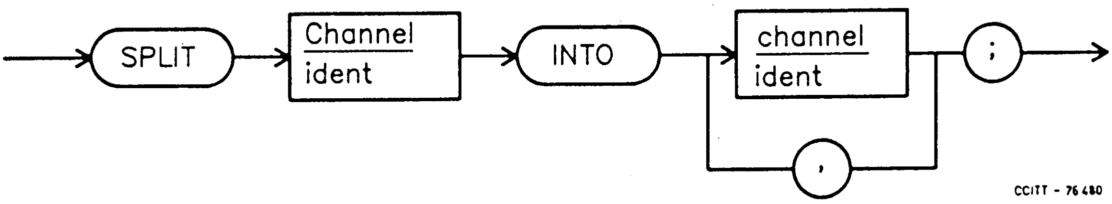



BLOCK DEFINITION
BLOCK
Block name
end
signal definition
process definition
procedure definition
data definition
block substructure definition
ENDBLOCK
Block name
;
CCITT - 76460

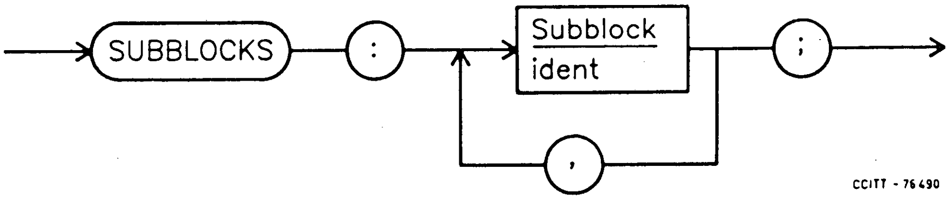
BLOCK SUBSTRUCTURE DEFINITION



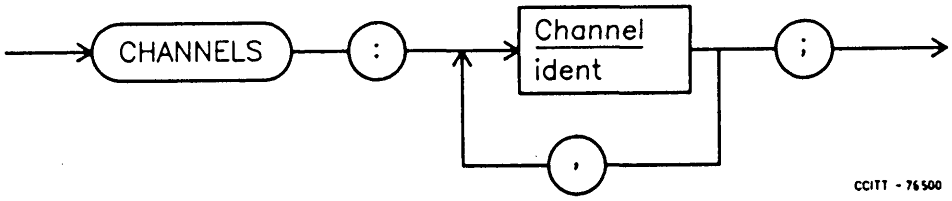
CHANNEL SPLITTING



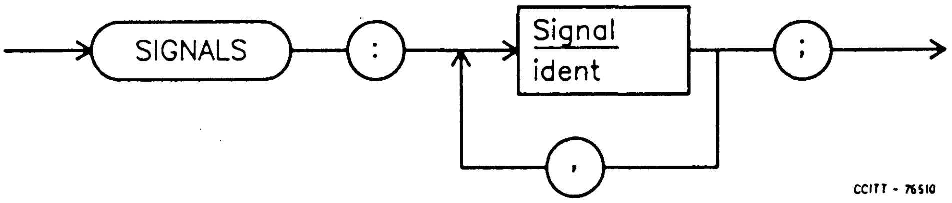
SUBBLOCK SPECIFICATION



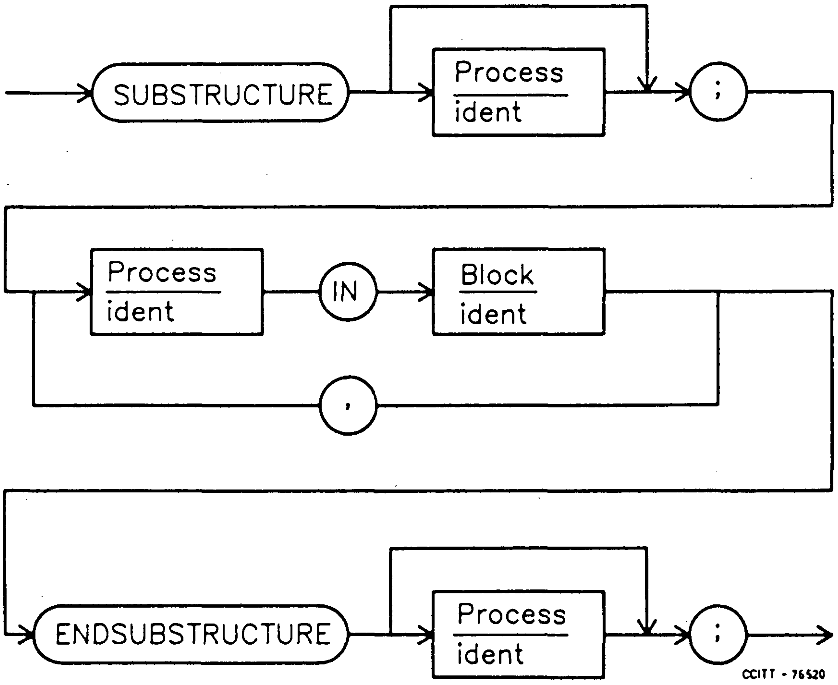
CHANNEL SPECIFICATION



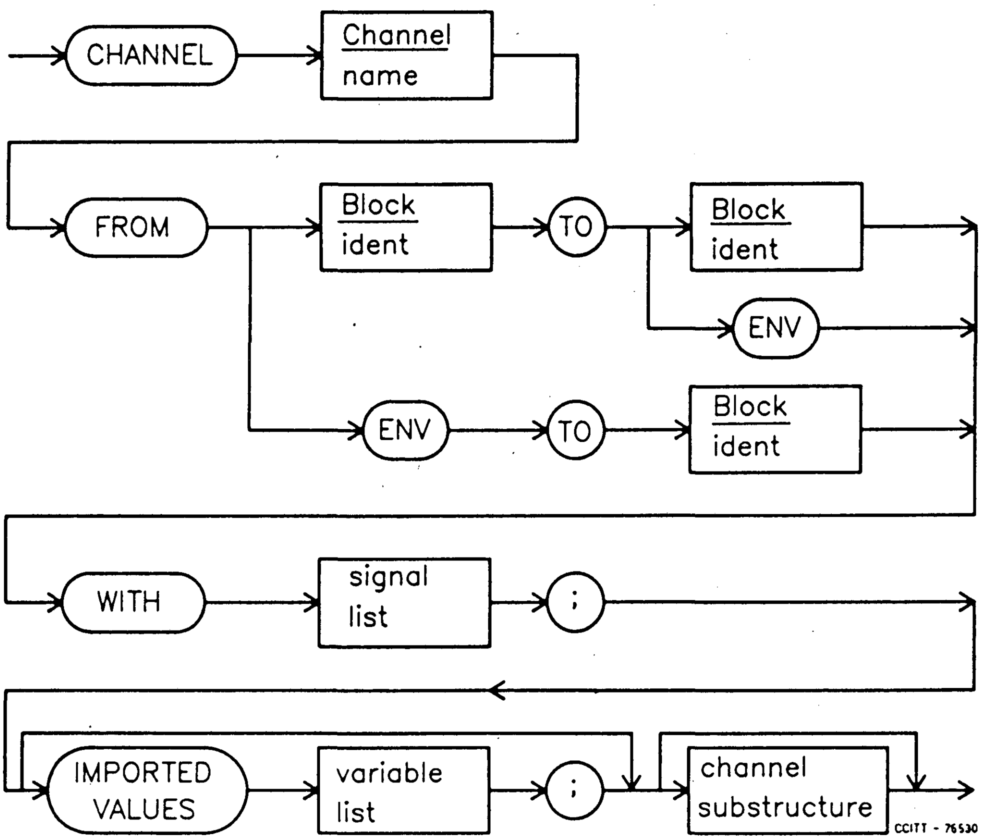
SIGNAL SPECIFICATION



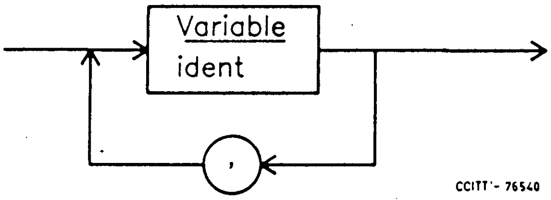
PROCESS SUBSTRUCTURE DEFINITION



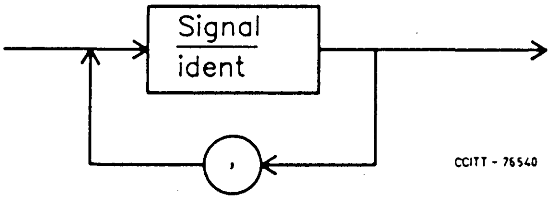
CHANNEL DEFINITION



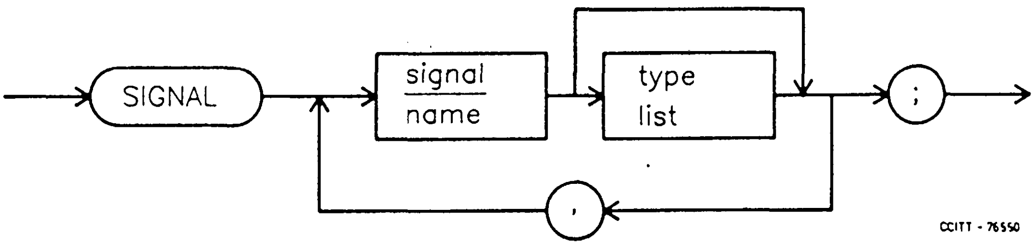
VARIABLE LIST



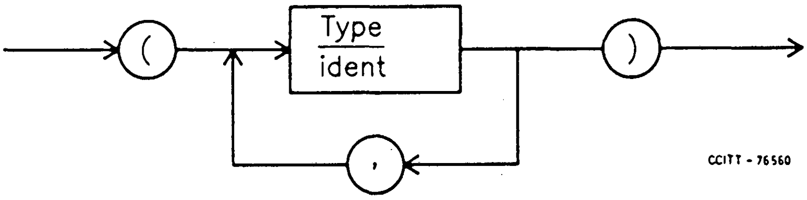
SIGNAL LIST



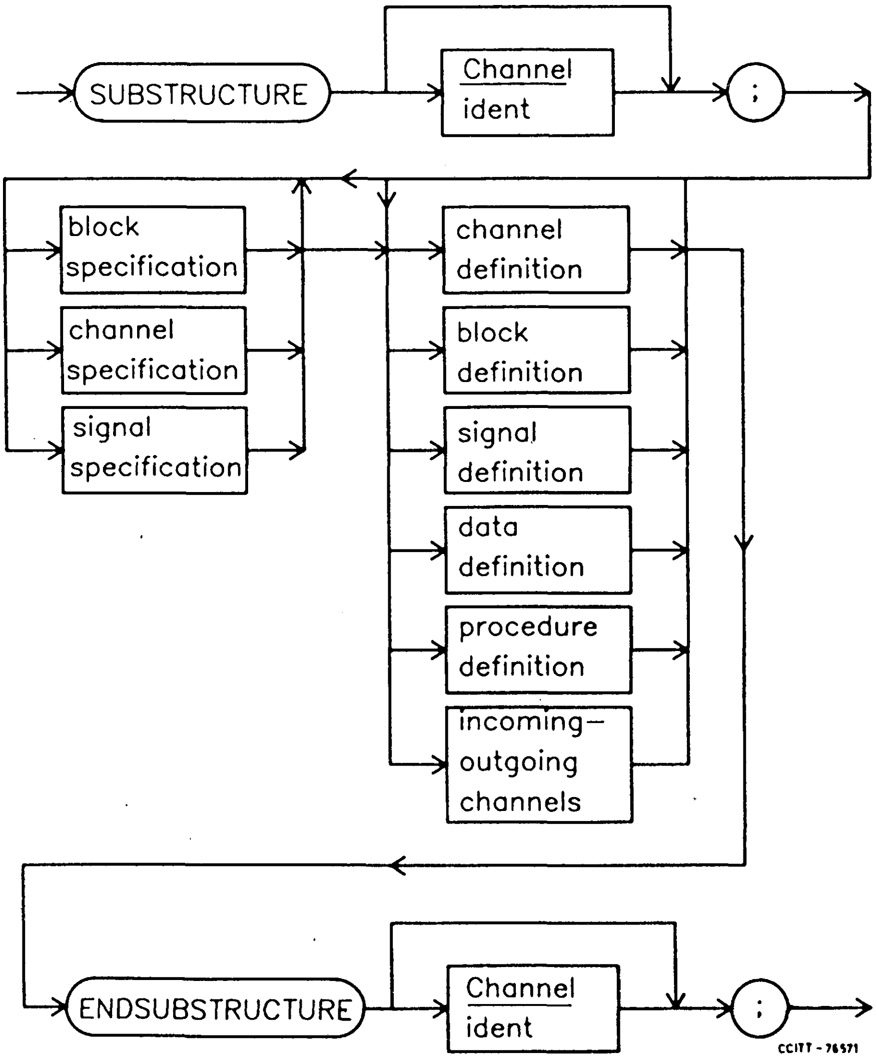
SIGNAL DEFINITION



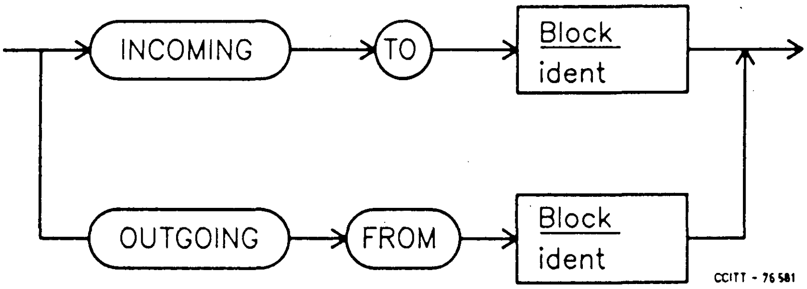
TYPE LIST



CHANNEL SUBSTRUCTURE DEFINITION

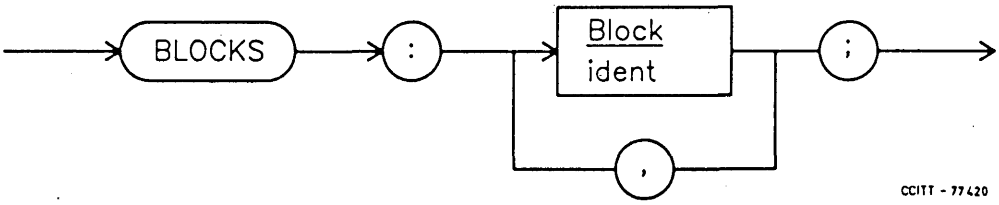


INCOMING–OUTGOING CHANNELS



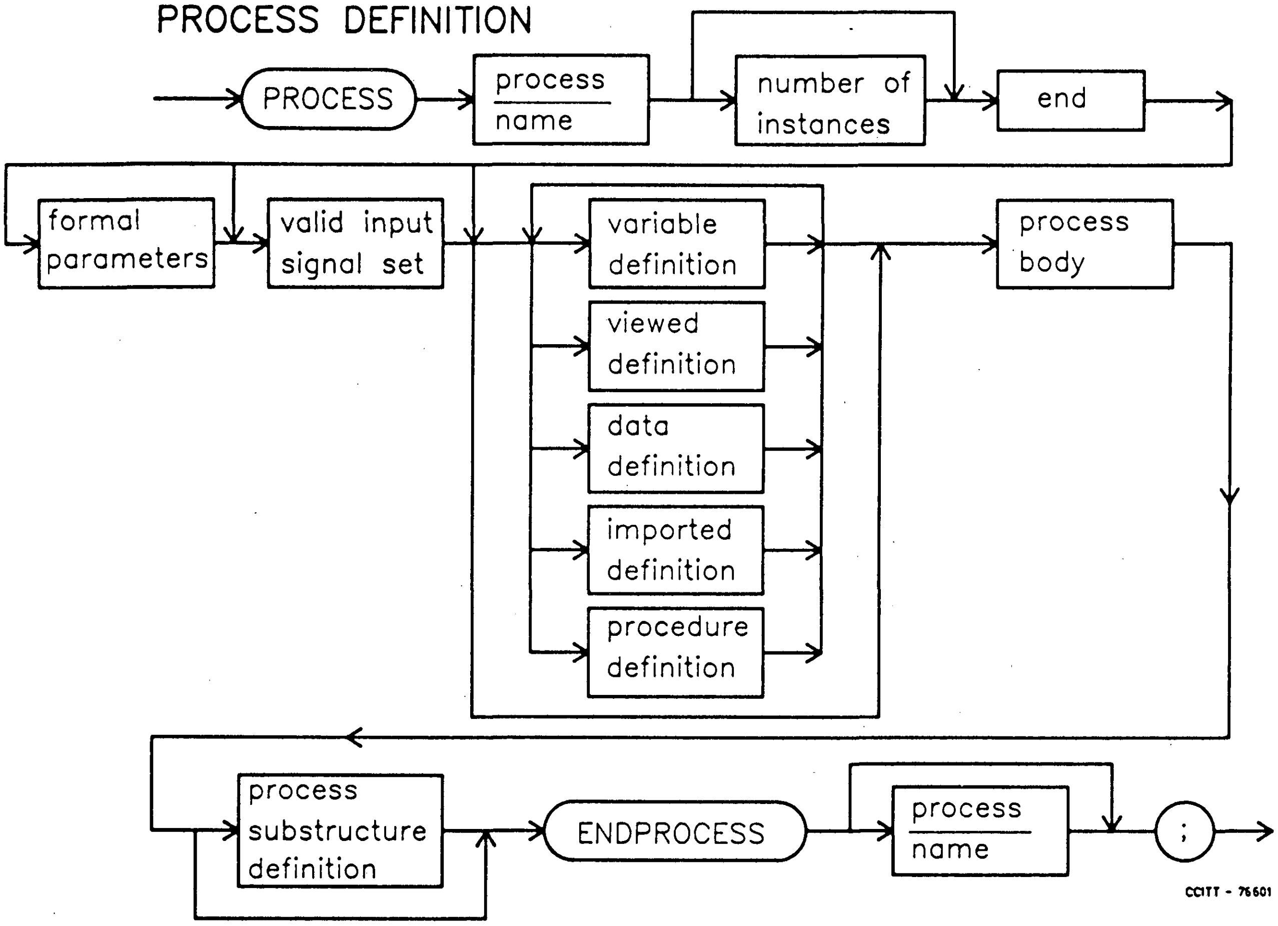
CCITT - 76581

BLOCK SPECIFICATION



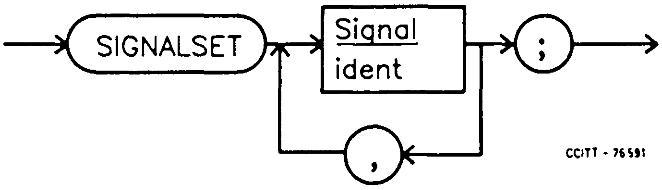
CCITT - 77420

PROCESS DEFINITION

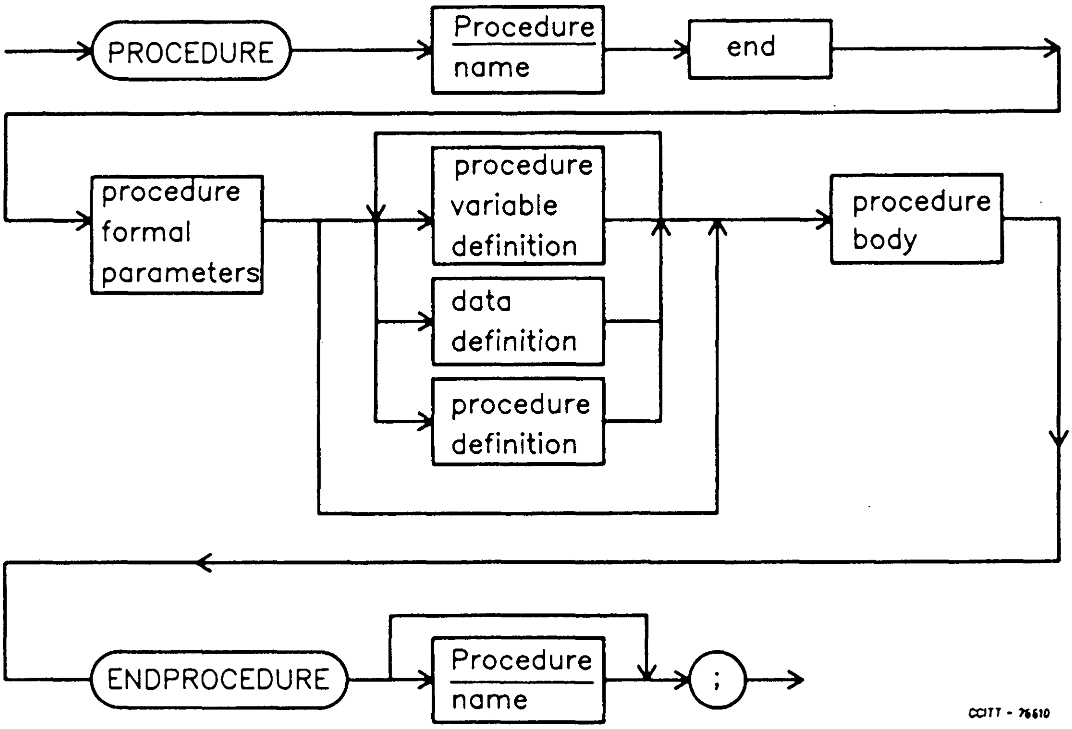


CCITT - 76601

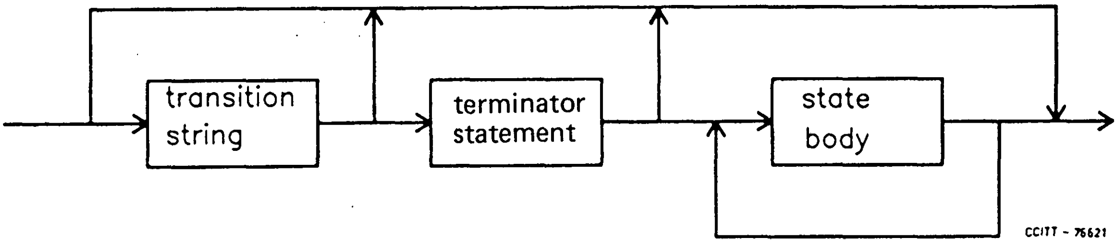
VALID INPUT SIGNAL SET



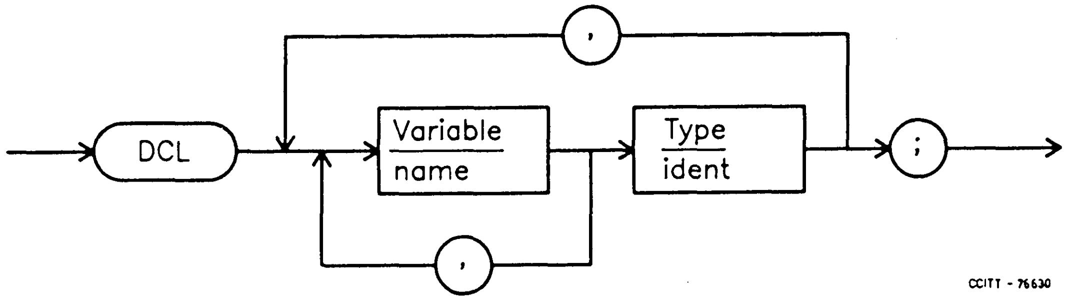
PROCEDURE DEFINITION



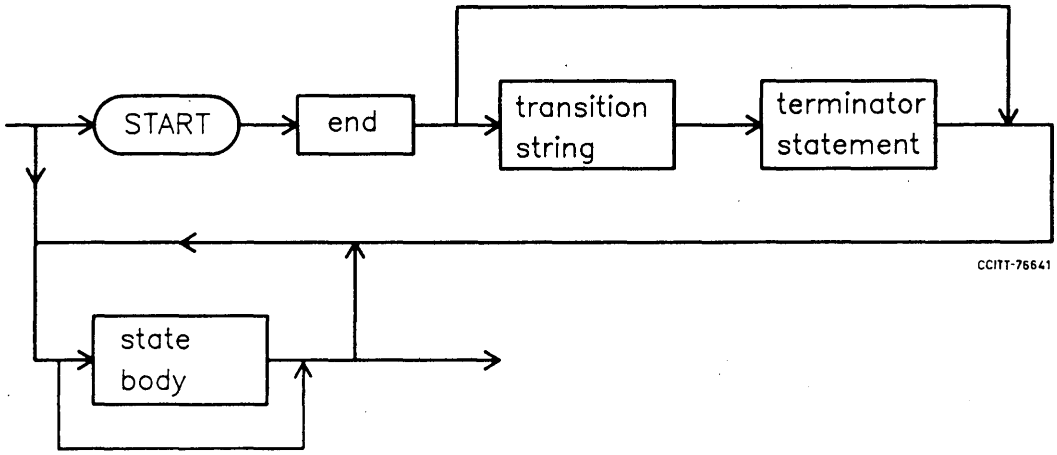
PROCEDURE BODY



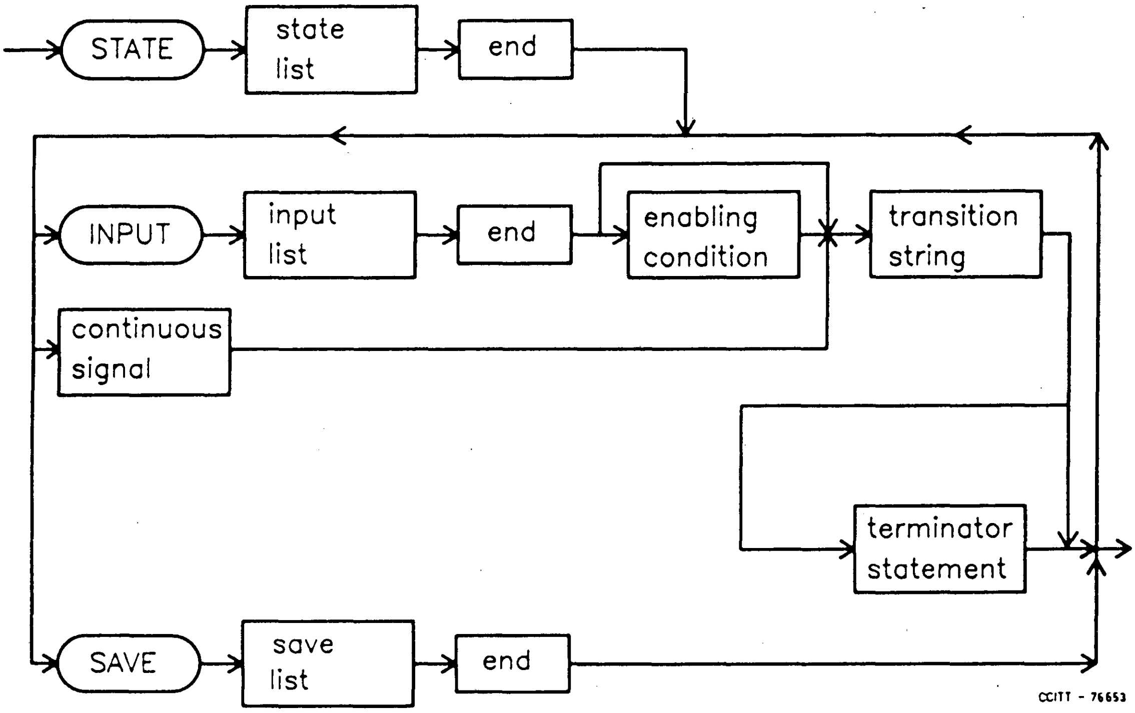
PROCEDURE VARIABLE DEFINITION



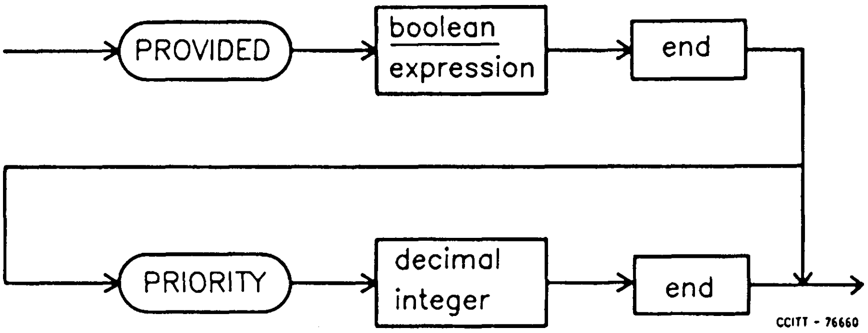
PROCESS BODY



STATE BODY

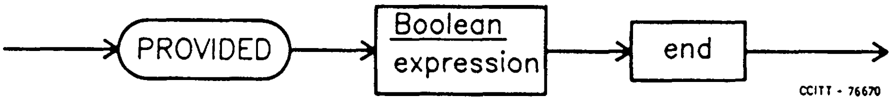


CONTINUOUS SIGNAL



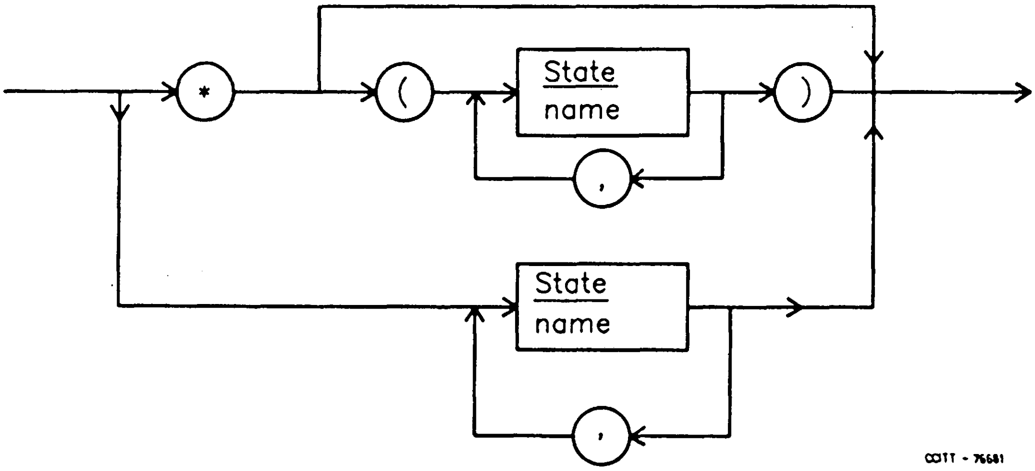
CCITT - 76660

ENABLING CONDITION



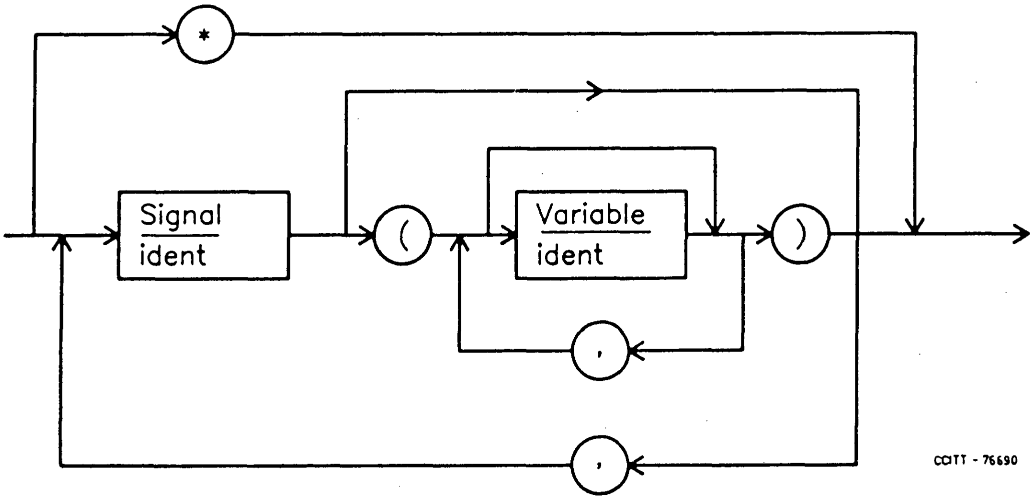
CCITT - 76670

STATE LIST



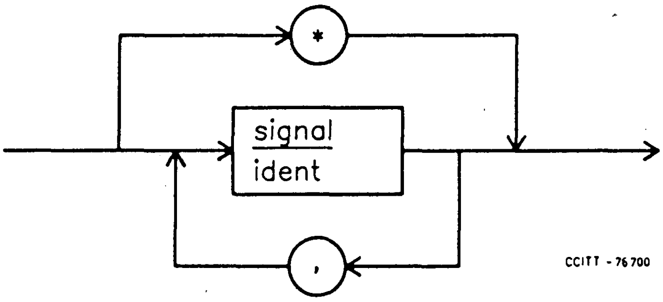
CCITT - 76681

INPUT LIST



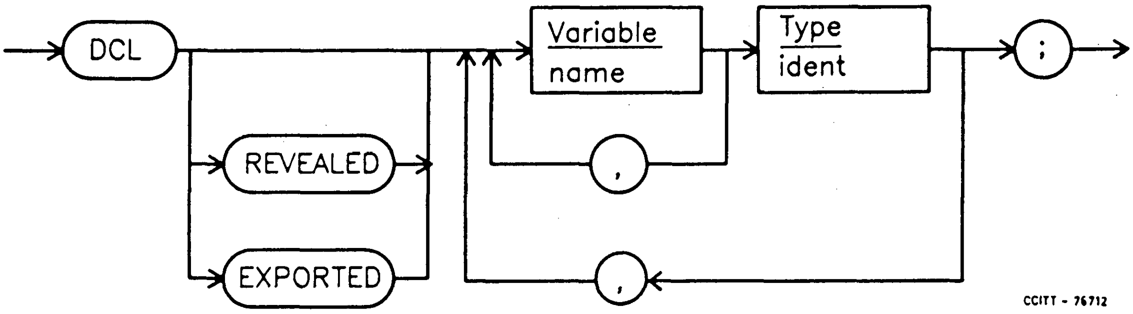
CCITT - 76690

SAVE LIST



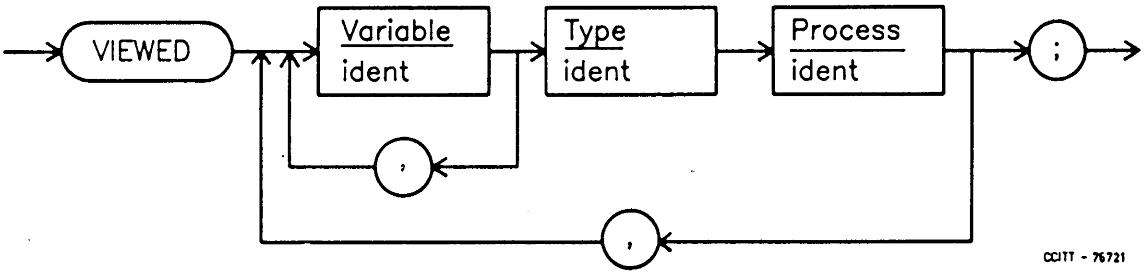
CCITT - 76700

VARIABLE DEFINITION



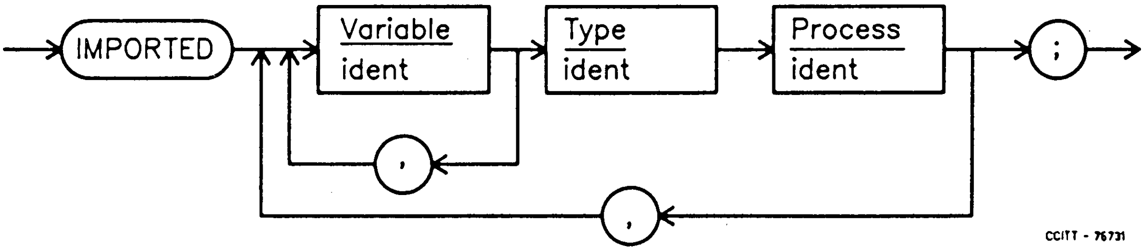
CCITT - 76712

VIEW DEFINITION



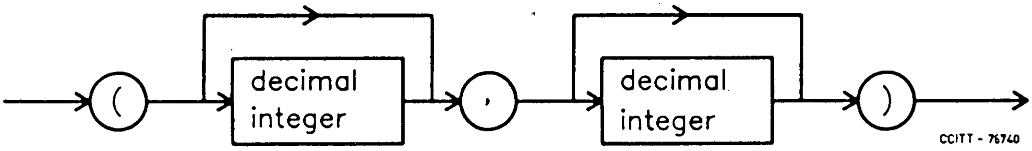
CCITT - 76721

IMPORT DEFINITION



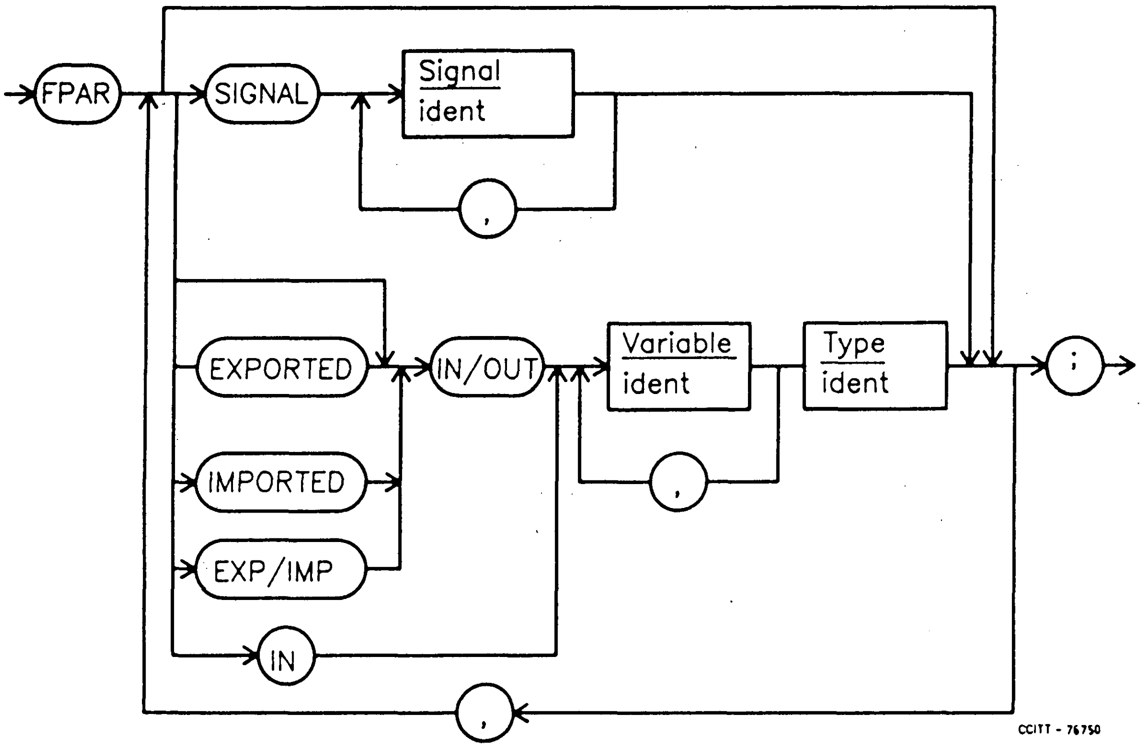
CCITT - 76731

NUMBER OF INSTANCES

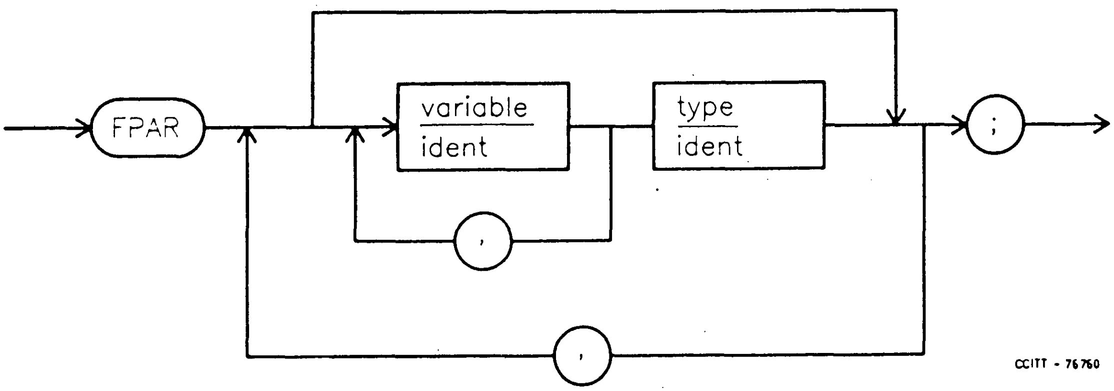


CCITT - 76740

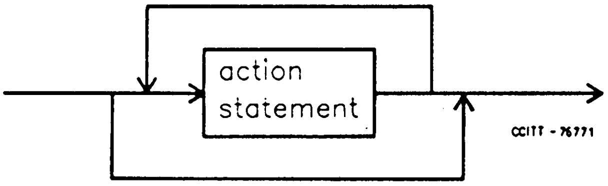
PROCEDURE FORMAL PARAMETERS



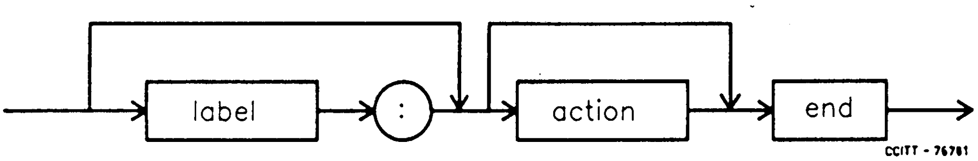
FORMAL PARAMETERS



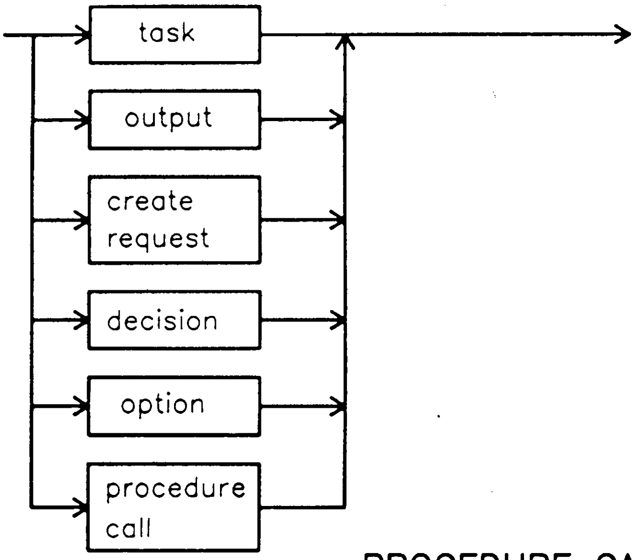
TRANSITION STRING



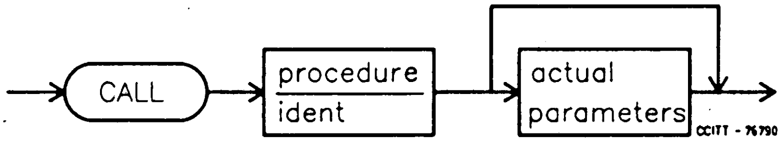
ACTION STATEMENT



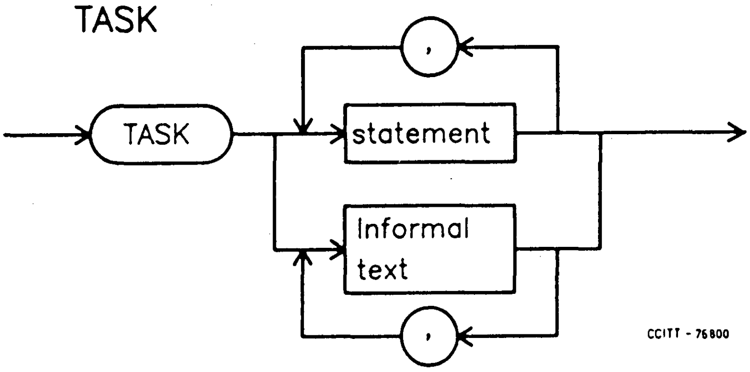
ACTION



PROCEDURE CALL

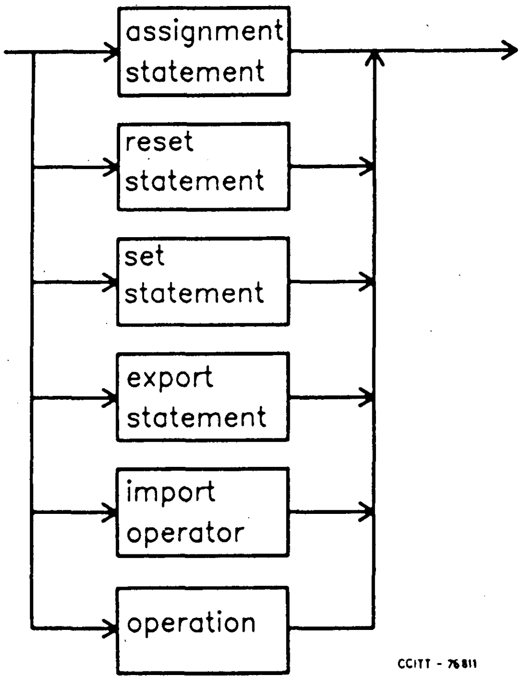


TASK



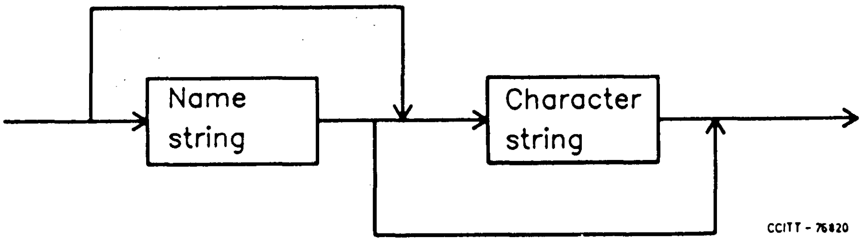
CCITT - 76800

STATEMENT



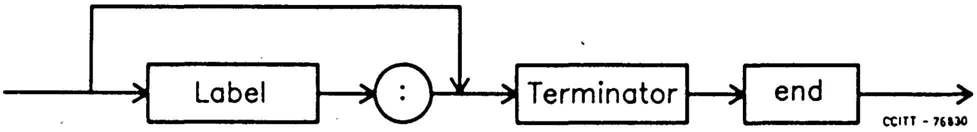
CCITT - 76811

INFORMAL TEXT

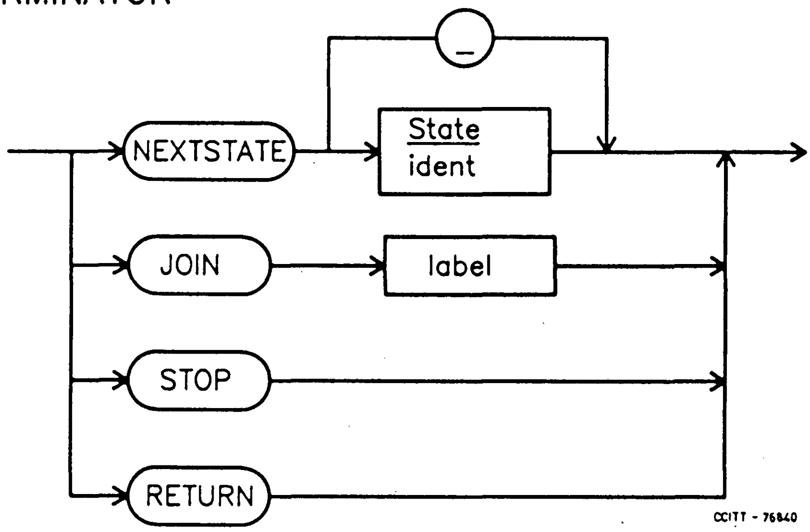


CCITT - 76820

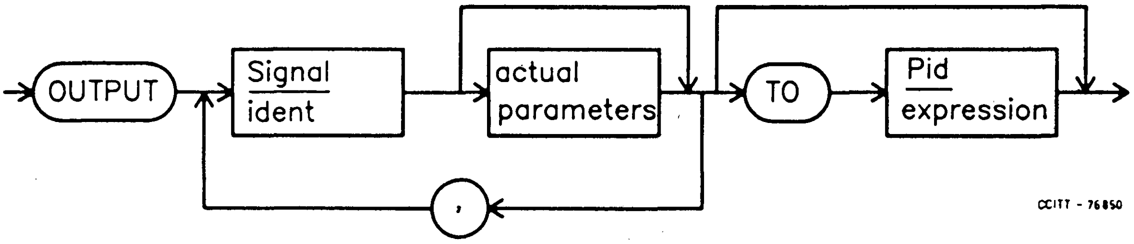
TERMINATOR STATEMENT



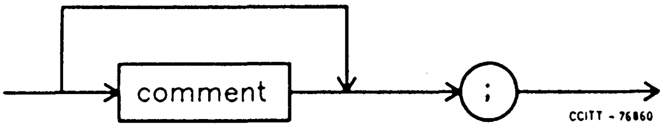
TERMINATOR



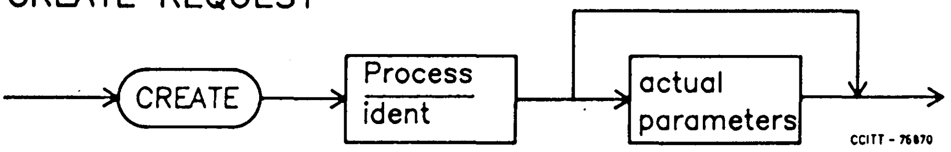
OUTPUT



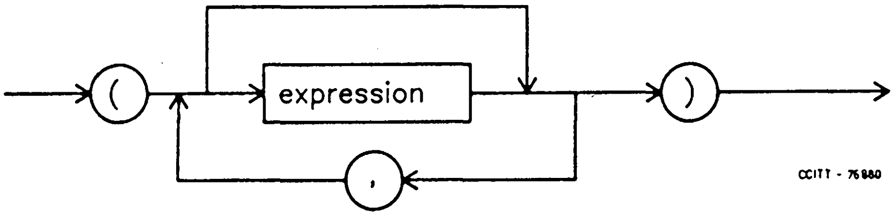
END



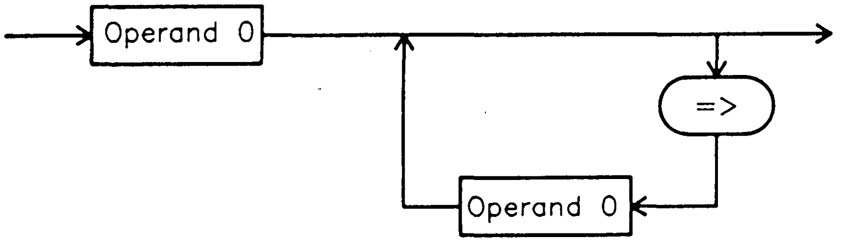
CREATE REQUEST



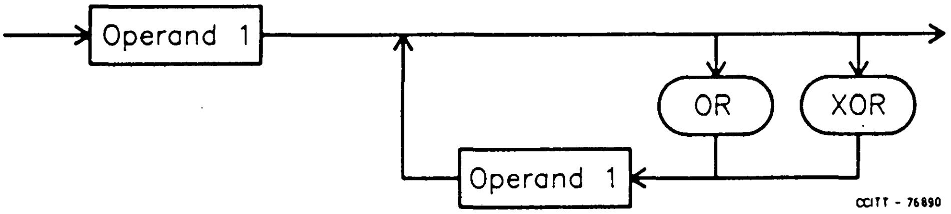
ACTUAL PARAMETERS



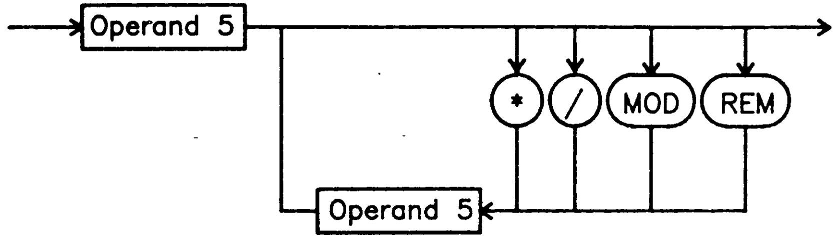
EXPRESSION



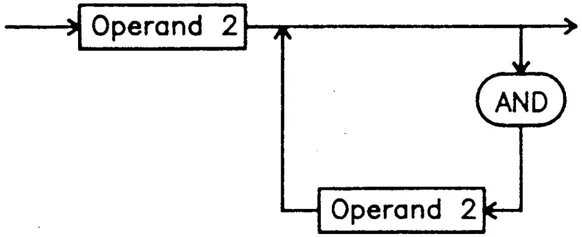
OPERAND 0



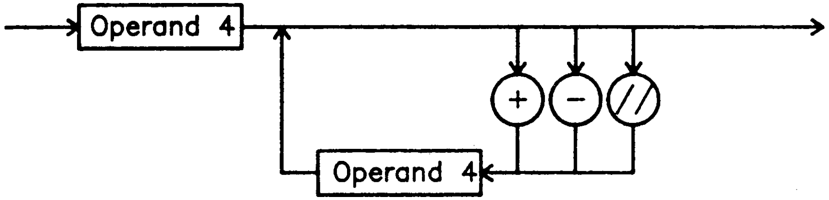
OPERAND 4



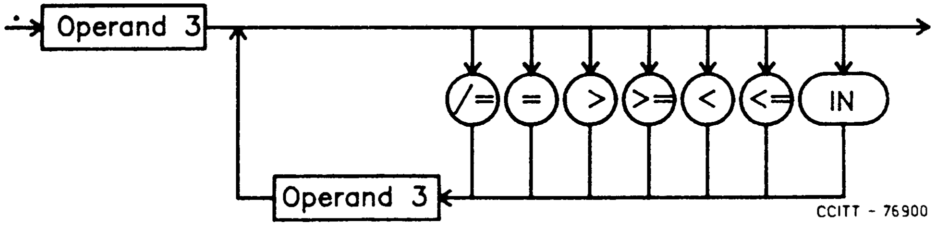
OPERAND 1



OPERAND 3

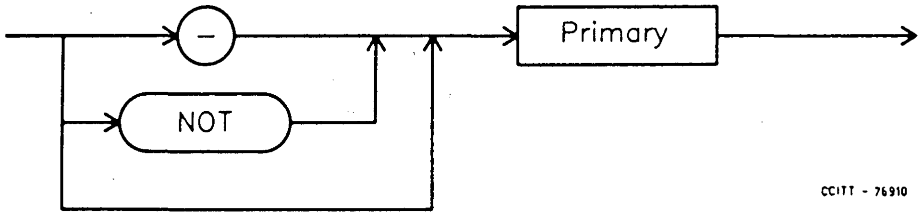


OPERAND 2



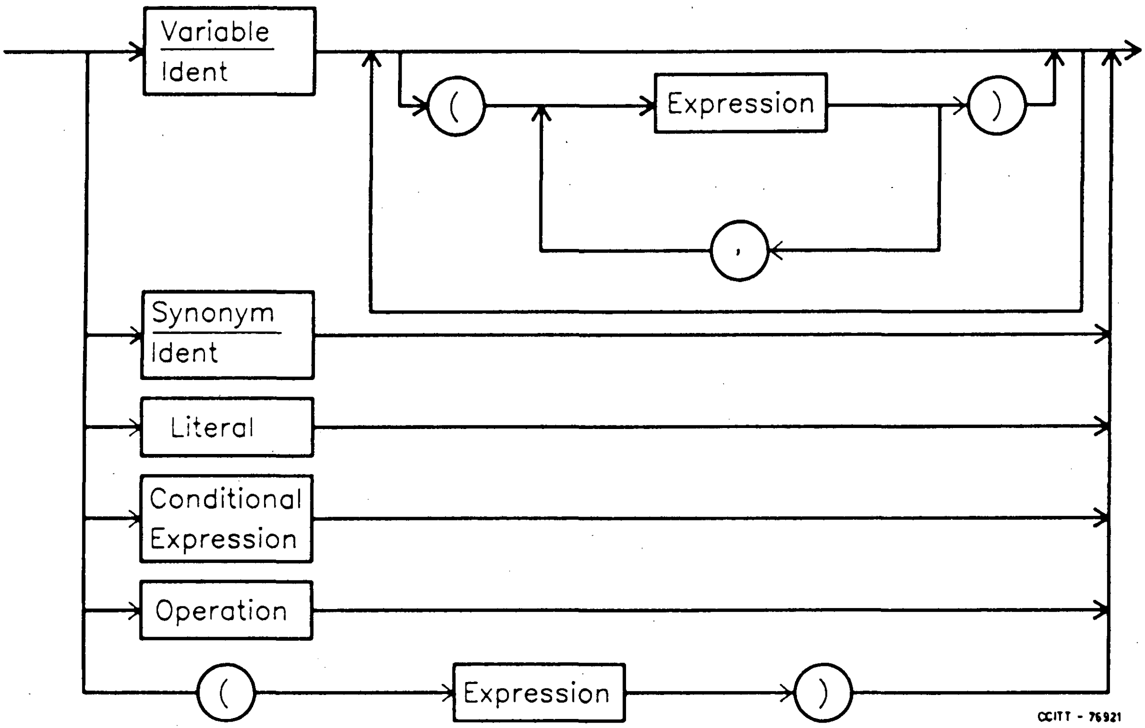
CCITT - 76900

OPERAND 5



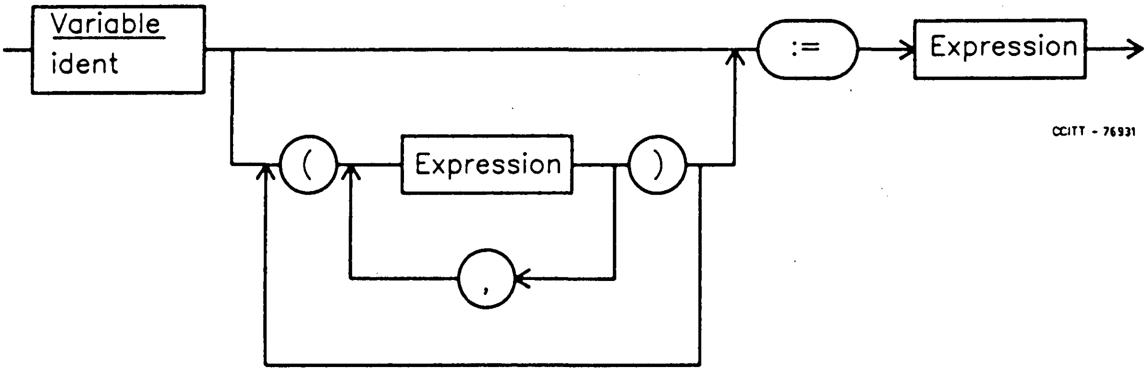
CCITT - 76910

PRIMARY



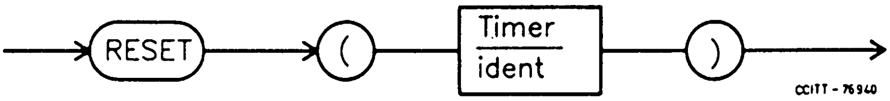
CCITT - 76921

ASSIGNMENT STATEMENT



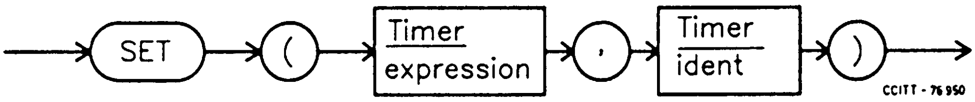
CCITT - 76931

RESET STATEMENT



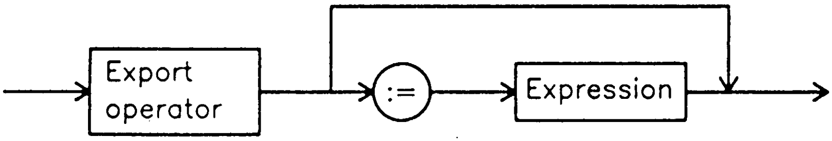
CCITT - 76940

SET STATEMENT

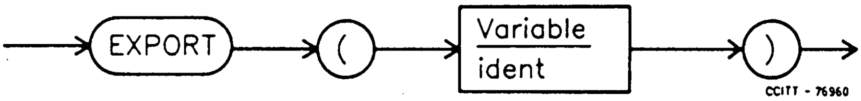


CCITT - 76950

EXPORT STATEMENT

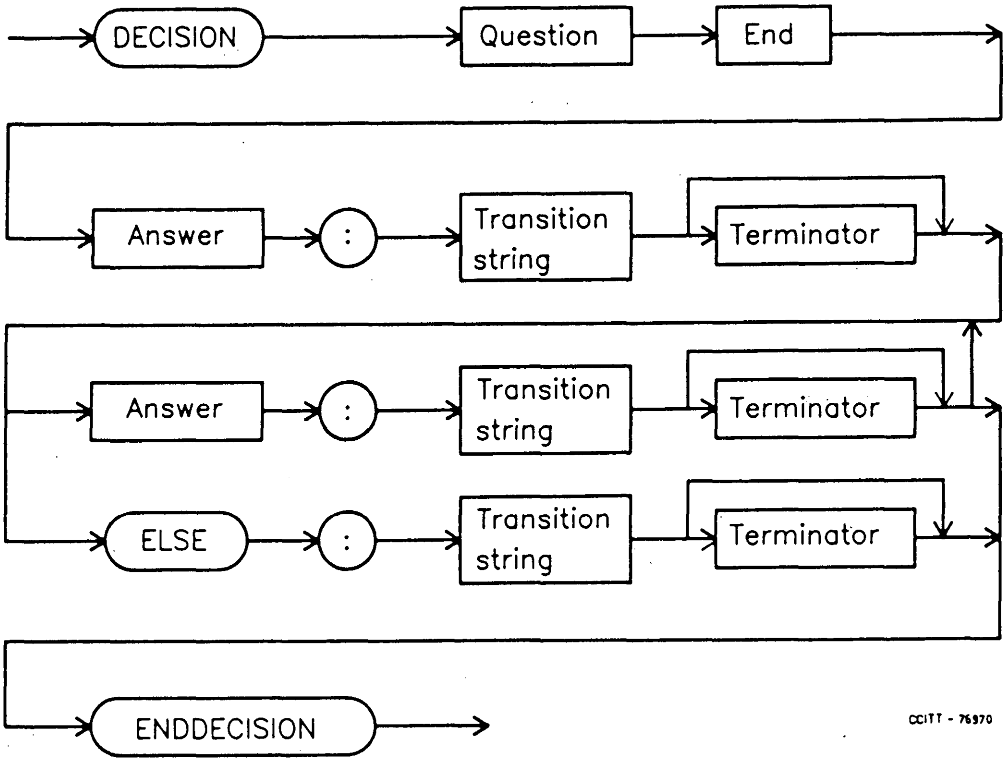


EXPORT OPERATOR

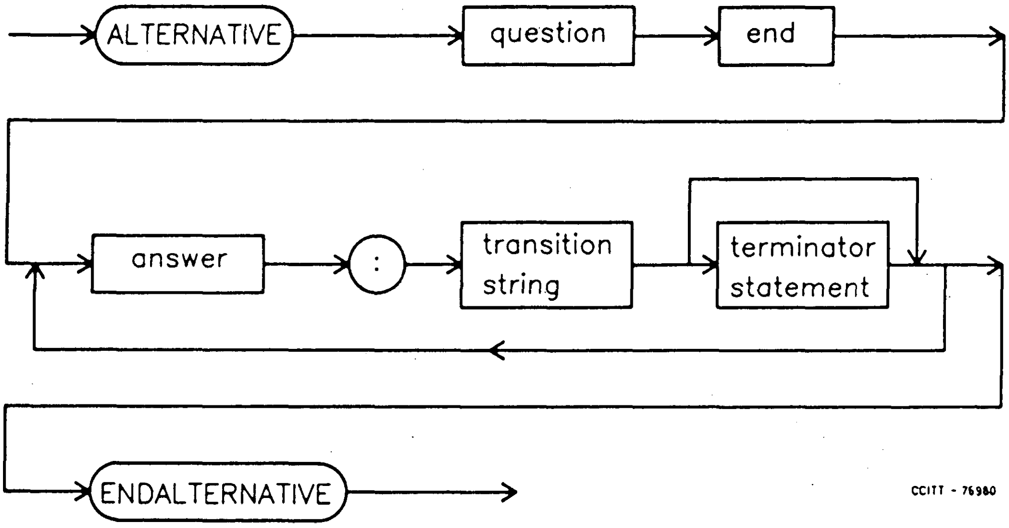


CCITT - 76960

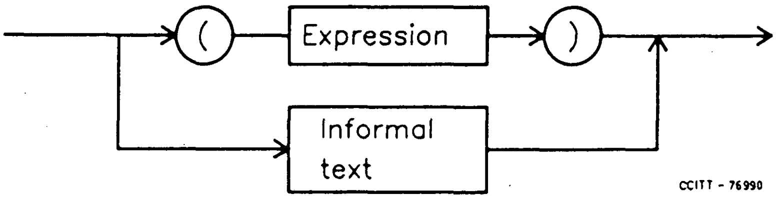
DECISION



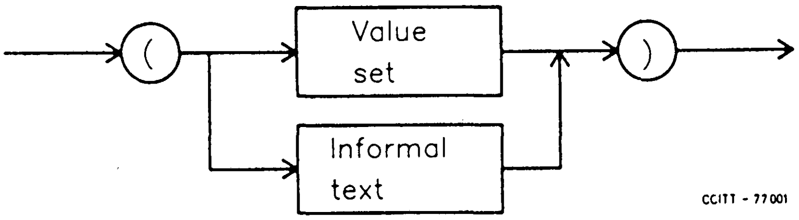
OPTION



QUESTION

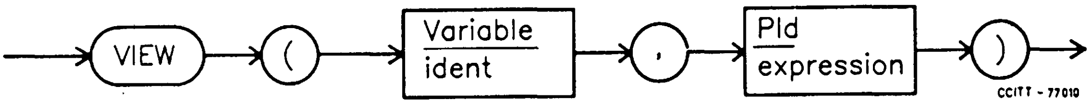


ANSWER



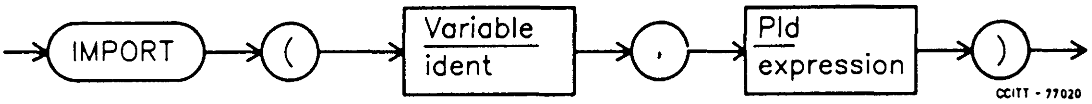
CCITT - 77001

VIEWING OPERATOR



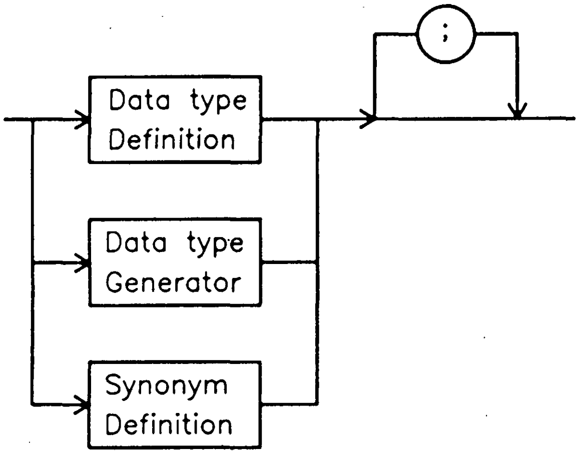
CCITT - 77010

IMPORT OPERATOR

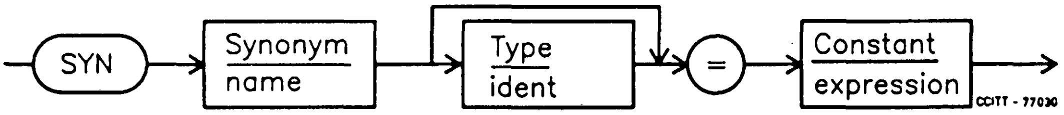


CCITT - 77020

DATA DEFINITION

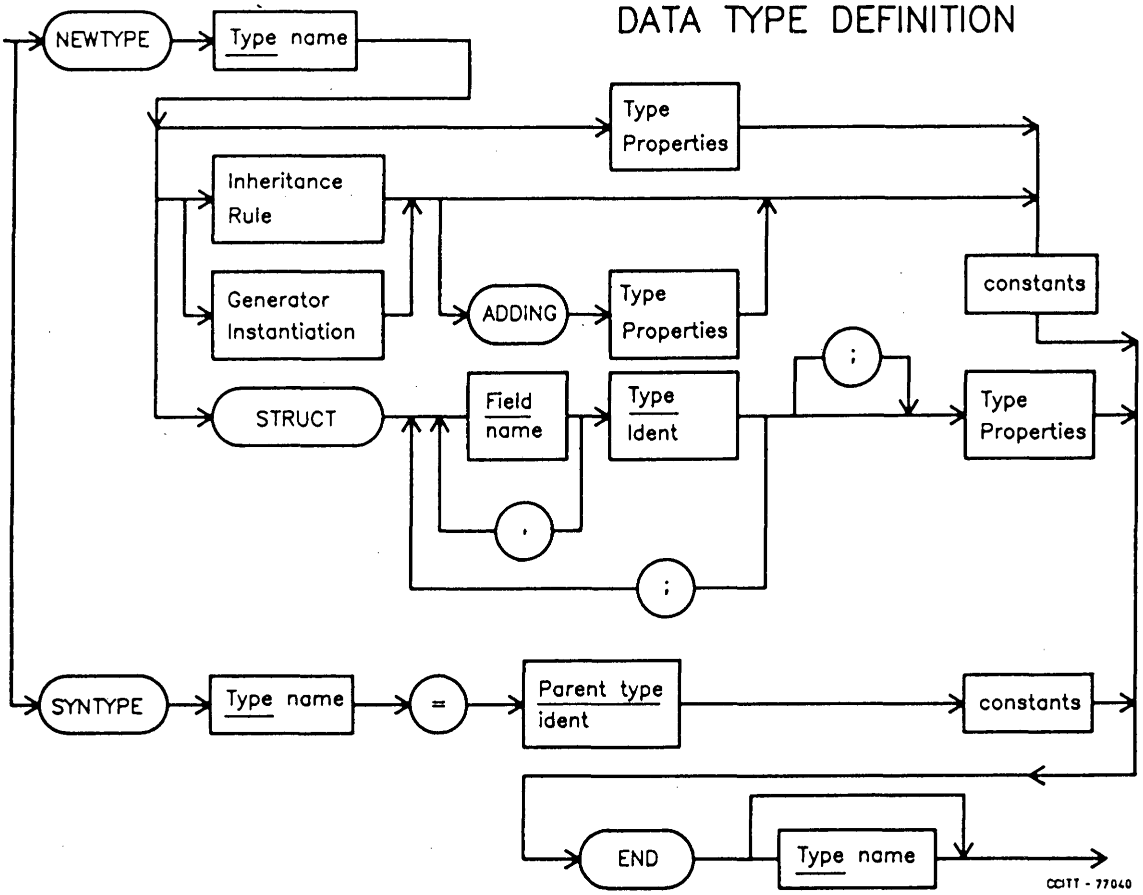


SYNONYM DEFINITION

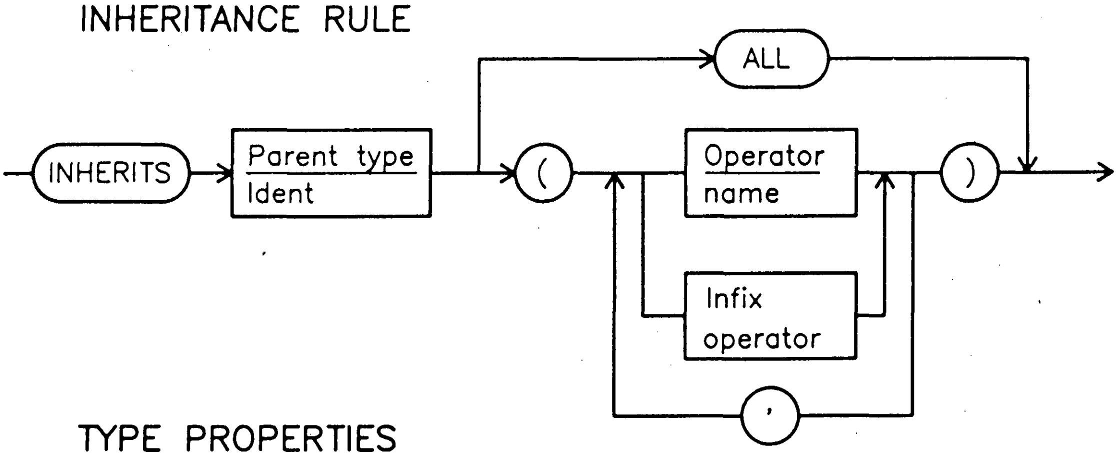


CCITT - 77030

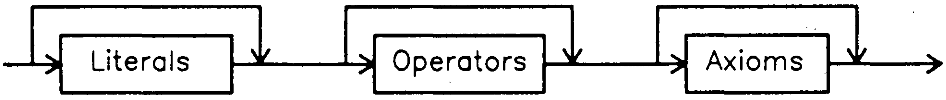
DATA TYPE DEFINITION



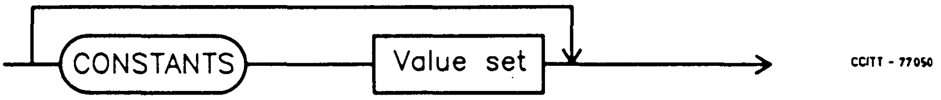
INHERITANCE RULE



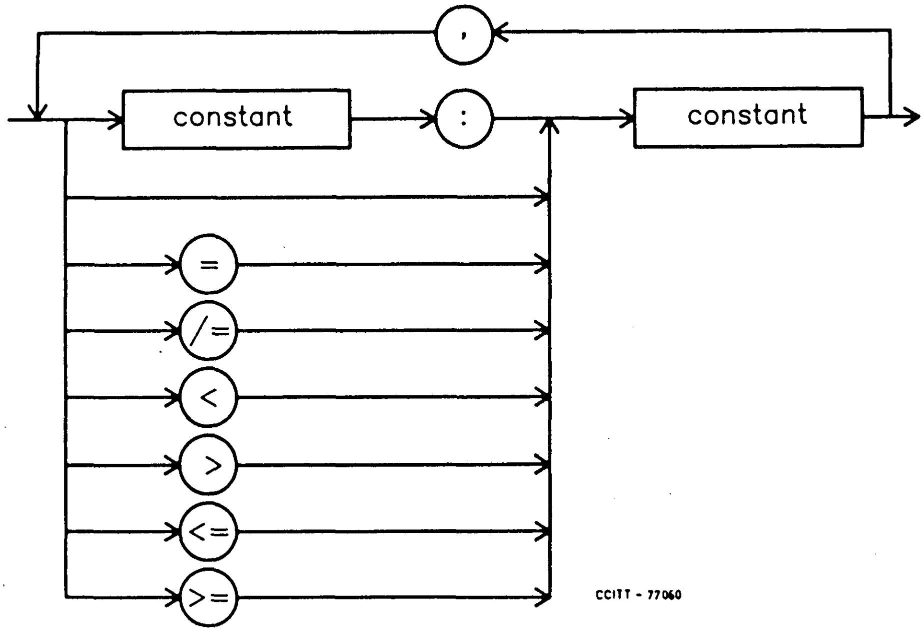
TYPE PROPERTIES



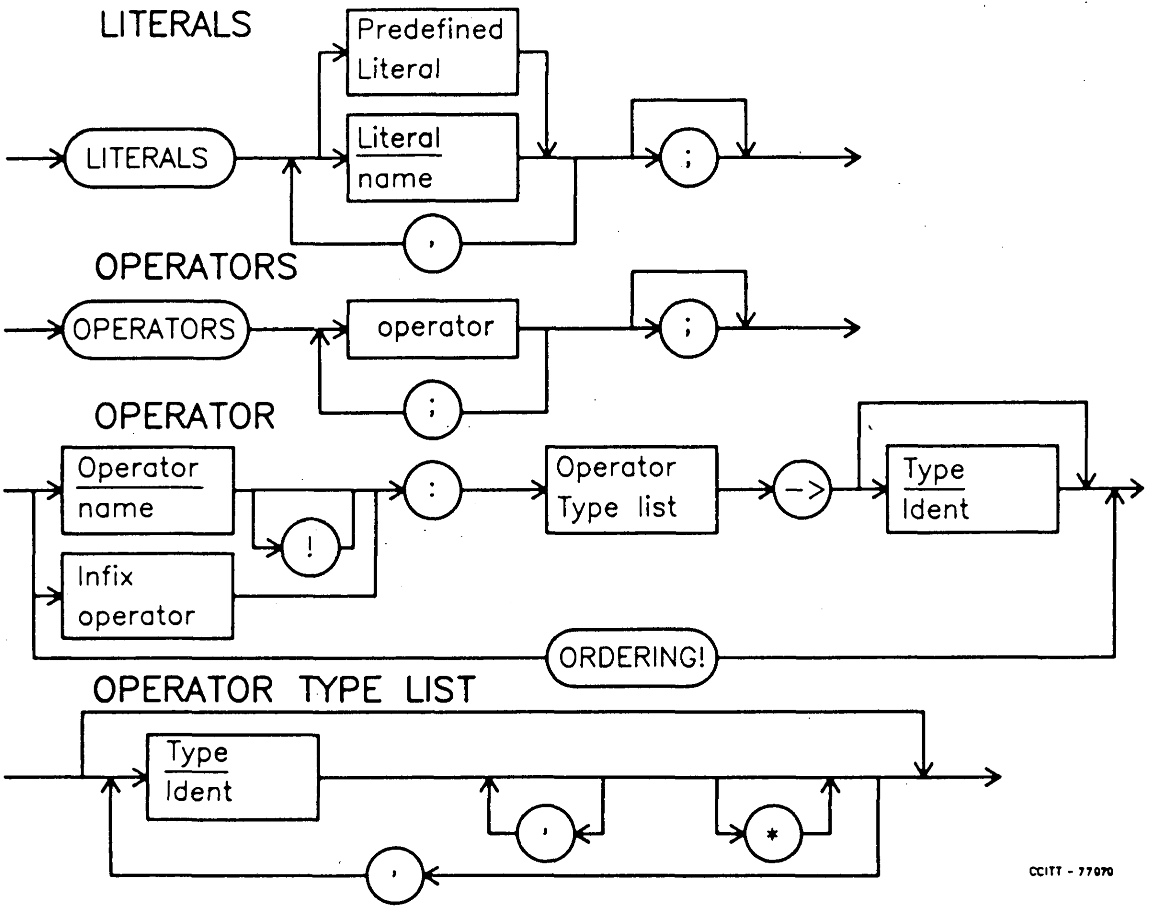
CONSTANTS



VALUE SET

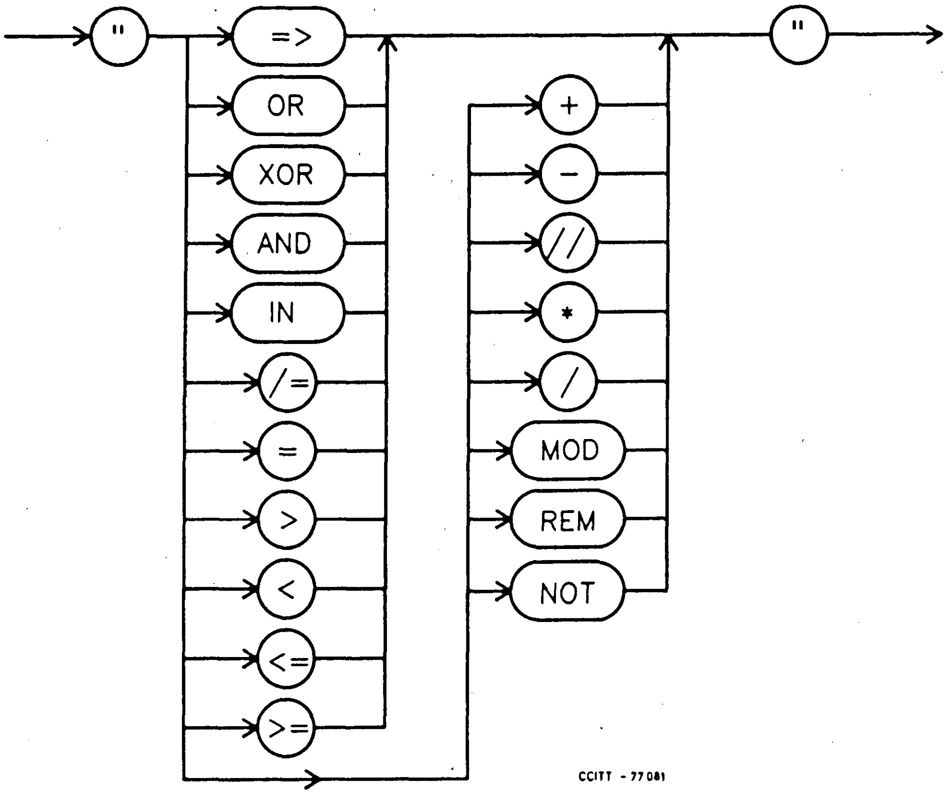


CCITT - 77060



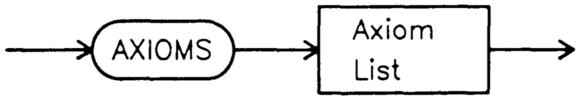
CCITT - 77070

INFIX OPERATOR

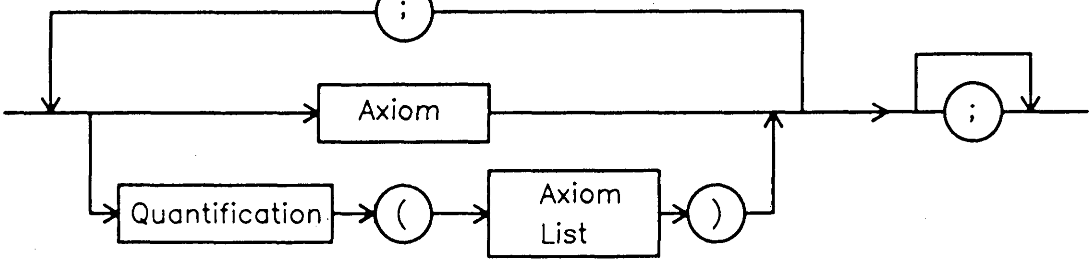


CCITT - 77081

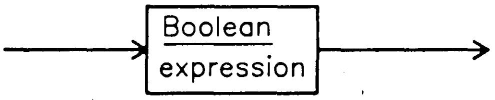
AXIOMS



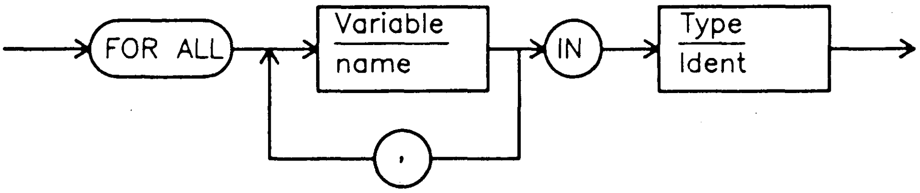
AXIOM LIST



AXIOM

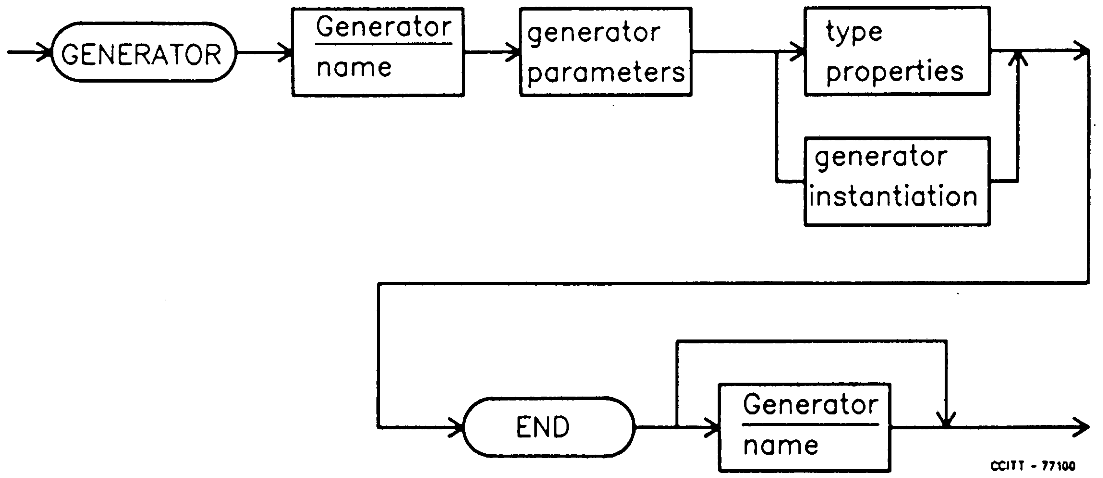


QUANTIFICATION

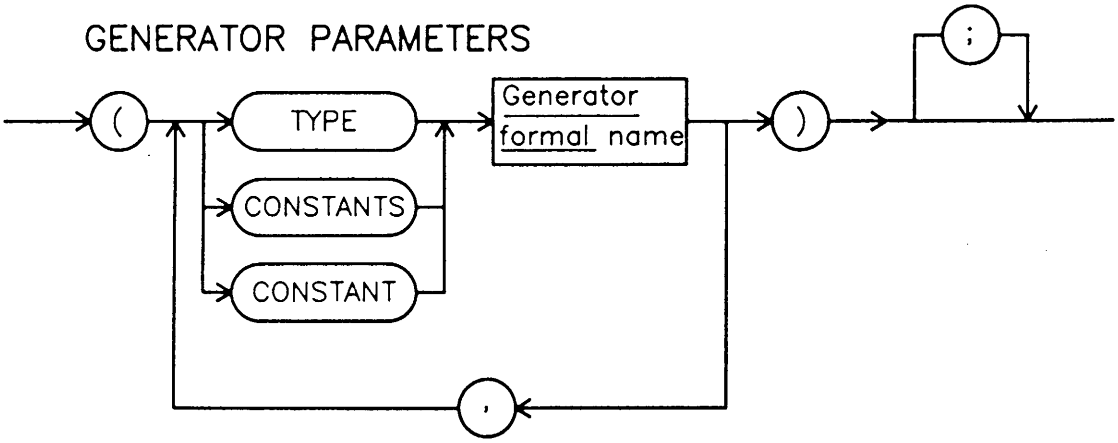


CCITT - 77090

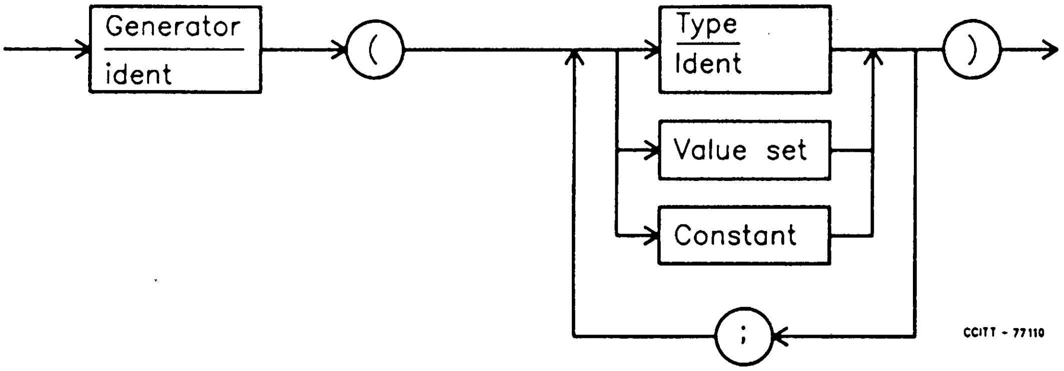
DATA TYPE GENERATOR



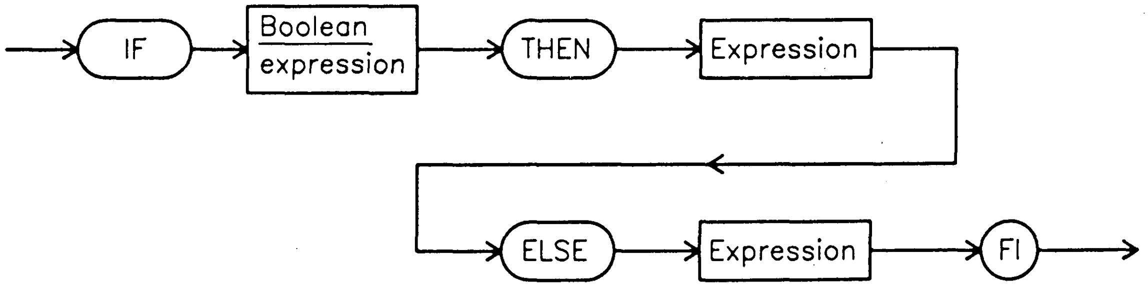
GENERATOR PARAMETERS



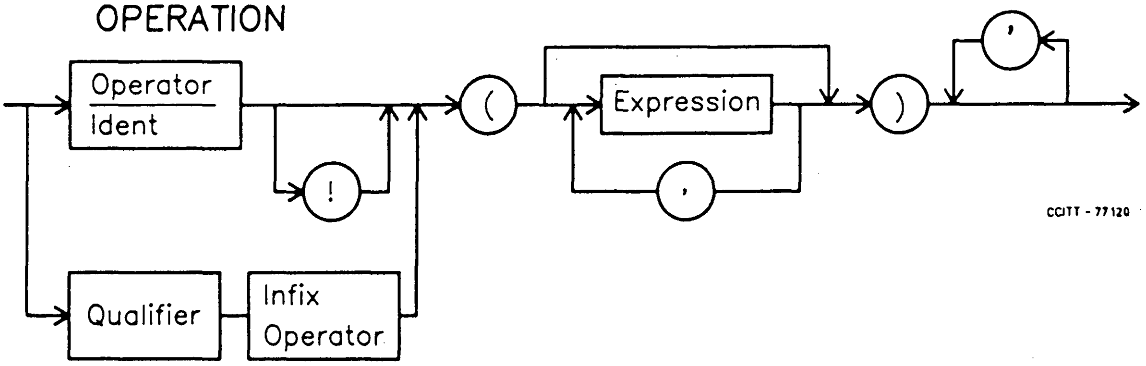
GENERATOR INSTANTIATION



CONDITIONAL EXPRESSION

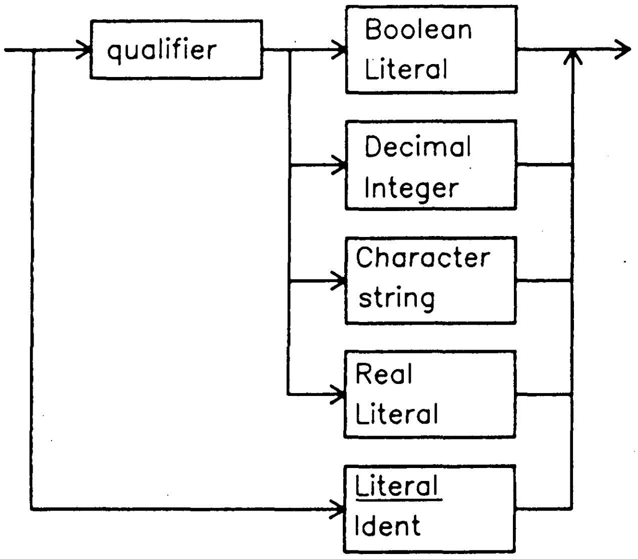


OPERATION

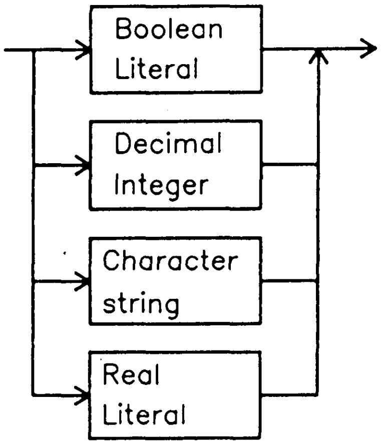


CCITT - 77120

LITERAL

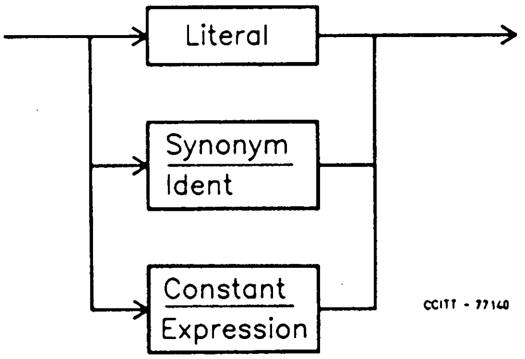


PREDEFINED LITERAL

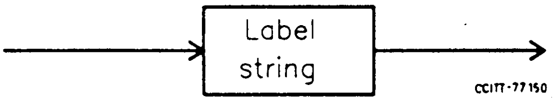


CCITT - 77130

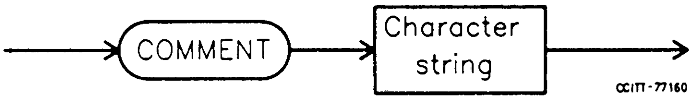
CONSTANT



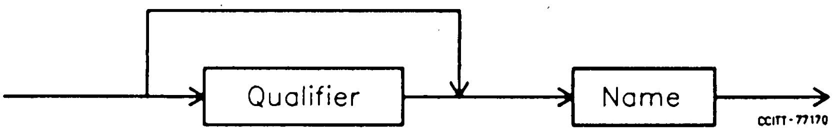
LABEL



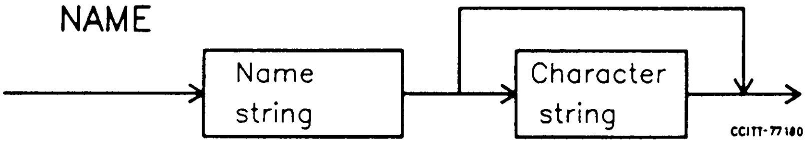
COMMENT



IDENT



NAME



C2.2 *Lexical rules*

1) All the punctuation marks (e.g. , . ; ’ : ! = ( )) and operation symbols (e.g. +, –, *, <, > . . .) are lexical units which may take the place of spaces.

2) Two lexical units must be separated by one or more spaces.

3) Keywords belong to the same lexical category as namestring, and they are reserved.

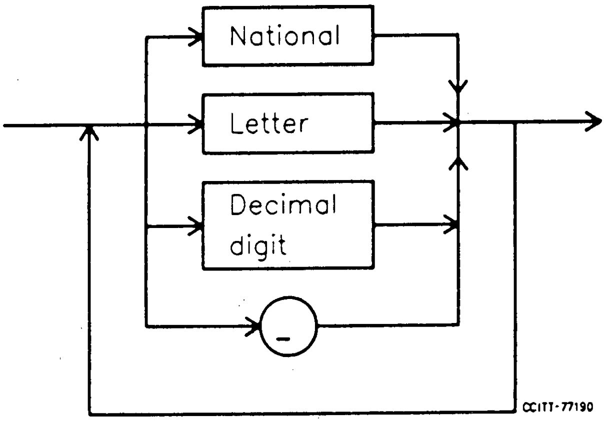
4) Outside lexical units several spaces have the same “meaning” as one space.

5) Tabulation characters (VT, HT, CR, BS . . .) may be considered as spaces.

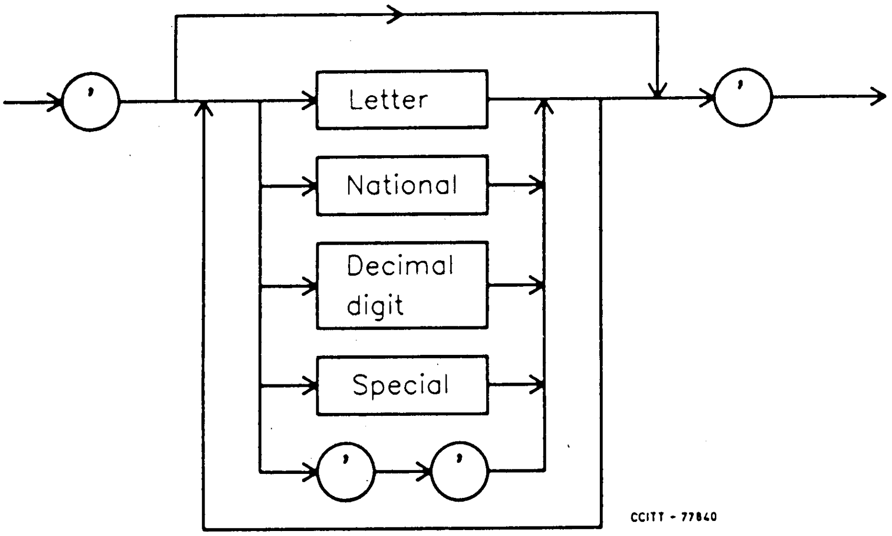
6) All letters and nationals are always interpreted as if uppercase, except within a charstring.

7) 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 '*/'.

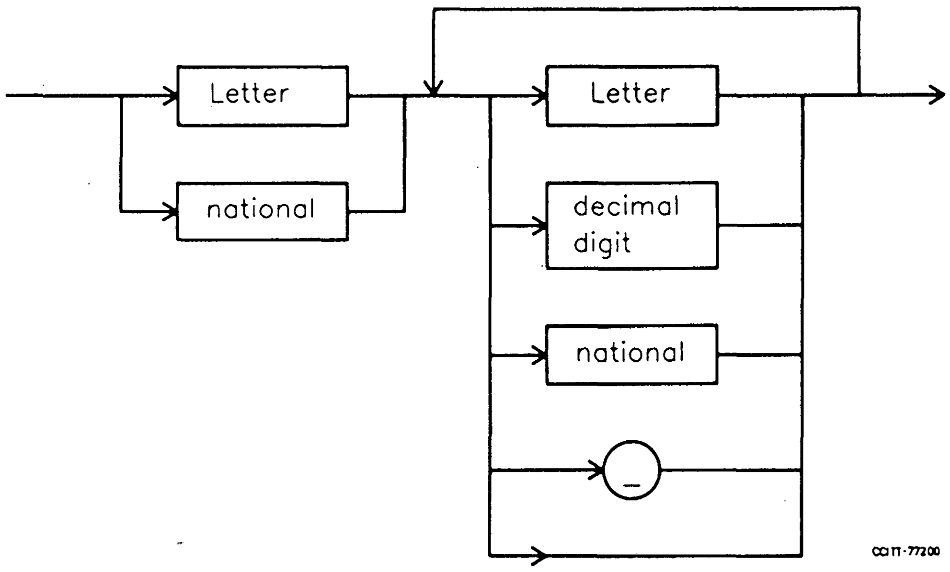
LABEL STRING



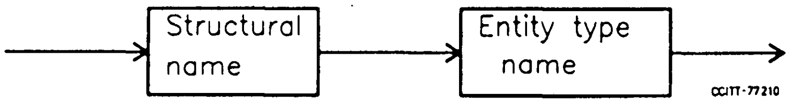
CHARACTER STRING



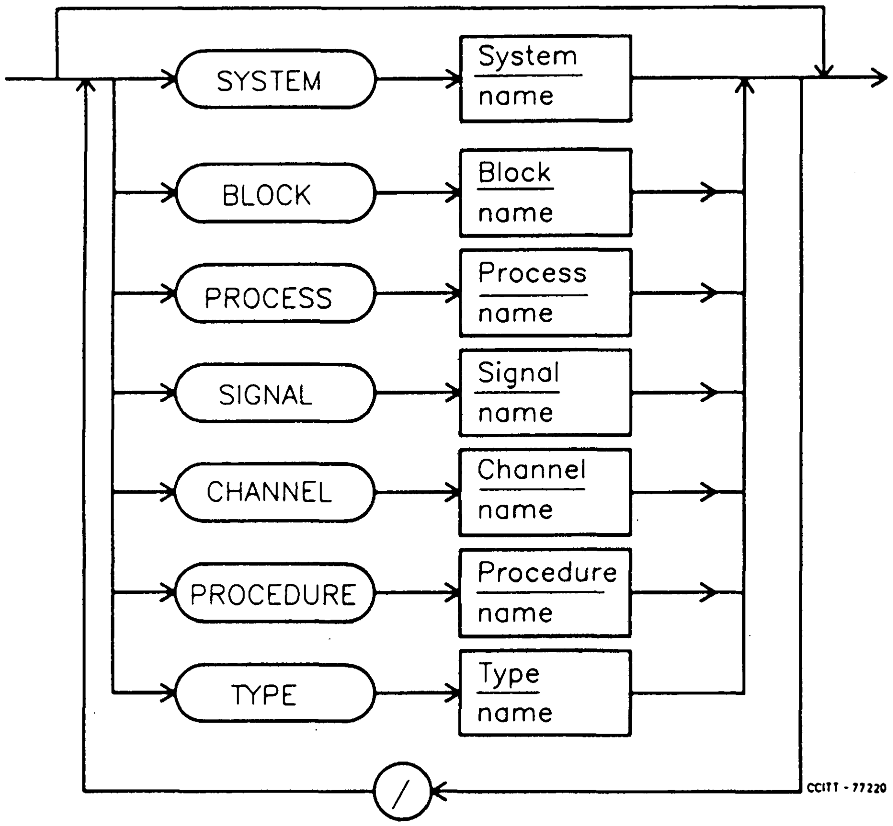
NAME STRING



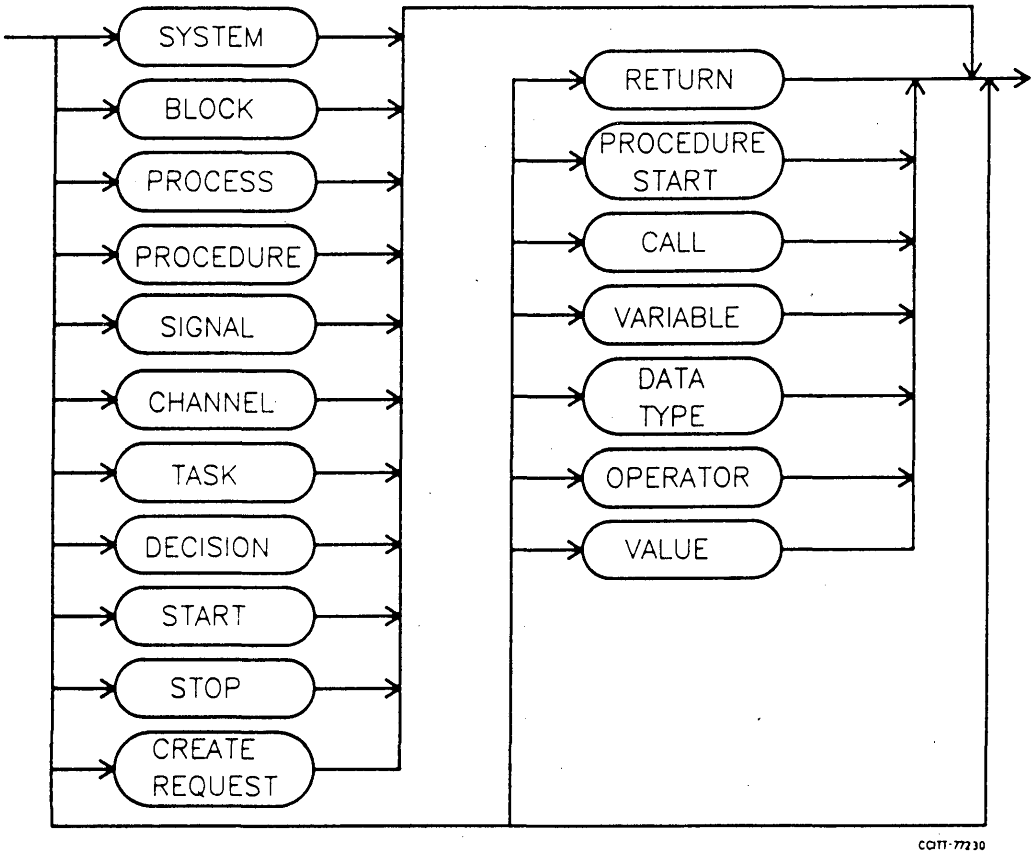
QUALIFIER



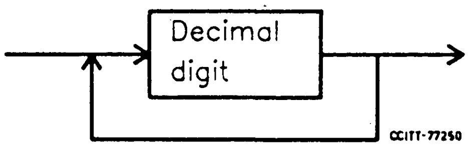
STRUCTURAL NAME



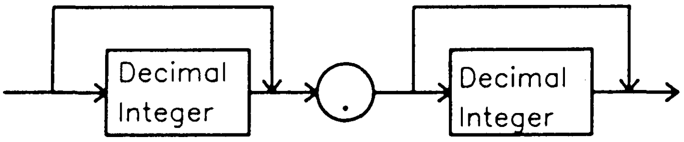
ENTITY TYPE NAME



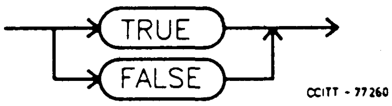
DECIMAL INTEGER



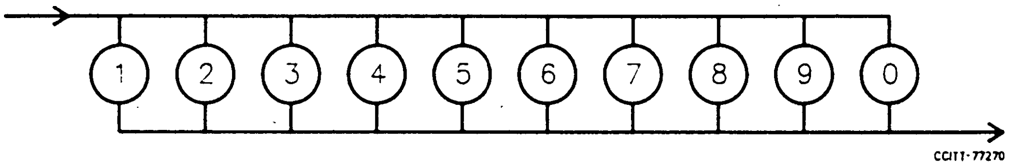
REAL LITERAL



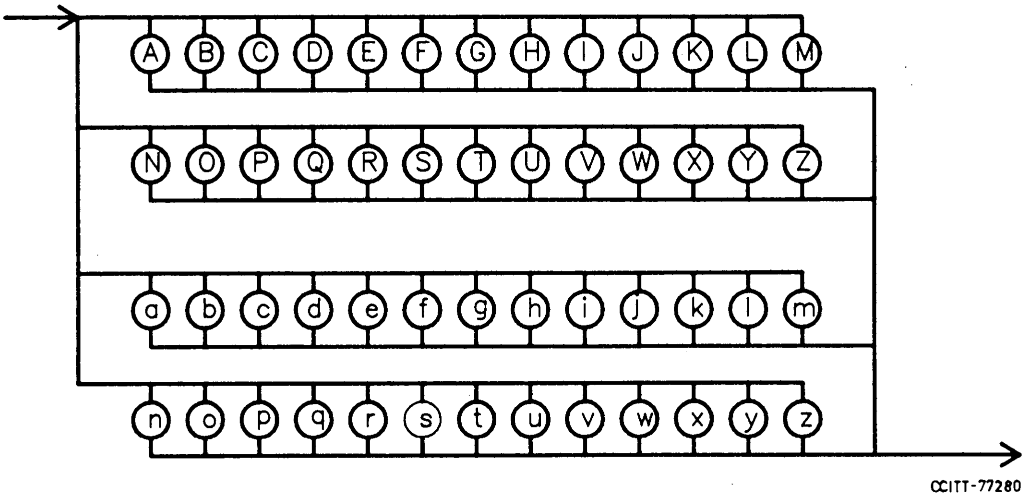
BOOLEAN LITERAL



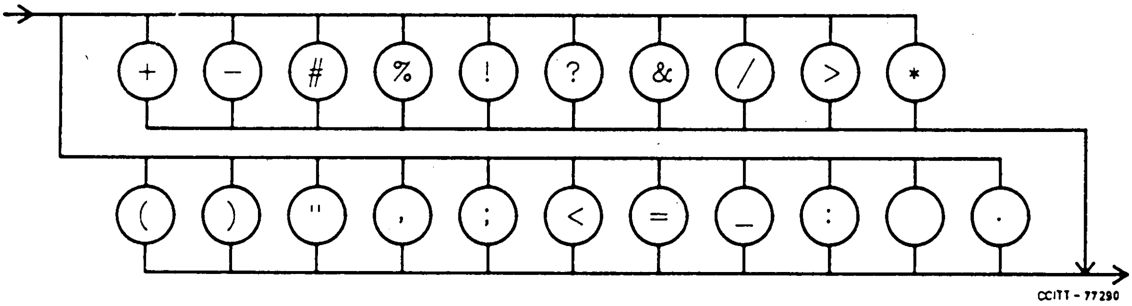
DECIMAL DIGIT



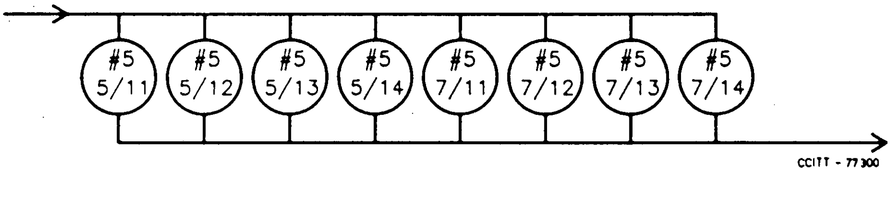
LETTER


A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m
n o p q r s t u v w x y z
CCITT-77280

SPECIAL

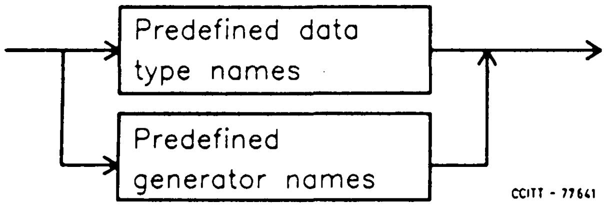


NATIONAL

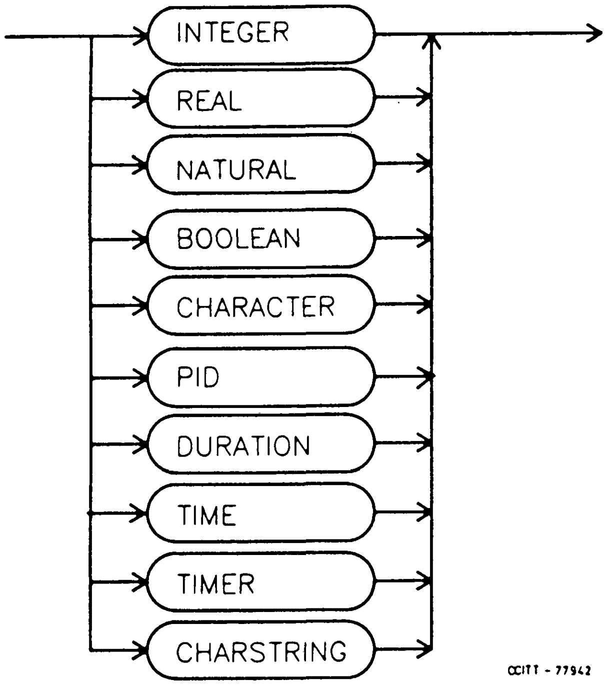


Note – The above referenced positions refer to the positions in the CCITT International Alphabet No. 5 reserved for national use.

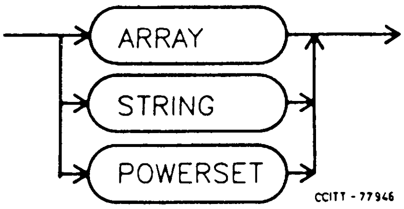
PREDEFINED TYPE NAME



PREDEFINED DATA TYPE NAMES



PREDEFINED GENERATOR NAMES



C2.3 *Reserved words used in PR*

SYSTEM
ENDSYSTEM
BLOCK
ENDBLOCK
SUBSTRUCTURE
ENDSUBSTRUCTURE
SPLIT
INTO
SUBBLOCKS
CHANNELS
SIGNALS
IN
CHANNEL
FROM
ENV
TO
WITH
IMPORTEDVALUES
SIGNAL
INCOMING
OUTGOING
BLOCKS
DECISION
ELSE
ENDDECISION
ALTERNATIVE
ENDALTERNATIVE
VIEW
EXPORT
IMPORT
CREATEREQUEST
CALL
DATATYPE
VALUE

PROCESS
ENDPROCESS
PROCEDURE
ENDPROCEDURE
DCL
START
STATE
INPUT
SAVE
PROVIDED
PRIORITY
REVEALED
EXPORTED
VIEWED
IMPORTED
FPAR
EXPORTED/IMPORTED
IN/OUT
IN
TASK
NEXTSTATE
JOIN
STOP
RETURN
OUTPUT
CREATE
COMMENT
RESET
SET
EXPORT
PROCEDURESTART
VARIABLE
OPERATOR

Note 1 – 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 macro name is not mandatory.

Note 2 – In the block definition diagram, the following rule is valid:

- if there is no block substructure definition, there must be at least one process definition (that can be explicitly defined in another module),
- if there is the block substructure definition, there is a process substructure definition for each of the process definitions contained in the block definition.

ANNEX D

(to Recommendations Z.100 to Z.104)

SDL User Guidelines

The present User Guidelines can be divided into three parts.

The first part consists of the table of contents (D.0), the preface (D.1), the introduction (D.2) and the application areas of SDL (D.3) giving information of SDL, contents and application areas.

The second part consists of a general explanation and the basic concepts of SDL (D.4) and an SDL structuring (D.5), giving information on how, and by what concepts, systems can be modelled in SDL. This is a general part which does not give details of the concrete forms of SDL.

The last part consists of guidelines for representing systems using SDL/GR (D.6), guidelines for representing systems using SDL/PR (D.7), mappings between SDL/GR, SDL/PR and CHILL (D.8) and application examples of SDL (D.9) giving guidance on the use of the two existing forms of SDL, i.e. SDL/GR and SDL/PR. Finally a section on tools for SDL is provided (D.10).

TABLE OF CONTENTS

Page

D.1 *Preface*

The CCITT Specification and Description Language, known as SDL, was first defined by Recommendations Z.101 to Z.103 in 1976 (the Orange Book, Volume VI.4), later extended in Recommendations Z.101 to Z.104 in 1980 (the Yellow Book) and further extended and reorganized into Recommendations Z.100 to Z.104 in 1984 (the Red Book, this fascicle).

It is evident to Study Group XI that user guidelines are needed to facilitate the use of the SDL in application to a wide range of telecommunications switching systems. The aim of the user guidelines is to assist users in the understanding of the SDL Recommendations and in their application to various areas. It is difficult to produce a comprehensive set of user guidelines because the area of application of SDL itself is still evolving. Consequently, the user guidelines in this fascicle will themselves require improvement and extension during the following study period 1984-1988, and contributions to the user guidelines based upon practical experience with SDL will be welcomed by the CCITT.

It is evident now, in 1984, that the SDL is being used more widely by the CCITT and its member organizations, and that the range of applications of SDL will continue to increase. These user guidelines are intended to assist people who are considering or starting to use SDL, by supplementing SDL Recommendations Z.100 to Z.104 with useful advice and helpful examples. It is realized that there will be some overlap between the user guidelines and the Recommendations; this is believed to be desirable, to make the user guidelines self-sufficient and readable. Nevertheless, it is the Recommendations which are the leading document.

D.2 *Introduction*

D.2.1 *General*

SDL can be used both in specification (to specify the required behaviour of a system) and in description (to describe the actual behaviour of a system). SDL has been designed to suit the specification and description of behaviour of telecommunications switching systems, but can also be used in other applications. As a matter of fact, SDL is well-suited to all systems whose behaviour can be effectively modelled by extended finite-state machines and where the focus is to be placed especially on interaction aspects.

SDL also can be the base for a methodology of documentation to represent completely a system specification or description. In this context the meaning of specification and description is related to their use in a system life cycle. Both define the functional properties of a system in an abstract way. The description will usually include some design dependent aspects (e.g. error handling) and will be more complete with respect to functional details. Both should be related to the concrete system design in a consistent manner. Thus both serve as specifications before the system is implemented and as documentation (descriptions) afterwards.

SDL may be used to represent, at various levels of detail, the functional properties of a system, or a function or facility, in either specifications or descriptions. The functional properties consist of some structural properties (block interaction diagrams) as well as behaviour. By "behaviour" is meant the way of reacting upon received signals (inputs), i.e. doing actions, e.g. sending signals (outputs), asking questions (decisions) and doing tasks.

Specifications may be very broad and general when an Administration wishes to explore the possibilities of updating a system with new features, new services, new technology, etc., while allowing the supplier to offer a wide range of design solutions. This type of specification is often not very detailed. The other extreme is a specification in which an Administration is requesting a replacement or addition to an existing exchange. This specification would probably have a greater level of detail, because of the very detailed specification of interfaces necessary.

A specification and a description might be identical. In any case it is preferable in new development that the design is derived from the specification so that compliance is ensured.

In general, descriptions are written by suppliers in response to a specification (although they can be written to describe systems that the supplier wishes to sell). A description will usually have more levels of detail than the specification because of the need to describe the detailed behaviour of the system.

It is to be noted also that SDL provides means of describing a system with various degrees of formality.

First, it is possible to describe a system using the SDL constructs with associated natural language. The resulting description conveys information only to a reader with knowledge of the context but not to a machine. Very limited checks can be performed automatically.

Second, it is possible to associate with the SDL constructs formal statements consisting of elements of defined types and operators on these elements. The properties of these elements are not specified: an example is "Connect A-B" where A and B are of type subscriber and connect is an operation allowed for that type. The resulting description conveys information to readers who know the meaning of the operators used. A machine can understand the description to a certain level and can perform checks on it, but it cannot perform complete checks nor "implement" the system because the properties of the operators are unknown.

Third, it is possible to also provide all the properties of all the operators. In this latter case the description is completely formal and a machine can perform all the checks and conceptually implement the systems described.

Depending on the goal, the descriptions can be tailored to the user needs using these different levels of formalism. Of course, the more formal the description the more cumbersome it is to read by a human being.

D.2.2 *Syntax forms of SDL*

SDL as defined by Recommendations Z.100 to Z.104 is a unique language which has two different syntaxes, both based on the same semantic model. One is called SDL/GR (SDL *G*raphical *R*epresentation), being based on a set of standardized graphical symbols and rules. The other is called SDL/PR (SDL textual *P*hrase *R*epresentation), being based on program-like statements. Both represent the same SDL concepts.

D.2.3 *SDL as based on an Extended Finite-State Machine model*

In the application of SDL, the system to be specified is represented by a number of interconnected abstract machines. A complete specification requires:

1) the definition of the structure of the system in terms of the machines and their interconnections;
2) the dynamic behaviour of each machine in terms of its interactions with the other machines and the environment; and
3) the operations on the data associated with the interactions.

The dynamic behaviour is described with the aid of models which define the mechanisms for the operation of the abstract machines and the communication between machines. The abstract machine used in SDL is an extension of the deterministic Finite-State Machine (FSM). The FSM has a finite internal state memory and operates with a discrete and finite set of inputs and outputs. For each combination of input and state, the memory defines an output and the next state. Transitions from one state to another are usually regarded as taking zero time.

A limitation of the FSM is that all information which needs storage must be represented as explicit states. Although it is possible to represent most systems in this way, it will not always be practical. There may be many values to remember which are significant for the future sequencing but do not contribute greatly to the overall understanding of the system. This information should not be a part of the explicit state space as it will clutter the presentation. For such applications, the FSM may be extended with auxiliary storage and auxiliary operation on that storage. Address information and sequence numbers are examples of information suitable for storage in auxiliary memory.

The SDL Recommendations define two auxiliary operations which may be included in transitions of the Extended Finite-State Machine (EFSM), i.e., decisions and tasks. "Decisions" inspect parameters associated with inputs and information in auxiliary memory when such information is important for the sequencing of the main machine. "Tasks" perform functions such as counting, operating on auxiliary memory, and manipulating input and output parameters.

In SDL, the interactions between the machines are represented by signals, i.e., the EFSM's receive signals as input and generate signals as output. The signals are composed of a unique signal identifier and optionally a set of parameters. The SDL allows for the possibility of non-zero transition time, and defines a first-in first-out conceptual queueing mechanism for signals which arrive at a machine while it is executing a transition. Signals are considered one at a time in order of their arrival.

D.3 *Applicability of SDL*

Figure D-1 shows a range of possible uses of SDL, in the context of the purchase and supply of telecommunications switching systems.

In this figure, the rectangles illustrate typical functional groups, whose precise names may vary from organization to organization, but whose activities would be typical of many Administrations and manufacturers. Each of the directed lines (flow lines) represents a set of documents passing from one functional group to another; SDL can be used as part of each of these sets of documents. The figure is intended to be merely illustrative and is neither definitive nor exhaustive.

The areas of applications are the ones effectively modelled by communicating Extended Finite-State Machines, e.g. telephone, telex, data switching, signalling systems (e.g. Signalling System No. 7), interworking of signalling systems and data protocols, user interfaces (MML).

When particularly considering SPC switching systems, examples of functions which can be documented using SDL are: call processing (e.g. call handling, routing, signalling, metering, etc.), maintenance and fault treatment (e.g. alarms, automatic fault clearing, system configuration, routine tests, etc.), system control (e.g. overload control) and man-machine interfaces. Application examples of SDL can be found in § D.9.

Specification of protocols using SDL is dealt with in CCITT X-series Recommendations and auxiliary user guidelines for application of SDL in this field are annexed to these Recommendations.

D.4 *General explanation and concepts of SDL*

This paragraph contains the concepts of SDL without giving details of the concrete forms, i.e. SDL/GR and SDL/PR. These details are described in §§ D.6 and D.7 respectively.

D.4.1 *Overview of SDL*

SDL is developed as a means of representing the functional properties as specified and as implemented in telecommunication system.

A system specification, or description, comprises two broad categories of information. The first category comprises the functional specification or functional description. The second category comprises the more general parameters of the system and details such information as the environmental conditions (temperature limits, humidity, etc.), transmission limits, exchange grade of service, which are not represented in SDL.

One of the major goals in developing SDL was to allow users to be able to take large systems and break them down into pieces that are small enough to be easily understood by the users of SDL (both the author and reader). Thus in SDL each system is composed of blocks. The blocks are connected together by channels which are used to show the communication path between blocks. These blocks and channels can be further subdivided into channels and blocks.

At the lowest level (and often at higher levels), blocks contain one or more processes. Processes are the communicating Extended Finite State Machines that are used to model the dynamic behaviour of the system. Processes may also be large and complex. Procedures allow one to further reduce a process into smaller manageable pieces.

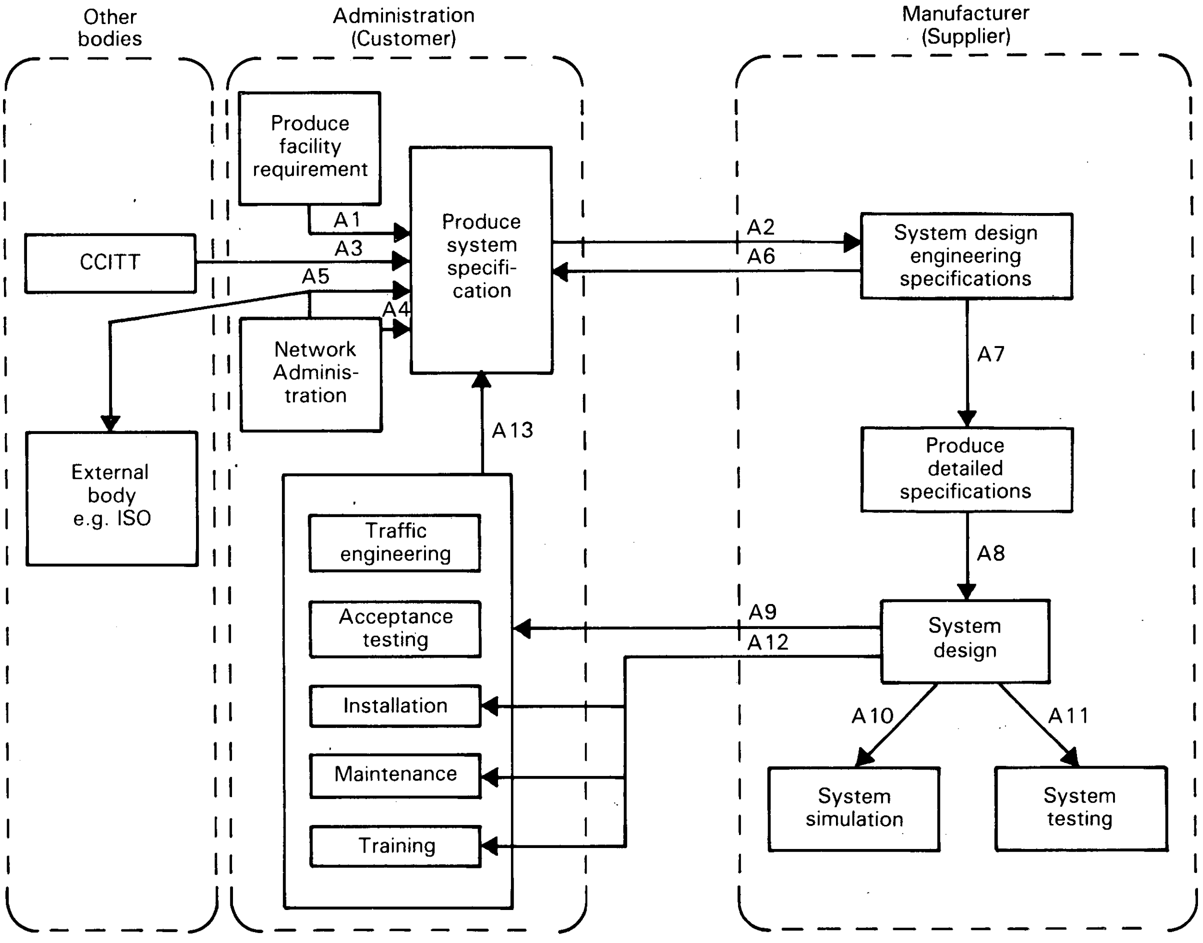
D.4.2 *Structuring SDL systems*

D.4.2.1 *General*

This chapter will discuss some SDL constructs that allow different structures of a system, so that starting from a broad overview of the system structure, more and more details about the structure can be provided showing, for each part, its internal structure. The minimum structure of a system in SDL is what is described in Recommendation Z.101, i.e. a system consists of a set of blocks connected with channels – and the blocks contain processes (see Figure D-2).

For systems not needing further partitioning the concepts of Z.102 are not necessary. Concepts for partitioning in several levels of detail are covered by Recommendation Z.102.

Looking globally at the internal structure of all the parts of a system we see different, more or less detailed, structures of the same system. We say that the system structure is represented at a different level of details.



CCITT-74 680

A1 An implementation-independent and network-independent specification of a facility or feature
A2 An implementation-independent but network-dependent system specification, including a description of the system environment
A3 CCITT Recommendations and guidelines
A4 Contributions to the system specification, showing the network administration and operational requirements
A5 Other Recommendations that are relevant
A6 Description of an implementation proposal
A7 A project specification
A8 A detailed design specification
A9 A complete system description
A10 Appropriate system and environment description documentation for system simulation
A11 Appropriate system and environment description documentation for system testing
A12 Installation and operation manuals
A13 Contributions to the system specification from specialized functional groups within the Administration

Note 1 — Iteration is possible at all levels.

Note 2 — In some circumstances, SDL documentation that is here shown as being internal to one organization, e.g. A1, A7, A8, could be supplied to another organization.

FIGURE D-1

General scenario for the use of the SDL

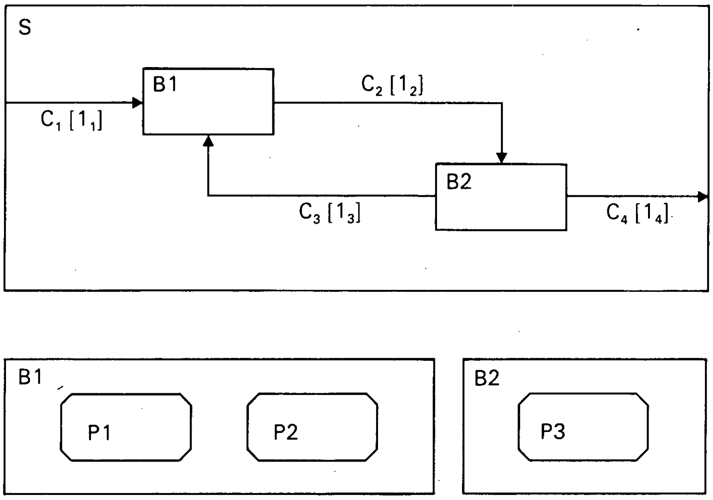


FIGURE D-2

Example of minimal system structure

Note that the internal structure of a system part, provides more details about the structure, not necessarily more details about the behaviour of the system. Conceptually, it is possible to distinguish the aspect of a more detailed representation of the behaviour (e.g. the handling of a new signal) from the aspect of a more detailed structure (e.g. covering both structure of parts and of system behaviour) but in practice the two aspects are usually merged together, so that with new details about the system structure we also provide new details on the system behaviour. For clarity, this chapter will address the aspect of structuring only.

D.4.2.2 *Criteria for partitioning*

The technique of starting with a high level view of a system representation and breaking it down into manageable pieces is called partitioning. This process of partitioning adds structure to a system.

The criteria leading to the partition of the system representation are several, including:

a) to define blocks or processes of intellectually manageable size;

b) to create a correspondance with actual software and/or hardware divisions;

c) to follow natural functional subdivisions;

d) to minimize interaction between blocks;

e) to reuse already existing representations (e.g. a signalling system);

f) to map a description with a certain specification.

The actual criteria adopted may depend on whether a specification or a description is being considered and on the degree of detail required.

Each block can be further partitioned using the same or different criteria.

Since the relationship between levels will depend on the chosen partitioning criteria, it is important to state clearly which criteria have been chosen in order to allow easy comprehension of the representation.

The partitioning criteria depend upon the user but some limitations exist to ensure a correct representation in SDL. These will be discussed in the following paragraphs.

D.4.2.3 *Block partitioning*

A block can be partitioned into a set of blocks and channels in almost the same way that a system is partitioned into blocks and channels (see Figure D-3).

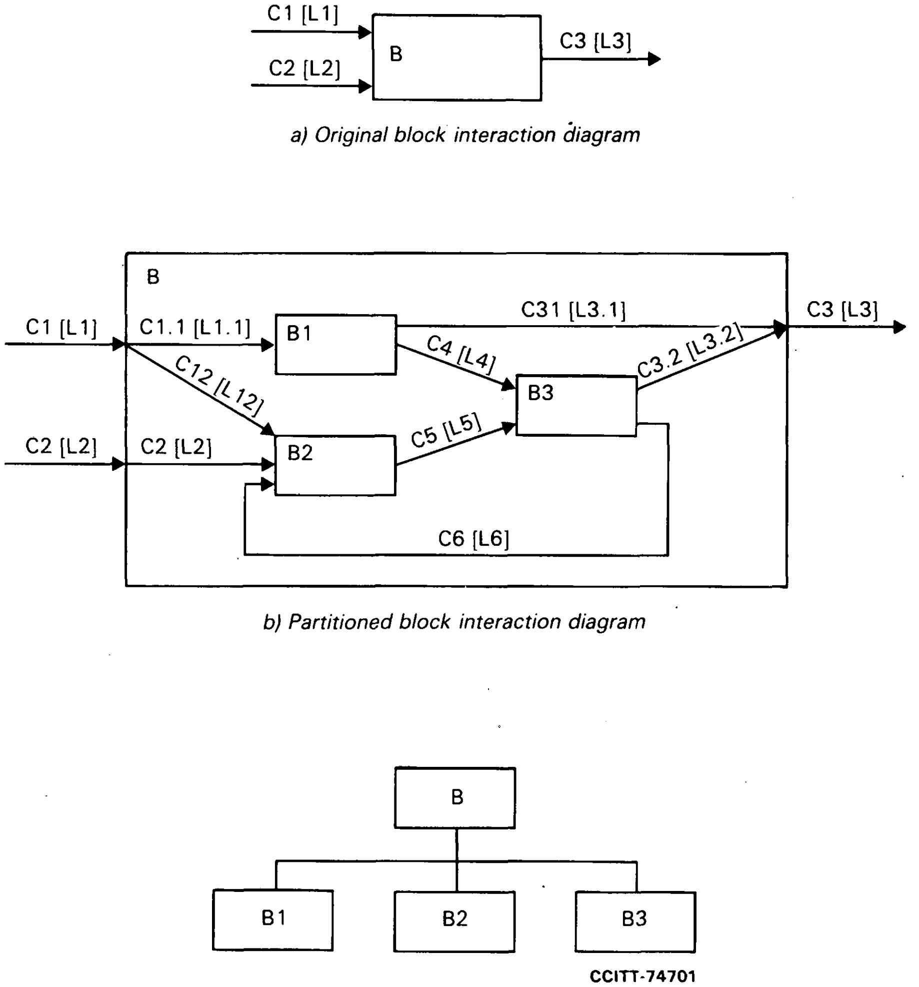
In the partitioning process there are some important rules to remember:

1) The subchannels (e.g. C1.1 and C1.2 in Figure D-3) connected to an incoming channel (e.g. C1) must contain no new signals in their signal lists and their signal lists must contain all the signals in the original channel. Thus for the example shown in Figure D-3, L1.1 and L1.2 contain all the signals in L1. In addition, no signal contained in L1.1 can appear in L1.2.

2) The subchannels (e.g. C3.1 and C3.2) connected to an outgoing channel (e.g. C3) must contain no new signal names in their signal lists and their signal lists must contain all the signal names in the original channel. Thus L3.1 and L3.2 contain all the signal names in L3. Unlike the subchannels connected to an incoming channel, L3.1 and L3.2 may contain the same signal names.

3) If the new channels (e.g. C4, C5 and C6) have signal names in their signal lists that were not known to the original block, these new signals have to be defined in the internal part definition of the original block (B).

4) If the original block contains processes, three options are available. First, each process can be assigned directly to one of the new subblocks. Second the original processes can be discarded and be replaced by new processes in the subblocks. Third, they can be partitioned using the partitioning rules for process partitioning (see § D.4.2.4).



a) Original block interaction diagram

b) Partitioned block interaction diagram

c) Block tree diagram

FIGURE D-3

Block partitioning

5) Data definitions in the parent block are available to its subblocks, so that each of them can use a data type defined in the parent block without having to redefine it.

6) If a type defined in the parent block is redefined in a subblock, the new definition applies to the defining subblock while the old one holds for the other subblocks. A redefinition made just for the sake of characterizing a subblock should be discouraged, because it can be overlooked by a reader who will assume the old definition as valid. In some cases of redefinition should be made, particularly when a refinement in the behaviour is involved. Care should be taken to emphasize this redefinition by appropriate annotations.

D.4.2.4 *Process partitioning*

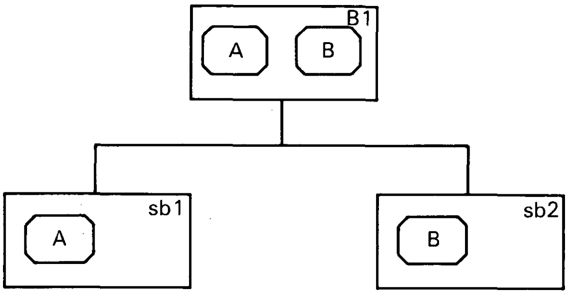
Processes are the means of representing (part of) the functional behaviour of a block.

Looking at this behaviour it may be considered necessary to partition it into sub-behaviours. This is equivalent to say that we partition a process into sub-processes.

The resulting sub-processes should represent (all together) the same behaviour as the behaviour of the parent process. Each signal received by the parent process should be received by one and only one sub-process, and each signal generated by the parent process should be generated by at least one of the sub-processes.

The partition of a process may be a consequence of the partitioning of the block containing the parent process or may be independent of the partitioning of any other construct, i.e. can be made on its own.

D.4.2.4.1 A process can be partitioned as a consequence of a block partitioning. First of all it should be clear that the partitioning of a block does not imply necessarily the partitioning of the process(es) representing its behaviour. It may be the case that a process represents the behaviour (or part of the behaviour) of one of the subblocks. In this case the process does not need to be partitioned (see Figure D-4).



CCITT-74710

FIGURE D-4

Partition of a block with no partition induced on the processes of the block

In cases where the behaviour represented by the process corresponds to the behaviour of two or more subblocks then the process must be partitioned into as many processes as required by the rule that each process can represent the behaviour of (part of) a single block only.

If the block has been partitioned into "n" subblocks each of the processes associated to that block can be partitioned into "m" subprocesses where "m" can be any number >0 independently of "n". In addition it should be noted that each of the processes associated to the block is partitioned independently of the others, so that the partition of one may result into 2 subprocesses, another may result into 4 subprocesses and a third one into one subprocess.

If a process is partitioned into one subprocess only, from the structure viewpoint, the subprocess is equal to the parent process.

D.4.2.4.2 A process can be partitioned independently of the blocks to which it is associated. In this case the sub-processes resulting from the partitioning of the parent process replace it in the same block to which it was associated.

D.4.2.4.3 The reasons behind the partitioning of the process in parallel with the partitioning of the block can be to represent the behaviour of the various subblocks. The reason for partitioning a process independently of the block can be to simplify the representation by isolating particular aspects of the behaviour.

Note that if these aspects are considered as subbehaviours (such as frequency validation or digit recognition) it would be much better to represent them by means of procedures and macros.

On the contrary if the behaviours isolated are "main" behaviours all having the same dignity (importance) they are better represented by subprocesses. In the former case the subbehaviours may handle the same input signals of the main process whilst in the latter case each subprocess has a set of input signals that is disjointed from the sets of input signals of the other subprocesses.

Examples of this latter case may be the partitioning of the "call handling process" into the "subscriber handling subprocess" the "routing subprocess" and the "junction handling subprocess" (see Figure D-5).

In partitioning a process, there are some important rules to remember:

1) The partitioning of a process induces a partitioning on the set of the incoming signals, so that a certain subset of signals will be received by a certain subprocess, another subset by another subprocess and so on. The subsets are disjointed and globally contain the signal set received by the parent process. They may also contain other signals produced by the partitioning; these signals are used to transfer information between the subprocesses.

2) Data definitions by the parent process are usable by the sub-processes without having to be redefined. If a data type defined in the parent process is redefined in a sub-process, the new definition applies to the defining sub-process only, and the old one holds for the other sub-processes.

3) All variables have to be redefined in the sub-processes. Procedures defined in the parent process or in the block associated to the parent process are visible to the sub-processes.

4) For each process definition, it is possible to have a variable number of process instances. When a process is partitioned into subprocesses, a create request for the process causes the creation of one instance of each of the subprocesses listed in the process substructure definition. Each of these subprocesses is a process and possesses all the characteristics of any SDL process.

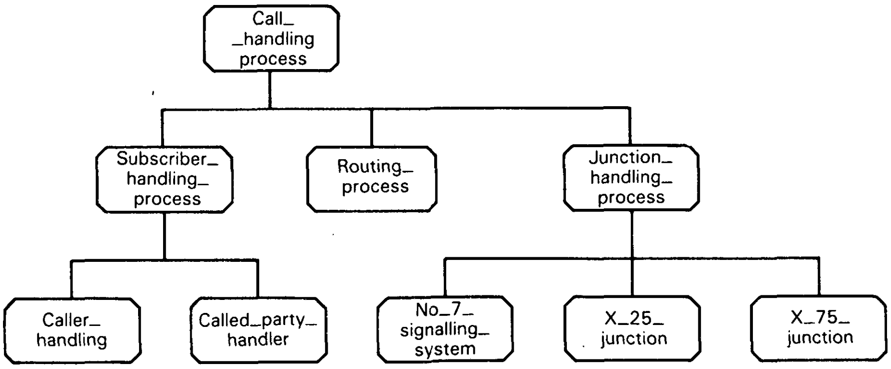


FIGURE D-5

Example of process partitioning

D.4.2.5 *Channel partitioning*

A channel can be partitioned independent of the blocks it connects. This allows the representation of the behaviour of the channel when conveying signals. To get an exact representation of the way through which a signal is conveyed, in some cases it may be necessary to represent the channel behaviour. This is made by considering the channel as an item of its own having as environment the two blocks it connects as shown in Figure D-6.

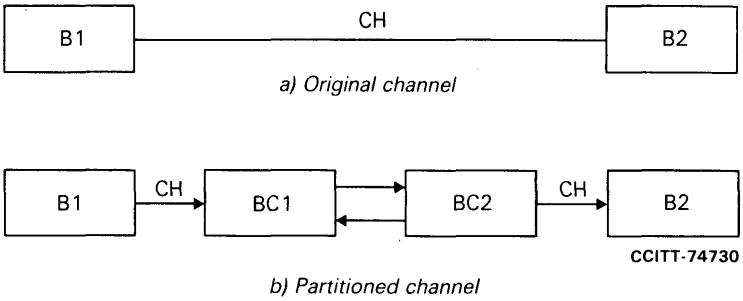


FIGURE D-6

Example of partitioning a channel

Looking at a channel in this way, we can show its structure (the blocks into which it is structured) and for each block its interconnection with the other blocks and the processes representing the behaviour of the blocks.

In the partitioning channel the incoming and outgoing channels are not split into multiple subchannels but must connect to only 1 block. With the exception of this restriction, the partitioned channel has the same attributes as a partitioned block. The blocks and channels contained in the channel interaction diagram may be further partitioned as well.

D.4.2.6 *Mutual influence of the partitioning on blocks, processes and channels*

So far we have considered the partitioning of blocks, processes and channels independently of each other. In reality these activities have closed relations. As has been seen the partitioning of a block usually induces a partitioning on the interconnecting channels and on the processes associated.

Conversely, the partitioning of a process singles out particular behaviours that are usually handled by specific parts of a system (each one mappable onto the concept of block).

The driving force behind the partitioning of a system representation may differ, as explained in § D.4.2.2 but the result is the partitioning of all the entities into which a system is represented.

This is discussed in more detail in the following sub-chapters. The partitioning of blocks, processes and channels should be self-evident, when facing aspects such as complexity, management or mirroring the actual system structure.

The SDL Recommendations provide constructs, rules and notations for the partitioning of these entities, but users will have to face the fact, that other SDL entities are involved in the partitioning and the consequent aspect of relating the various entities one to the other.

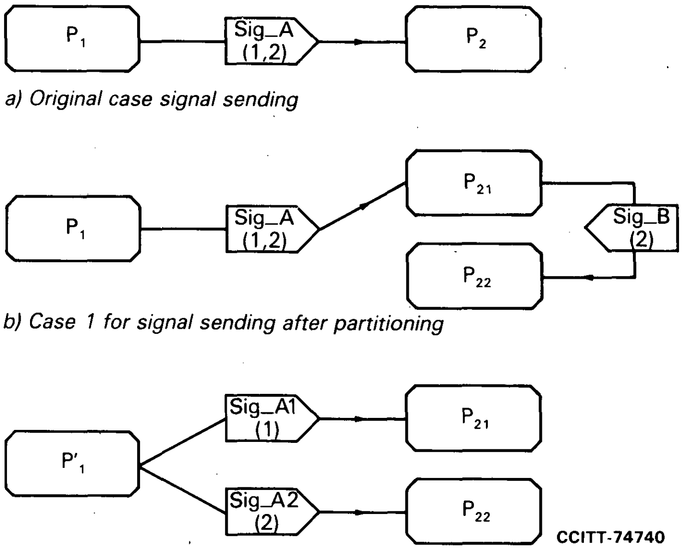
D.4.2.6.1 *Signals*

A signal conveys information from a process to another (the environment behaviour can be modelled by one or more SDL processes). This information can be very complex and is conveyed at a certain time in the life of the system, i.e. when the sending process interprets a certain transition and after that when the receiving process reaches a certain state.

Partitioning a process into subprocesses may require that the information “packed” in a signal has to be distributed to more than one subprocess.

This can be solved in two ways. In the first way the signal is not affected, it will be received by one subprocess and the subprocess will generate the signals necessary to convey the part of the information the other subprocesses may need. The second way affects the signal. In this case it is partitioned into a number of signals, one for each of the subprocesses requiring the information.

The two mechanisms are shown in Figure D-7.



a) Original case signal sending

b) Case 1 for signal sending after partitioning

c) Case 2 for signal sending after partitioning

FIGURE D-7

Mechanism of signal sending when process partitioning occurs

The two mechanisms explained seem equivalent and it would seem that the second can always be mapped into the first but there is a difference in that the former performs a sequential transfer of information, the latter performs a potentially parallel transfer of information. Also in the second case the sending process is not identical to the process in the original case because the two processes send different signals.

Apart from the above consideration there is a case in which the second mechanism is the only one that can be used. Imagine that at a high level of description we use a powerful signal with several types of information packed together.

When partitioning the system representation (thus giving more details on the structure) we can be in the situation where the information packed within the signal cannot be packed any longer due to the fact that they are not available in the same place or/and at the same time. This can be the result of the partitioning being the information located into different blocks (different signal origin) or available at different times. An example of this latter case may be the signal "called party selection" at the high level becoming at the lower level a sequence of the signals "digit". Note that we could imagine a packing procedure that accumulates all the "digits" and then sends the "called party selection" but this may be a distorted representation or even worse could be an impossible representation in case where we want to show that a release back may occur from the outgoing junction after each digit.

An example of the former situation may be the signal "line set up completed" that at a lower level is partitioned into a set of signals originated by different subblocks each one keying a part of the total set up.

As shown by the examples, there are several cases in which a signal should be partitioned as result of the partitioning of the system.

The SDL Recommendations do not recommend any particular notation even though some of these were proposed and discussed by the group defining the SDL.

It is advisable to annotate in SDL representations with various levels those signals which result from the partitioning of a signal at an upper level. This greatly increases the readability of the representation bridging together the various levels.

The annotation should state the parent signal for a certain signal when this signal is defined. Also a forward reference indicating for each signal the signals resulting from its paritioning.

Note that the partitioning of a signal affects both the sender and the receiver, as well as the channel which now contains new signal names. In a situation where several signals are partitioned the benefits deriving from maintaining the old channel, adding to its extremities subchannels, are lost and in a sense this representation becomes disadvantageous being now necessary to redefine part of the old channel. The extent of this redefinition depends on the number of signals partitioned.

The second approach indicated in Figure D-7, to substitute the old channel with new ones, is more attractive in this case for its simplicity considering that both the sender and receiver are affected anyhow.

D.4.2.6.2 *States*

The partitioning of a process into subprocesses can also be considered on its components, i.e. states, decisions, tasks and data (inputs and outputs are affected by the partitioning of the signal).

A state should be the representation of a logical situation of a process, e.g. the conversation. phase or the testing phase. The partitioning of the process into subprocesses causes the partitioning of the logical states into a set of states over several subprocesses (not necessarily over the totality of the subprocesses).

This has effects both on the human interpretation and on the machine interpretation of the correlation between levels.

Increasing the number of details, there is a tendency in the human beings to lose the grip on the general situation and this may lead to undesired effects (compare how many times an updating to a program which was supposed to be perfect, generates undesired effects in the global system!).

The spreading of details logically grouped, has an even more deleterious effect on the capacity of understanding a global situation. The SDL organization of a representation on several levels strives to avoid this problem, but it is essential to correlate the various levels to one another.

The correlation can be achieved by appropriate notations referring to the more abstract representation giving the overview (the upper level) and by referencing the other parts that have to be considered together.

The aspect of machine interpretation is completely different from human interpretation. The machine does not have any overview, i.e. all the items are taken into account, each being considered with the same attention as the others, and they are taken into account one at the time, one relation at the time. For this reason the spreading of information each one being related to the other does not affect the machine interpretation (let us forget about time consumption in the interpretation). The problem with the machine is the one of formalizing the partitioning of the process, so that the machine can understand that what was previously referred to as "idle" is now a set of "idle" states spread over several subprocesses and so on.

The only possibilities for making a machine aware of this correspondence is by declaring it explicitly (most unefficient and seldom safe for the high chances of forgetting some declarations) or by formalizing the rules of partitioning a process into subprocesses.

This issue is tackled (and solved) by methodologies and is something that is not intentionally, dealt with by SDL Recommendations.

D.4.2.6.3 *Decisions*

Decisions may be affected by the partitioning of a process if the data upon which the decision is made have been partitioned. If a data is no longer available to a subprocess which has to make a decision upon it, a signal has to be sent to require the value of the data and the subprocess must wait for the answer. The mechanism of signal sending (request) and reception (answer) can be hidden by declaring the data as imported and actuating the decision on the imported data.

Note that as sub-processes usually are allocated in different blocks, the viewing mechanism of shared data is not usable.

D.4.2.6.4 *Tasks*

The task is the representation of a set of actions not having a direct effect outside the process. When the process is partitioned the set of actions performed by a task can also be partitioned in such a way that some parts are performed in a subprocess, others in different sub-processes and so on.

Note that the splitting takes place only in the subprocesses of that process. If a certain set of actions was performed in a given process the same set is performed by the totality of the subprocesses of that process. In this case there is a slight difference with the state, whose partitioning may depend on the partitioning of other processes, because a state of a process is part of a logical status which may include more processes.

Coming back to the task, the partitioning of the actions performed over several subprocesses will usually require the introduction of new signals to trigger the execution of a subset of actions in a subprocess upon the termination of the actions performed by another subprocess.

The partitioning of a task is usually a side effect of the partitioning of the data associated to a process. Again a design/representation methodology may provide formal notations to trace the partitioning of a set of actions over several subprocesses.

The same advice given for the annotation of signals to help the reader are applicable to task partitioning.

D.4.2.6.5 *Data*

Data are associated to a process. When we partition a process we also partition its data. There is no formal way in SDL to indicate this partitioning. Each of the subprocesses needs to define its variables and the new synonyms and types if any.

Note that the partitioning of data affects not only the actions of the process (a certain output carrying an information cannot be sent if the information is no longer available to that subprocess, a task operating on that information cannot be performed etc.) but also the partitioning of data may affect other processes if exported or revealed data are involved.

The importing processes should be notified of the partitioning in the exporter, so that they can address the new subprocess to get the information. A data which was viewed may not be viewed any longer if the subprocess owning it is allocated in a different subblock of the viewer and therefore its value should be imported/exported or explicit signal interchange may be performed.

D.4.2.7 *System representation in case of partitioning*

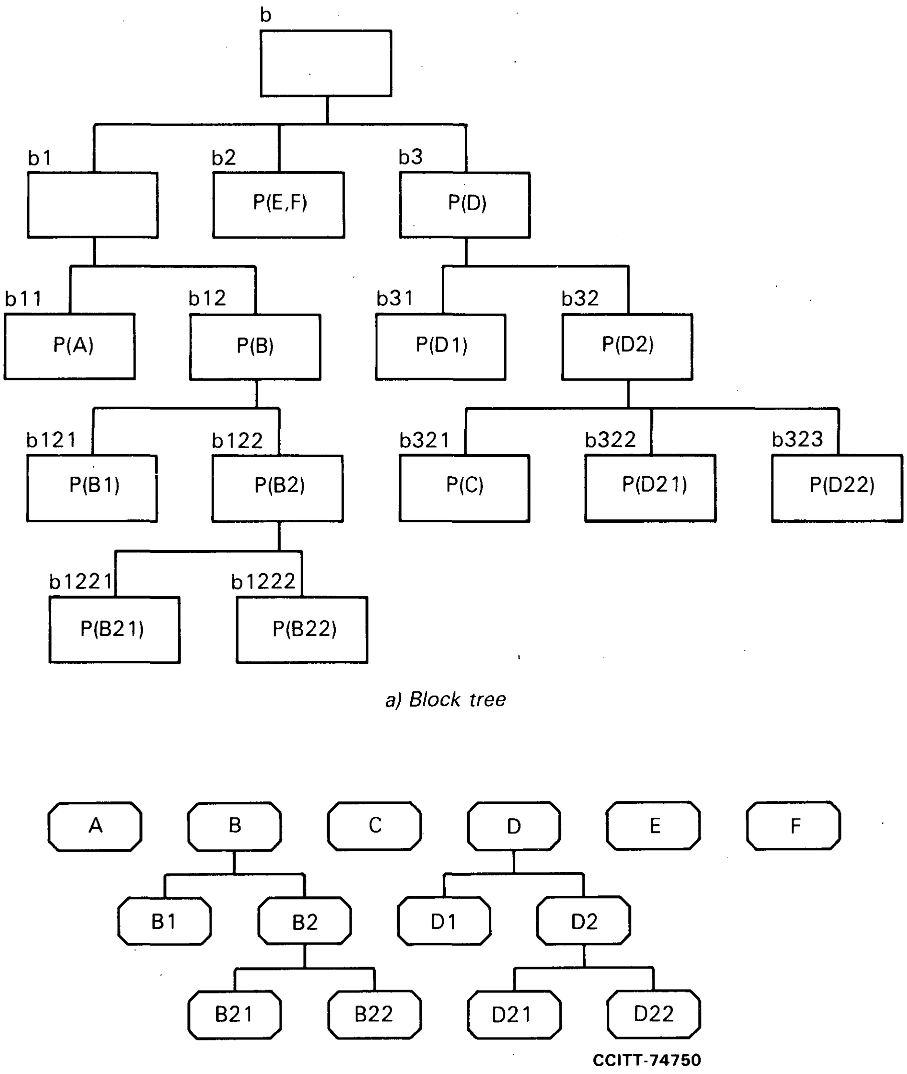
In the cases where a system is represented as a set of blocks interconnected by channels, and where the behaviour of each block is expressed by one or more processes, we have a single level representation. This means that we can see all the representation elements at the same level. When we introduce the partitioning, we insert a hierarchial relation between the various documents. We will have a document containing the representation of the structure of the system where the system is composed of "n" blocks.

A different document may present the system as composed by a different set of blocks some being derived from the blocks contained in the previous document (some blocks in the previous document have been substituted by the subblocks obtained by the partition of the blocks). This latter document must be related to the previous one.

It is not just a matter of relating the documents to each other to obtain a complete representation of the system, but also they should be organized in such a way that it is possible to access the system representation by levels, starting with a general overview and moving to more and more detailed representations. This implies the grouping of the various documents so that they form various levels of system representation.

Not all the levels should contain the same elements. At a first level the system representation may consist of the block and channel representations without including the processes describing the behaviour of each block. At a lower level we may wish to include the representation of the behaviour of some blocks but not the one of others, or we may even wish to represent part of the behaviour of a block but not the complete behaviour, which will be provided only at a lower level. The lowest level of representation (the more detailed one) should include the complete representation of the behaviours of all the blocks, i.e. the complete set of processes expressing this behaviour.

The first side effect of this type of representation by levels is that we have to indicate the association of process to block. The partitioning of the system (considered as a single block) generates a tree structure; the partitioning of the "n" processes representing the behaviour of the system generates "n" trees and each of the processes of these trees should be associated to a block shown in the block tree. In addition we may have new processes introduced at a certain level and possibly further partitioned from that level on. Each of these, if partitioned, generates a tree structure and again each process in the structure should be associated to a block in the block tree as shown in the Figure D-8.



a) Block tree

b) Process trees

Note — Blocks b and b1 do not have associated processes.
Block b2 contains two processes (E, F).
Block b321 contains process C which appears at that level only.

FIGURE D-8

Allocation of processes to the blocks

Looking at the block tree we can make several considerations.

First of all the tree always has one and only one root, the system block. This is so, even in the cases where a system is represented from the beginning as composed of several blocks. In the block tree these blocks will be represented at, for instance, level 1. The root block may consist of the system name only. The channels definitions may be given for the blocks at level 1 even though this is not required unless the blocks have associated processes.

A second point comes up looking at the leaves of the tree: not all of them are at the same level. This is the result of a different number of partitioning on the blocks of the tree. The number of partitioning depends on several aspects, most of which are under the subjective judgement of the specifier/designer.

The SDL only requires that leave blocks can stay without being further partitioned only if their behaviour is completely specified (i.e. the processes definitions associated are sufficient to represent its behaviour). Consequently a leaf block must contain at least one associated process.

When a system representation is given on several abstraction levels, we can select any of these levels to represent the system. The selection of a certain level implies the consideration of the blocks at that level, of their associated processes and of all the blocks which are leave blocks at upper levels together with their associated processes (see Figure D-9).

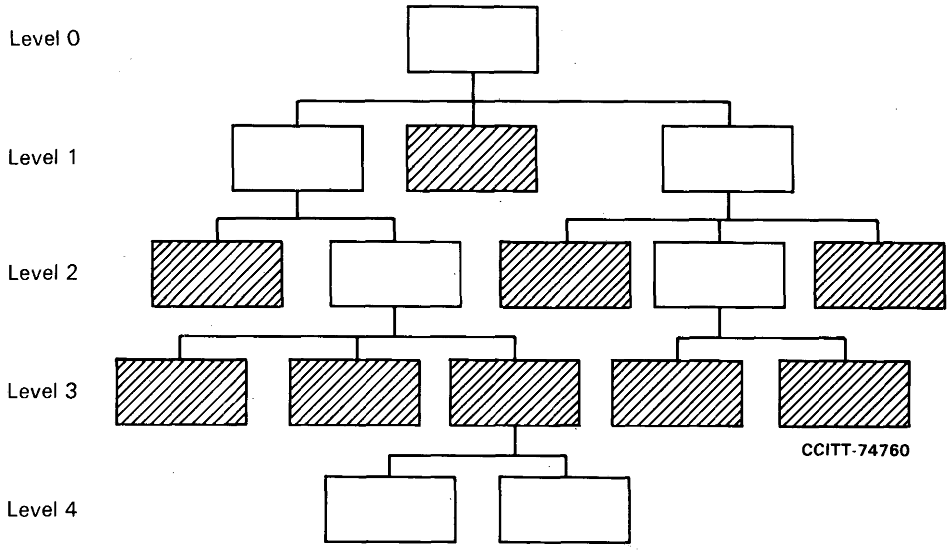


FIGURE D-9

Representation ▨ level 3

A system representation at a certain level may be incomplete in the sense that some of the blocks at that level do not have associated processes of that those associated are not completely representing their behaviour.

The reasons for talking about representation levels and of selecting a certain level of representation are that:

- We may have reached in our design a certain "level" of detail, which is represented by that level of representation (in this case the blocks at that level are leave blocks and they are not complete because further work is needed!).
- We want to look at the system representation at a certain level of detail; and therefore we select that level of representation which corresponds in the best way to the abstractions we are looking for. Note that in some cases a certain representation level may consist of documents having different levels of abstraction. A part of the system may be represented in a very detailed way at level 2, whilst another part may be still abstract at level 4. This means that when we select a representation at level three we can have very detailed parts together with parts that can only be considered as overview.
- The representation/design methodology can be such that each level has a precise meaning, e.g. level one corresponds to the specification, level two provides the overall system structure, level three provides the module structure (racks, software functions), level four provides the detailed structure (printed boards, procedures, software modules). In this case the selection of a certain level corresponds to a certain reader's need; in this case it should be noted that the methodology will avoid the situation of discrepancies in the level of details of part composing a certain representation level.

In addition to the total system representation and to the one given by levels, SDL has the concept of "consistent system representation" (see Figure D-10).

We define with that any representation of the system considering it, as a single level representation where all the blocks can be taken from any level of the system structure given that:

- A block can be chosen to be part of the consistent system representation if it can be considered as a leaf block (either it is a leaf block or it has associated all the processes required to represent its behaviour).
- If a block is chosen all the blocks obtained from the partitioning of its parent block should be included either directly or by including their sons.
- All the documents defining the signal flowing on the channel connecting a block in the representation should be provided. This may imply, depending on the partitioning strategy chosen, that once a block has been taken, its parent should also be taken at least for the parts defining the data and the signals.

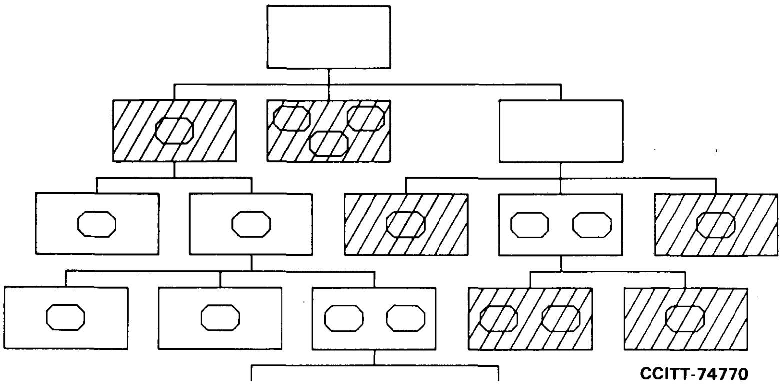


FIGURE D-10

Example of consistent system representation

In cases where some channels have been partitioned considering them as systems, the respresentation of each of these "systems" has to be provided.

The representation has the same type of documents as the usual system representation. Notation should be added to relate these systems to the main system. In a sense we may consider these systems as inner systems with respect to the "main system". Each of these systems can have several levels of representation and may also have inner systems if some of the channels contained in them are further dealt with as a system of its own (see Figure D-11).

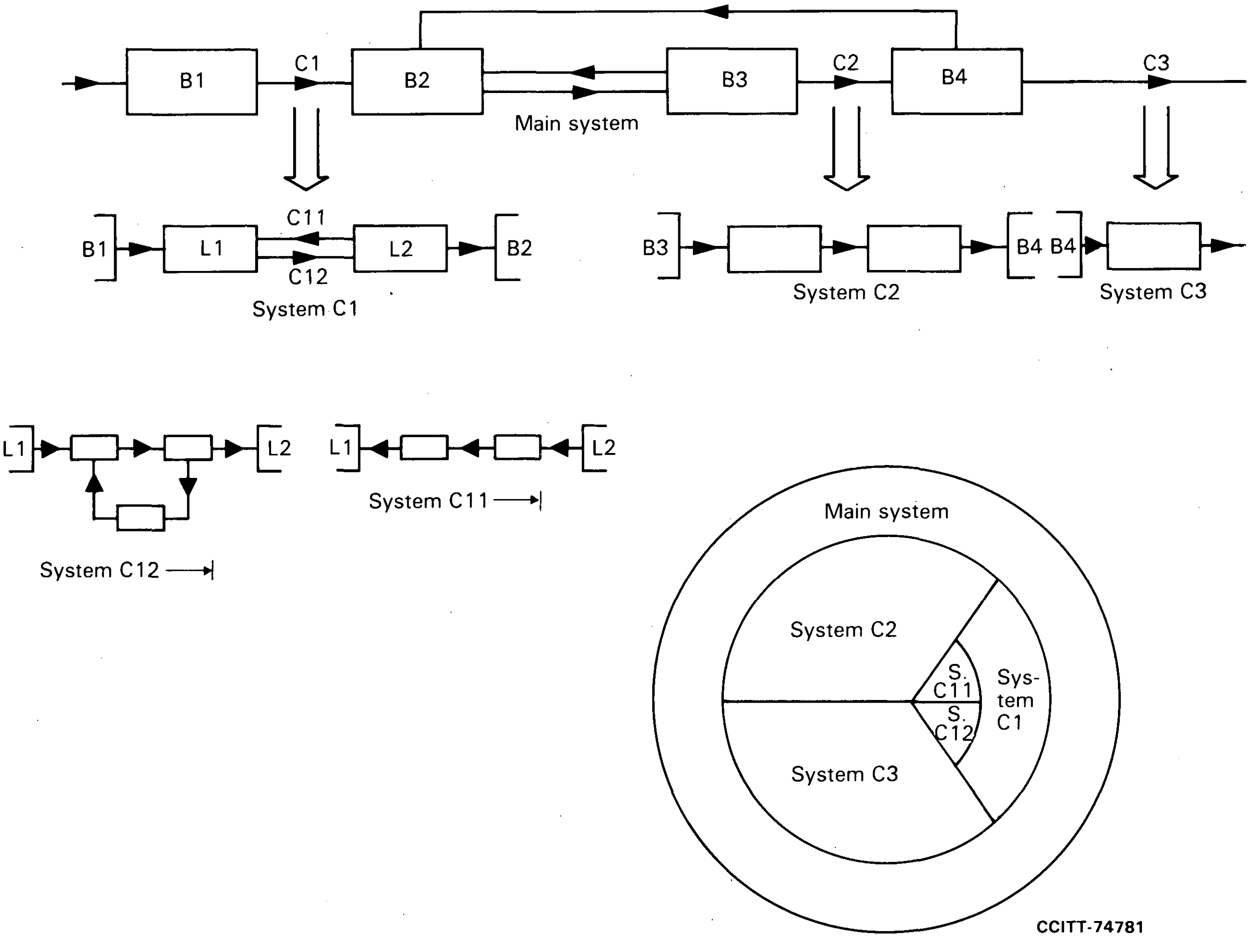


FIGURE D-11

Channel partitioning as virtual system representation

D.4.3 *Concepts of SDL*

This § will present each of the SDL concepts and will give some general guidelines on how to use each of them. For the examples, SDL/GR will mostly be used. The same guidelines apply for the SDL/PR as well.

D.4.3.1 *System*

As has been stated earlier, SDL models systems. Thus the system is what an SDL specification or description is defining. As such, an SDL system may model a part of a telephone system (or exchange) or a complete network of telephone systems or parts of a multiplicity of telephone exchanges (e.g. the trunk controllers at both ends of a trunk). The key thing is that from an SDL point of view, the SDL system contains everything the specification or description is trying to define. The environment is outside the specification and is not defined in SDL.

The system interfaces with the environment through channels. In theory, only a single incoming and a single outgoing channel are required to interface with the environment. In practise, channels are usually defined for each logical interface to the environment.

Each system is composed of a number of blocks connected together by channels. Each block is independent from every other block. The only means of communication between processes in two different blocks is by sending signals over the channels.

D.4.3.2 *Block*

As was shown in the chapter on structuring in SDL, a block can contain a structure of 1 or more processes. The complete definition of a block requires that the blocks at the bottom of the block tree diagram contain 1 or more process definitions.

Within a block, processes can communicate with one another either by signals or shared values. Thus the block provides not only a convenient mechanism for grouping processes, but also, a boundary for the visibility of data. For this reason, care should be taken when defining blocks to ensure that the grouping of processes within a block is a reasonable functional grouping. In most cases it is useful to break the system (or block) into functional units first and then define the processes that go into the block.

D.4.3.3 *Channel*

Channels are the means of communication between blocks. Normally, a channel is a functional entity that may be used to denote a specific information path. In fact, through the partitioning of channels, it is possible to formally specify the behaviour of each channel.

Channel definitions contain a signal list that lists all the signals that can be carried by the channel. This signal list is provided as a means of ensuring that every signal sent by a process at one end of the channel can be received by the process in the block at the other end of the channel. Thus the channel definition becomes part of the interface definition for each block. In large multi-person projects, early agreement on the signals in a channel and on the definition of those signals reduces the probability that two processes will not be able to communicate with one another.

D.4.3.4 *Process*

A process is an extended finite state machine and defines the dynamic behaviour of a system. Processes basically are in a state awaiting signals. When a signal is received, the process responds by performing the specific actions that are specified for each type of signal that the process can receive. Processes contain many different states to allow the process to perform different actions when a signal is received. These states provide the memory of the actions that have occurred previously. After all the actions associated with the receipt of a particular signal have occurred, a new state is entered and the process waits for another signal.

Processes can either exist at the time the system is created or they can be created as the result of a create request from another process. In addition, processes can live forever or they can stop by performing an internal stop action. Some important attributes of process creation are:

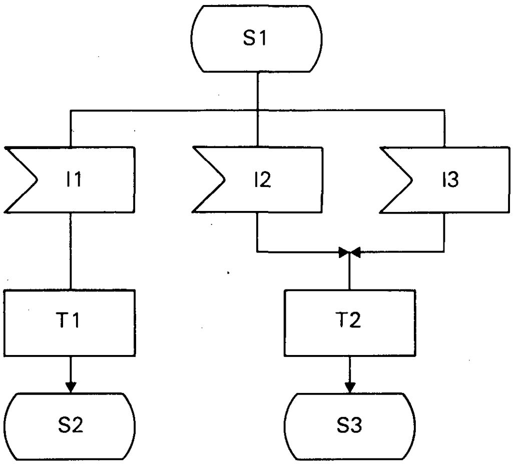
1) After the system is created, processes can only be created by another process in the same block. One means of allowing creation of processes in other blocks, is to have a special process in every block that will cause the creation of a process when it receives a signal from a process in another block. In many instances, this special process is some sort of "operating system" process.

2) Once created processes have a lifetime of their own. Processes die only when they perform a stop action during a transition. One way to model systems where outside process kill operations are allowed, is to have a special kill signal. When the kill signal is received, the process performs a stop action.

3) When created, a process knows its own identity and the identity of its parent. Without being given any other information, it cannot communicate with any other processes. There are two ways to solve this problem. First, one or more of the process parameters contains the identity of the processes that this process will communicate with. Second, it can receive a signal that contains a process instance identifier as one of its actual parameters.

D.4.3.4.1 *States*

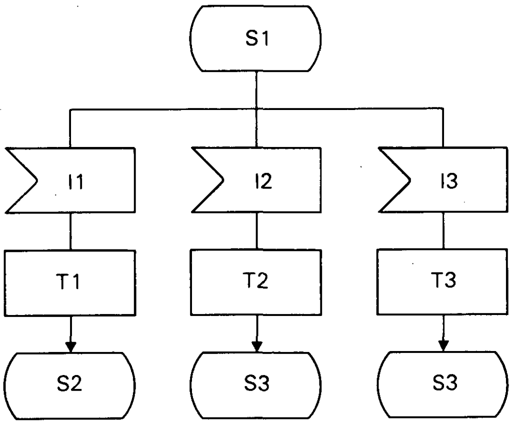
A state is a point in the process where no actions are being performed but where the input queue is monitoring for the arrival of incoming signals. Based on the signal identifier given in the input signal, the arrival of the signal will cause the process to leave the state and perform a specific sequence of actions. A signal which has arrived and caused a transition has been "consumed" and ceases to exist. During a transition, a process does not know explicitly which input signal caused the transition. This can only be inferred from context (i.e. this transition can only occur if a particular signal had been received). In Figure D-12, task T1 is performed only if I1 is received. However, task T2 will be performed if either I2 or I3 are received. However, task T2 will be performed if either I2 or I3 are received. If it is important for T2 to know which input was received, then it is better to design the process as shown in Figure D-13.



CCITT-74790

FIGURE D-12

Performing a task dependant of two out of three received signals but independant of which one



CCITT-74800

FIGURE D-13

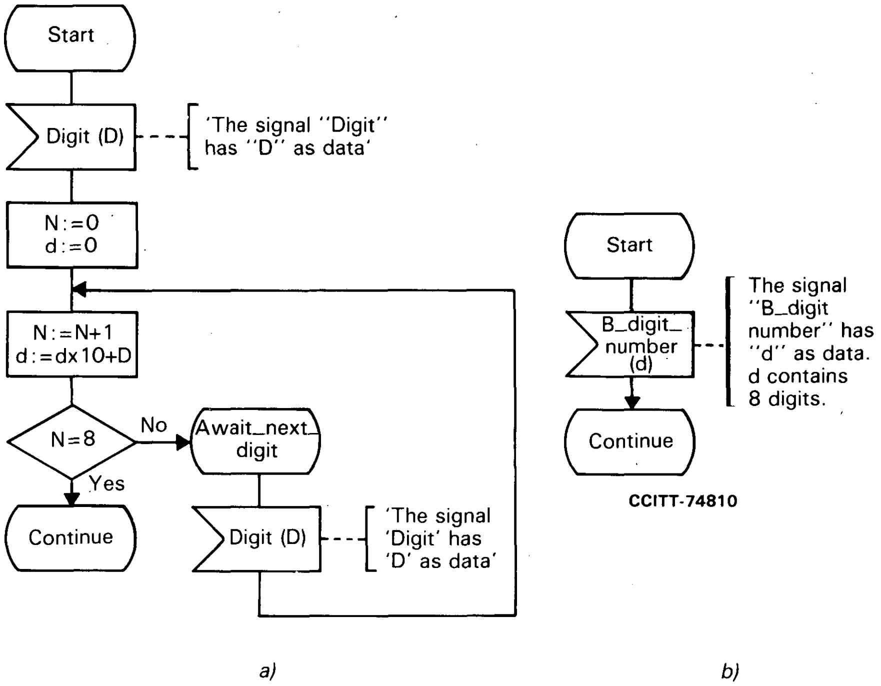
Performing a task dependant on the received signal

D.4.3.4.1.1 *Determination of the required states*

Usually the author of an SDL diagram has some flexibility available in the approach when defining the states of a process. He may require a strategy enabling him to identify the states of the process and this strategy can be informal or formal. Good judgement (obtained through practice) is required in order to produce SDL diagrams which are neither unnecessarily complicated through the identification of too many distinct states or which fail to exploit the inherent advantages of SDL through having an artificially reduced number of states. Before the author starts to draw the diagram, necessary preliminaries (discussed in § D.4.2) must have been completed, for example:

- the structuring of the system in functional blocks;
- the representation of the functional blocks by means of one or more SDL processes per block;
- the choice of input and output signals;
- the use of data in the process.

All the above factors have a crucial effect in determining the states of each process. The effect of the "choice" of input signals upon the number of states in an SDL diagram is shown by the two examples in Figure D-14.



Start
Digit (D)
'The signal "Digit" has "D" as data'
N:=0
d:=0
N:=N+1
d:=dx10+D
N=8
No
Yes
Await_next_digit
Continue
Digit (D)
'The signal 'Digit' has 'D' as data'
Start
B_digit_number (d)
The signal "B_digit number" has "d" as data. d contains 8 digits.
Continue
CCITT-74810
a)
b)

Note — Examples a) and b) represent the same function with different levels of details. As a consequence the number of states varies.

FIGURE D-14

Reception of an 8-digit telephone number

D.4.3.4.1.2 *Reduction in the number of states*

Having used a strategy for identifying the states of a process, the author of an SDL diagram may feel that "too many states" have been used. The number of states is important because the size and complexity of an SDL diagram is often closely related to the number of states. There may be some means available for reducing the number of states, but the fact that an SDL diagram is complex is not, by itself, a reason for changing it, as the complexity of the diagram may simply be a reflection of the inherent complexity of the process which it defines. In general the set of states should be chosen to give maximum clarity to the sequential interaction between the process and its environment. Such clarity is in general not achieved by minimizing the number of states. The number of independent sequences handled by a process has a multiplicative effect on the number of states. It is therefore desirable to treat independent sequences in separate processes as this will reduce the number of states and increase clarity.

The number of states can be reduced by separation of common functions, by merging states or by using the procedure concept. Particular data structures can also lead to a reduced number of states. Alternative representation can benefit from the use of macros.

D.4.3.4.1.3 *Separation of common functions*

It may become clear when planning an SDL diagram, that to define a particular and repetitive aspect of a process will require the representation of repetitive states. In Figure D-15 part of a line-signalling process SDL diagram is shown which illustrates the requirement that a line signalling tone must be present for a particular length of time before the line signal is considered to have been detected.

To specify this an intermediate state is required between the state No _ line _ signal _ detected and the state Conversation. Let us assume that in a complete diagram, such a common function would have to be repeated at every point where the signal is detected. An alternative way is to define a separate process which is responsible for monitoring the line signalling tone and detecting signals on the basis of the specified recognition time. The existence of this new process would enable the diagram shown in Figure D-15 to be drawn as shown in Figure D-16. (In a given context, the figures can be made equivalent, at the expense of introducing a new signal Valid _ line _ signal.)

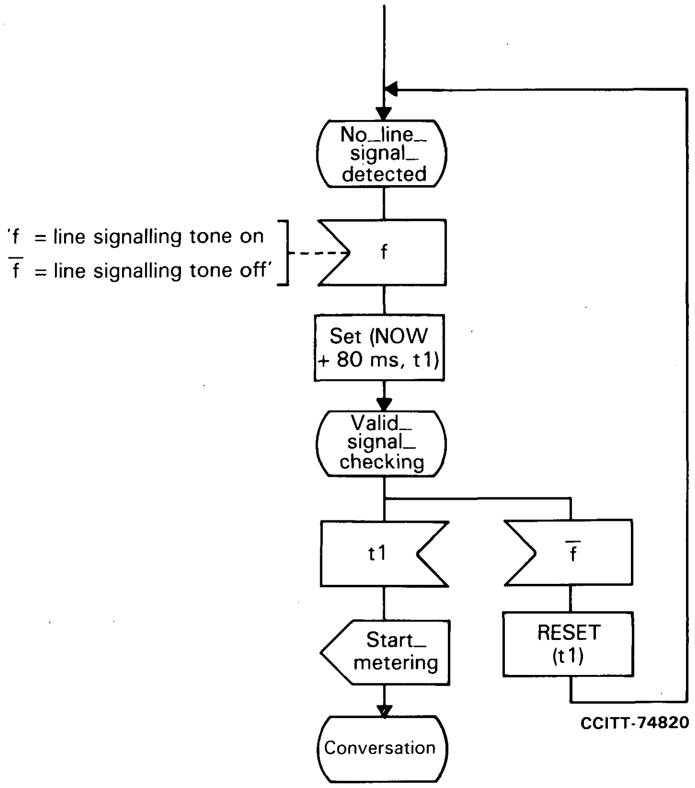
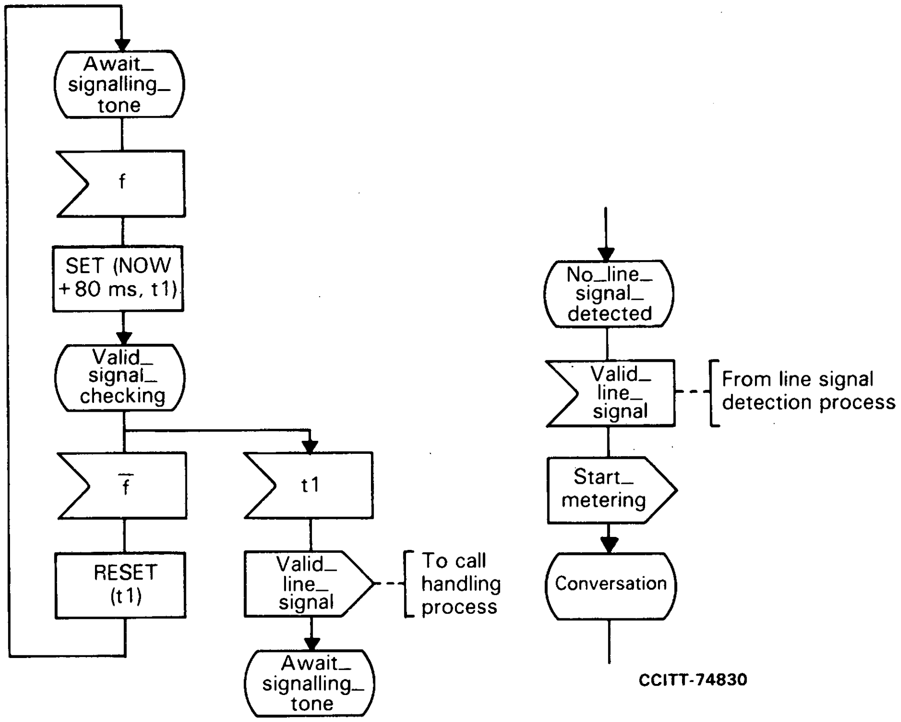


FIGURE D-15

Example of an SDL diagram for a composite call-handling and line signal detection



a) Line signal detection process

b) Call handling process

FIGURE D-16

Example of the segregation of a common function (line signal detection) to avoid repetitive states such as Valid_signal_checking (compare with Figure D-15)

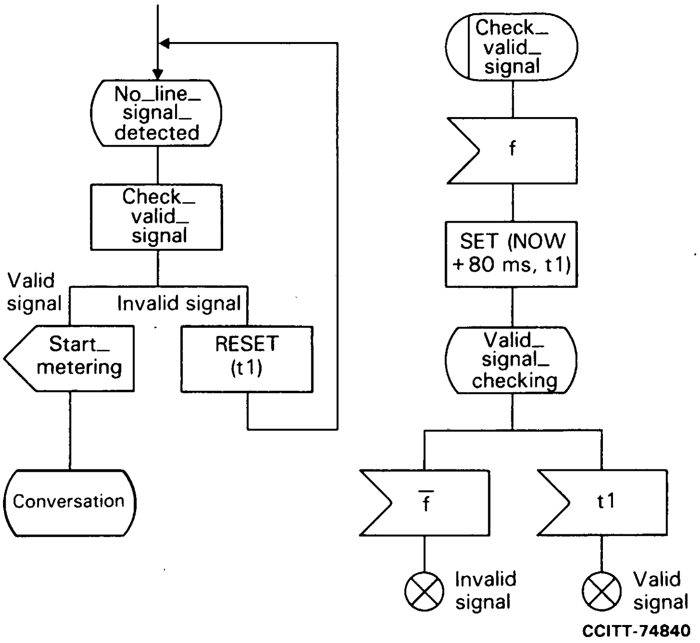


FIGURE D-17

The example of Figure D-16 with the use of macros

The slight difference between the process shown in Figure D-15 and the two shown in Figure D-16 should be noted.

The process in Figure D-15 starts metering immediately upon reception of the t1 whilst the call handling process (process b) in Figure D-16) starts metering upon the reception of the valid _ line _ signal and this is generated and sent upon reception of the t1 by the process a) of Figure D-16. This implies that in this second case the metering starts after t1 + signal generation, sending and reception time. In addition there may be other signals that have been queued in the time it takes to generate and send the valid _ line _ signal.

A second aspect to be noted is that this separation of a subfunction would not be possible, if the signal to be received by the subfunction has to be received by the main function as well, because a signal is always directed to one process only.

If in the example the same signal f has to be received in another state, where it is not required to validate it for time t1 it would not have been possible to separate the validation subfunction into another process.

In general, the solutions with separate processes are useful in cases where signals are to be processed in a manner which is independent of the states in the main process. In this case pre- and post-processes can handle the detailed sequences and relieve a main process of the details. This will often provide a useful modularity too because particularities of e.g. signalling systems can be isolated from the more service oriented main process.

A different solution to this problem would be to use the MACRO notation as shown in Figure D-17. In this case we obtain the desired compactness in the diagram without altering in the slightest the semantics of the original one. Furthermore the MACRO can be called from several states, if the logic of the process should require it.

D.4.3.4.1.4 *Merging of states*

If in an SDL diagram the future of two states is the same, then, irrespective of their history, they can be merged into one without affecting the logic of the diagram.

Figure D-18 shows part of an SDL diagram with two states which differ in their history but whose futures are "identical". In Figure D-19 the two states have been combined to form one state. This is a fairly trivial example where the reduction in complexity is small but the technique can be used to obtain significant simplification. The SDL semantics do not allow a decision following a state to determine the history of the process before the state (i.e. for this example, was A6 or B4 sent?) unless this information had been explicitly stored prior to entry into the state. Note that naming of data in an input causes the value to be stored.

A state should have a clear relation with the possible logical situations of the process, and therefore it is not advisable to merge different logical situations in one state.

Care should be taken when a merged diagram is changed later on. The user should investigate whether the intended change effects the original two (or more) branches of flow in the same way or not.

D.4.3.4.2 *Inputs*

Paragraph D.4.3.4.2 has been written to explain the input concept and the use of inputs in SDL diagrams without the save concept. The save concept and the use of inputs and saves together in an SDL diagram is covered in § D.4.3.4.3.

D.4.3.4.2.1 *General*

An input symbol attached to a state means that if the signal named within the input symbol arrives while the process is in this state the transition which follows the input symbol should be interpreted. When a signal has triggered the interpretation of a transition, the signal no longer exists and is said to have been consumed.

A signal may have associated data. For example, a signal named "digit" serves not only to trigger the execution of a transition by the receiving process, but also to carry with it the value of the digit (0-9), data which can be used by the receiving process.

To make these data items available to the process, they must be named in the input symbols within parentheses. By doing so the data is available during the transition in action. If the data is not explicitly stored it will not be accessible in another transition afterwards.

One item of data is separated from the others by commas. Examples of the data reception from inputs are shown in Figures D-20, D-21 and D-22.

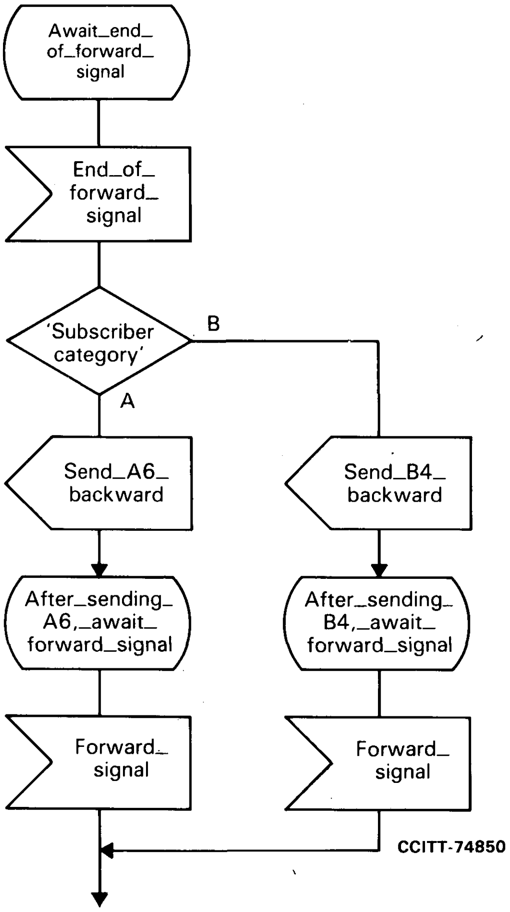


FIGURE D-18

Example of part of an SDL diagram before merging states

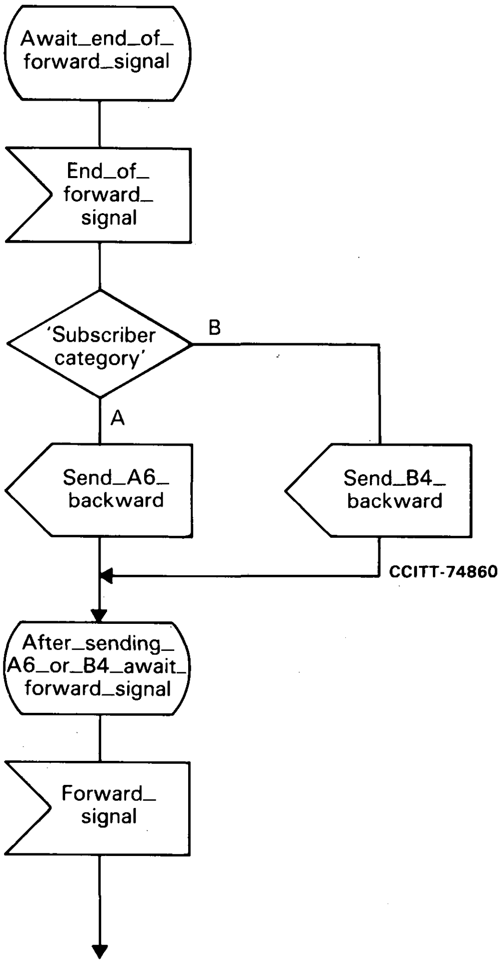


FIGURE D-19

Example of a part of an SDL diagram with merged states

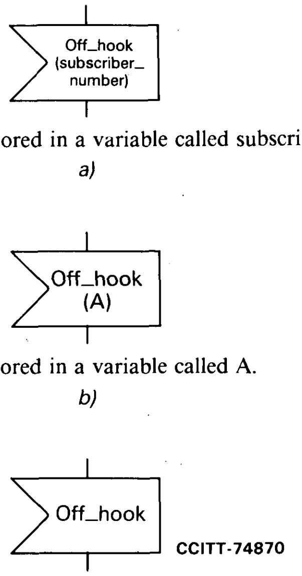
Named data becomes available to the receiving process when the input is interpreted.

Figure D-20 shows reception of the signal *off_hook*. The signal *off_hook* has associated data (subscriber_number) with the value 9269. This signal may be received in three ways as shown in a-c of the figure.

Figure D-22 shows how to send and receive several data items in one signal. Each item must be separated from the next by a comma. In Figure D-22 c) we show how to ignore unwanted data items by leaving a gap in the item list.

Note that in the output signal we could write expressions for E1, E2 or E3, but in the input signal we must use variables to receive the values sent.

In SDL it is unnecessary to draw input symbols to represent signals whose arrival would necessitate a null transition (i.e. transition containing no actions and which leads back to the same state). The convention is that for any signal not shown in an explicit input symbol at a particular state, there exists at that state an implicit input symbol and null transition. By this convention, the two diagrams shown in Figure D-23 are logically equivalent and either can be used.



Note — The value 9269 is stored in a variable called subscriber_number.

a)

Note — The value 9269 is stored in a variable called A.

b)

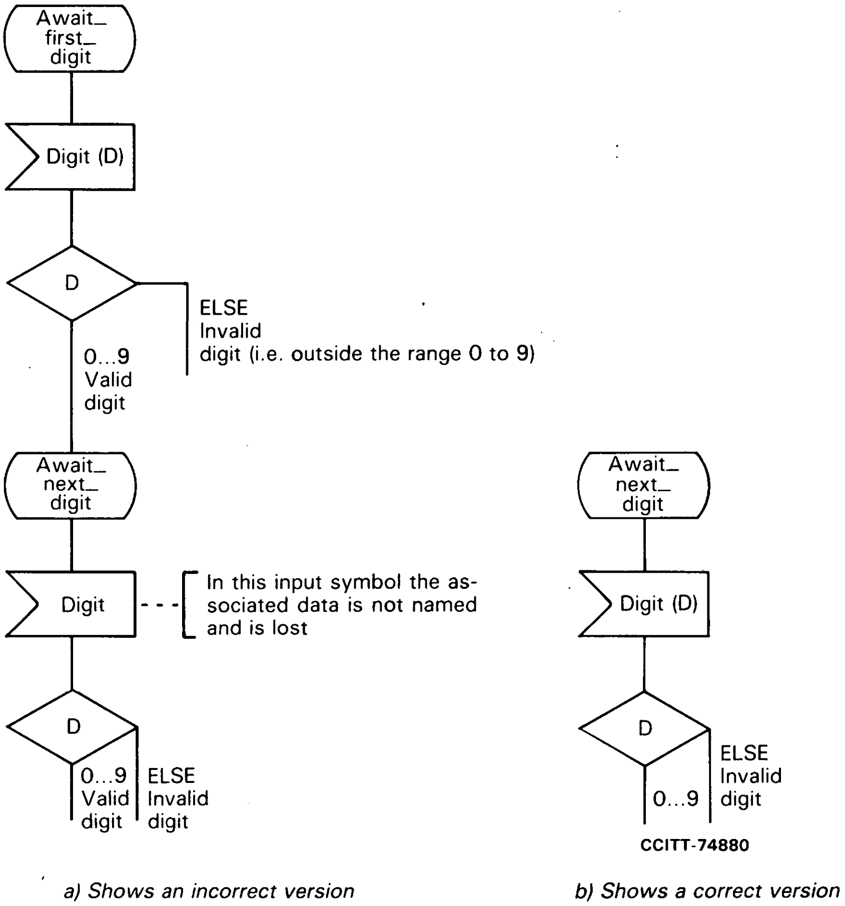
Note 1 — There is no data name and the value 9269 is lost and is not available to the receiving process.

Note 2 — The signal name (off_hook) must correspond to the output signal name but data names can be chosen either to correspond or not.

c)

FIGURE D-20

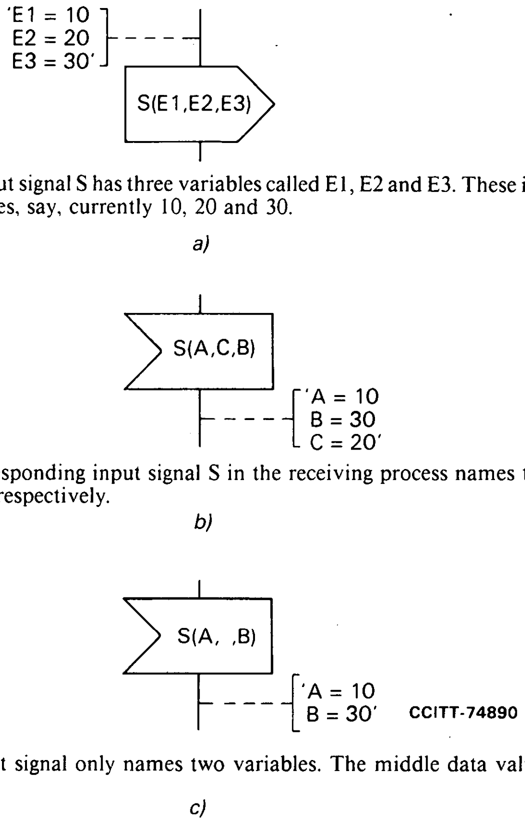
Example of data reception in a process



Note 1 — In a) the second decision will use the value of D obtained from the first digit.
Note 2 — In b) the value of D will be that of the current digit. Values of previous digits will be overwritten.
Note 3 — As D is declared as a formal parameter to digit, the value of D is automatically stored.

FIGURE D-21

Part of a digit-receiving process



Note — The output signal S has three variables called E1, E2 and E3. These items bind to three values, say, currently 10, 20 and 30.

a)

Note — The corresponding input signal S in the receiving process names these items A, C and B respectively.

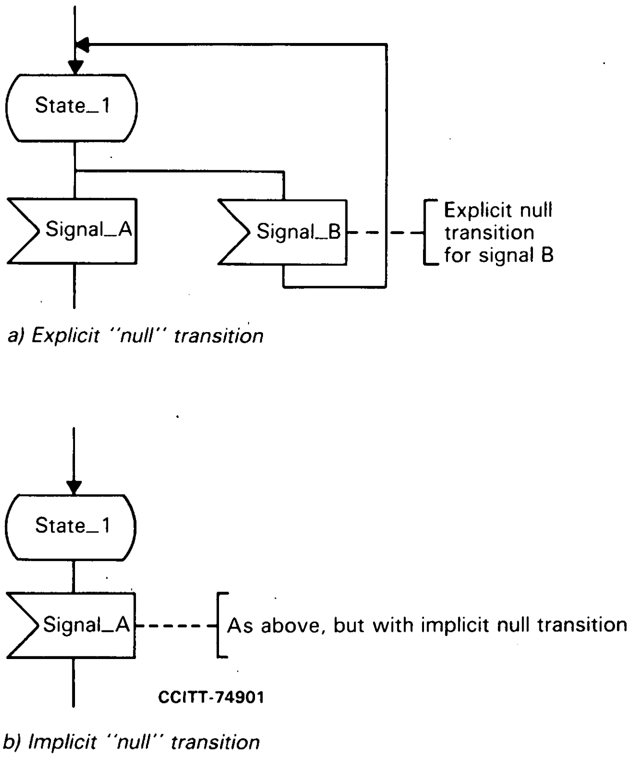
b)

Note — This input signal only names two variables. The middle data value is lost.

c)

FIGURE D-22

A signal with several items of associated data


State_1
Signal_A
Signal_B
Explicit null transition for signal B
State_1
Signal_A
As above, but with implicit null transition
CCITT-74901

a) Explicit "null" transition

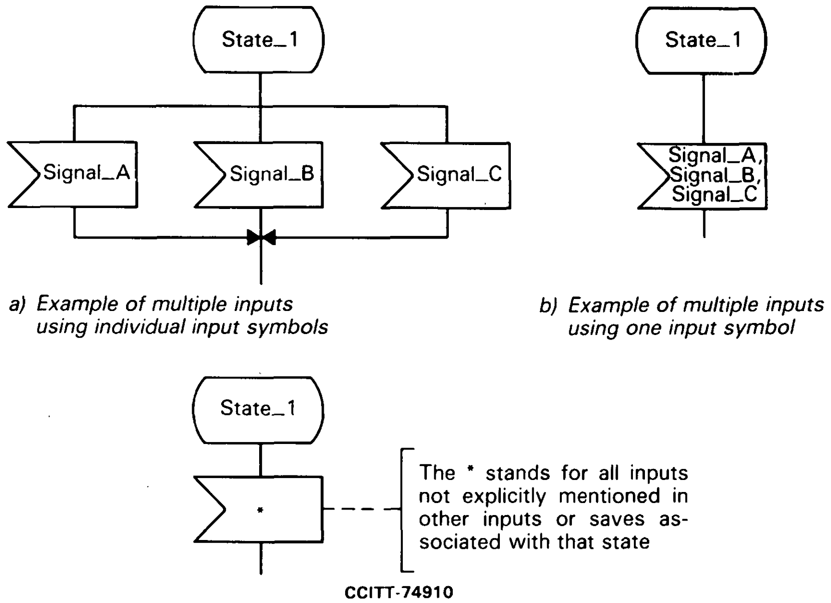
b) Implicit "null" transition

Note — If data is associated with signal B, it is lost in both cases. If however a data name is shown (e.g. Signal_B (x)) in case *a)*, the data value is kept. In this case the two cases are no longer identical.

FIGURE D-23

Explicit and implicit representation of a "null" transition

Where a number of inputs lead to the same transition, all the relevant signal names may be placed within one input symbol. Figure D-24 illustrates this point and the diagrams in the figure are logically equivalent. If all the signal names are mentioned they must be separated by commas.



a) Example of multiple inputs using individual input symbols

b) Example of multiple inputs using one input symbol

The * stands for all inputs not explicitly mentioned in other inputs or saves associated with that state

c) Example of multiple inputs using asterisk notation

FIGURE D-24

Alternative representation of multiple inputs

D.4.3.4.2.2 *Implicit queuing mechanism*

One or more signals may be waiting for consumption when a process reaches a new state. This means that signals must be queued in some way, or they will be lost. When a signal arrives to the destination block it is given to the input queue of the receiving process. The SDL semantics define a first-in first-out conceptual queuing mechanism for each process where signals are considered for consumption by a process in the order of their arrival at that process. When the process reaches a state, it is given one and only one signal from the queue. This means that if the queue is not empty, the process consumes the first signal from the queue. If the queue is empty, the process waits in the state until a signal arrives to the queue, whereafter it is consumed by the process.

Figure D-25 uses the conceptual queue to explain the operation of the depicted SDL process where transition times are non-zero. It should be noted that:

- The save concept is not used and therefore the signals are consumed in the order in which they arrive.
- The sequence of signal arrival is important. Had "C" arrived before "B" in the transition between State 1 and State 2, the state sequence would have been 1, 2, 3 instead of 1, 2, 4.
- Because the queue is not empty when the process arrives at both State 2 and State 4, the process does not wait at either of the states.
- It is not possible to assign any priority to a signal.

If transition times are zero, then every signal will be consumed at the time it arrives at a process, unless the save operation is used (§ D.4.3.4.3).

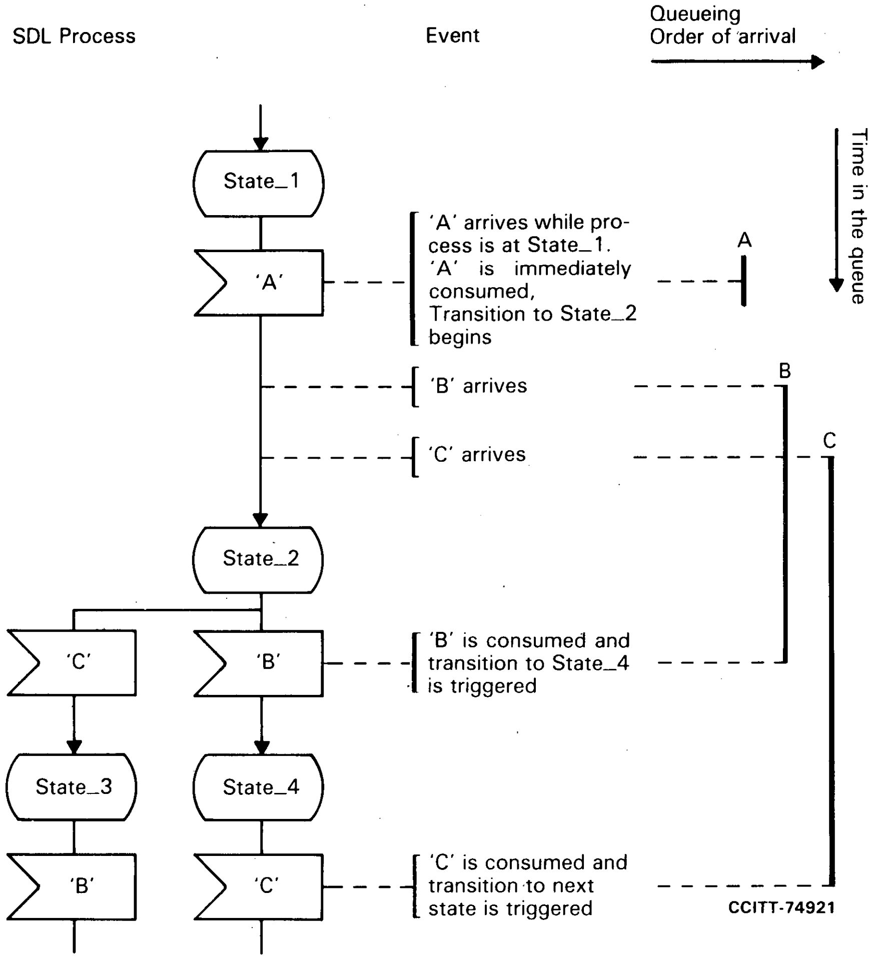

SDL Process
Event
Queueing
Order of arrival
Time in the queue
State_1
'A'
'A' arrives while process is at State_1. 'A' is immediately consumed, Transition to State_2 begins
A
'B' arrives
B
'C' arrives
C
State_2
'C'
'B'
'B' is consumed and transition to State_4 is triggered
State_3
State_4
'B'
'C'
'C' is consumed and transition to next state is triggered
CCITT-74921

FIGURE D-25

Example of the operation of the implicit queueing mechanism

D.4.3.4.2.3 *Reception of signals not normally occurring*

In each state all possible signals must be shown implicitly or explicitly. Exceptions (unexpected but theoretically possible signals, undefined signals or logically wrong signals at that place etc.) can occur in almost all states. Normally the author does not show such possibilities, with the effect that such a signal will be discarded if it arrives. If the author however wants to include exceptions in his diagram *all* states must be expanded with an extra input.

Another possibility is to make use of multiple appearances of a state (see § D.6.3.6.5.1) together with the "all" symbol (*) (§ D.6.3.6.21). For example if the signal A_ON-HOOK can be received in all states and if the subsequent actions are identical, the method shown in Figure D-26 might be used.

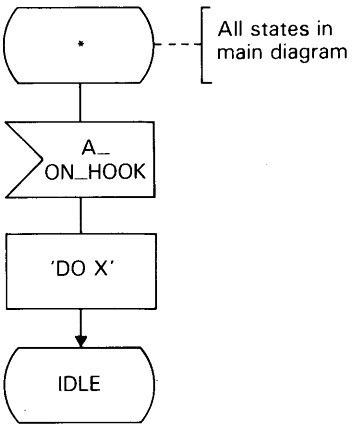
D.4.3.4.2.4 *Simultaneous arrival of signals*

The Recommendation Z.101 covers the possibility that signals can arrive simultaneously at a process and states that they will be ordered arbitrarily.

If a user designs a process which may get simultaneous signals, he should take care that the order of arrival cannot upset the desired operation of the process.

SDL do not recommend priority of signals, i.e. that simultaneous arrival of signals means that one is chosen indeterministicly.

If several signals are available when the process enters a state, only one signal is presented to the process and thus recognized as an input. The SDL semantics implies that the other signals are in fact retained.



CCITT-74930

FIGURE D-26

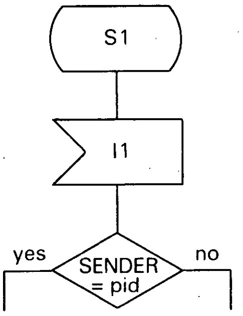
Example of handling of signals to appear in several states

D.4.3.4.2.5 *External versus internal inputs*

An external signal is a signal between processes of different functional blocks; an internal signal is a signal between processes in the same functional block. Semantically there is no diffrence between external and internal signals. In the previous recommendations of SDL, however, a distinction has been made syntactically. These differences in syntax have been removed, and today internal and external signals have identical syntaxes. The previous GR syntax for an internal signal (two vertical lines in the symbol) is no longer used. Old diagrams using internal signals are allowed but the internal signals are interpreted in the same way as external signals.

D.4.3.4.2.6 *Sender identification*

Each signal carries with it the process instance identifier of the sending process. When a signal is received, a process variable called SENDER takes on the value of the sending process's process instance identifier that is contained in the signal. Figure D-27 shows an example of how this may be used.



CCITT-74940

FIGURE D-27

Use of SENDER variable

D.4.3.4.3 *Saves*

The save concept allows the consumption of a signal to be delayed until one or more other signals, which arrive subsequently, have been consumed. As discussed in § D.4.3.4.2 unless the save concept is used, signals are consumed in the order in which they arrive.

The concept of save can be used to simplify processes in cases where the relative arrival order of some signals are not important and the actual arrival order is indeterministic.

At every state, every signal is treated in one of the following ways:

- it is shown as an input symbol or;
- it is shown as a save symbol or;
- it is covered by an implicit input leading to an implicit null transition.

The operation of the implicit queuing mechanism, introduced in § D.4.3.4.2, also applies to the save concept. On arrival, signals enter the queue and when the process reaches a state, the signals in the queue are reviewed one at a time and in the order in which they arrived. A signal covered by an explicit, or implicit, input symbol is consumed and the related transition executed. A signal shown in a save symbol is not consumed and remains in the queue in the same sequential position and the next signal in the queue is considered.

Figure D-28 shows an example of an SDL process which incorporates a save symbol. It should be noted that signals S and R are consumed in the order R, S – the reverse order to that in which they were received. A single save symbol can save a signal only while the process is at the state where the symbol appears, and saves it for the duration of the transition to the next state. At the next state, the signal will be consumed via an explicit or implicit input (as shown in Figure D-28, unless either the save symbol with the signal name is repeated, or there happens to be another saved signal available for consumption ahead of it in the implicit queue (as shown in Figure D-29).

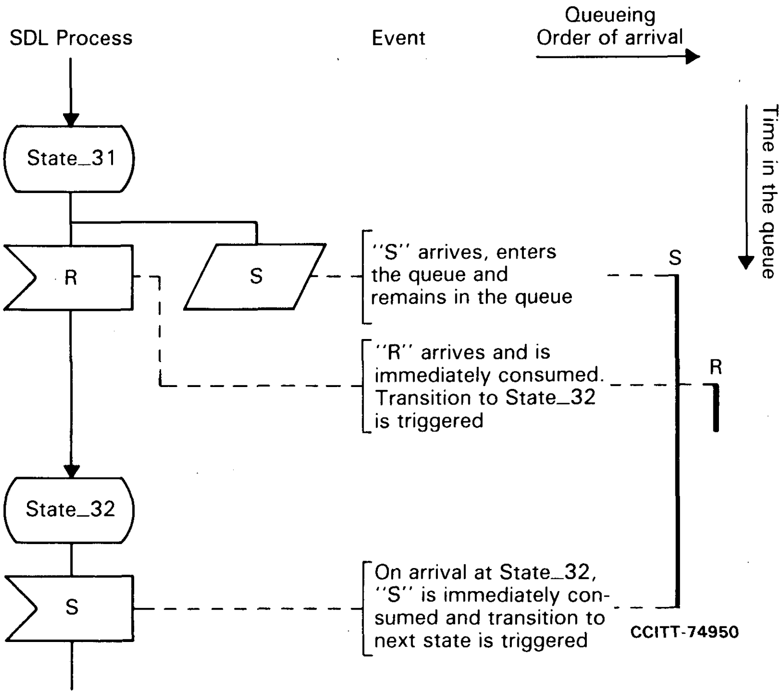

SDL Process
Event
Queueing
Order of arrival
Time in the queue
State_31
R
S
"S" arrives, enters the queue and remains in the queue
S
"R" arrives and is immediately consumed. Transition to State_32 is triggered
R
State_32
S
On arrival at State_32, "S" is immediately consumed and transition to next state is triggered
CCITT-74950

FIGURE D-28

Example of an SDL diagram with save symbol showing operation of implicit queueing mechanism

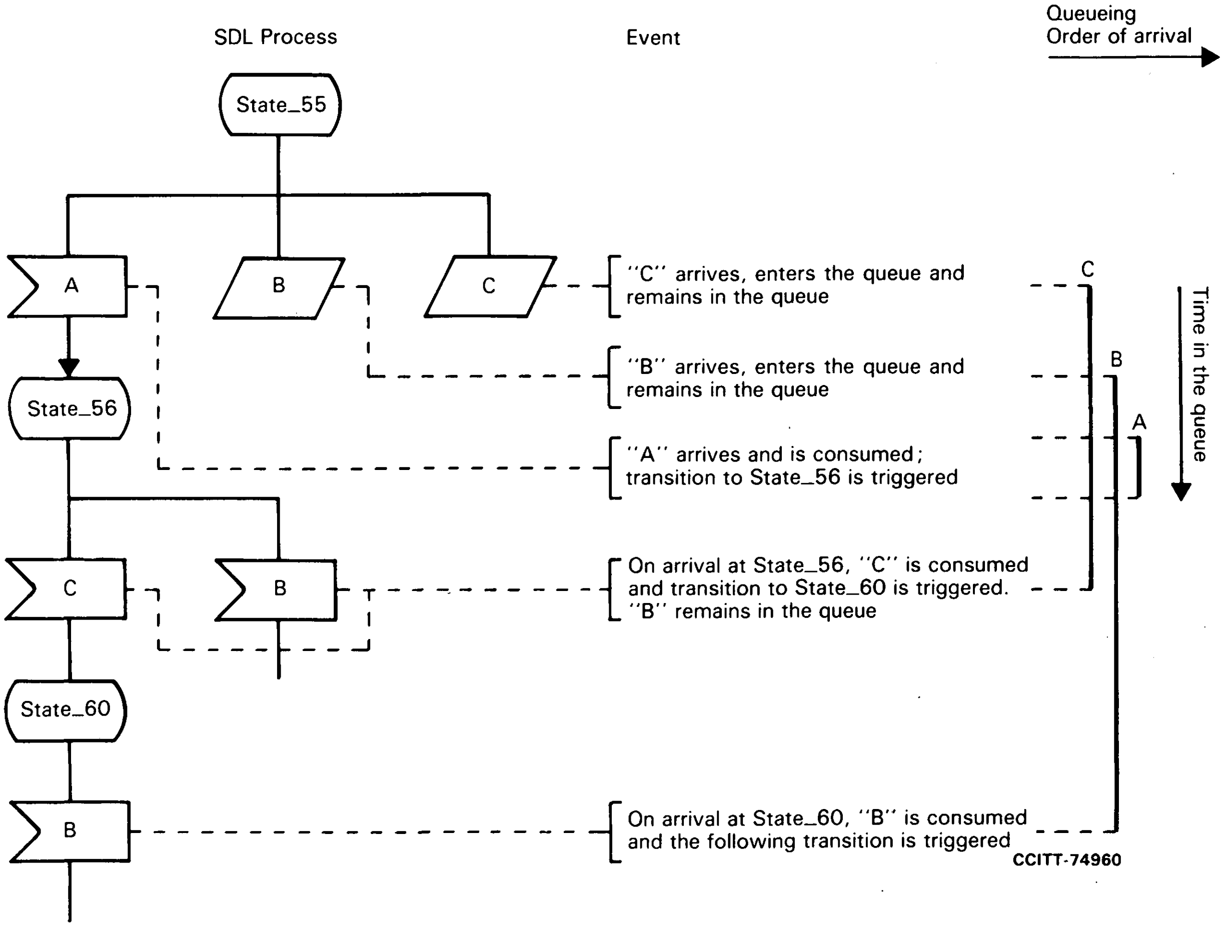

SDL Process
Event
Queueing
Order of arrival
State_55
A
B
C
"C" arrives, enters the queue and remains in the queue
"B" arrives, enters the queue and remains in the queue
State_56
"A" arrives and is consumed; transition to State_56 is triggered
C
B
On arrival at State_56, "C" is consumed and transition to State_60 is triggered. "B" remains in the queue
State_60
B
On arrival at State_60, "B" is consumed and the following transition is triggered
C
B
A
Time in the queue
CCITT-74960

FIGURE D-29

Second example of the use of the save symbol

A saved signal becomes available to a process only through a corresponding input symbol (explicit or implicit). In particular no questions about a saved signal may be asked in a decision prior to its recognition as an input; nor is its associated data available.

At a state where more than one signal is to be saved, either a save symbol may be given for each signal, or they may all be shown within one save symbol.

If several signals are to be saved, the semantics of the save symbol implies that the order of their arrival is preserved.

A third example of the use of the save is given in Figure D-30 with a description of the operation of the conceptual queuing mechanism in Figure D-31.

The save can be used to simplify diagrams. For example by saving a signal it is possible to avoid receiving it and having to store its associated data until the next state.

Although the save can be used at every level of description, at lower levels it may be necessary to describe the actual mechanism which implements the save concept.

If the save is not used with care, the queue of saved signals may grow very big or may remember signals too long so that an old signal is consumed where a fresh one was needed.

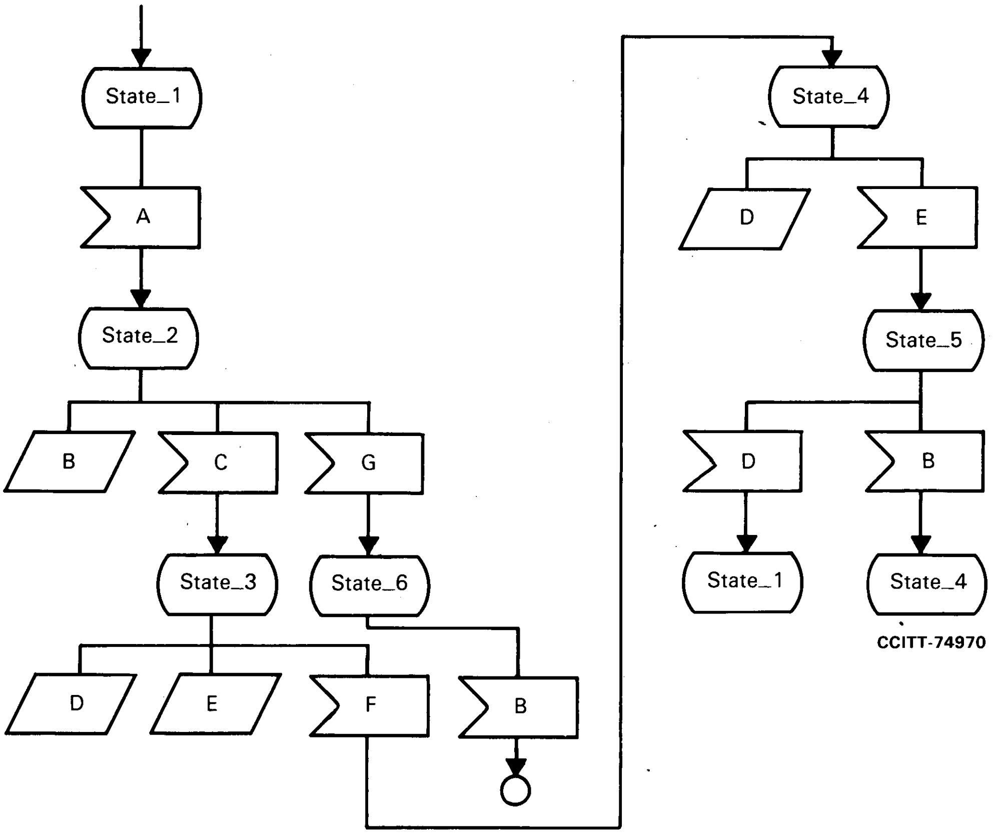
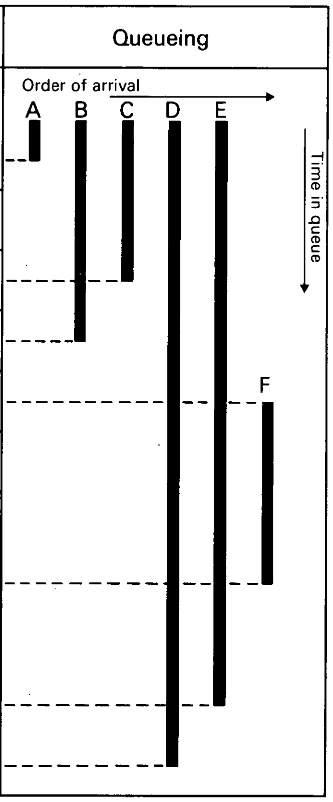


FIGURE D-30

Example of a more complex SDL diagram with numerous inputs and saves

Current state	Event	Queueing
State 1	(Process arrives at State 1 with signals "A", "B", "C", "D", "E" in queue) The first signal in queue, "A", is consumed (explicit input) and transition to State 2 is triggered.	
State 2	The first signal in queue, "B", appears in a save symbol and remains in queue.	
State 2	The second signal, "C", is consumed (explicit input) and transition to State 3 is triggered.	
State 3	The first signal in queue, "B", is consumed (implicit input) and null transition is triggered.	
	"F" arrives and enters queue.	
State 3	(On reaching State 3 again) the first signal in queue, "D", appears in save symbol and remains in queue.	
State 3	The second signal, "E", appears in save symbol and remains in queue.	
State 3	The third signal, "F", in consumed (explicit input) and transition to State 4 is triggered.	
State 4	The first signal in queue, "D", appears in a save symbol and remains in queue.	
State 4	The second signal, "E", is consumed (explicit input) and transition to State 5 is triggered.	
State 5	The first (and only) signal in queue, "D", is consumed (explicit input) and transition to State 1 is triggered.	

CCITT-39510

FIGURE D-31

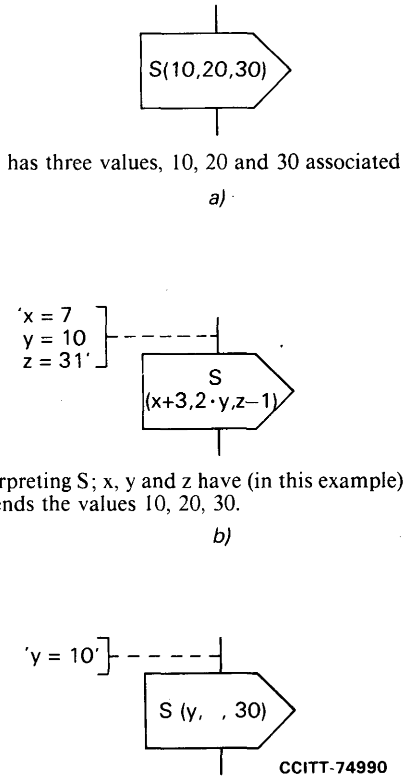
Operation of conceptual queueing mechanism

D.4.3.4.4 *Outputs*

An output symbol represents the sending of a signal from one process to another. Because control over the reception and consumption of the signal is associated with the receiving process (see § D.4.3.4.2) the semantics directly relating to the output symbol is relatively simple. From the point of view of the sending process an output can often be regarded as an instantaneous action which, once completed, has no further direct effect on the sending process, which will not be directly aware of the fate of the signal.

If the author has difficulty in deciding whether an action should be represented as an output or a task, § D.4.3.4.6 should be consulted. An output action represents the sending of a signal and the association of data values if any. Data values may be associated with an output signal by placing values in parentheses, or by placing expressions having values in parentheses (see Figure D-32 and Figure D-22).

Each output must be directed to a specific process instance. Since it is usually impossible to know the process instance identifier for any process at the time a specification or description is produced, the normal method of addressing signals is to use a variable or expression in the TO symbol or keyword. Figures D-33, D-34 and D-35 show some examples. In Figure D-33 the process parameter output_to is given the value of a process instance identifier at the time the process is created. Output_to is then used within the process as the link between this process and its connected process. In designing the system, care should be taken to ensure that the type of the process indicated by output_to is able to receive the signals being sent. In Figure D-34 the built-in process variable is used to send a signal back to the process that sent the signal that was just received. In Figure D-35, the signal is addressed to the process's most recently created offspring process.



Note – Signal S has three values, 10, 20 and 30 associated with it.

a)

Note – On interpreting S; x, y and z have (in this example) the values 7, 10, 31 respectively. S sends the values 10, 20, 30.

b)

Note – On interpreting S, y has (in this example) the value 10. S sends the values 10 and an undefined value and 30.

c)

FIGURE D-32

Output with associated data

```
PROCESS X; (output_to PID);
        .
        .
        .
OUTPUT SIGNAL TO output_to;
        .
        .
        .
```

FIGURE D-33

Addressing signals using formal parameters

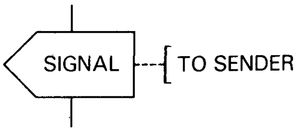
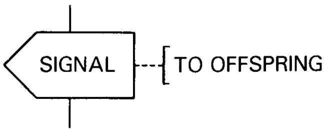


FIGURE D-34

Addressing signals back to SENDER



CCITT-75000

FIGURE D-35

Addressing a signal to an offspring process

D.4.3.4.5 *Enabling condition and continuous signals*

Enabling conditions allow conditional reception of signals based on the specified enabling condition. If the condition is true, the signal is received and the transition is interpreted. If the condition is false, the signal is saved and the process remains in the state until either another signal arrives or until the condition changes from being false to being true. This can be illustrated by the example in Figure D-36. The process starts in state S1. Over in process P2, the value of X is 9. When signal B arrives, the value of X is compared with 10. Note that since this is a value in another process, the import operation is required to determine its present value. Since X is not 10, the process remains in S1 until either another input is received or until P2 sends a new value of X. In this example, A arrives and causes a transition to S2. During the transition, X changed to 11 and thus now the condition attached to signal B, in state S2, is true. Since B is the first signal in the queue, the transition following it is performed and the process ends in state S3.

Some important attributes of enabling conditions are:

1) The enabling condition is tested only when the process arrives at a state. Thus if the value of X had changed from 9 to 11 and then to 12 while executing the transition following the reception of A, the process would have remained in S2.

2) Enabling conditions can only be based on expressions containing IMPORTED or local variables and not VIEWED variables. The reason for this is based on the underlying mechanism to implement the enabling condition. The enabling condition works using signals to determine when a value in the enabling condition has changed. For local variables, the condition is checked before entering the state. Since the values are local to the process, they cannot change while the process is in the state. Thus the condition need not be continuously monitored. For IMPORTED variables, SDL uses signals for changes to IMPORTED variables as a trigger to know when to check the enabling condition. VIEWED variables access the value of the variable directly. Since there is no sending of signals, once the process entered a state, there would be no way for the process to check the value of the variable.

3) While it is possible to use more than one enabling condition per state, it is not allowed that more than one enabling condition is used for the same signal. Thus the condition shown in Figure D-37 is not allowed. If multiple conditions are required for a given signal, they can be combined in one Boolean expression as shown in Figure D-38. This eliminates the ambiguity that occurs when two or more conditions are true at the same time. In Figure D-38 the user has decided that the condition on X has a higher priority than the condition on Y. So if both were true, the C1 path would have been followed. Once again the reason for this is that the underlying SDL system does not have priorities attached to signals.

Continuous signals have the same basic properties as the enabling condition except that no signal is attached to it. Thus when entering the state with no signals in the queue which can cause a transition, the continuous signals are checked and if one is true, the transition following it is performed. This can be illustrated by the example in Figure D-39. Initially, the process is in state S1, and the value of X is 9. When signal A arrives, it causes the transition to S2. During the transition, X changed to 11. Since there were no other signals in the queue, the transition to S3 is performed.

Some important attributes of continuous signals are:

1) as with enabling conditions, the value of the condition is checked only on arriving at a state;

2) again, only IMPORTED and local variables can be used in the continuous signals;

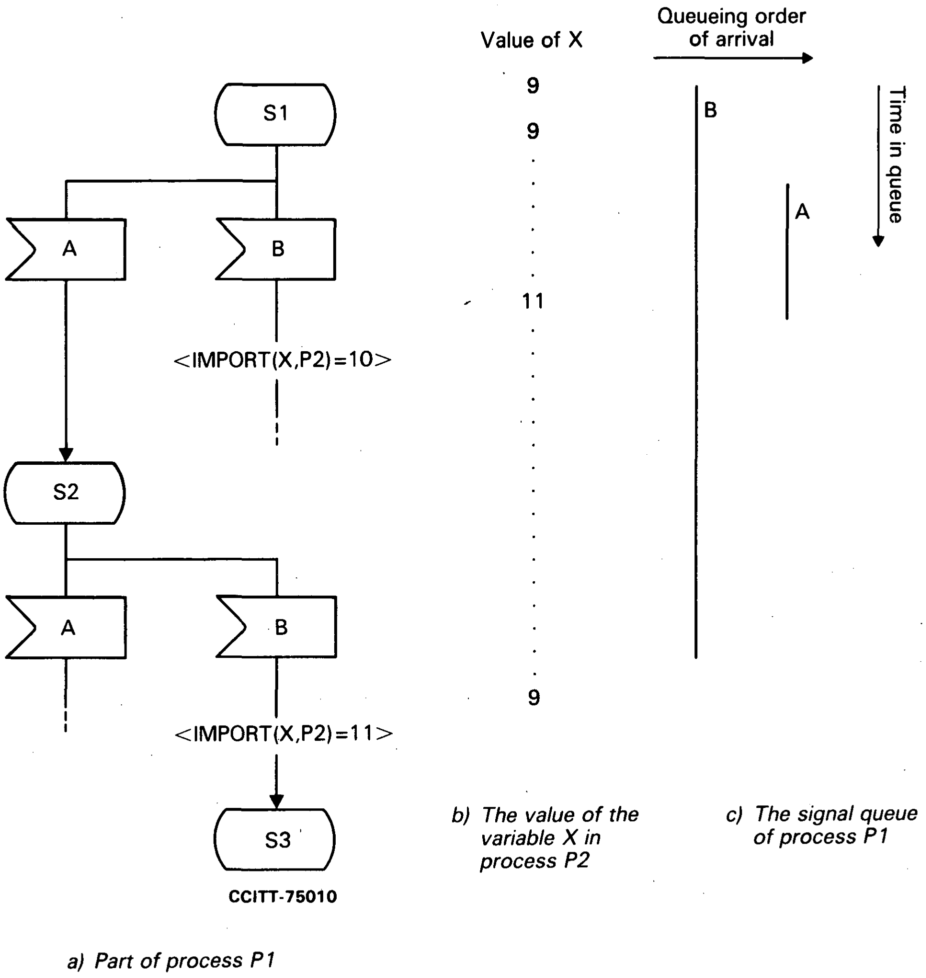
3) multiple continuous signals are allowed for each state. When more than one continuous signal is attached to a state, the continuous signal with the lowest priority number will be tested first. No two continuous signals may have the same priority number. In all cases, the priority of the continuous signal is lower than that of any other signal. The use of priorities on continuous signals is the only place where priorities are used in SDL. Once again, this is caused by the underlying system of SDL; however, the way that continuous signals are modelled in SDL by using the signals sent when importing variables, allows priorities to be allowed for continuous signals and in fact makes it necessary to prevent ambiguity, when more than one continuous signal is present. This is illustrated in Figure D-40. Initially, the process is in state S1, and its local variables have the values of 10 for X and 11 for Y. Since both continuous signals are true, the one with the higher priority (lower priority number) is chosen and the transition to S2 is performed. In S2, the condition on Y is no longer true and so even though the priority of the continuous signal for X is lower than the one for Y, the transition following it is performed and the process arrives at state S3.

D.4.3.4.6 *Task*

A task is used to represent operations on data performed on a transition, with the exception of output signal generation and decisions. Data can only be changed by the process which owns the data in question. The nature of the tasks appearing in any diagram will be determined by the nature of the process being described and the level of detail required. Examples of action which may be covered by tasks are:

a) hardware actions such as send_busy_tone;

b) the manipulation of data.



a) Part of process P1

b) The value of the variable X in process P2

c) The signal queue of process P1

FIGURE D-36

Enabling condition

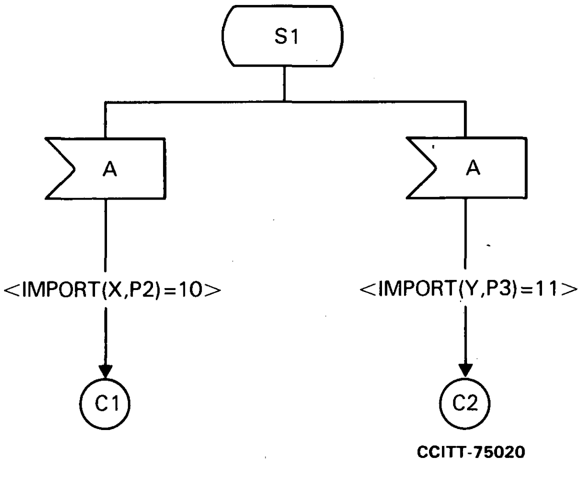


FIGURE D-37

Illegal enabling condition

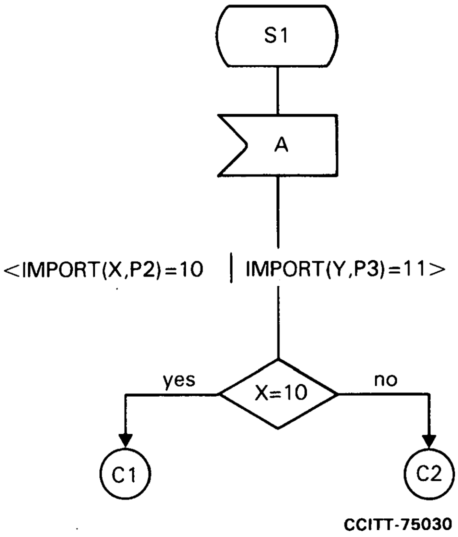


FIGURE D-38

Correct solution for Figure D-37

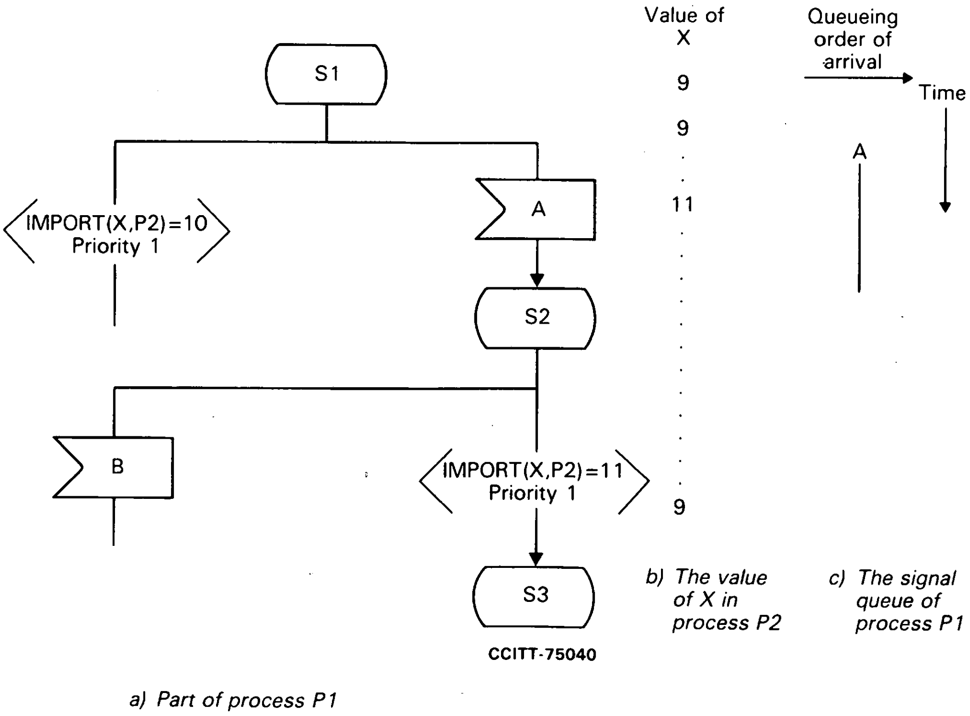


FIGURE D-39

Continuous signals

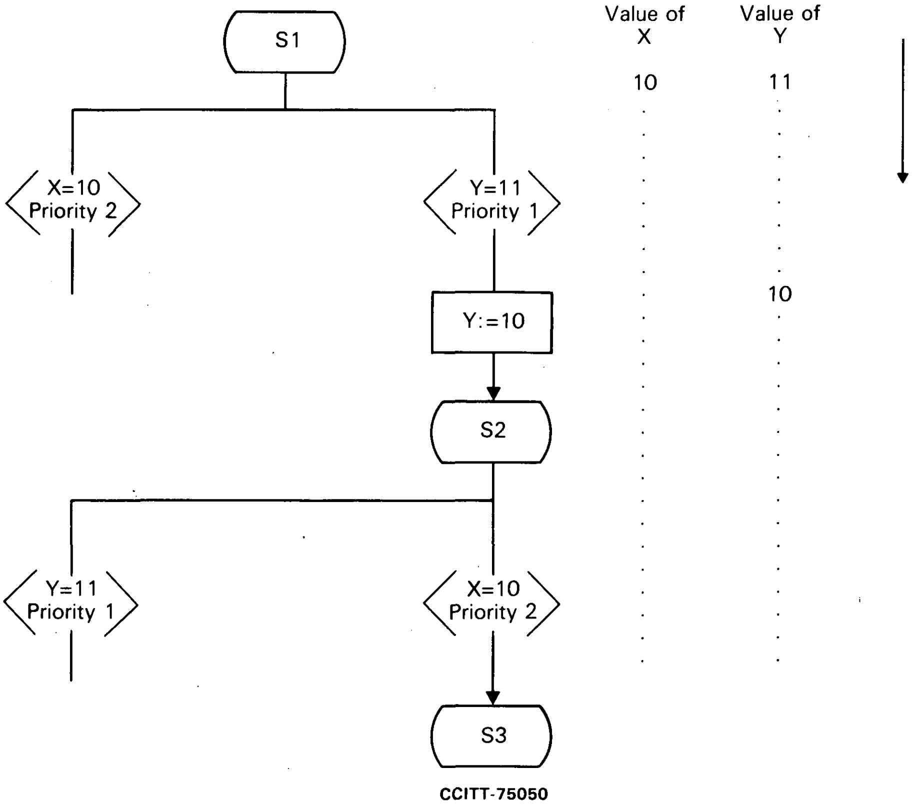


FIGURE D-40

Continuous signals with priority

SDL users may occasionally have difficulty in deciding whether some aspects of the system being defined should be represented by a task or a separate process. Consider the process shown in Figure D-41; should "connect_switch_path" be represented as a task or as a separate process? If a separate switch path control process has not been identified, then the task symbol would be appropriate (see Figure D-41 a)). If a separate switch path control process is identified, then signals communicating with the control process must be used (see Figure D-41 b)).

D.4.3.4.7 *Decisions*

A decision is an action within a transition which asks a question regarding the value of data items available to the process at the instant of executing the action. The process proceeds to one of the two or more paths following the decision according to the answer. SDL diagram authors should ensure that processes are defined so that they cannot attempt to execute decisions for which the answer (or data) is not available; such decisions would make the diagram invalid and could cause considerable confusion.

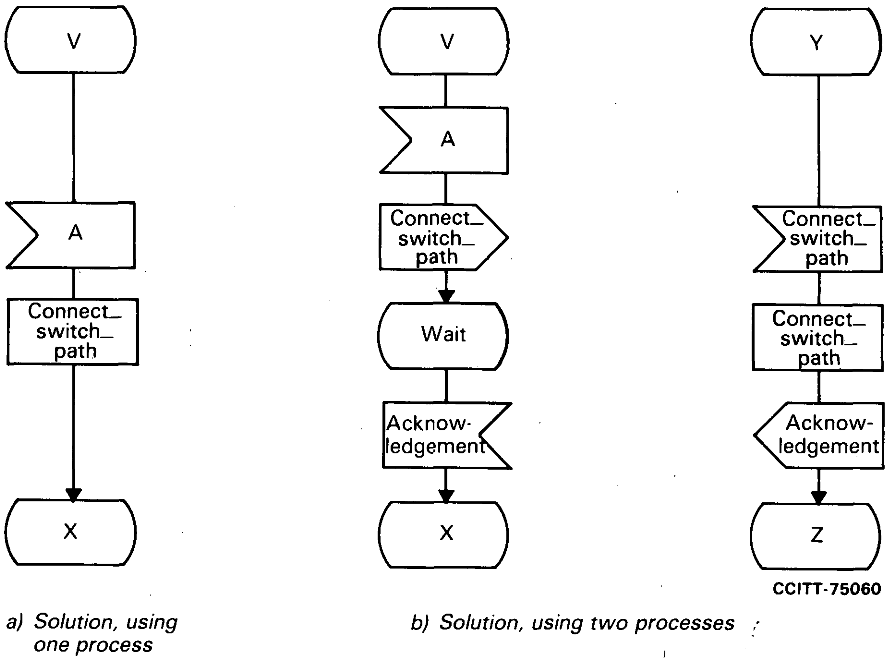


FIGURE D-41

Two possible solutions for 'connect_switch_path'

Some examples of use of decisions are shown in Figure D-42. The variable X in the examples may have the values 1, 2 or 3.

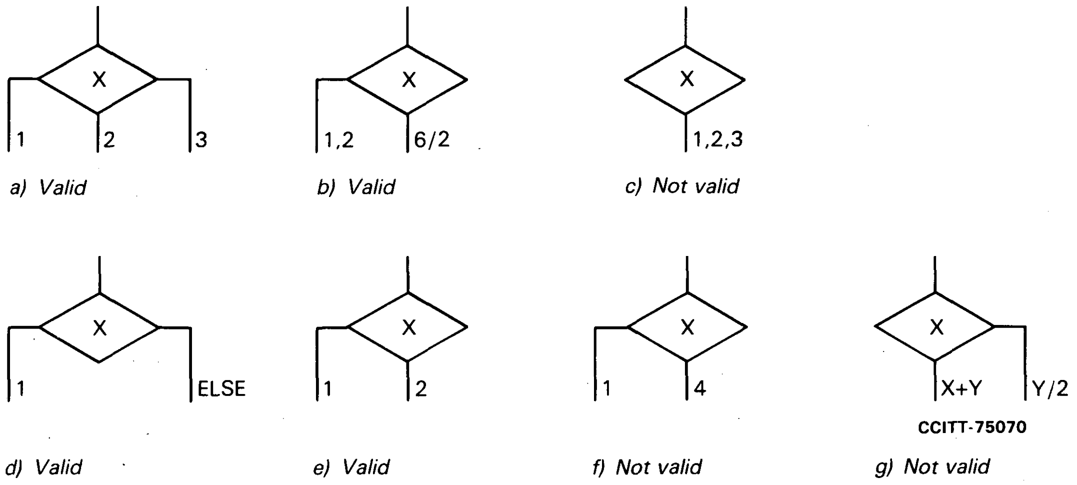


FIGURE D-42

Examples of use of decision

Alternative c) is not valid as the decision is not followed by at least two paths. The e) alternative is valid as in SDL it is a convention that if the choice of alternative is undecidable it will be interpreted as an implicit alternative leading directly to a stop symbol. Alternative f) is not valid as one of the alternatives is out of range. The g) alternative is not valid as the conditions on the alternatives contain variables and thus they are not possible statically to evaluate.

If an answer is leading back to the decision in the same transition some actions must be performed which influence the question in the decision. However, even with this rule deadlocks might be created as shown in Figure D-43. Care should therefore always be taken when having answers leading back to a decision in the same transition.

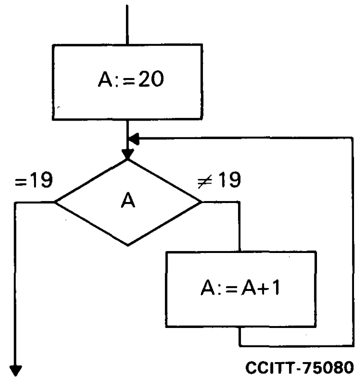


FIGURE D-43

Example of a legal use of a decision which creates a deadlock

Decisions can be made using any values currently available to the process. This includes:

- data stored by a task;
- data received by an input;
- data set when the process was created (data passed as an actual parameter);
- shared data.

The question in a decision may ask for the value of an expression and the expression can include constants and any of the above kinds of data values.

D.4.3.5 *Procedures*

Procedures are one technique for adding structure to processes. In may ways, procedures in SDL are similar to procedures in programming languages. The user is advised, however, that there are some major differences and that caution is advised. With that in mind, it is possible to begin a discussion on procedures.

Syntactically, a procedure is very similar to a process. The procedure graph has a procedure start node which differs slightly from the process start node and it has a return node instead of a stop node. Semantically, it is quite different, although identical nodes of processes and procedures may have the same meaning.

Procedure calls may occur wherever a task node is allowed in either a process or procedure graph. In some sense, a procedure can be interpreted as a task with the following exceptions:

1) A procedure can contain states and thus will receive signals. Thus a part of the procedure definition provides an additional-save-set which lists the signals that are to be saved when the procedure is called. Since procedures may be defined at the system level, it is important to list all the signals that must be saved in any of the processes that call the procedure.

2) A procedure can send output signals, the originating process instance identifier is the process instance identifier of the process which called the procedure.

3) A procedure has its own local variables and does not have access to any of the process variables unless the variables were passed to the procedure as IN/OUT parameters.

4) To allow the procedure to be called in a number of places, SIGNAL parameters are allowed to create synonyms for the actual signal names for the calling process and the signal names used in the procedure.

When a procedure is called, the procedure environment is created and the procedure begins to be interpreted. Interpretation of the procedure continues until the return node is reached. While the procedure is being interpreted all signals addressed to the process are either saved or received by the procedure. The procedure does not have its own input queue, but uses the input queue of the process that called it. This is why it is important to have the proper additional-save-set for a procedure. The additional save-set is constructed dynamically when the procedure is called, and consists of all valid input signals of the calling procedure that are not given as parameters to the call. If the signal is not in either that set or in the normal save-signal-set for the state node in the procedure graph, the signal will be discarded.

An example of the use of procedures can be found in § D.9. The example shows the use of procedure for protocol specifications.

D.4.3.6 *Expressing time in SDL*

There are several places in SDL where the time representation may be needed:

a) the activation of a process at a certain time (absolute time or relative time);

b) the time consumed in executing transition actions;

c) the time taken to transfer a signal from one process to another process, which may involve a channel.

D.4.3.6.1 *Timers and timeouts*

The need to measure time and request timeouts in a system is met by timers and a set of operations performed upon them.

"Timer" is a predefined type, and all use of timers in a process must be declared. The operations "SET" and "RESET" may be performed on timers. The SET operation requests a timeout to occur at a specified time, and the RESET operation cancels any requested timeout. (Note that a SET operation includes implicitly a RESET operation of any not expired timings on the Timer.)

```
DCL T1 Timer;

SET (NOW + 20 s, T1);

RESET (T1).
```

FIGURE D-44

Example of declaration and operation of a timer

In the SET operation an absolute time must be specified. Relative time is transformed to an absolute time value by adding the operation "NOW", which yields the current time. For example, 20 seconds from now is expressed: NOW + 20 s.

Timers are only monitored when the process is in a state. A signal is put into the input queue of the process when a timer is checked and found equal to, or less than, the current time. The timer is then implicitly reset.

The timeout signal has the same name as the timer and carries no data values.

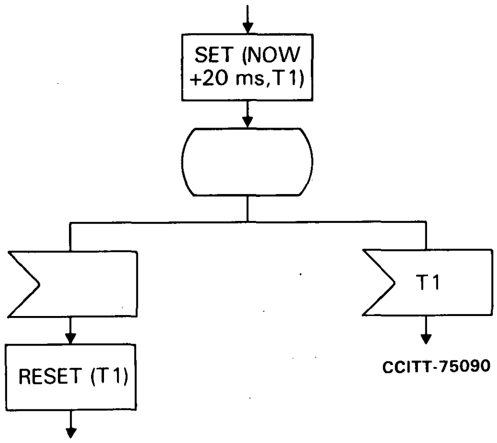


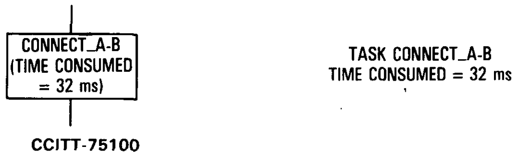
FIGURE D-45

Example of use of timers

D.4.3.6.2 *Specifying time consumed by actions*

When using SDL to simulate the performance of systems the time consumed in transitions must be specified. The current SDL does not cater for this, however the following guidelines are provided.

In such cases a time value (the length of time consumed in executing a given action) is associated, as a parameter, to the name of the action. This optional parameter can be identified by a keyword (e.g. TIME CONSUMED = time value).



a) An action which takes 32 ms is represented by a task symbol which includes time consumption

b) The same action represented in SDL/PR

FIGURE D-46

Indication of transition time

The indication of time as a parameter associated with the name, instead of a comment, is because the comment should provide explanation for human beings and does not have to be understood by a machine.

D.4.3.6.3 *Signal transfer time*

The time elapsing from the generation (output) to the consumption (input) of a signal is very complex to express because it depends on several factors.

There may be an absolute time consumed by the action of transporting the signal from one block to another and this part may be expressed (if the channel is explicitly represented) as the sum of times consumed by the actions performed by the channel.

There is a relative time consumption depending on the system load and on the signal handling strategy. Also the time spent in the activation of the receiving process depends on the operating system features and on process priority, system load, concurrent events, etc. This time can only be guessed and is relevant only in system capacity studies.

Another time which cannot be statically determined is the one spent in queues awaiting the consumption of the preceding signals.

Normally, time is not easily representable statically and it is of relevance for simulation and capacity studies only.

However, providing that all actions at the more detailed level of system description have the indication of time consumption and that the description contains a representation of channels (operating system), it is possible to simulate the system and to evaluate it from the capacity view-point with various techniques.

D.4.4 *Text associated with SDL constructs*

The conventions for text associated with SDL constructs as described in the following, apply both to GR and PR.

Text in SDL may either be:

- formal text;
- informal text;
- comments.

Formal text is text that may be formally interpreted. Informal text may only be informally interpreted. Comments are only annotations and may not be interpreted. The meaning of a diagram should be the same with or without comments.

D.4.4.1 *Formal text*

Formal text is represented using the SDL/PR syntax. In SDL/PR the permitted combinations of text are defined in the syntax. In SDL/GR the formal text associated with a graphical construct should either be contained in that symbol or in a connected text extension symbol (see § D.6.3.6.19).

In formal text upper and lower case letters may be freely mixed. There is no difference if a name is represented in upper or lower case. However, a consistent use of upper and lower case letters is recommended.

D.4.4.1.1 *Name*

A name represents the identification of the entity to which it is associated within the context. A name is required for:

BLOCK, PROCESS, SIGNAL, CHANNEL, STATE, SAVE, INPUT, OUTPUT, MACRO, CREATE, START, PROCEDURE and CONNECTOR.

A name must begin with a letter and may contain letters, digits, national characters and underscores. Note that spaces are not allowed.

Examples of legal names are:

CALL_HANDLER
RINGING_TONE

Examples of illegal names are:

START CHARGING	(it contains a space)
1982	(it starts with a digit)

The INPUT name identifies the received SIGNAL and vice versa the OUTPUT name identifies the sent SIGNAL, so that OUTPUT – SIGNAL – INPUT bind the same name. Connectors (JOINs in PR) have an associated name. This name may begin with a digit. Connectors are also peculiar in the fact that they do not have an associated text string.

All keywords in SDL/PR are reserved words, and may not be used as names. All reserved keywords are given in the SDL/PR summary.

D.4.4.1.2 *Formal parameters*

Formal parameters are used in statements or in process and procedure start symbols.

Formal parameters are indicated by the keyword "FPAR".

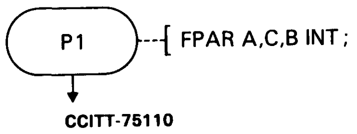


FIGURE D-47

Example of formal parameters

See § D.4.3.5 on the use of formal parameters.

D.4.4.1.3 *Actual parameters*

Actual parameters are used in constructs such as create request, procedure call, output, etc. Actual parameters are delimited by a pair of parentheses.

OUTPUT digits (a,b,c,d);

FIGURE D-48

Example of actual parameters in an output

D.4.4.1.4 *Statements and expressions*

The text in tasks and decisions follow the SDL/PR syntax for statements and expressions.

D.4.4.1.5 *Definitions and declarations*

Signal definitions, data type definitions and variable declarations are always represented by the SDL/PR syntax for those constructs.

D.4.4.2 *Informal text*

Informal text associated with GR symbols or PR statements may only be informally interpreted. The SDL/PR syntax for text strings is used for informal text, i.e. the text is enclosed by a pair of apostrophes.

Examples of text strings are:

'Is subscriber free'

'$a^2 + 2ab + b^2$'

Note that any character is allowed within the text string with the exception of '(apostrophe). If required ' can be represented by a couple ".

Example:

John's house. – ' John"s house'

If only informal text is used throughout the SDL representation the apostrophes may be omitted. If this is the case it should be indicated in the beginning of the SDL representation.

Informal text may have any form suitable to convey information. In this case the writer assumes some common basis of understanding between himself and the reader. Note that the reader can be a machine. In this sense informal text must be sufficiently formal in the reader's understanding in order to be unequivocally understood (Figure D-49).

Examples of informal texts are:

- TASK 'do while x <5.....'
- INPUT x 'contains subs _ identity'
- DECISION 'subscriber busy?'
- (busy):

FIGURE D-49

Examples of informal text

CHILL text associated with SDL statements is one example of "informal" text from an SDL point of view, though it is a very formal way of representing actions.

There is a very clear distinction between informal text and comments.

Comments are introduced to help readers in understanding the representation, their absence or presence does not alter the representation.

The informal text is part of the representation itself, so that we cannot understand it without considering the associated text (both formal and informal).

D.4.4.3 *Comments*

Comments in SDL do not have any formal or informal meaning. They are just introduced to help readers better understand the GR-diagram or PR-text.

In SDL/GR comments may be inserted anywhere in a diagram. Comments are recognized either by being associated with a comment symbol or by being delimited by '/*' and '*/' (see Figure D-50).

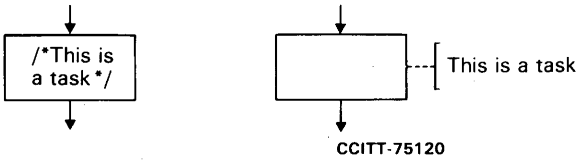


FIGURE D-50

Examples of comments in GR

In PR, comments are either identified by the keyword "COMMENT" or delimited by '/*' and '*/'.

```
TASK A:=2 COMMENT this is a task;
or
TASK A:=2 /* this is a task */;
```

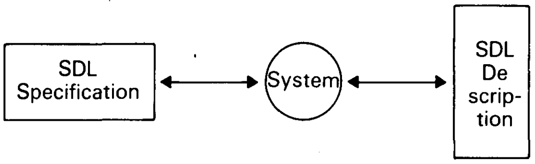
FIGURE D-51

Example of comments in PR

D.4.5 *Undefined situations*

D.4.5.1 *General*

In the lifetime of systems, from initiation to removal from service, failures normally occur. Some of these (and for software systems, most of these) are due to undefined situations in the specification. It is therefore important to check an SDL specification to find as many of these as possible before implementation. Afterwards, when a specification has been implemented and a description has been provided, it should be checked that the description describes the system 100% – not more or less!



CCITT-75130

FIGURE D-52

One to one correlation between specification, system and description

D.4.5.2 *Examples of undefined SDL situations*

D.4.5.2.1 A data object which tries to assign a value outside its range, or a value not belonging to its type. For example, the variable SUBSCRIBER_NO has the type INTEGER_RANGE (100000 : 999999) and tries to assign the value FALSE, 1 000000, 87 or BUSY.

D.4.5.2.2 A signal is sent to a non-existent process.

D.4.5.2.3 A signal sent to a (newly) dead process instance. However, it might not be an error of the system – but in the SDL specification/description it is an undefined situation, which must be treated by the underlying system/simulation program.

D.4.5.2.4 A signal addressed to a non-existent channel.

D.4.5.2.5 An error in the behaviour of the physical system will also be present in the SDL description if a one-to-one correlation has been obtained.

D.4.5.2.6 Directing signals to a channel or to a process without having declared the signals, channel or the process properly.

D.4.5.2.7 A decision for which the answer or data is not available when the process attempts to execute the decision.

D.4.5.2.8 A try to instantiate a process for which the maximum number of instances has already been reached.

D.4.6 *Guidelines on primitive data in SDL*

The advanced data concept is treated in § D.4.7.

D.4.6.1 *General*

In SDL, communication between processes takes place by use of signals, by shared values and by exported/imported values.

A signal is initiated by a process with the use of an output. The signal is a flow of data conveying information to another particular process. If the receiving process recognizes this signal, by means of an input, the data associated with this signal becomes available to this process.

A process can read a data value of another process if it belongs to the same block and if that data is declared as a shared value.

In addition, a process can reveal a data item by declaring it exportable. All other processes with a corresponding import declaration are given copies of the data value upon request.

From the point of view of a particular process, and at a particular time, there are two major and complementary categories of data:

1) data available to this process;
2) data not available to this process. (This second category includes data retained but not yet available to this process in the form of an arriving signal.)

Only available data can be used by a process in performing any of its actions: decisions, tasks or outputs.

When used in a decision or output the data itself, at this level of abstraction, is not modified. A special task, however, can create, store, modify or destroy data.

Examples of treatment of data are shown in Figure D-53 (for a decision), in Figure D-54 (for an output) and in Figure D-55 (for a task).

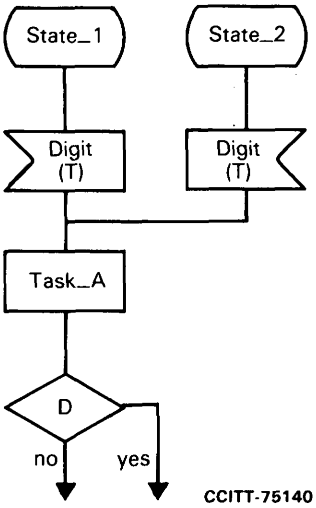
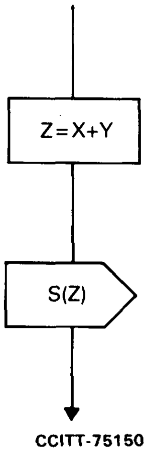


FIGURE D-53

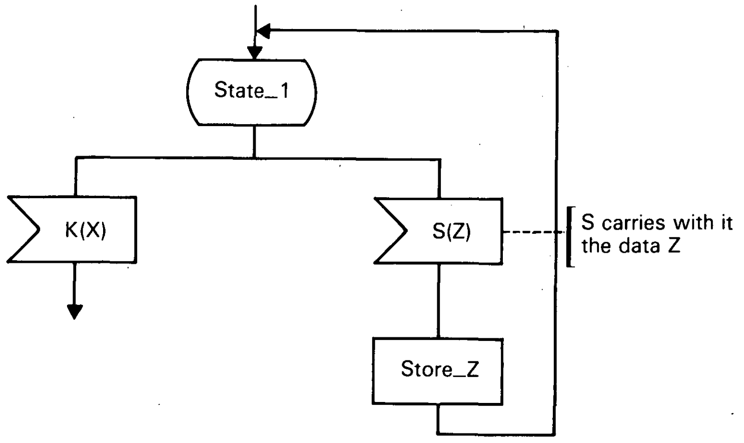
Questioning of data with inputs by means of a decision



Note — S is the signal name. Z is the associated data.

FIGURE D-54

Sending data by means of an output



CCITT-75160

Note – S and K are signal names. Z and X are the associated data.

FIGURE D-55

Storing incoming data for future use by means of a task

The concepts of shared values and imported/exported values are available as alternatives to signals as a means of interchanging information.

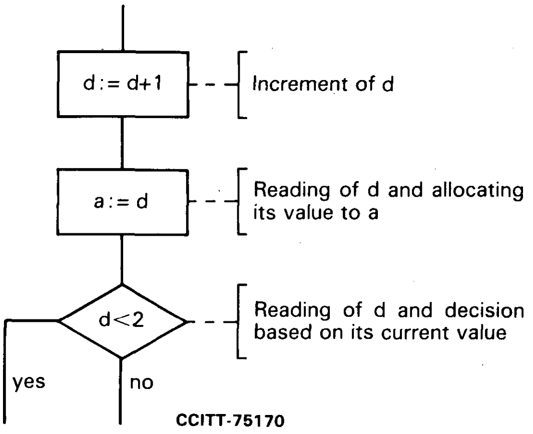
D.4.6.2 *Data handling inside a process*

Three possible ways of interprocess communication exist; namely via SIGNALS, SHARED VALUES or IMPORTED/EXPORTED VALUES. These three techniques differ substantially. For example, an IMPORTER of certain data items cannot have the same data items revealed as SHARED VALUES.

Data is local to a process instance, which means that all data have one and only one owning process instance.

The owner process instance can change the actual value of the data.

Example: If data items a and d are integers and local to process P, the examples in Figure D-56 of sequence of actions are legal inside P.


d := d+1
Increment of d
a := d
Reading of d and allocating its value to a
d<2
Reading of d and decision based on its current value
yes
no

CCITT-75170

FIGURE D-56

Example on actions upon local data in a process

D.4.6.3 *Data handling between processes*

Two processes can exchange data otherwise than by means of signals. A process can read a data value of another process if it belongs to the same block and if that data is declared as a "shared value" (§ D.4.6.3.1). In addition, a process can read a data value of another process if that data is declared as an "exportable value" (§ D.4.6.3.2). The process that does not own the data cannot modify the data value, but can make a local copy which can be manipulated like all other local data items.

A data item cannot be both imported and shared.

D.4.6.3.1 *Reading of shared values*

A process can reveal some or all of its data values by declaring them as "shared values", which has the effect that all other processes (belonging to the same block as the revealing process) have the possibility of reading the shared value in the same manner as if this shared value was a local value. This means that the value which a viewing process is seeing is always the same value as that seen by the revealing process.

There are some limitations to what can be revealed.

D.4.6.3.2 *Reading of exportable values*

A process can declare some or all of its data as "exportable data", which has the effect that all other processes with a corresponding "importable data definition" are allowed to get a copy of the data value upon request time. This means that the copy of the data value, given to an importing process upon request, remains constant even if the actual data value is changed later by the exporting owner process.

D.4.6.3.3 *Example of differences between use of shared and exportable values*

In the example of Figure D-57 there exists only one instance of process 1. If more than one instance can occur, some means of uniquely identifying the relevant data item *d* must be provided.

D.4.6.3.4 *Shared values*

D.4.6.3.4.1 *General*

The SDL user may find that the definition of shared data permits an easy way of specifying communication between two processes. However, a number of problems arise in implementing systems so *specified*, and this section is intended to guide users in avoiding and overcoming such problems. *Describing* systems in SDL which have implemented shared data is less likely to prove difficult, because the problems will have been overcome in the implementation and it should be possible to map the solution chosen onto SDL.

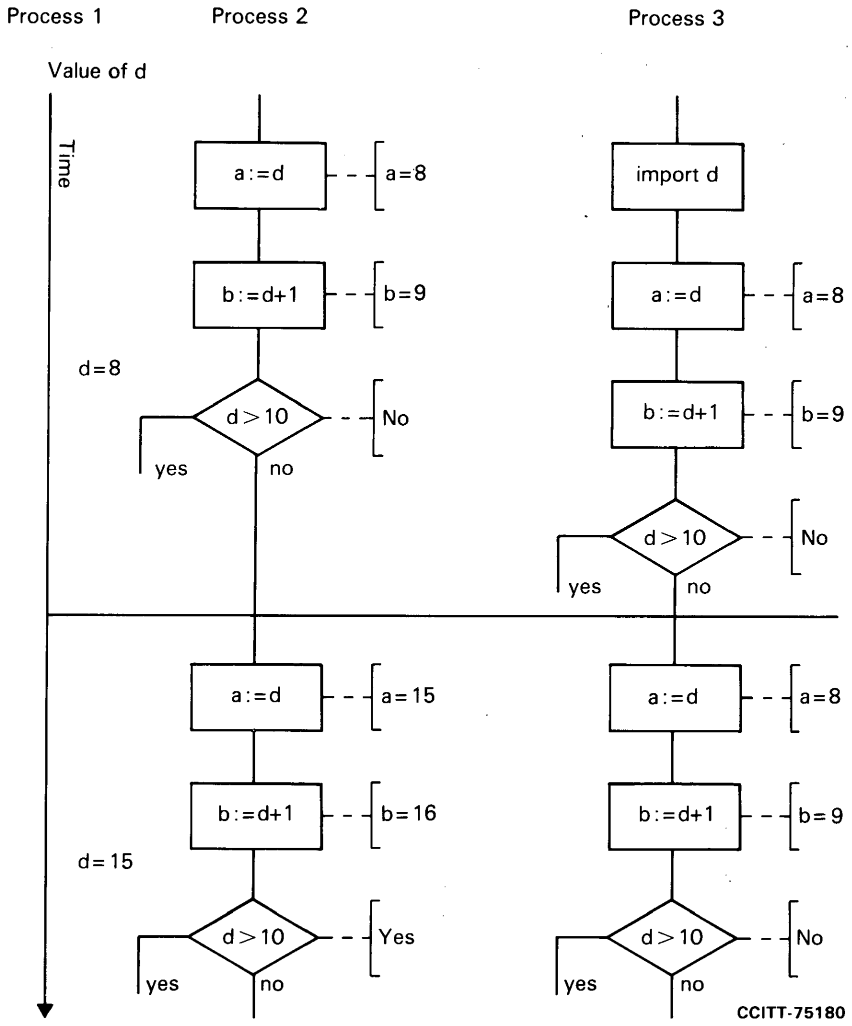
In the remainder of this section it is supposed that a process A owns data which it reveals to a process B which views it.

D.4.6.3.4.2 *Creation problems*

The result of an attempt to view data in a process before the process and the data is created is an SDL error. The user may avoid this problem in two ways. Either:

- ensure that the revealing process instance A is created and has initialized the relevant data before the viewing process instance B, or
- ensure that B does not enter transitions which use shared data until A has been created and has initialized the relevant data.

A simple way to achieve the former case is to make A the parent (or an ancestor) of B, or to let A be created at the same time as the system (implied creation). In the latter case it may be arranged that the relevant transition in B can only be triggered by a signal from A.



Note — Process 1 is the owner process of d. d is shared with process 2 and made exportable. Process 3 has d as an importable value.

FIGURE D-57

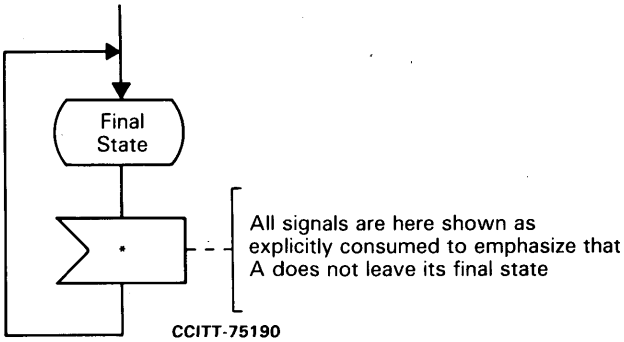
Example of differences between use of shared and exportable values

D.4.6.3.4.3 *Process stop problems*

The process A will no longer be able to reveal data after it reaches a stop symbol. Any attempt to view data would then be an SDL error. The user may avoid this problem in two ways. Either:

- do not use a stop symbol in A at all. In Figure D-58 process A is shown in a dormant state with shared data visible, but not capable of further change, or
- ensure that B is aware that A is about to stop and does not make any further attempts to view the relevant data. An example using signals is shown in Figure D-59.

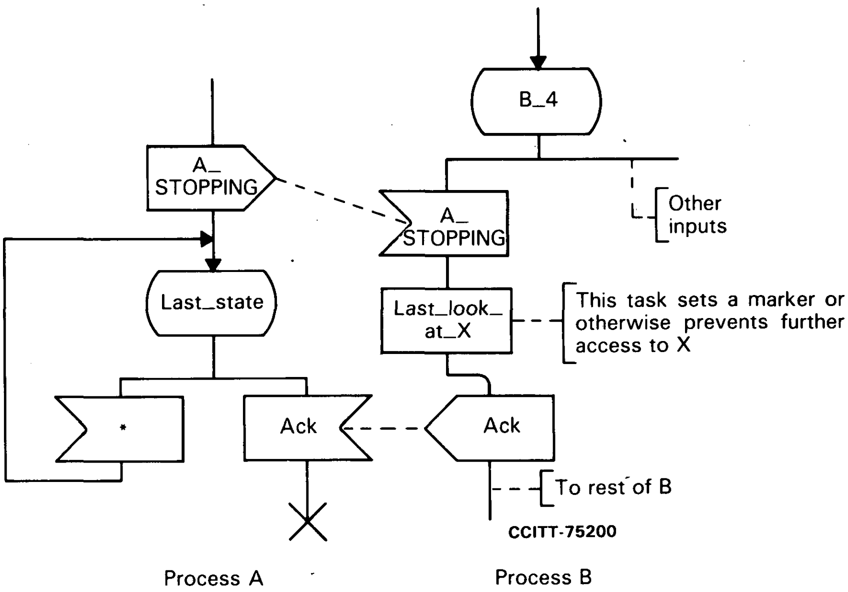
The first solution has the disadvantage for the implementor that A has not released the data storage it was using.


Final State
*
All signals are here shown as explicitly consumed to emphasize that A does not leave its final state
CCITT-75190

Note — Process A has data X which is revealed.

FIGURE D-58

A process having a dormant state, still able to reveal data but not to change it



Note — Process A has data X which it reveals to process B.

FIGURE D-59

Process A notifies B and gets agreement before it stops

D.4.6.3.4.4 *Problems due to multiple instances of B*

If the process B has several instances simultaneously alive, say B1, B2, B3 ... all viewing A's data, solutions to the earlier problems in § D.4.6.3.4.2 need a slight modification.

The easiest way to avoid any instance of B attempting to view A before A exists and initialized the relevant data, will be if A is the parent of all the B's.

Alternatively if A has to signal to each B it can only do so after being notified by the parent (B) of the existence and process identifier of each B.

There also has to be a slight modification to the process stop solution in § D.4.6.3.4.3. Clearly the dormant state solution in Figure D-58 is still correct.

The solution in Figure D-59 is no longer satisfactory. A solution on the following lines could be constructed.

a) A exists before B's parent.

b) A is informed by B's parent sending a signal, each time a new B is created.

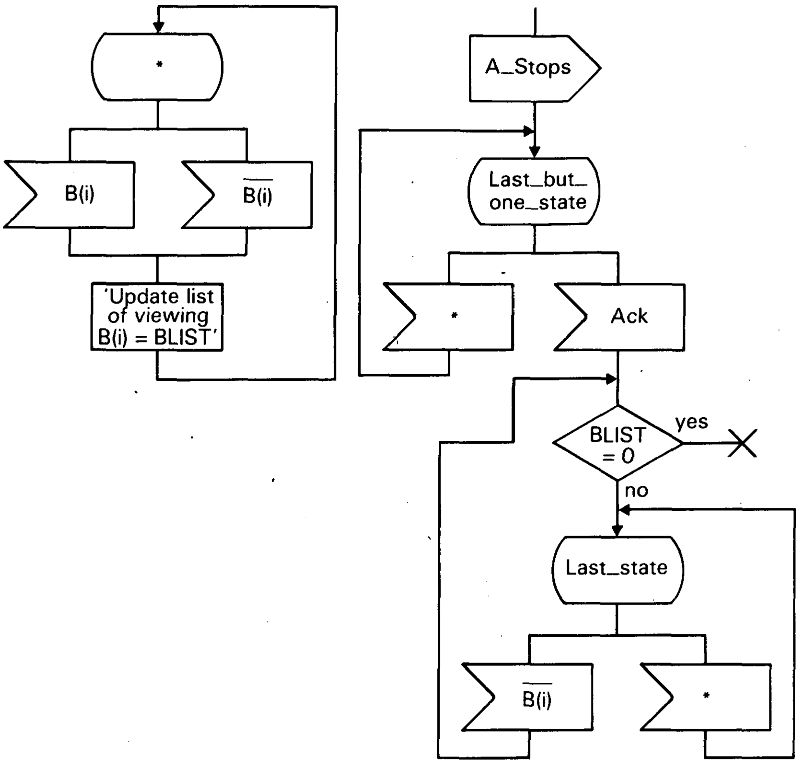
c) A is informed by each B sending a signal when it is going to stop – or at any rate when it will not view A's data again.

d) A keeps count of how many Bs are alive and viewing or potentially viewing.

e) A sends a signal to B's parent to stop the creation of more Bs and gets an acknowledgement that no more will be created.

f) When the count of Bs potentially viewing reaches zero, A can stop.

This is shown in Figure D-60 concerning the A process.



CCITT-82790

FIGURE D-60

Two diagrams of process A

D.4.6.3.4.5 *Problems due to multiple instances of A*

If the process A has several instances simultaneously alive, say A1, A2 ... and B wishes to view data, then B has to specify which A holds the relevant data. This means that B must know the process identity of the relevant A, and must be certain that the A whose data is to be viewed exists.

A possible solution to the problems can be constructed as follows.

a) A's parent sends a signal to B containing the process identifier of one of its offspring (say A6) holding the data B wishes to view.

b) B views the data.

c) A6 sends a signal to B to inform B that A6 is about to stop and that data should be viewed no more.

d) B acknowledges by replying to A6 with a signal.

e) A6 stops.

D.4.6.3.4.6 *Problems due to multiple instances of A and B*

The most likely practical case concerning multiple instances of both process A and B, is when the data sharing relation is between pairs of As and Bs. Problems in § D.4.6.3.4.2 and § D.4.6.3.4.3 may be avoided by the following:

a) As soon as a new instance of A, say Ai is created, it creates a new instance of B, say Bj. Ai will then know, the process identity of Bj as offspring, and Bj will know Ai as parent.

b) Bj views the data.

c) When Bj has finished viewing the data for the last time it sends a signal to its parent Ai and stops (or continues with other transitions).

d) Ai stops.

D.4.6.3.4.7 *Sharing several items of data*

Consider the case in which A owns several items of data, say X1, X2 and X3 which are all to be viewed by B. For simplicity suppose there is only one instance of process A and one of process B.

If A updates X1, X2 and X3 while B is trying to view them, it may happen that B gets a mixture of new and old values, i.e. an inconsistent set.

In some systems, consistency is more important than having up-to-date values, and in such cases means to enforce consistency should be applied. Two possibilities are discussed here:

a) Instead of sharing X1, X2, X3 we define a composite structure X which has X1, X2, X3 as fields and/or substructures.

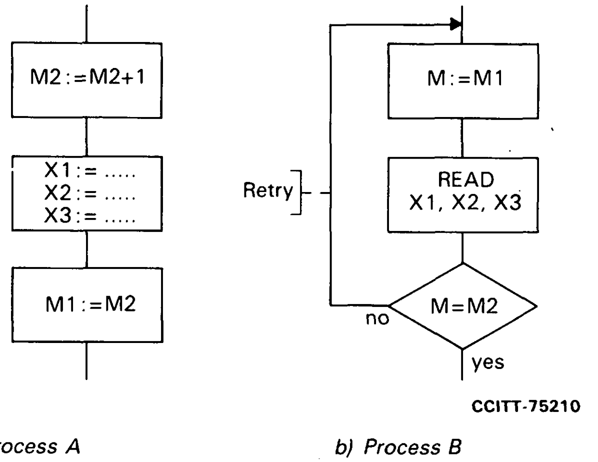
   X is revealed by A and viewed by B.

   A now obtains updated values for X1, X2, X3 and when all are updated, assigns all new subfields to X in a single assignment (assuming that SDL takes assignment as atomic).

   In this way an inconsistent set is never viewable.

However the problem has not really been solved – merely passed on to the implementor since in most implementations assignments are only atomic if the fields are in one machine word.

b) In order to solve the problem as posed, we need to specify some form of lock in SDL. A simple deadlock free example is shown in Figure D-61 which requires the introduction of two markers M1 and M2.



a) Process A
Updating X1, X2, X3. The importance of the sequence, first M2, then the Xs, then M1 is emphasized by using separate tasks

b) Process B
Reading X1, X2, X3. Tries again if it finds that M1 differs from M2

Note – Process A reveals M1, X1, X2, X3, M2 to Process B. M1 and M2 are of type INTEGER and initially set to zero.

FIGURE D-61

A lock to ensure consistent values

The markers must be implemented so that they can be atomically modified (i.e. in one word).

The implementor should examine the frequency and time taken for an update and investigate the probability of repeated attempts and ensure that he gets his required performance.

D.4.6.3.4.8 *Conclusions*

The above paragraphs reveal some problems in specifying systems using shared data, and give some examples of ways of avoiding them. Neither the problems nor the solutions given are exhaustive, and users may find other ways of avoiding the problems as well as other problems to avoid. In any event it is desirable to select solutions which harmonize with other aspects of the system being designed, so as to reduce complexity. Users are advised to analyse the behaviour of systems employing shared data with care.

Finally it is perhaps worth pointing out that most of these problems are avoided by using signals to send data.

In SDL, data values are regarded as "indivisible", i.e. a data object either has one value or another in its range. When implementing data objects, an implementation may modify only one part of a complex object at a time. When such objects are shared, the implementor should provide mechanisms for ensuring consistency (e.g. locks, etc.).

Note – This problem is normally avoided by exportable data since the owning process will not be able to reply to a request at a time of data changing (when the data may be inconsistent). A data value is changed totally in one transition and as the process can only be interrupted in a state the data will be consistent. "Mutual exclusion" is thus ensured.

D.4.7 *Guidelines on advanced data in SDL*

D.4.7.1 *Definition of data types*

D.4.7.1.1 *General*

SDL provides a number of pre-defined data types. It is possible, however, for users to define new data types. Once defined, these types may be used when declaring variables, signals, etc. in the same way as the pre-defined types.

As with most SDL concepts, the type definition mechanism can be used at a number of levels of abstraction, ranging from informal definitions of the basic syntax, to a fully formal definition of type syntax and semantics.

Types may be "parameterized", thus providing "building blocks" from which an even wider class of types may be created. These are known as type generators and, like SDL processes, must be instantiated to create one or more similar types with slightly different properties. The built-in type Array is an example of this since its "contents" are supplied only when it is used.

The facility to define new data types should be used whenever a highly rigorous, unambiguous specification is essential. Often, just stating the name of a type does not allow all readers of a specification to determine the exact properties of this type, even for types which are quite familiar. In these cases the advanced SDL data concepts should be used.

D.4.7.1.2 *Introduction to abstract types*

All data types in SDL are defined as abstract types. A type defines a set of values and a set of operations which may be performed on these values. For example, the set of values for type Boolean are True and False and the operations are the traditional Boolean operations, Not, And, Or, etc. Although seemingly restrictive at first glance, the abstract type concept is widely accepted in modern language theory, and allows the user to define *any* data type required. The data structures in traditional programming languages are merely a subset of abstract types. The behaviour of the operations can be defined informally, via "comment" text, or by a formal set of axioms.

This scheme allows definition of types in a way totally independent of their implementation, and thus the use of data types in SDL does not influence the final choice of implementation techniques.

D.4.7.1.3 *Data definition*

Data definitions are divided into three distinct categories: data type definitions, data type generators and synonym definitions.

D.4.7.1.3.1 *Synonym definition*

Synonym definitions allow a name to be equivalenced to the value yielded by an expression (which is dependent on context). "Strong typing" requires the type of the expression to be unambiguously determinable, if not, the type must be explicitly stated (e.g. the expression '4' is ambiguous since it may belong to type Real or type Integer).

Synonym definitions would normally be used as a "shorthand" notation where an expression is used in a number of places or to provide a centralized point for easy modification of a frequently accessed constant.

Example:

```
SYN Single Integer = 1
SYN BlockSize = 512 * BitsPerByte
SYN Legs = (4 * Dogs) + (2 * Birds)
```

D.4.7.1.3.2 *Data type definition*

The data type definition mechanism represents the principal technique for formally defining new types in SDL. Two alternatives are provided, *newtype* and *syntype.*

A *syntype* type defines a type which represents a subset of the values of some already existing type (known as the parent type), typically used like a programming language "range restriction" to allow tighter control over the values a variable may take.

This allows "structural equivalence" among types, i.e. types are compatible if they have the same underlying structure (parent type) even though they have different names.

Example:

```
SYNTYPE Digit = Integer
    CONSTANTS 0:9
END Digit

SYNTYPE WarmColours = Colours
    CONSTANTS Yellow, Orange, Red
END WarmColours
```

A *newtype* type introduces a totally new data type, that may or may not be based on an existing type. New literals and operator names can only be introduced via *newtype* definitions.

Newtype definitions provide for "name equivalence" among types, i.e. variables are only considered to have the same type if their declared types have textually the same name.

A *newtype* may be defined independently of all other types, as an extension to an existing type, or as an instantiation of a type generator or the built-in type generator provided for STRUCT. In the first two cases an important feature is the ability to define properties of the type as formal axioms. These optional axioms define the semantics of the operations defined for the type by stating the effect of combining operations. Thus all operations are defined in terms of each other. The type Boolean is used as the basis for definition of all abstract types, providing the nucleus of the total set of axioms for all types (although Boolean need not appear in the axioms for a particular type, recursively examining types will ultimately show that all types are defined in terms of Boolean).

Construction of these axioms requires care on the part of the SDL user to avoid two significant pitfalls: incompleteness and inconsistency. Both result in a potentially ambiguous SDL specification. In the first case, unfortunately, the set of axioms is, perhaps, insufficient to completely characterize the type being defined. In the second case the axioms conflict with each other, clearly invalidating the specification. Unfortunately, detection of these problems must be left to the SDL user: SDL provides no automated techniques for detection of incompleteness of inconsistency in algebraic type definitions; the user must rely on intuitive evaluation of a given set of axioms.

D.4.7.1.3.3 *Data type generator*

Informally a data type generator is a "template" from which any number of new types can be created. The generator parameters are textually replaced throughout a textual copy of the original generator and this new generator can then be interpreted as part of the type definition in which the generator instantiation is found.

Clearly this avoids redundant duplication of type definitions in cases where a number of types are required which are identical except for a part of the type which does not affect the basic axioms.

D.4.7.2 *Examples of data definition*

In the following, some examples of data definitions are given. They are not exhaustive; their only purpose is to provide a flavour of the power of the data definition mechanism adopted by SDL.

Beside the type definitions, examples of declarations of data objects of those types and of operations on them performed, are given.

Accordingly to the SDL form used, declarations and operations will be contained (GR form) or not (PR form) in graphical symbols retaining however the same syntax.

D.4.7.2.1 *Example 1: Definition of a subscriber meter*

Here a possible definition of a subscriber meter is given. The subscriber meter is defined by relying on the general properties of the natural type and restricting the set of allowable operations to only addition.

DEFINITION

```
NEWTYPE SUB_METER
INHERITS NATURAL ("+")
END SUB_METER
```

EXAMPLE OF USE

```
DCL A_METER SUB_METER;
.
.
.
A_METER : = A_METER +10
```

D.4.7.2.2 *Example 2: Definition of a generic type "bidimensional array"*

Here a bidimensional array is defined. The definition is general and not providing any restriction on the ranges of the two indexes, nor on the components of the array. This allows this type to be instantiated using whichever range and component type.

Three distinct operations are allowed (besides the implicit one which is the assignment operation): declare, insert, extract.

It is possible to declare new objects of that type, insert a new value in a selected position of the array and to extract a value from a selected position in the array respectively.

Accordingly to the syntax of the data definition for each allowed operator the domains and codomain of the operator is given in the *OPERATORS* part of the definition[1]. The *axioms* part gives the semantics of the operators.

Each operator name which is followed by an exclamation mark (in this case all of them) indicates that this name is used in the definition and a different name is used in the concrete syntax when using objects of that type.

[1] When no result type is stated (after the arrow) the result is a replacement in the parameters which are primed.

DEFINITION

```
GENERATOR BIDI_ARRAY (TYPE index1, TYPE index2, TYPE A_TYPE);

    OPERATORS
        INSERT! : BIDI_ARRAY', index1, index2, A_type → ;
        EXTRACT! : BIDI_ARRAY', index1, index2 → A_type;

    AXIOMS
        EXTRACT!(DECLARE!(v), l1,l2)=Error!;
        EXTRACT!(INSERT!(A,lp1, lp2, E1),lpr,lpc)=
                    if lpr=lp1 and lpc=lp2 then E1 else
                    Extract!(A,lpr,lpc)fi;

END BIDI_ARRAY
```

EXAMPLE OF USE

```
SYNTYPE INDEX = INTEGER
    CONSTANTS 1:20
END INDEX

NEWTYPE
    ARRAY2 BIDI_ARRAY (INDEX; INDEX; SUB_METER);
END ARRAY2
.
.
DCL DOUB_COL ARRAY2;
DCL A SUB_METER;
.
.
.
.
A :=DOUB_COL(5,16)
.
.
.
DOUB_COL(5,16):=A
.
.
```

D.4.7.2.3 *Example 3: Definition of type bit*

The newtype bit is defined as having only the values 0 and 1. The flip operator is introduced to set/reset the value of a bit (the codomain is missing) and all other operations (Not, And, Or, "+") return a value.

DEFINITION

```
NEWTYPE bit
    Literals (0,1);
    OPERATORS
        NOT : bit → bit;
        AND : bit,bit → bit;
        OR : bit,bit → bit;
        "+": bit,bit → bit;
        FLIP : bit' → ;

AXIOMS
    Flip (0)'=1;
    Flip (1)'=0;
    Not (1)=0;
    Not (0)=1;
    And (A,B)=if A=0 then 0 else B fi;
    Or (A,B)=if A=1 then 1 else B fi;
    "+" (A,B)=if A=0 then B else if B=1 then 0 else 1 fi;

END bit
```

EXAMPLE OF USE

```
DCL CNTRL_BIT BIT;
DCL TMP_BIT, DUM_BIT BIT;
.
.
.
.
TMP_BIT :=NOT(CNTRL_BIT)
DUM_BIT :=OR(TMP_BIT, CNTRL_BIT)
.
.
.
.
FLIP(CNTRL_BIT)
.
.
.
```

D.4.7.2.4 *Example 4: Definition of type byte*

A byte type has been defined as being an array of bits (newtype bitarray) as an array it inherits all the properties of arrays. Other operations allowed are: Shr, Shl, And-Mask providing shift to left n positions ($0 \leqslant n \leqslant 7$), shift to right n posiions ($0 \leqslant n \leqslant 7$) and the AND operation with another byte respectively.

DEFINITION

```
NEWTYPE bitarray (integer, bit) END bitarray;

NEWTYPE byte INHERITS bitarray ALL
    ADDING OPERATORS
                            shr : byte', integer → ;
                            shl : byte', integer → ;
                            And-Mask : byte', byte → ;
    AXIOMS
                            extract!(shr(B,1)',J)=  if J<0 or J>7 then Error!
                                                    else if J=7 then 0 else extract!
                                                    (B,J+1)' fi
                                                    fi;
                            extract!(Shl(B,1)',J)=  if J<0 or J>7 then Error!
                                                    else if J=0 then 0
                                                    else extract (B,J-1)!
                                                    fi
                                                    fi;
                            shr(B,n)'=  if n<0 or n>7 then Error!
                                        else if n=0 then B
                                        else shr(shr(B,1)',n-1)'
                                        fi
                                        fi;
                            shl(B,n)'=  if n<0 or n>7 then Error!
                                        else if n=0 then B
                                        else shl (shl(B,1)',n-1)'
                                        fi
                                        fi;

FOR ALL i IN INTEGER (extract!(And-Mask(B,MB)', i)
                                 = IF i<0 OR i>7 THEN error!
                                   ELSE
                                   and (extract!(B,i),extract!(MB,i)) FI;
                                   /* When two bytes are added together, each bit in the resulting byte is equal
                                   to adding together the corresponding bits in the original bytes */

end byte
```

EXAMPLE OF USE

```
DCL F_BYTE, S_BYTE, T_BYTE BYTE;
DCL TMP_BIT BIT;
.
.
.
shr (F_BYTE, Z)
.
.
shl (S_BYTE, 3)
.
.
.
And-Mask (F_BYTE, S_BYTE)
.
.
.
TMP_BIT := F_BYTE (7)
.
.
```

D.4.7.2.5 *Example 5: A simplified definition of a byte type*

SDL data definitions allow users to define "informally" data types. Such a facility may be very useful, especially when a stepwise refinement method is wanted for defining the system. This example shows a way of "informally" defining the same byte type as defined in example 4. (Note – the definition of the type bitarray is assumed.)

DEFINITION

```
NEWTYPE byte INHERITS bitarray ALL
        ADDING OPERATORS
          shr : byte', integer → ;
          shl : byte', integer → ;
          And-Mask : byte', byte → ;

     /* Shr provides shift n positions, with 0≤n≤7, towards right */

     /* Shl provides shift n positions, with 0≤n≤7, towards left */

     /* And-Mask provides the and operation on corresponding bits in two different bytes; the result is stored in
        the first operand */

end byte
```

D.4.7.2.6 *Example 6: Definition of a subscriber*

A very simple example of the definition of a subscriber is given. Up to this point a subscriber is a structure (record in many programming languages; or more formally a collection of several fields each of which may have a different type and may be selected when needed) containing a number field (the telephone number of the subscriber) and a status field (the status of the subscriber). These two fields are previously defined.

The only operations allowed on a subscriber object are connect and disconnect; they have the effect of setting the status of a subscriber to busy, or to free, respectively.

DEFINITION

```
NEWTYPE digitstring string(digit) END digitstring;
NEWTYPE status_id LITERALS(free, busy) END status_id;

NEWTYPE subscriber
    STRUCT   number digitstring;
             status status_id;

    OPERATORS
             connect : subscriber', subscriber" → ;
             disconnect : subscriber', subscriber" → :
```

```
    AXIOMS
               S=connect(S1,S2)' => S1(status)=busy;
               S=connect(S1,S2)" => S2(status)=busy;
               S=disconnect(S1,S2)' => S1(status)=free;
               S=disconnect(S1,S2)" => S2(status)=free;

end subscriber
```

EXAMPLE OF USE

```
DCL SUB_EX, CALD_SUB SUBSCRIBER;
DCL NUM DIGITSTRING;
.
.
.
SUB_EX (STATUS) := FREE
.
.
.
NUM := SUB_EX (NUMBER)
.
.
.
CONNECT (SUB_EX, CALD_SUB)
.
.
.
```

D.4.7.2.7-8 *Examples 7 and 8: A more detailed definition of a subscriber*

Here two other examples of informally defined data types are given. They refer to the possibility of structuring. Each new type is defined as containing as a field the old one. Such a feature allows the user to concentrate on particular aspects of the data type, deferring to a more detailed level other aspects.

DEFINITION

```
NEWTYPE subscriber_rev1
    STRUCT   first_app subscriber;
             counter sub_meter;

END subscriber_rev1
```

This definition adds the metering facility to a subscriber, here called subscriber_REV1.

```
NEWTYPE subscriber_REV2
    STRUCT   old_sub subscriber_rev1;
             enabling Three_cond;

END subscriber_rev2
```

This adds enabling conditions to a subscriber (field enabling); The Three_cond type may be defined as:

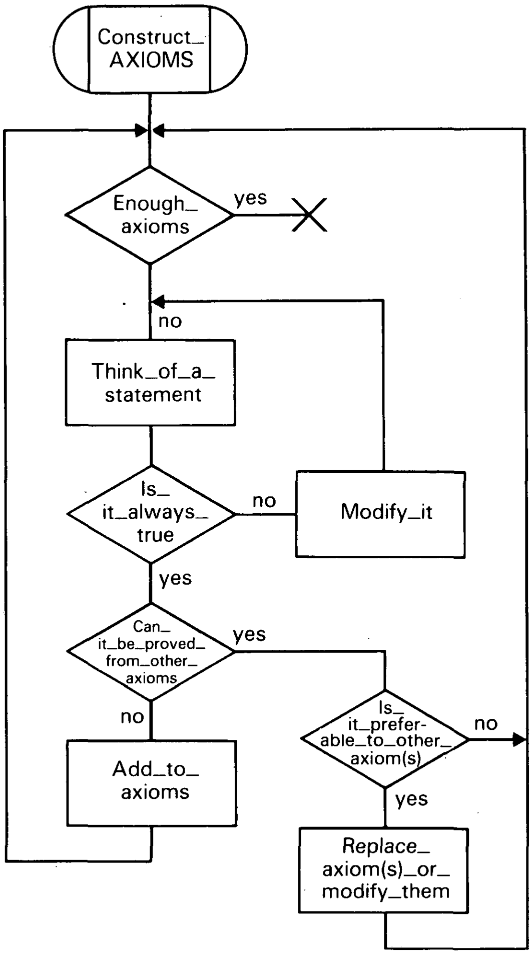
```
NEWTYPE Three_cond LITERALS (LOCAL_AB, LONG_AB, INT_AB) END Three_cond;
```

D.4.7.2.9 *Example 9: A procedure for constructuring axioms for a NEWTYPE*

When introducing new data types it is essential to set up enough axioms, which means that:

- each operator appears at least once in the set of axioms;
- true statement (other than the axioms) are proveable;
- no inconsistency can be detected,

in order to fulfil the requirements of the new data type.



CCITT-82800

D.5 *Documentation*

D.5.1 *What is a document?*

A document is defined by ISO as "a limited and coherent amount of information stored on a medium in a retrievable form". It should therefore be considered as a logical unit which is strictly delimited. Documents are used for conveying all information relating to a system which is specified or described using SDL.

When paper is used as the physical medium for storing a document, the term document is often incorrectly applied to the sheets of paper rather than their logical contents. With the growing use of magnetic storage media, the term is being returned to its original meaning.

D.5.2 *Introduction*

This § D.5 discusses documentation aspects. It is concerned with the logical organization of documents rather than their physical organization. This is left to the users discretion. The similarity in requirements of both the logical and physical organization of documents means that some useful hints may be offered in the following text to aid a user in setting up a physical organization for documents.

D.5.3 *Why do we need documents?*

By splitting the information relation to a system into a number of documents, the system can be made more readable and manageable. Since SDL has no formal concept of a document, the user is forced to use an informal mechanism or to borrow one from another language (preferably CHILL). It is only when specifying or describing a complete system as a single textstring using the PR-form, that it is not necessary to split the system into separate documents.

D.5.4 *Document structure*

The set of documents covering the whole system definition can be considered as a set of structures.

A document structure, where documents refer to sub-documents, is attached to each block in the system structure including the system level. See Figure D-62.

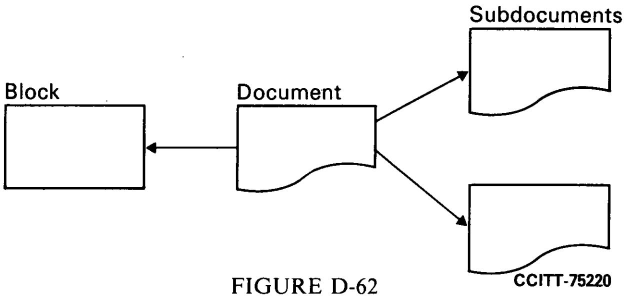


FIGURE D-62

A document structure

D.5.5 *Referencing mechanism*

All documents must be precisely related to each other. Particularly since updates to one can have an impact on the others.

Each document must have a unique identification and its boundary must be clearly specified. The boundary could be indicated by giving the page numbers for sheets belonging to the same document. For a one page document, such as a block interaction diagram, the frame symbol could indicate the boundary.

The same document can appear as a sub-document to more than one block in the same system.

§ D.9, "Examples", gives two approaches for referencing. One approach is based on PR-syntax, the other uses the comment mechanism in GR-diagrams. For more details see § D.9.

D.5.6 *Types of documents*

The different types of diagrams, which are introduced in the GR-syntax are generally documents according to the given definition. They should also include an indication of their boundary. The documents, which together cover all definitions introduced in a block definition should, as already stated, form a document structure. In this structure at least the process definitions should be placed in different documents. If a block interaction diagram is used, the channel definitions are also covered in the same document. The signal definitions and data definitions should preferably be grouped together in separate documents.

The procedure definitions and macro definitions should similarly form sub-documents to the process document.

The definitions for those signals forming a signal list can be placed in separate documents called: Signal list X, Y, Z, etc. A possible document structure for a block is given in Figure D-63 below:

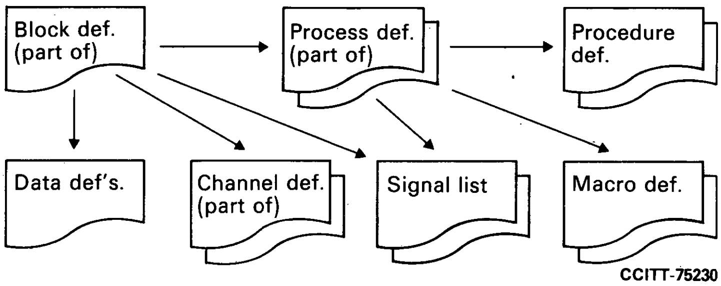


FIGURE D-63

A possible document structure for a block

D.5.7 *Mixing of GR and PR*

The process definition can be partially defined using the GR-form. The process graph is covered either by the process diagram (in GR) or by the process body (in PR). To be able to mix the definition of the process graph given in GR with the rest given in PR, the process graph definition should form a document of its own.

A system or block definition can also be partially defined using the GR-form. The block interaction diagram should then form a document. Since the block interaction diagram covers only parts of a system or block definition, the remaining parts should be given in separate documents.

When mixing GR and PR, the PR representation has to be split into several documents. Unfortunately, the PR syntax does not allow for this, so the splitting must be performed in an informal way.

Methods of forming syntactically connect parts of a PR repesentation are discussed in § D.7.

D.6 *Guidelines for representing systems using SDL/GR*

Chapter D.6 of the SDL User Guidelines explains the use of the graphical syntax representation of SDL, the SDL/GR.

D.6.1 *Why graphical syntax?*

A graphical language has the advantage that it clearly shows the structure of a system and allows the control flow to be easily visualized. As a design tool SDL should be in a form which allows the user to clearly and concisely express his ideas. The graphical syntax allows this and is more in line with the traditional representation of extended finite state machines, whereas the textual phrase representation is best suited for machine use.

The graphical representation syntax of SDL (SDL/GR) is the original form of SDL. It was devised during 1973 to 1976 and first appeared in the 1976 version of the Z.100-series of Recommendations.

The graphical representation syntax was derived from the graphical languages developed by different organizations for their own use.

D.6.2 *SDL/GR diagrams*

The diagrams explained in this § are:

Block interaction diagram which describes the internal structure of a system or a block;

Block three diagram which describes the partitioning of a system in terms of blocks and sub-blocks;

Process tree diagram which defines the partitioning of a process in sub-processes;

Channel sub-structure diagram which defines the internal structure of a *channel*;

Macro definition diagram which defines a set of symbols. Each appearance of a reference to the macro definition should be replaced with the definition before it is interpreted;

Procedure diagram which defines the behaviour of a *procedure* execution;

Process diagram which defines the behaviour of all *process instances* of that *process definition.*

Signal definitions and data definitions are defined using the SDL/PR syntax (see § D.7.3.2), and are referenced to from the diagrams in which the information is used.

D.6.2.1 *General guidelines*

General guidelines for drafting graphical diagrams are:

- they should be drafted so that the normal sequence of reading is top-down/left-right;
- diagrams should not contain too many details; it is often suitable to partition large diagrams into sub-diagrams covering different parts or aspects;
- preferably a connected segment of a diagram should be able to be contained on one page;
- comments may be inserted anywhere in a diagram, either using the GR comment symbol or the PR comment syntax;
- text should preferably be placed in the symbol the text is associated with, if this is not practical, the text may be placed in a text extension symbol connected to the symbol.

D.6.2.2 *Referencing*

In the current set of SDL Recommendations there are no means provided to make references between GR diagrams themselves, between definitions in SDL/PR or between GR diagrams and PR definitions. Both in order to make a complete SDL representation of a system, and to structure the documentation of a system, it is necessary to have a mechanism for referencing.

A referencing mechanism may be devised in several ways. It depends on the peferred organization of documentation and methodologies. As there is no mechanism provided in the Recommendations it is left to the user's discretion to choose a mechanism.

An example of the use of references in the GR syntax is to use notes or the comment syntax as shown in Figure D-64.

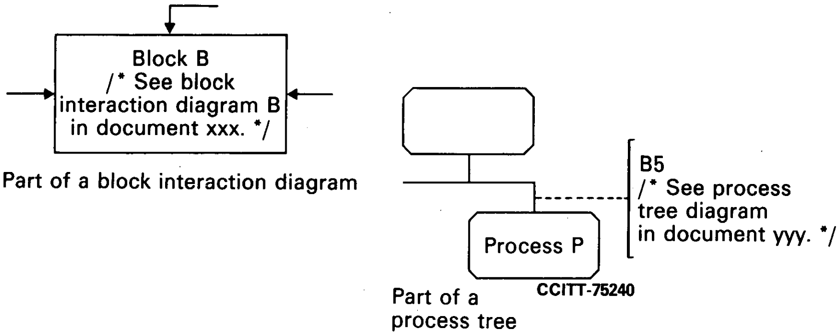


FIGURE D-64

Example of using notes or comments as references

Several other methods are possible, other examples are found in the examples in § D.9.

D.6.2.3 *Template*

A template for the symbols of the SDL graphical representation has been provided in the inside back cover of this fascicle. A schematic design for the template is given in Figure D-65.

In the figure the symbols for Input, Output, Decision, Alternative, Process, Start, Task, State, Save, Connector and Stop are shown directly, in three sizes, i.e. 20 × 40 mm, $20/\sqrt{2} \times 40/\sqrt{2}$ mm and 10 × 20 mm. The internal ratios are also shown for the largest size.

The symbols for Procedure Call, Macro and Create can be constructed from the task symbol by drawing the indicated extra horizontal or vertical line(s).

The symbols for Procedure Definition and Macro Inlet can be constructed from the Start Symbol by drawing the indicated extra vertical line(s).

The symbol for Macro Outlet can be constructed from the Connector symbol by drawing the indicated extra vertical line.

The return symbol is the combination of the Connector and the End Process symbol.

The Comment symbol is drawn using either end of the Task symbol.

The Enabling Condition symbol can be drawn by using the Stop symbol.

The All "symbol" (asterisk) is not included in the template because it should be printed with the text associated with the symbols.

The symbol for a Channel and the line to the text expansion symbol is a solid line.

The signal line and the line to a Comment symbol is a dashed line with the ratio 1:1.

All the recommended symbols can be seen in Figures D-209 and D-210.

D.6.3 *Using the SDL/GR*

For a description of all used symbols in the SDL/GR see the graphical syntax summary (*Annex C* to the Recommendations Z.100 to Z.104).

D.6.3.1 *Expressing structure in SDL/GR*

The partitioning of a system into blocks and channels, and the optional further partitioning of the blocks and channels, is explained in this section.

It should be noted that when a system is partitioned into several levels of increasing detail it may contain alternative diagrams (see Recommendation Z.102). When there are two alternatie descriptions the less detailed description is to be considered as an overview.

D.6.3.1.1 *Block interaction diagram*

The *Block interaction diagram* shows the internal structure of a *system* or a *block* in terms of *channels* and *blocks.* The set of *processes* contained in blocks may be represented.

The *processes* which may create other *processes* may also be represented.

The diagram is composed of:

- the frame symbol (mandatory for blocks but optional if used around a system)
- block symbol
- channel symbol
- signal lists

and optionally

- process symbol
- signal route symbol
- request symbol
- environment symbol.

Figure D-66 is a simple example of a block interaction diagram representing a small system (S). It consists of two blocks (B1 and B2). Note that for a system the enclosing frame symbol is optional.

Following the top-down/left-right convention, names should preferably be placed in the upper left corner of, or besides, the named object. Referenced definitions may be placed outside the diagram, as for the signal list L_1 in the example.

Signal definitions and data definitions, which are described in PR-syntax, may either be put in the diagram, then preferably in the upper left corner, or outside the diagram. If definitions are placed outside the diagram, and it is not obvious where to find them, the diagram should contain a reference to where they are to be found.

In the block symbol the contained processes may be shown, as indicated in Figure D-67.

In order to make more readable diagrams, the diagrams may be partitioned so that the interior of blocks are shown separately. For example, the segment shown in the Figure D-67 may form a sub-diagram, shown separately, to the block interaction diagram.

It is often useful to show several levels of structure in the same block interaction diagram. This is done by replacing a block symbol in a diagram by the block interaction diagram for that block. This is exemplified in Figure D-68.

This may be repeated any number of times.

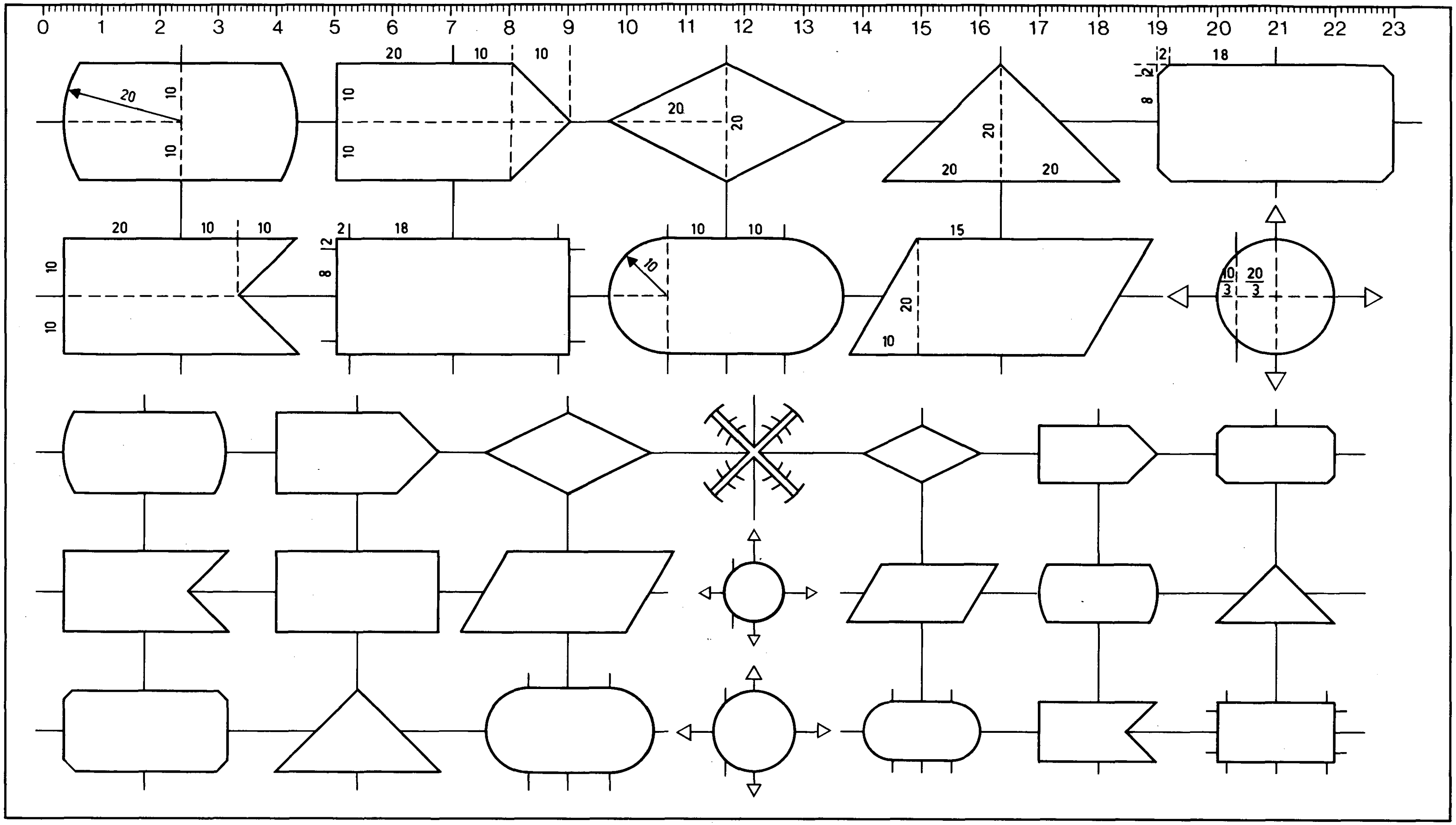
In order to have a well structured, and readable diagram, the following should be noted:

- the diagrams should not be made too complex to be contained in one page;
- use of the possibility of partitioning the diagram into sub-diagrams reduces the amount of detail and makes the diagram more readable;
- the macro may be used as a tool to reduce the complexity of large diagrams.

D.6.3.1.2 *Block tree diagram*

A block tree diagram represents the structure of a system in terms of blocks and sub-blocks. The intention with the diagram is to give the reader an overview of the total structure of a system.

The diagram is a hierarchical tree of block symbols and "partitioned into" lines.



CCITT-75251

Note 1 – Ratios height : length = 1 : 2

Note 2 – Three sizes shown i.e. length 40 mm, 28 mm and 20 mm. 40 mm/$\sqrt{2}$ = 28 mm ; 28 mm/$\sqrt{2}$ = 20 mm, etc. This permits photoreduction from A3 to A4 paper with compatible symbol sizes.

FIGURE D-65

Schematic design of the SDL template

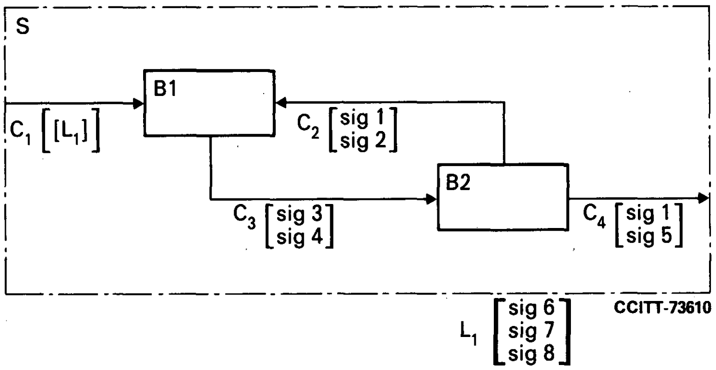


FIGURE D-66

Example of block interaction diagram representing a system

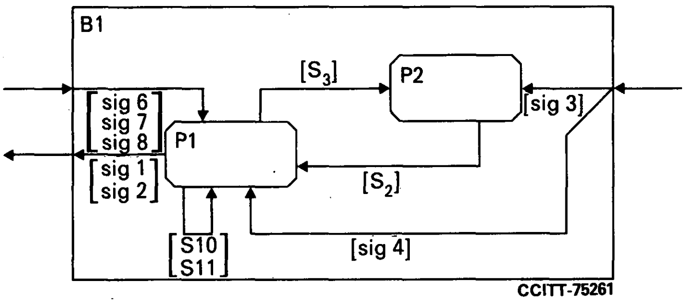


FIGURE D-67

Segment of a block interaction diagram

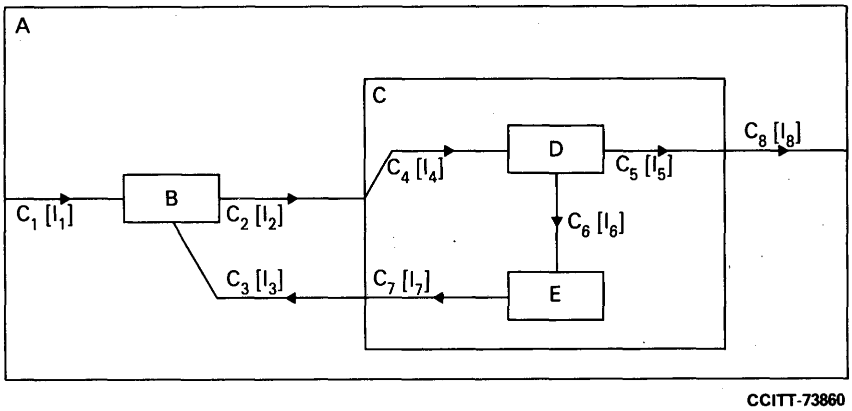


FIGURE D-68

Example of representing several levels of structure

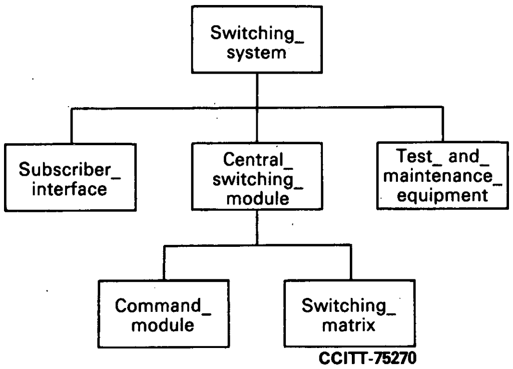


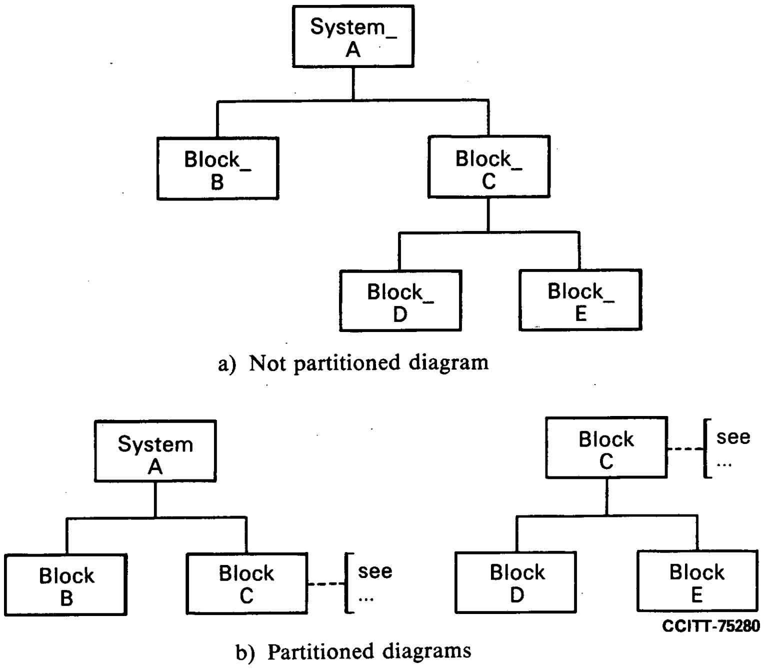
FIGURE D-69

Example of block tree diagram

The diagram should preferably be drawn with all the block symbols having a uniform size. This allows the blocks at the same level of partitioning to appear as a uniform level in the diagram. If the diagram is so large that it requires more than one page it should be partitioned into "partial" block tree diagrams as shown in Figure D-70.

It is often useful to partition a block tree diagram into partial diagrams.

The splitting into several partial diagrams is done so that the first diagram, having the system as root, is chopped off so that a set of further partitioned blocks appear as not partitioned. The blocks, where the original diagram was chopped off, appear as roots in diagrams showing the further partitioning.



a) Not partitioned diagram

b) Partitioned diagrams

FIGURE D-70

Example of partial block tree diagrams

If partial diagrams are used, and it is not obvious that a block is further partitioned and/or where to find the continuing diagrams, references should be inserted using the comment symbol.

D.6.3.1.3 *Process tree diagram*

A process tree diagram describes the partitioning of a process into sub-processes, and shows where the processes are allocated. The diagram is a hierarchical tree of process symbols and "partitioned into" lines. The allocation of the processes is described using the comment symbol.

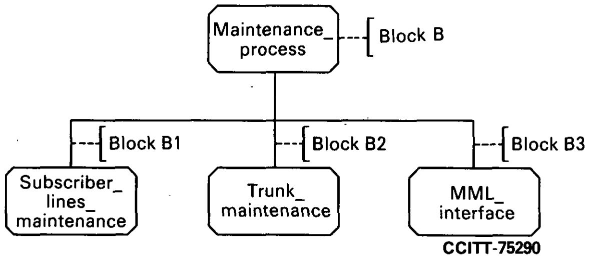
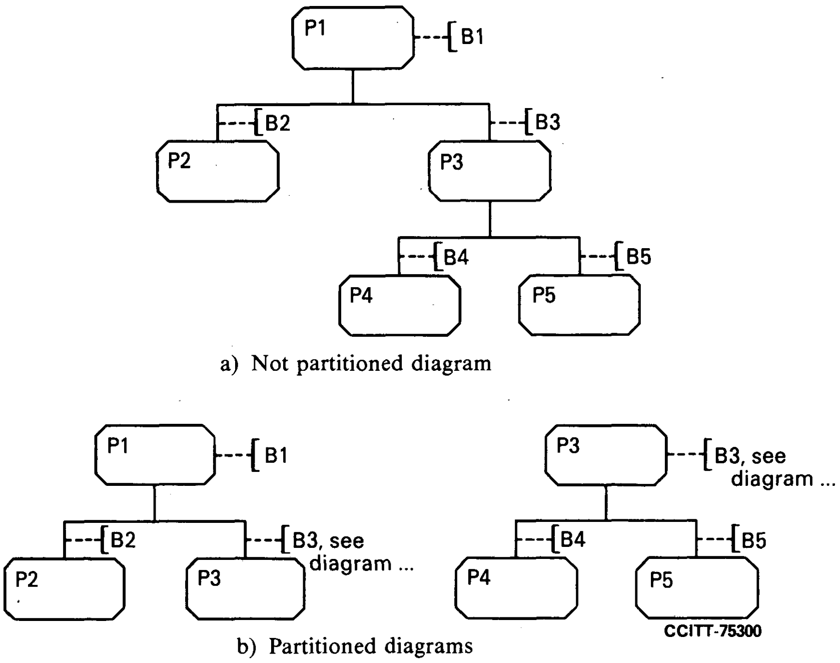


FIGURE D-71

Example of process tree diagram

To be complete the root should be a process which is not a sub-process of any other process. Also, all of the leaf-processes should not be further partitioned. Preferably, the diagram should be drawn with all the process symbols having a uniform size, so that all the sub-processes of the same level of partitioning appear as a consistent level in the diagram.

If the diagram becomes so large that it may not be contained on one page, or for purpose of readability, the diagram may be split into partial diagrams. A set of partial diagrams is obtained by letting a set of further partitioned processes appear as not partitioned in the original diagram. These processes appear as roots in other diagrams describing the further partitioning.



a) Not partitioned diagram

b) Partitioned diagrams

FIGURE D-72

Example of partial process tree diagrams

If partial diagrams are used, and it is not obvious that a process is further partitioned or where to find the continuing diagram, references should be inserted using comment symbols.

D.6.3.1.4 *Channel sub-structure diagram*

A channel sub-structure diagram describes how a channel is partitioned into sub-components. The diagram resembles the block interaction diagram. All the guidelines given in § D.6.3.1.1 are also valid for the channel sub-structure diagram.

In the block interaction diagram where the partitioned channel appears there should be a reference to the channel sub-structure diagram describing the partitioning. See Figures D-73 and D-74.

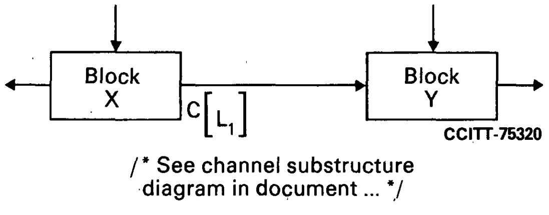


FIGURE D-73

Segment of a block interaction diagram containing a partitioned channel

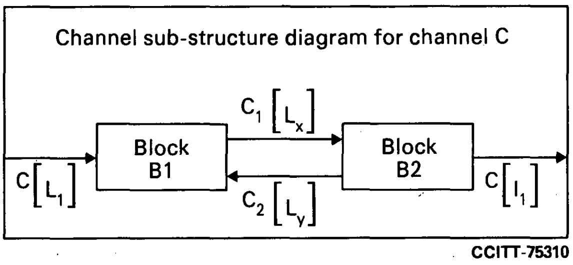


FIGURE D-74

Example of channel sub-structure diagram

If it is obvious that the channel is partitioned and it is also obvious where to find the sub-structure diagram, the reference may be omitted.

D.6.3.2 *Signal definition*

There is no special graphic syntax for signal definitions. In the SDL/GR diagrams that use signal definitions the SDL/PR syntax is used (see § D.7.3.2). The PR text may be inserted into the diagrams but preferably the text should only be referenced to from the diagrams.

The diagrams that may reference signal definitions are:

- block interaction diagrams
- channel sub-structure diagrams.

When the definitions appear in the diagram they should preferably appear in the upper right of the diagram, as shown in Figure D-75.

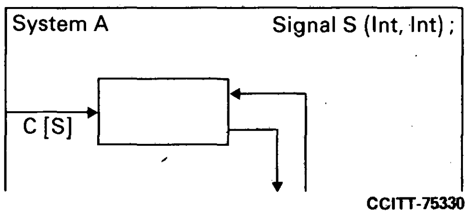


FIGURE D-75

Segment of a block interaction diagram containing a signal definition

Large volumes of PR text in a diagram will decrease the readability of the diagram.

It is preferable to replace this PR text with a reference to the location of the PR text, e.g.:

/*For signal, type and procedure definitions see document . . . */

D.6.3.3 *Data definitions*

There is no special graphical syntax for data definitions in SDL/GR. When data definitions are needed in GR, the PR syntax is used (see § D.7.3.4).

There are two types of data definitions:

- data type definitions
- variable definitions.

As for signal definitions the PR text is preferably referenced to from the GR diagrams.

D.6.3.3.1 *Data type definitions*

Data type definitions may be referenced in the following SDL/GR diagrams:

- block interaction diagrams
- channel sub-structure diagrams
- macro definition diagrams
- process diagrams
- procedure diagrams.

The same guidelines as for inserting signal definitions also apply for data type definitions.

D.6.3.3.2 *Variable definitions*

Variable definitions may be contained in the following SDL/GR diagrams:

- process diagrams
- procedure diagrams.

The same guidelines as for inserting signal definitions apply for variable definitions.

D.6.3.4 *Macro definition diagrams*

A macro definition diagram defines the set of symbols that each reference of that macro should be replaced with before interpretation.

The diagram can have one or more inlet symbols, followed by a flow line leading into the diagram, and one more outlet symbols, following a flow line leading out from the diagram. Apart from that, a macro definition diagram may contain any set of graphic symbols and text in any combination. The correctness and meaning of the macro can only be decided after that replacement has taken place.

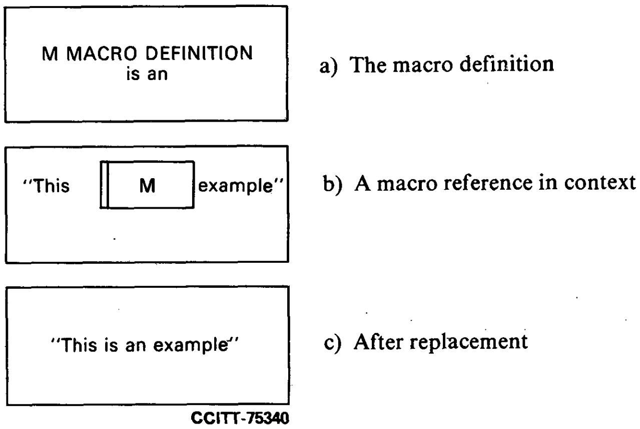


FIGURE D-76

Example of use of macro

The general guidelines for well structured diagrams apply for the macro definition diagram as well.

D.6.3.5 *Procedure diagram*

The procedure diagram defines the behaviour to be performed at each call of that procedure. Procedures in SDL are similar to procedures in CHILL and other programming languages. Procedures are intended to:

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

b) maintain 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.

As procedure definitions may be contained in system definitions, block definitions, process definitions and procedure definitions, the same procedure may be called from several processes. The use of common procedure libraries is allowed.

The diagram defining the procedure is similar to a process diagram. The differences are that the start symbol of the process diagram is replaced by the procedure start symbol and the stop symbol is replaced by the return symbol.

The same guidelines as for process diagrams are also applicable for procedure diagrams (see § D.6.3.6). An example of a procedure diagram is shown in Figure D-77.

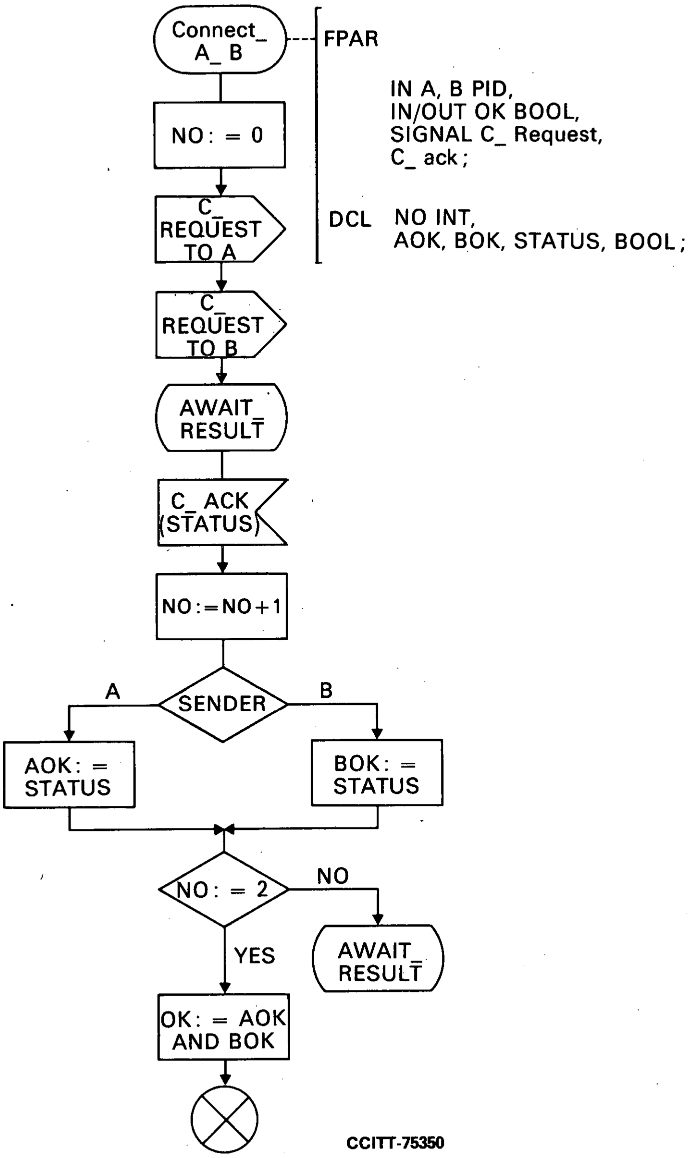


FIGURE D-77

Example of procedure diagram

D.6.3.6 *Process diagram*

A process diagram defines the behaviour of a process using SDL/GR. Auxiliary overview documents are often useful as leading documents when the processes described are large or complex. At the end of this section some examples of such documents are given.

D.6.3.6.1 *Versions of SDL/GR process diagrams*

If in an SDL/GR diagram, transitions are described exclusively by explicit action symbols, this is called the transition-oriented version of SDL/GR.

If, on the other hand the states are described using state pictures and the difference in state pictures is used to imply transition actions, this is called the state-oriented version of SDL/GR.

A combination of both versions is called the combined version.

Examples of these three versions are given in Figures D-78 to D.80.

The transition-oriented version is suitable when the sequence of actions is important and detailed state descriptions are less important.

The state-oriented version is suitable when the sequene of actions within each transition is not important, when pictorial explanation is desirable, and when compact representation is desirable. It is available for applications for which suitable pictorial elements are defined (see § D.6.3.6.22).

The combined version is suitable when the redundancy implied is helpful.

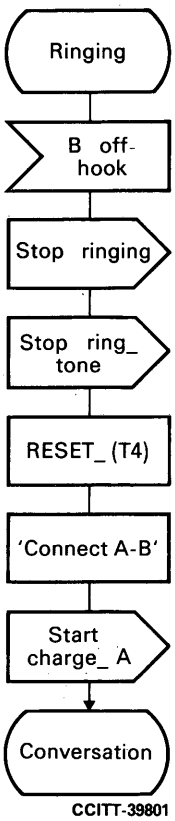
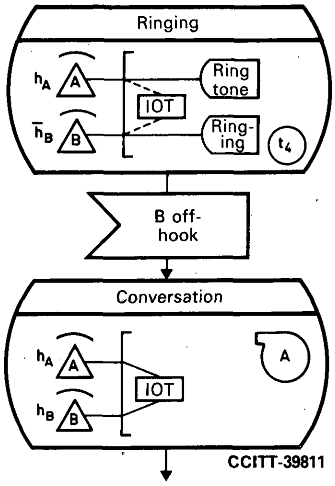


FIGURE D-78

Transition-oriented version



IOT: Input/output trunk

FIGURE D-79

State-oriented version

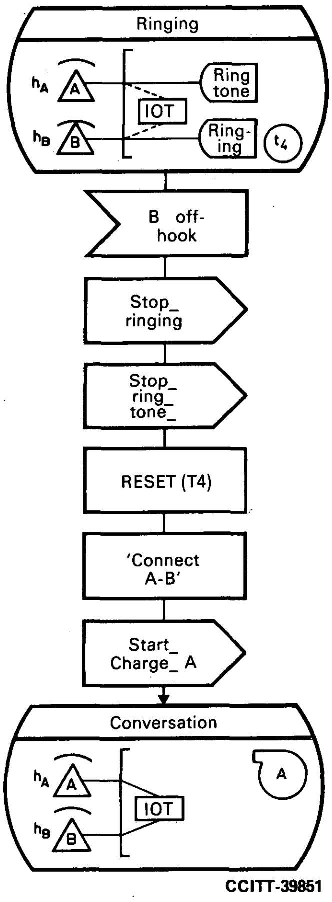


FIGURE D-80

Combined version

D.6.3.6.2 *Symbols and connection rules*

The symbols allowed in a process diagram and how they are to be connected can be found in Annex C to the Recommendations Z.100 to Z.104.

D.6.3.6.3 *Well structured diagrams*

The following is a set of general guidelines for drafting well structured diagrams.

D.6.3.6.3.1 *Starting and ending of a diagram*

Process instances may be explicitly created, and in this case the process diagram should be drawn to begin with the start symbol. For processes which have no explicit instance creation, there is a starting state, identified by its position at the top in the process diagram.

It is normally preferable to draw an SDL diagram with the flow from the top to the bottom of the page.

A diagram may be split into several pages. And flow from one page to another can only be made by connectors (§ D.6.3.6.16) or by multiple appearances of the same state (§ D.6.3.6.5).

D.6.3.6.3.2 *Alternative arrangements of a diagram*

The suspension of a process is represented by the state symbol. Whether suspensions should be emphasized (by disconnecting the diagrams at each state) or not, is left to the SDL-user. According to the SDL rules there are three possible methods:

Method A

The diagram is interrupted at every state symbol and will continue again with that state symbol.

Using Method A, the complete SDL diagram consists of a set of *n* diagrams where *n* is the number of states of the process. Each diagram begins with a different state, shows all transitions from that state and ends with the states of those transitions. The same state can appear in several diagrams.

Examples are shown in Figure D-81.

Note that the transitions, together with the inputs, are only indicated by directed lines.

If the process has a start, the start symbol and starting transition will also be shown as a separate diagram (see Figure D-81 b)). Similarly if the process has a stop, states and transitions leading to stop symbols will be separately shown (see Figure D-81 c)).

Method B

The diagram is uninterrupted.

In method B the complete diagram is drawn as a fully connected graph. The example of Figure D-81 is redrawn in Figure D-82 as a fully connected graph.

Method C

The diagram is interrupted as some state symbols.

If the complete SDL diagram is only interrupted at some state symbols there are several equivalent ways of redrawing the Figure D-81. One of these, consisting of two diagrams, is shown in Figure D-83.

In Method C some states will appear in several diagrams. Using this method it is of course advisable to draw connections between those parts of the diagram which belong logically together. For instance, it is very common that a relatively small part of the process represents the behaviour of most usual situations. This part could be suitably drawn as a separate diagram.

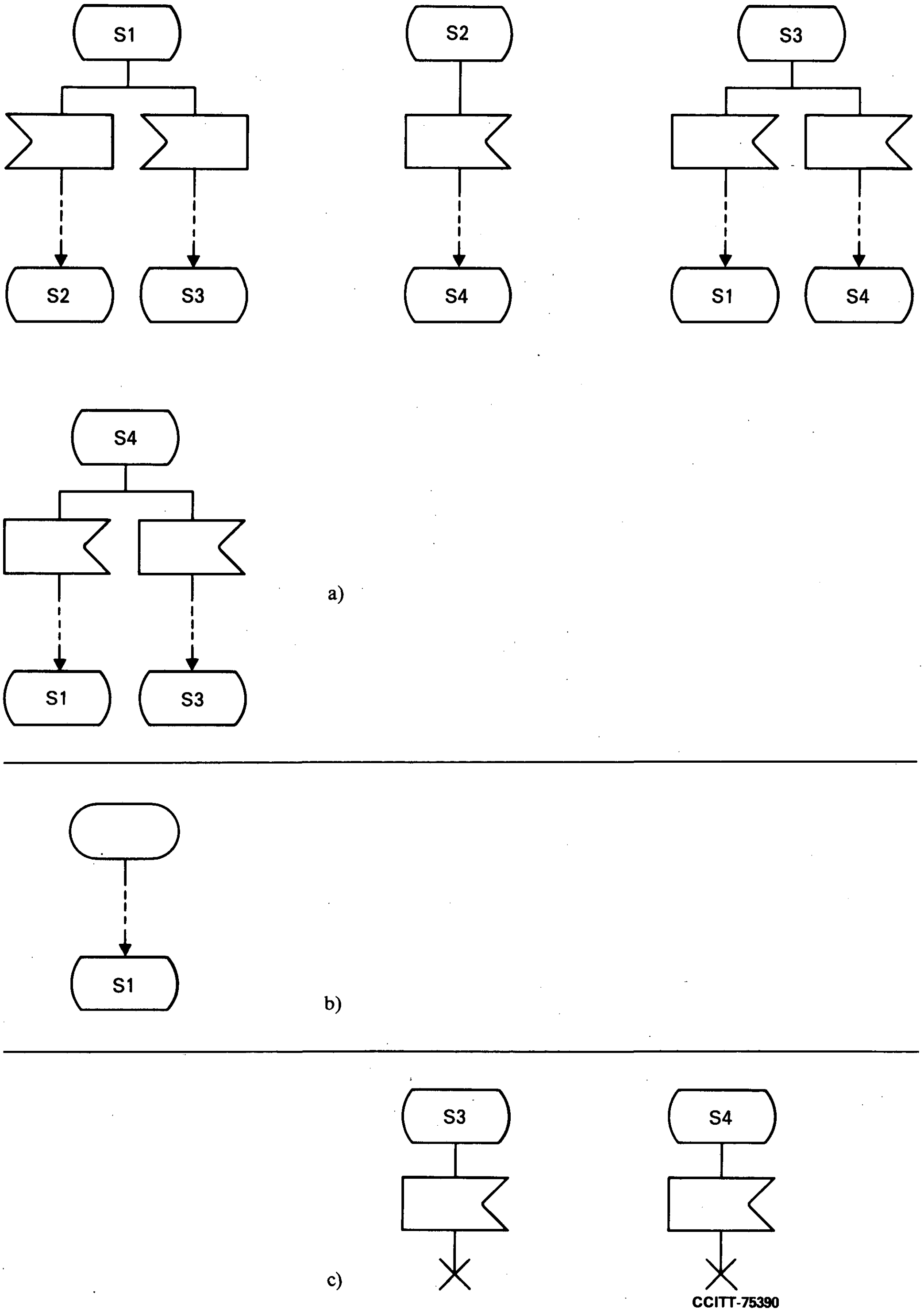


FIGURE D-81

Fully separated graphs

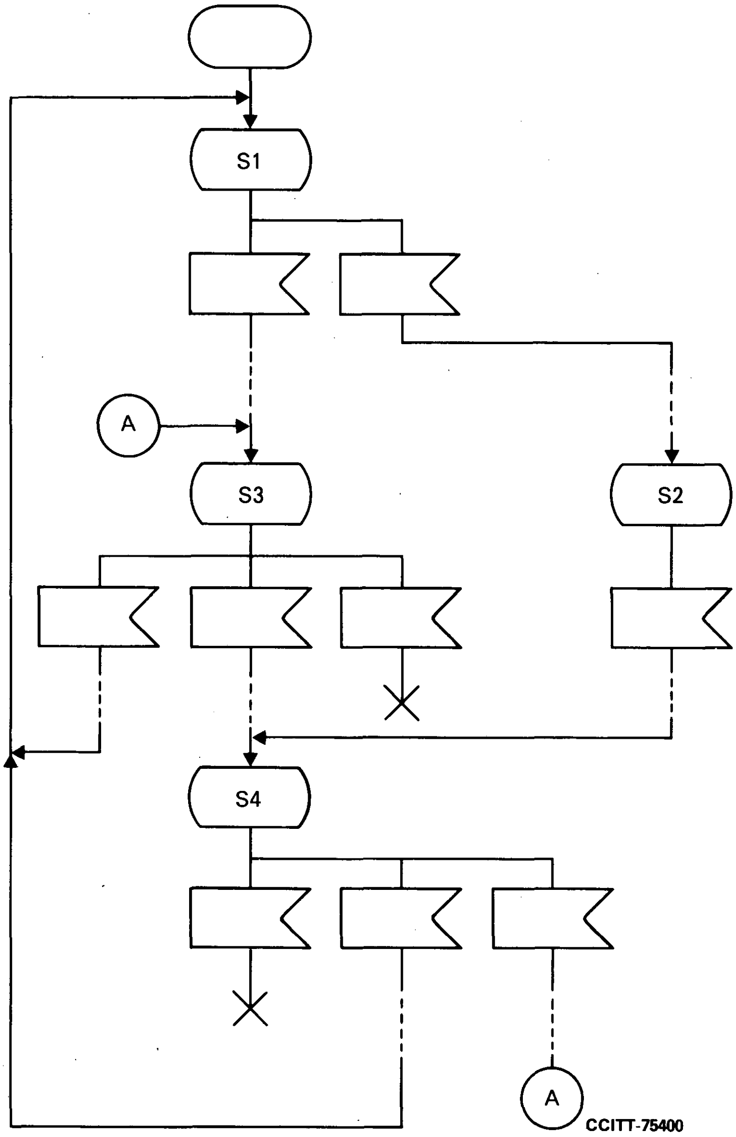


FIGURE D-82

Fully connected graph

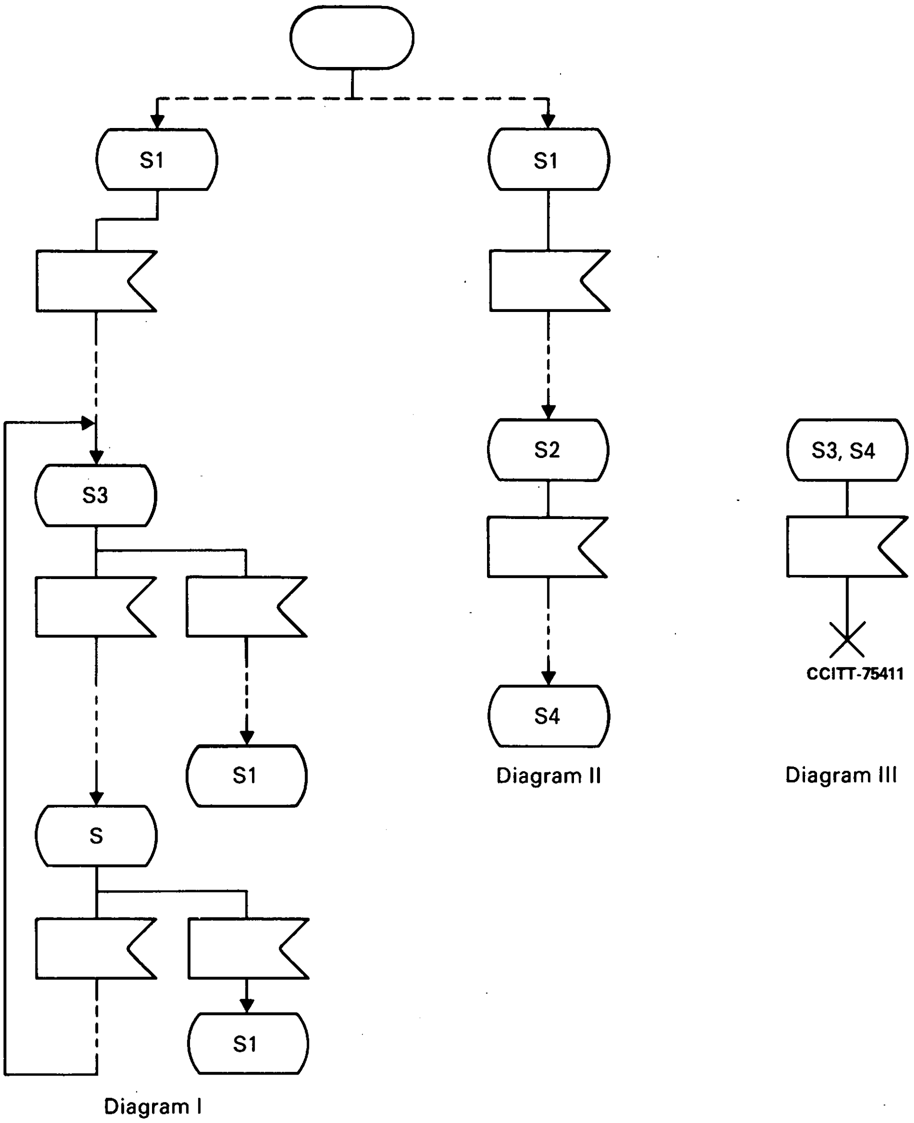


FIGURE D-83

Partly connected graphs

Comparison of the three methods

The advantage of Method A lies in the minimal time spent deliberating how to draw a diagram, and in the ease of modification and the ease of information retrieval.

The advantage of Method B lies in the total process overview, whereas Method C leads to the structuring of the process according to the predetermined criteria as to which parts of the process are of greatest interest.

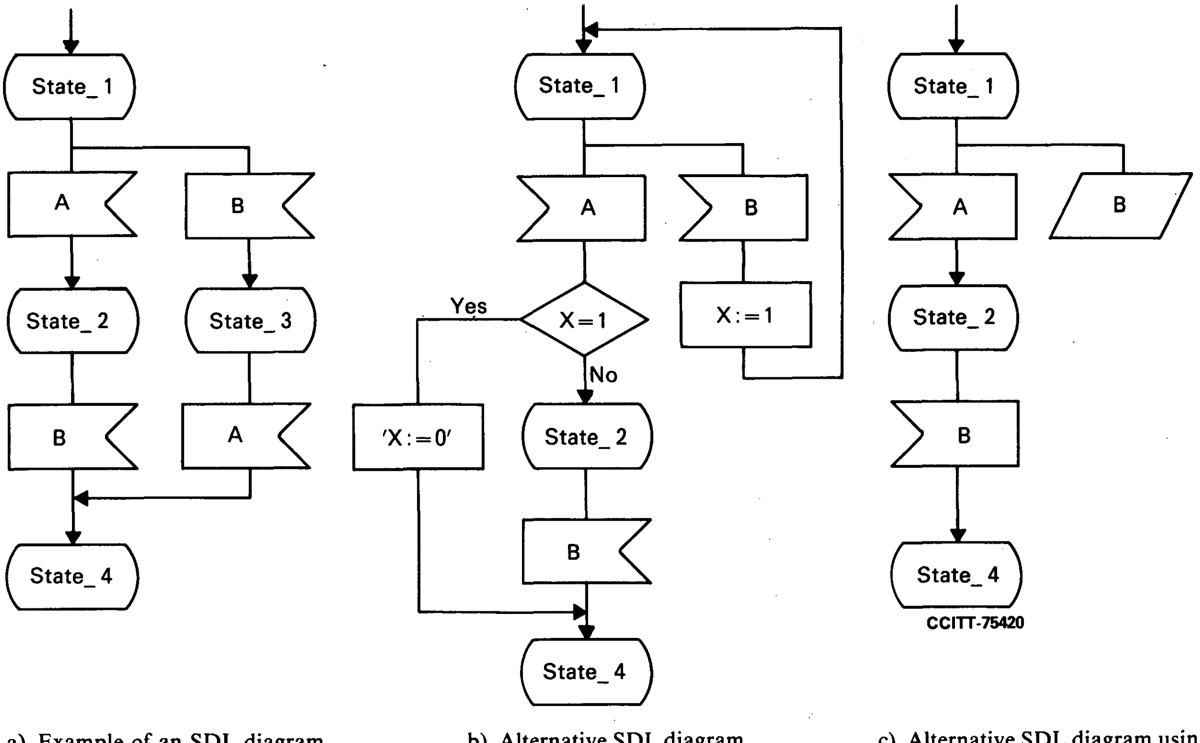
Clarity of diagrams

Since SDL diagrams are a medium of communication between the authors and readers, it is highly desirable that the diagrams are clear. The clarity of the diagram depends not only on the method of arranging the diagram but also on the complexity of the process, (e.g. the number of states, inputs and decisions) which in turn is a function of structuring.

As diagrams should be easy to read and comprehend, some care should be taken to achieve a suitable layout.

In search for the clearest method of representation, the author should be aware of different methods of using SDL. The examples shown in a), b) and c) of Figure D-84 can be considered to be alternative ways of representing a part of the process. As no tasks or outputs are performed (except for the artificial task of setting or resetting the variable X) the 3 examples are indeed logically identical.

Alternative c) is considered to be the clearest representation.



a) Example of an SDL diagram

b) Alternative SDL diagram incorporating a decision

c) Alternative SDL diagram using a save

FIGURE D-84

The three diagrams show the process going from state 1 to 4 on receiving input A and B in random order

D.6.3.6.3.3 *SDL/PR text in process diagrams*

When describing a process definition in SDL/GR data type definitions and variable declarations are described in SDL/PR syntax. This text may be either included directly in the diagram or in a separate document. When the text is not included directly in the diagram there must be a reference to the document where it is contained. See Figures D-85 and D-86.

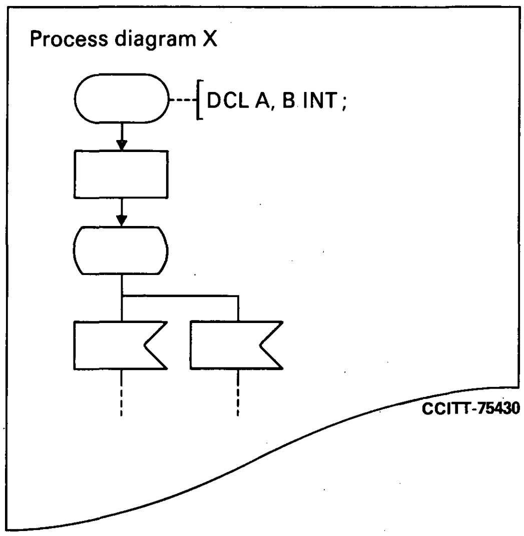


FIGURE D-85

Example of included SDL/PR text in a process diagram

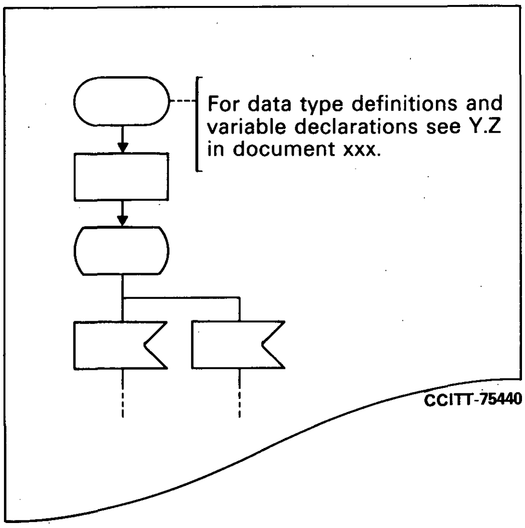

For data type definitions and variable declarations see Y.Z in document xxx.
CCITT-75440

FIGURE D-86

Example of references to definitions contained elsewhere

The method chosen is dependent on the amount of text to be included or referenced. Large amounts of included PR text may destroy the clarity of a diagram, on the other hand: if the volume of text is small it may be more readable to have the text included in the diagram.

D.6.3.6.3.4 *Procedure definitions*

When procedures are defined local to processes, the procedure diagrams may be either included in the process diagrams or referenced.

When procedure diagrams are included it is important that the graph defining the procedure is separated from the graph defining the process. One way of doing this is to form chapters or sections within the process diagram, as can be seen in Figure D-87.

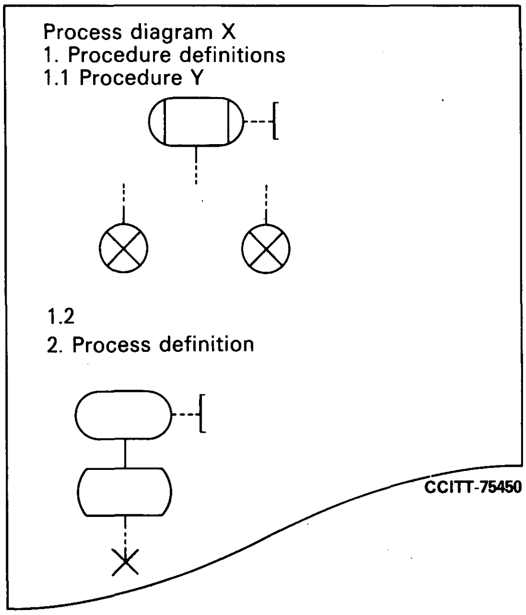


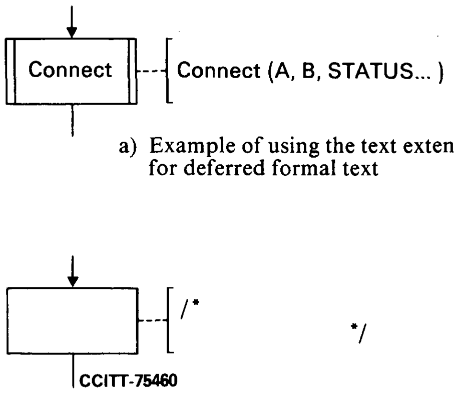
FIGURE D-87

Example of including procedure definitions in a process diagram

D.6.3.6.3.5 *Text in symbols*

Normally text associated with a symbol should be placed inside that symbol. This is, however, not always practical due to the amount of text and/or the size of the symbols.

If necessary, the text may be placed outside the symbol. If this alternative is chosen it must be made clear that the text placed outside the symbol is associated with the symbol and that it is not a comment. This is made using the text extension symbol as shown in Figure D-88.



a) Example of using the text extension symbol for deferred formal text

b) Example of using a 'note' – mechanism to associate in formal text

FIGURE D-88

D.6.3.6.4 *Creation of processes*

D.6.3.6.4.1 *Create request*

The creation of a new process instance is described by the create request symbol. It should be noted that process instances may only be created within the same block as the creating instance.

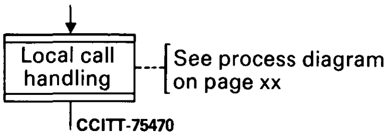


FIGURE D-89

Example of use of the create request symbol

A reference to the process diagram of the created process, should be inserted, as a comment, to make the diagram clearer. When actual parameters are required they should be supplied using the SDL/PR syntax.

D.6.3.6.4.2 *Start*

The start symbol of a process diagram describes the starting point of the behaviour of a process instance when it is created. The formal parameters of the process should be associated with the symbol.

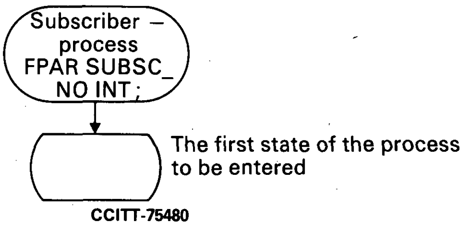
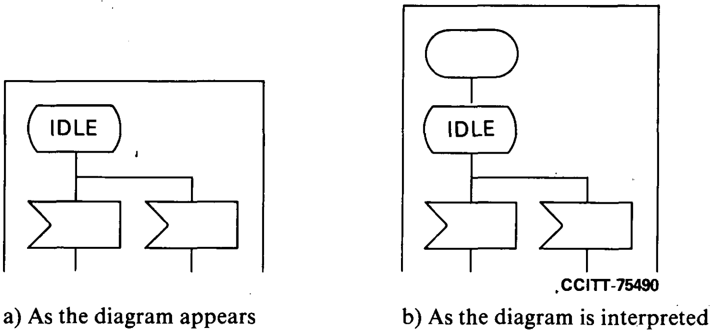

Subscriber – process FPAR SUBSC_ NO INT;
The first state of the process to be entered
CCITT-75480

FIGURE D-90

Example of the use of the start symbol

In order not to invalidate old SDL-diagrams the required start symbol may be implied; it is then assumed to appear before the "first" state symbol in the process diagram, where "first" means the first state symbol appearing at the top of the first page of the diagram.



a) As the diagram appears

b) As the diagram is interpreted

FIGURE D-91

Implied start symbol

The start symbol should appear as the first symbol of a diagram; hiding it inside the diagram may confuse the reader.

D.6.3.6.5 *States*

A state is represented by a state symbol and has associated input symbols and may also have save symbols. An example of a state is shown in Figure D-92.

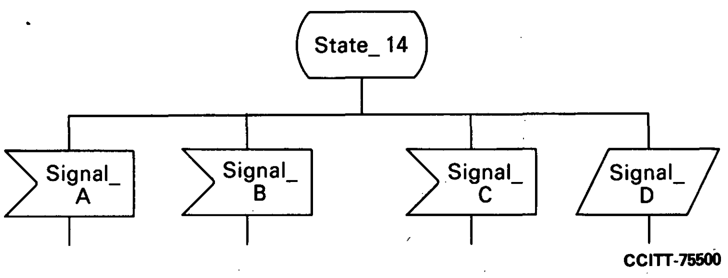


FIGURE D-92

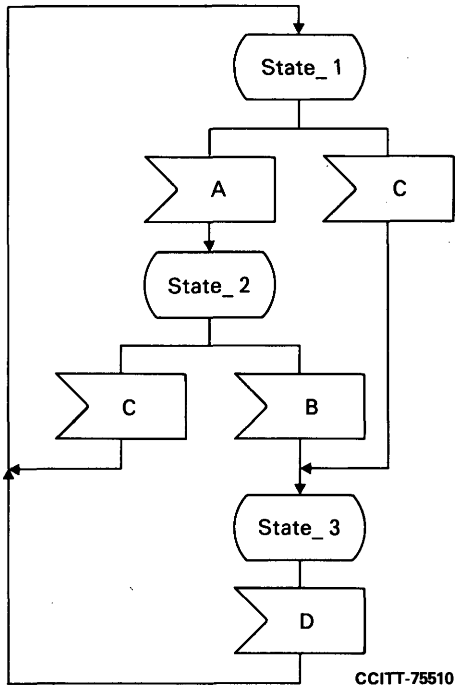
Example of use of state

D.6.3.6.5.1 *Multiple appearances and nextstate symbols*

For convenience, to simplify drawing or as an aid to better understanding, the same state may appear a number of times in an SDL diagram. Where this is done, the diagram is considered to be completely equivalent to the diagram which would result from merging all multiple appearances of the same state. Figures D-93 and D-94 give examples of this. In Figure D-93 b) a state symbol is used as a connector to the main state having the same name, when referred to in a nextstate symbol. In Figure D-94 a state is represented by multiple symbols each with only a subset of the inputs (or saves).

In Figure D-93 diagrams a) and b) are logically equivalent. Diagram a) contains only a single appearance of each state whilst diagram b) uses multiple appearances. In diagram b) the state has one main appearance where all of its related input (and save) symbols are shown. Where this state can be reached from other points in the diagram (as the terminating point of a transition) it is shown as a state without any associated inputs or saves. A comment referencing to the state symbol from the nextstate symbol will improve clarity, particularly when the appearances are on different pages.

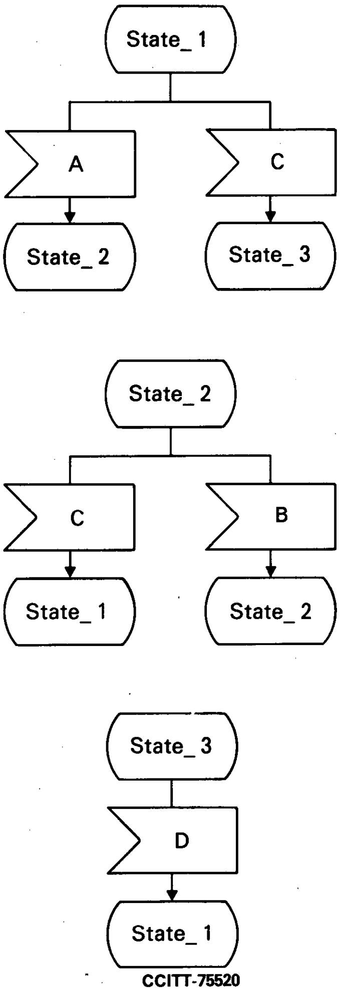
Figure D-94 uses multiple appearances of a state to build up the total set of inputs (and saves). Each occurrence of the state is shown with only a subset of these.



a)

FIGURE D-93

A complete diagram



b)

FIGURE D-93b

Diagram a) with main states and subsequent states used as connectors to the states

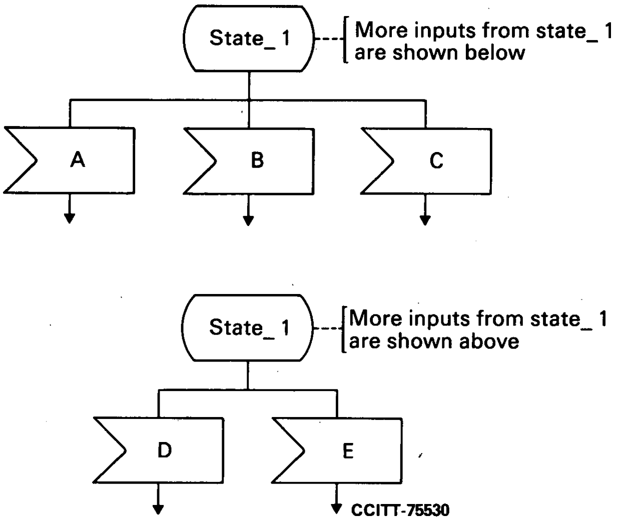

State_ 1
More inputs from state_ 1 are shown below
A
B
C
State_ 1
More inputs from state_ 1 are shown above
D
E
CCITT-75530

FIGURE D-94

Multiple appearances of a state for when all of the inputs can not clearly be drawn from the same symbol

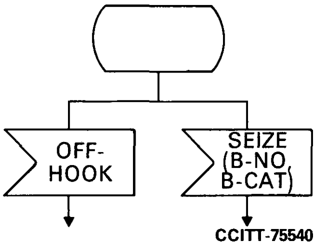
This approach has been successfully used where states have a relatively large number of input or saves but may have the danger that readers misinterpret the diagram if they are unaware that there are multiple appearances. To avoid this misunderstanding, the states showing only a subset of the inputs/saves should be annotated with a comment giving reference to the other states with their associated inputs and saves as shown in Figure D-94.

Multiple appearance may conveniently be used to focus the attention of the reader on certain aspects (e.g. the normal sequence of signals processed) deferring other aspects to other pages (e.g. alarm situation handling).

D.6.3.6.6 *Inputs*

The input symbol, connected to a state symbol, represents the input concept.

When data is conveyed by the signal to be received by the input, the list of local variables to share the values in it are given in the symbol using the SDL/GR syntax. If no date is conveyed the list may be omitted. If data is conveyed and the list is omitted it should be interpreted as that the values will be discarded when received. See the example of Figure D-95.



Note – There should be no arrowhead on the line leading to the input symbol

FIGURE D-95

Example of use of input symbol

If several signals may initiate the same transition after a state, all the inputs may be represented by the same input symbol combining a signal list. In the signal list all the names of the signals (and their lists of variables) appears separated by commas.

In earlier versions of SDL a distinction was made between external and internal inputs. The difference was that external inputs received signals sent from outside the block, in which the process was contained. As there is no semantic difference between signals originating from inside and outside the block these concepts have been removed. The special symbol earlier used to represent an internal input, if found in diagrams, is to be interpreted as an input symbol. (See Figure D-97.)

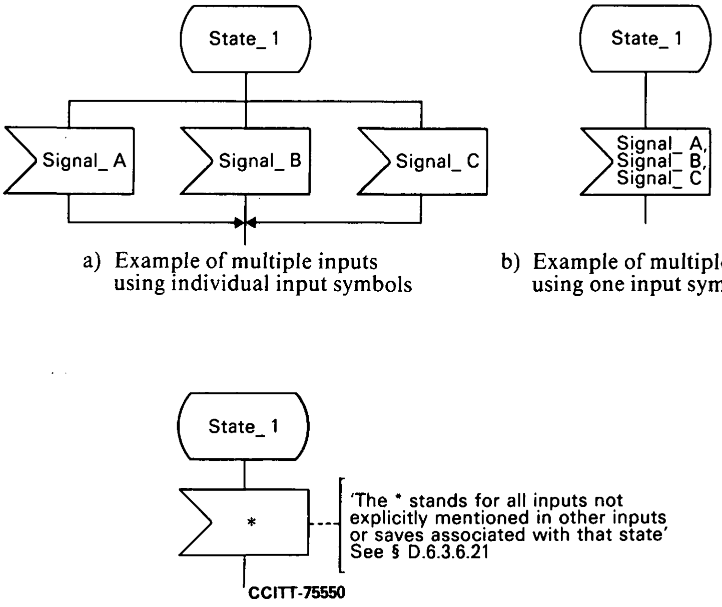
D.6.3.6.7 *Saves*

The save symbol represents the save-signals list of a state. The signals to be saved should be mentioned by name inside the symbol. If several symbols are used connected to one state symbol they are to be considered as one save symbol containing the list of names being the union of the names mentioned in all the symbols.

The names in the name lists should be separated by commas. There is also a short hand representation using the "asterisk notation" explained in § D.6.3.6.21.

D.6.3.6.8 *Outputs*

An output, sending a signal and providing it with values to be conveyed, is represented by the output symbol. The list of values to be conveyed is specified using the SDL/PR, and the list should preferably be contained in the symbol. The symbol must also contain the address of the process instance the signal is sent to. See examples in Figure D-99.



a) Example of multiple inputs using individual input symbols

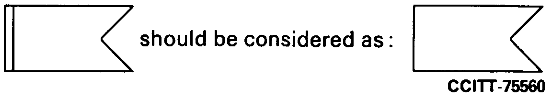
b) Example of multiple inputs using one input symbol

'The * stands for all inputs not explicitly mentioned in other inputs or saves associated with that state'
See § D.6.3.6.21

c) Example of multiple inputs using asterisk notation

FIGURE D-96

Alternative representation of multiple inputs

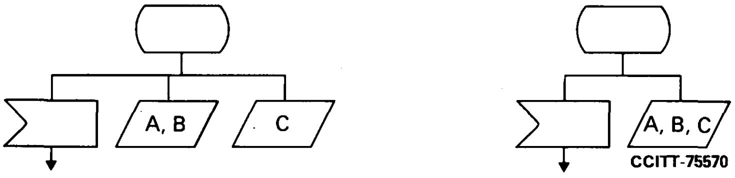


a) Old symbol

b) New symbol

FIGURE D-97

Equivalence of external and internal input

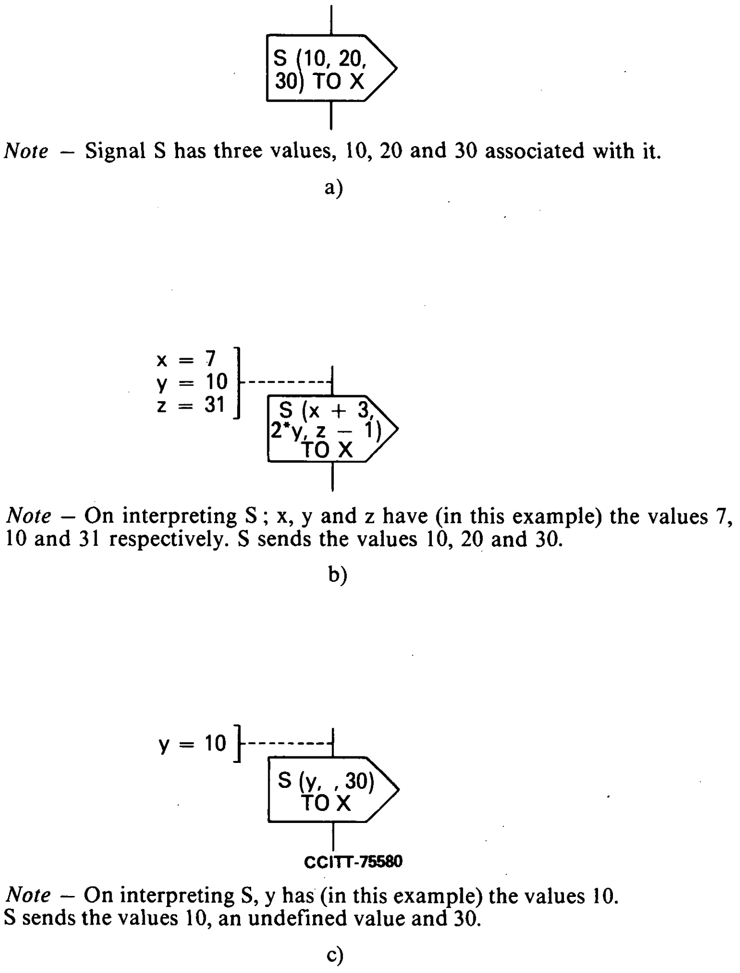


a)

b)

FIGURE D-98

Different representations of the same save list



Note – Signal S has three values, 10, 20 and 30 associated with it.

a)

Note – On interpreting S ; x, y and z have (in this example) the values 7, 10 and 31 respectively. S sends the values 10, 20 and 30.

b)

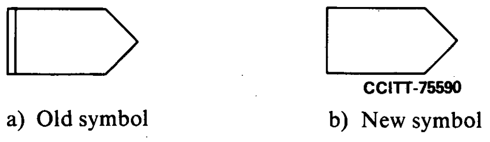
Note – On interpreting S, y has (in this example) the values 10. S sends the values 10, an undefined value and 30.

c)

FIGURE D-99

Output with associated data

In older versions of SDL a distinction was made between external and internal outputs in the same manner as for inputs. Any old internal output symbol found in diagrams should be interpreted as an output symbol. See Figure D-100.



a) Old symbol

b) New symbol

FIGURE D-100

Equivalence of external and internal output

D.6.3.6.9 *Enabling condition*

The enabling condition is represented by the enabling condition symbol which is a combination of the input symbol and a graphical syntax representing the Boolean condition.

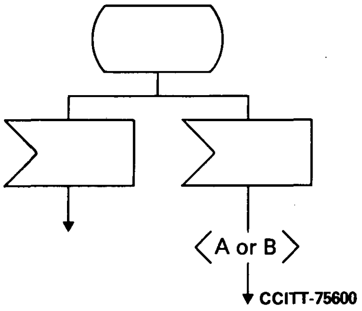


FIGURE D-101

The combination should be considered as one symbol, which is why there should be no arrowhead on the line connecting the two parts. See Figure D-101.

D.6.3.6.10 *Continuous signal*

The continuous signal is represented by the continuous signal symbol.

The contained expression should be Boolean.

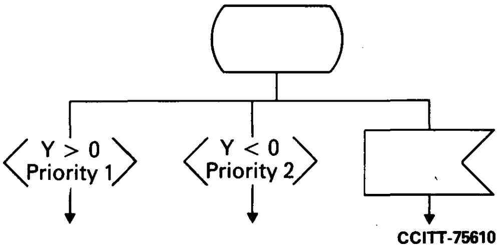


FIGURE D-102

Example of enabling conditions

The priority clause is mandatory if there is more than one enabling condition associated to a state.

D.6.3.6.11 *Task*

A task is represented by a task symbol. The description of the task, either in SDL/PR text on informal text should preferably be contained in the task symbol.

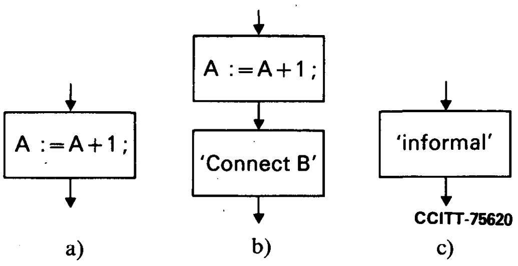


FIGURE D-103

Examples of task symbols

In diagrams containing both formal statements and informal text, the informal text should be enclosed between apostrophes.

D.6.3.6.12 *Decisions*

A decision is represented by a decision symbol.

The text of a question related to a certain decision is placed in the decision symbol. The questions need not contain question marks, but just the expression to be evaluated, e.g. no., A + B, Z. The symbol must have two or more branches. The answer to the question is placed directly beside and to the right of or on the top of the corresponding branch. The branches must together contain all possible answers. Each answer can be shown with either the value of the answer (e.g. 17, 12) or with an operator as a prefix (e.g. > 18, ≠34, = 25). Some possible formats are shown in Figure D-104.

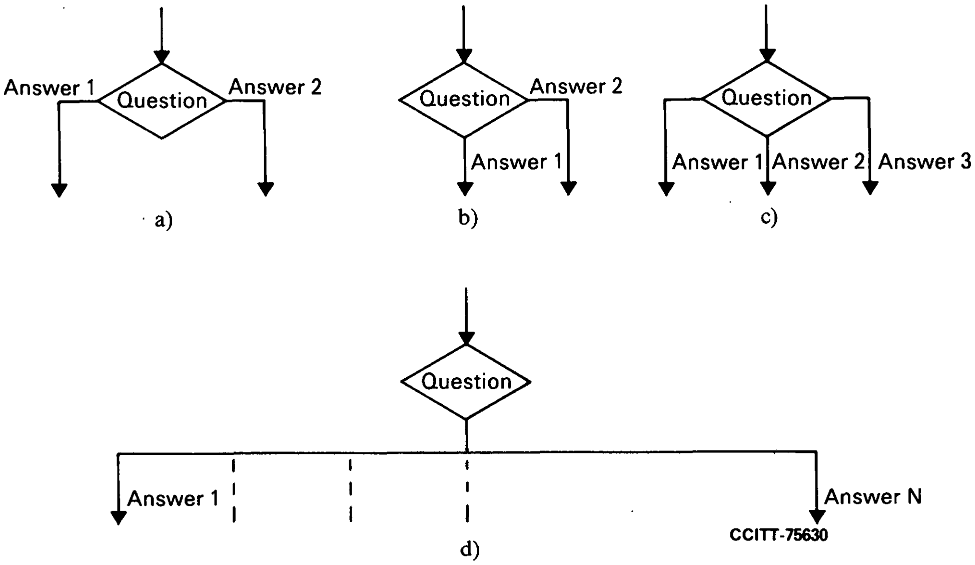


FIGURE D-104

Some possible formats for the decision symbol

Many questions are related to one specific condition in which the only possible answers are "Yes" or "No", i.e. TRUE or FALSE.

For example:

B subscriber busy

telephone set "on-hook"

digit 2 received

Z = 1

An example of this is given in Figure D-105.

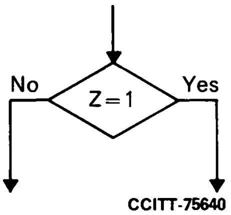


FIGURE D-105

Formatting of a simple binary decision

In the case of more than two answers, the formats in Figure D-106 are possible, assuming Z to be a positive integer. The ELSE answer means any answer not covered by the other answers, i.e. values different from the set (1, 2, 3, 9, 10, 11, 12, 13, 14, 15).

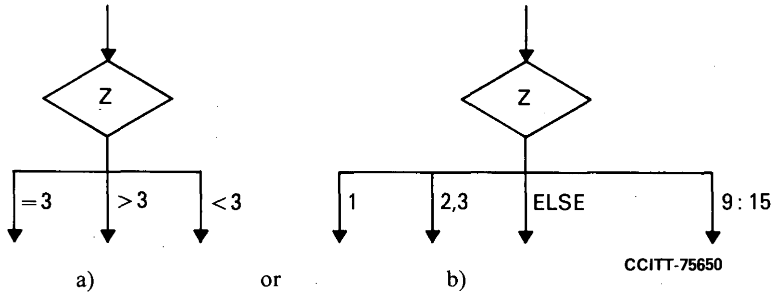


FIGURE D-106

Formatting of questions and answers

D.6.3.6.13 *Macro*

Macro symbols can be placed anywhere in a diagram. The correctness of the diagram can only be determined after the macro symbol is replaced by the contents of its definition. However, the preferred use of macros is in process diagrams where a task symbol could occur.

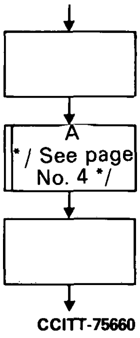


FIGURE D-107

Use of macro symbol

When using macros it is useful in the macro symbol to provide a reference of where to find the definition (Figure D-107 shows an example).

When using macros the user should be aware that macros may hide side effects. As a macro normally appears in the place of a task symbol, readers may assume the semantics of a task.

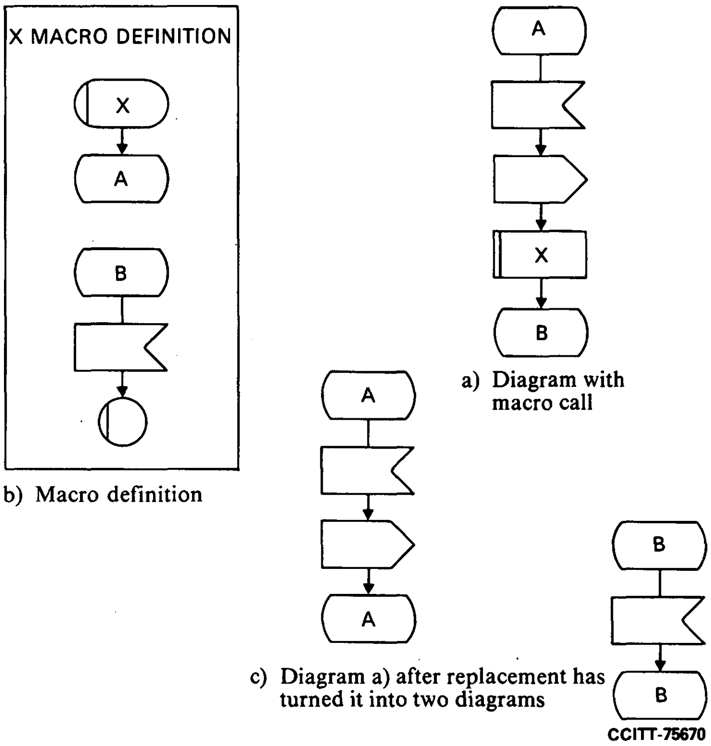

X MACRO DEFINITION
X
A
B
b) Macro definition
A
X
B
a) Diagram with macro call
A
A
B
B
c) Diagram a) after replacement has turned it into two diagrams
CCITT-75670

FIGURE D-108

Example of not recommended use of macros

In Figure D-108 is shown how the replacement of a macro call may change the sequence in the diagram. When macros contain states it should be noted that the transition, containing the reference, may be split into several transitions.

When macros contain several inlets and/or outlets they should be clearly identified by labels, preferably placed on the right of the in-on outgoing flow lines.

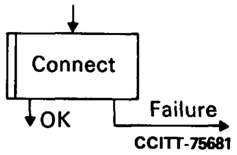


FIGURE D-109

Macro with several outlets

D.6.3.6.14 *Procedures*

The procedure call symbol represents the call of a procedure. The parameters provided with the call should preferably be placed in the symbol, if this is not possible or practical they may be placed in a text extension symbol at the side.

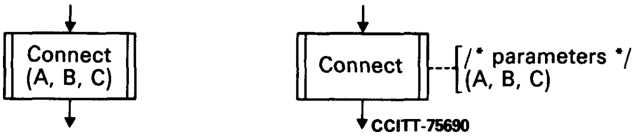


FIGURE D-110

Example of use of the procedure call symbol

The symbol may occur wherever a task symbol may be placed.

The procedure is a tool to reduce the complexity of a diagram, and to make use of standard solutions.

D.6.3.6.15 *Option*

The option symbol is used to describe several alternative process behaviours with one diagram. This is a useful facility if several process behaviours only differ in small parts. The symbol contains an option expression. This expression should be such that it can be evaluated before the process is interpreted, i.e. it must not be a dynamic condition. The expression should also be such that the evaluation leads to a discrete set of alternatives, which the outgoing flowlines can be labelled with.

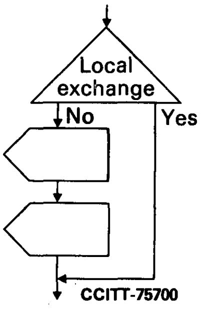


FIGURE D-111

Example of use of options

The option alternatives should preferably be placed to the right of the outgoing flowlines.

The diagram may not be interpreted until all option expressions are evaluated, and the diagram should then be considered as if all of the non-reachable branches following options were deleted.

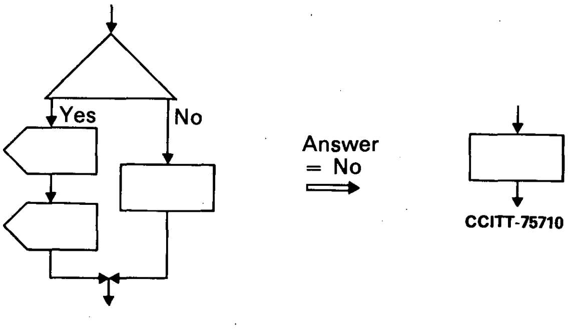


FIGURE D-112

Interpretation of options

D.6.3.6.16 *Connectors*

General

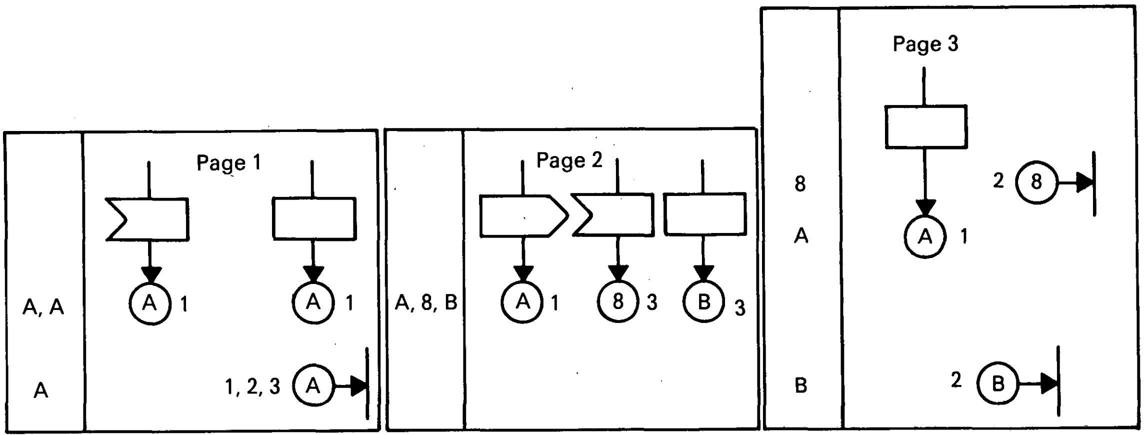
In drawing a large SDL diagram, it may also be necessary to split the diagram due to the lack of space. Connectors are used for this purpose.

Connectors should also be used to avoid the crossing of flowlines which would make diagrams somewhat unclear. It is normally preferable to draw an SDL diagram with the flow from the top to the bottom of the page.

Use of connectors to show implied flowlines

Any flowline may be broken by a pair of associated connectors, with the flow assumed to be from the out-connector to the in-connector. Each connector has a name. Associated connectors have the same name. For each name only one in-connector exists but there may be one or more out-connectors.

It is desirable that the page reference for the appropriate in-connector should be given next to each out-connector, and that the page reference(s) for the appropriate out-connector(s) should be given next to each in-connector. See Figure D-113. It is also desirable to have a connector reference column in the margin of the page indicating the horizontal place and the order of the connectors.

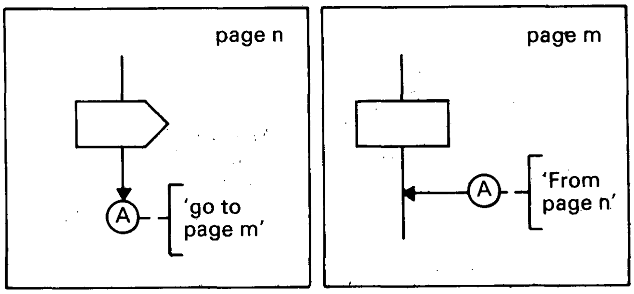


CCITT-75720

Note – An alternative and equally acceptable, method of drawing page references is to use comments, as shown in figure D-114.

FIGURE D-113

Method of drawing page references and connectors references



CCITT-75730

FIGURE D-114

Alternative method of drawing page references

Connectors can only be used for applying connections within the same process.

If major flow paths exist, attention can be concentrated on them by not breaking them up with connectors except at the top and bottom of pages. A minor flow path within an SDL diagram can instead be broken with an out-connector, the associated in-connector being placed on some other suitable page, such as a page following the end of the major flow paths.

D.6.3.6.17 *Divergence and convergence*

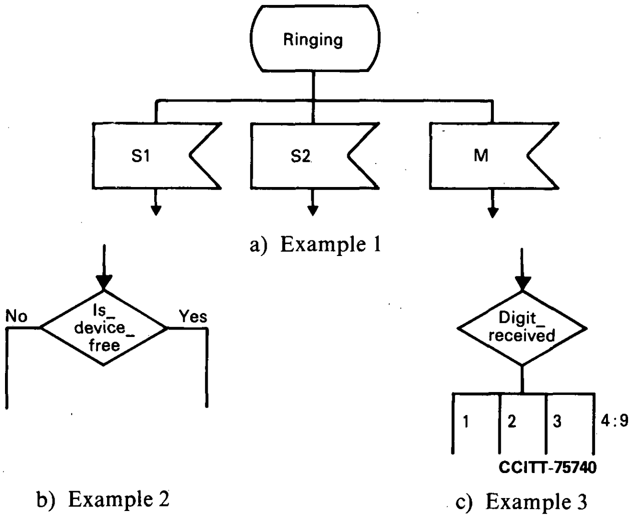
Divergence

Divergence within a transition of an SDL diagram may only occur:

a) between a state symbol and its associated input and save symbols, or;

b) immediately after a decision symbol.

Some examples of divergence are shown in Figure D-115.



a) Example 1

b) Example 2

c) Example 3

FIGURE D-115

Examples of divergence

Convergence

The point of convergence cannot occur between a state symbol and an input or save symbol, but may occur at any other point in an SDL diagram.

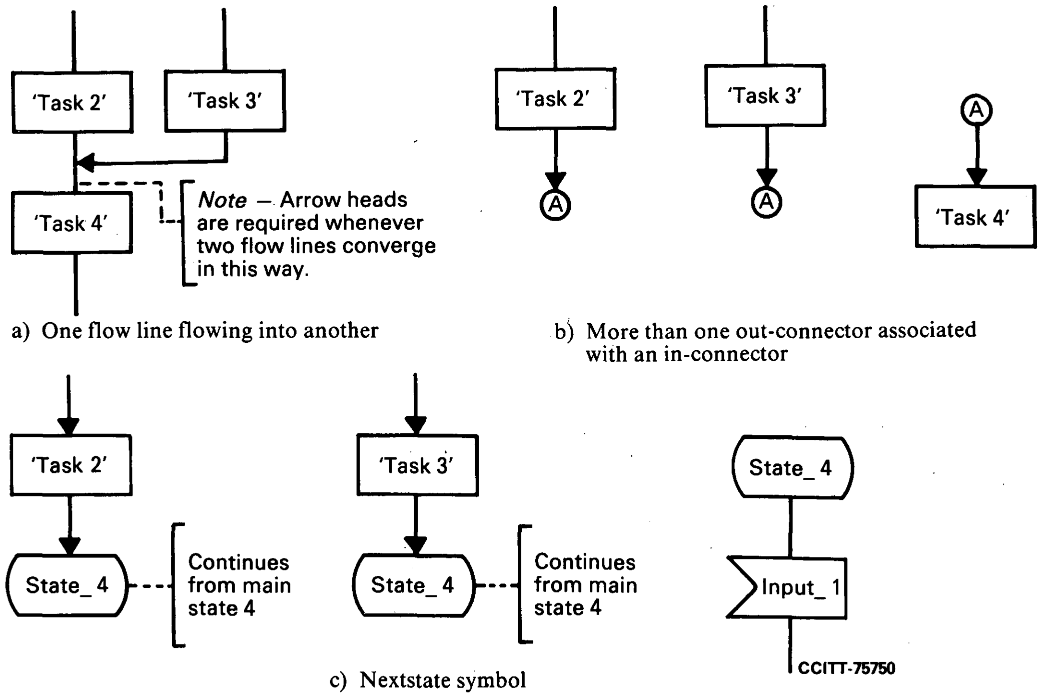
Convergence may appear in any of the ways shown in Figure D-116.

The use of convergence can reduce the number of symbols in an SDL diagram where a sequence of symbols and any associated text is repeated within the same SDL diagram as shown in Figure D-117.

D.6.3.6.18 *Comments*

Comments may be added to an SDL diagram to clarify some part of the diagram. Comments are to help the reader and have no effect on the interpretation of the SDL diagram. Comments should therefore neither contradict nor add anything to the semantics of the diagram.

A comment is attached to a flowline or an SDL symbol by a single square bracket connected by a dashed line to the flowline or symbol. See Figure D-118.


'Task 2'
'Task 3'
'Task 4'
Note — Arrow heads are required whenever two flow lines converge in this way.
A
State_ 4
Continues from main state 4
Input_ 1
CCITT-75750

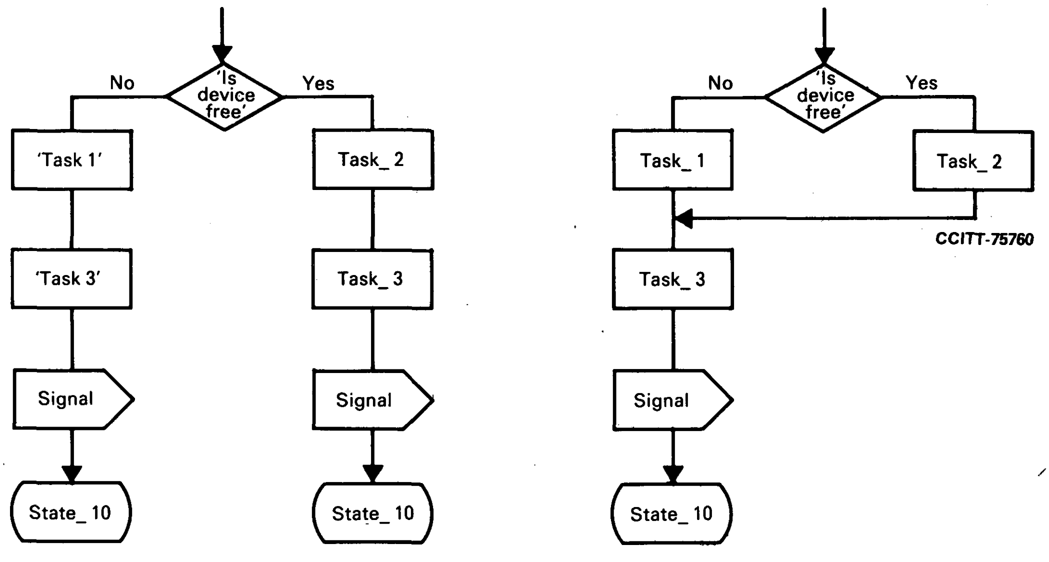
a) One flow line flowing into another

b) More than one out-connector associated with an in-connector

c) Nextstate symbol

FIGURE D- 116

Examples of convergence



a) Example a)

b) Example b)

Note — Example b) is equivalent to Example a)

FIGURE D-117

Use of convergence

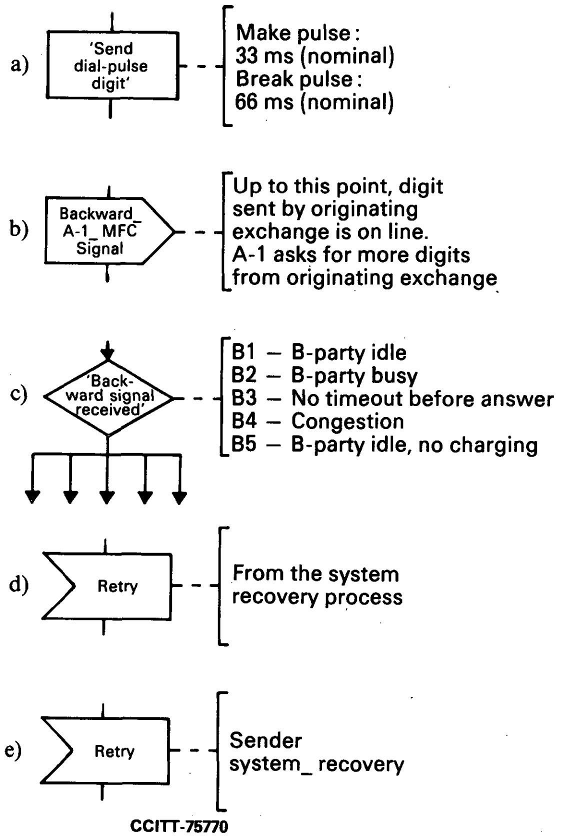

a)
'Send dial-pulse digit'
Make pulse: 33 ms (nominal) Break pulse: 66 ms (nominal)
b)
Backward_ A-1_ MFC Signal
Up to this point, digit sent by originating exchange is on line. A-1 asks for more digits from originating exchange
c)
'Back-ward signal received'
B1 – B-party idle
B2 – B-party busy
B3 – No timeout before answer
B4 – Congestion
B5 – B-party idle, no charging
d)
Retry
From the system recovery process
e)
Retry
Sender system_ recovery
CCITT-75770

FIGURE D-118

Examples of use of comments

Comments may also be inserted using SDL/PR syntax (/* . . . . */) as can be seen in Figure D-119.

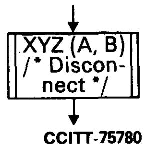


FIGURE D-119

Example of use of SDL/PR comment syntax in SDL/GR

D.6.3.6.19 *Text extensions*

Normally the text associated with a graphical symbol should be placed inside that symbol. However, this is often not possible or practical. An alternative is to place the text in a text extension symbol connected to the associated symbol. See Figure D-120.

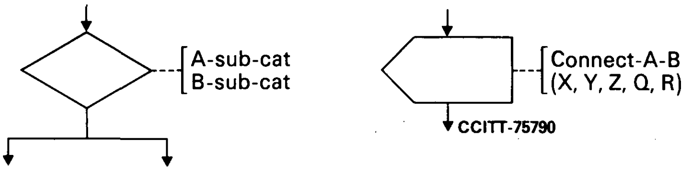


FIGURE D-120

Example of use of the text extension symbol

As the syntax is similar to the comment syntax it is important that the connecting line is drawn solid, in contrast to the dashed line of the comment symbol.

If informal text is used in text extension symbols it should be enclosed within apostrophes.

D.6.3.6.20 *Data*

Generally, the SDL/PR syntax is used for data constructs in SDL/GR. When using data in tasks, outputs, decisions or inputs, the appropriate SDL/PR text is associated with the symbol.

Data type definitions and variable declarations are either contained in separate documents and referred to, or contained in the process diagram. These definitions should be associated with the start symbol of the diagram.

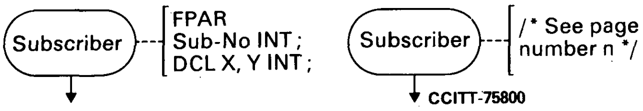


FIGURE D-121

Examples of data definitions in an SDL/GR process diagram

The association to the symbol is made using the text extension symbol as can be seen in Figure D-121.

D.6.3.6.21 *Shorthand notations*

In order to make diagrams more readable and to avoid long name lists the following shorthand notations are introduced. The symbols used are the asterisk (*) and the hyphen (-). Generally, "*" has the meaning of "all" or "all but", and the "-" the meaning of "the same".

The shorthand notations may be used in the STATE, INPUT, SAVE and NEXTSTATE symbols.

In the state symbol an asterisk may be used either alone, or combined with a state name list enclosed in brackets.

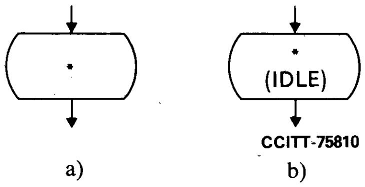


FIGURE D-122

Examples of "*"-notation in states

The asterisk in Figure D-122 should be interpreted as "all states" for a) and combination b) as "all states except IDLE". The notation is using the multiple appearance facility of state symbols (see § D.6.3.6.5).

In inputs an asterisk may only be used in one of the inputs connected to a state, provided it does not appear in a save symbol connected to the same state.

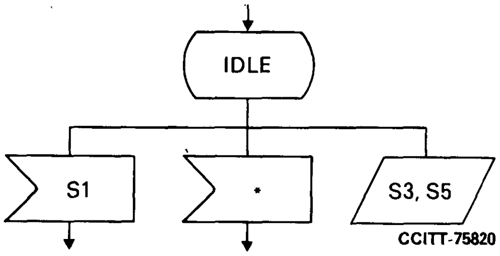


FIGURE D-123

Use of an asterisk in an input symbol

The interpretation of the asterisk in Figure D-123 is "all signals but the signals mentioned in other inputs or saves, i.e. S1, S3 and S5".

The use of an asterisk in a save symbol is similar. It may only appear in one of the saves connected to a state, provided it does not appear in any input connected to the same state.

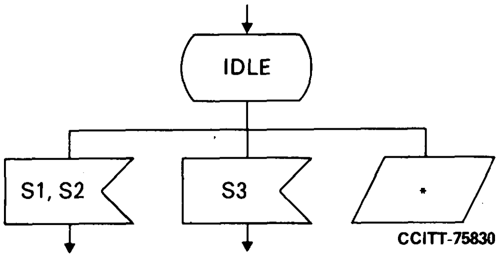


FIGURE D-124

Use of an asterisk in a save symbol

In Figure D-124 the interpretation of the asterisk is "all signals except S1, S2 and S3".

In a nextstate symbol a hyphen "-" may be used to represent "the same state as the one the transition originated from".

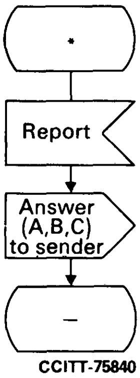


FIGURE D-125

Use of hyphen in a nextstate symbol

The interpretation of Figure D-125 is that in any state of the process, the signal "REPORT" can be received. The reception will cause the sending of signal "ANSWER" and the transition will terminate in the state it originated from.

In combination with the multiple appearance facility these shorthand notations have great expressive power. Note that simple additions to a diagram, using "*" and "-" may change the meaning of a diagram in an unexpected way.

As an example, the addition to the diagram shown in Figure D-126 will have the effect that in all states there will be no implicit transitions.

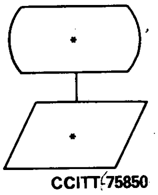


FIGURE D-126

An effect of the diagram is that "*" is not allowed in any input, state or save symbol throught the diagram

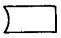
D.6.3.6.22 *State pictures*

D.6.3.6.22.1 *State picture option*

If the state picture option is chosen, it is useful to consider each state picture as consisting of the three types of elements shown in Figure D-127 a).

These three types are often combined to form a composite pictorial element that can be understood in isolation from the rest of a state picture. For example Figure D-127 b) gives the meaning of a multi-frequency code receiver that is awaiting a forward signal.

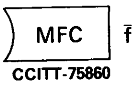
Note that the recommended pictorial element symbol for time supervising of a process incorporates the relevant input variable t_i.

1) Pictorial elements, e.g.  (meaning a signalling receiver);

2) Input variables, e.g. $\bar{f}$ (meaning that a forward signal has yet to be recognized); and

3) Qualifying text, e.g. MFC (meaning multi-frequency code).

a) The content of a state picture



b) A composite pictorial element

FIGURE D-127

Construction of pictorial elements

D.6.3.6.22.2 *Input variables*

Input variables are a valuable way of representing those conditions associated with the process which, if changed, will result in a transition by the process. Input variables should be shown using lower-case letters, in order to distinguish them readily from qualifying text (which should be shown using upper-case letters). (Changes in input variables correspond to input signals, which cause the process to leave its current state; changes in qualifying text do not correspond to input signals.)

D.6.3.6.22.3 *Qualifying text*

Qualifying text should be heavily abbreviated, and where possible should be placed inside appropriate pictorial elements. It is then quite obvious which pictorial elements are being qualified.

D.6.3.6.22.4 *Completeness of state pictures*

Sufficient pictorial elements should be assigned to each state picture in order to show:

a) what resources are being considered by this process during the current state. Examples of resources are: switching paths, signalling receivers, signalling senders and switching modules;

b) whether one or more timers are currently supervising this process;

c) in the case that this process concerns call-handling, whether call charging is currently in progress or not, and which subscribers are being charged during this state of the call;

d) the current categories (or status) of equipment allocated to this process, where this information affects the behaviour of this process;

e) the status of those input variables which are monitored by the process during the current state.

D.6.3.6.22.5 *Example*

An example of the application of the principle expressed above is illustrated by the state picture in Figure D-128. It will be seen that during this state:

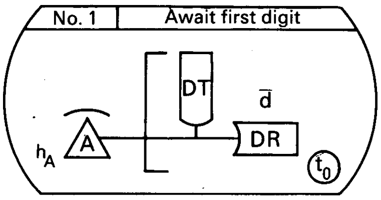
a) that the resources considered by the process consist of a subscriber's handset, a digit receiver, a dial tone sender and switching paths connecting these items;

b) that a timer T_0 (whose current condition is t_0) is currently supervising this process;

c) that no charging of the call is in progress;

d) that the subscriber has been identified as an A-party, but no other category information is taken into account;

e) that the following input variable conditions apply: h_A (i.e. the handset is off-hook, $\bar{d}$ (i.e. the digit receiver is awaiting a digit) and t_0 (i.e. the supervising timer T_0 is running).



CCITT-75870

FIGURE D-128

Example of a state in a call-handling process

D.6.3.6.22.6 *Consistency checking feature of SDL diagrams with pictorial elements*

If the principle expressed in § D.6.3.6.22.4 e) is followed, it is always possible to deduce the set of inputs which will cause the process to leave each of its states, simply by looking at the input variable shown in the state picture. For example, by looking at the state picture in Figure D-128, one can deduce that three different inputs will cause the process to leave this state: the handset condition changing to on-hook (input $\bar{h}_A$), the arrival of a digit (input d), or expiry of timer T_0 (input $\bar{t}_0$). In this way, some "surprise transitions" can be avoided when implementing systems from very complex SDL specifications.

It is clear that the pictorial is more compact and in a certain sense puts more information under the eyes of the readers; but at the same time one has to look very closely to the pictorial to work out the exact set of actions performed in the transition.

In addition it is not possible to tell, looking at the pictorial diagram alone, whether an action is performed through an output or a task (§ D.6.3.6.22.8).

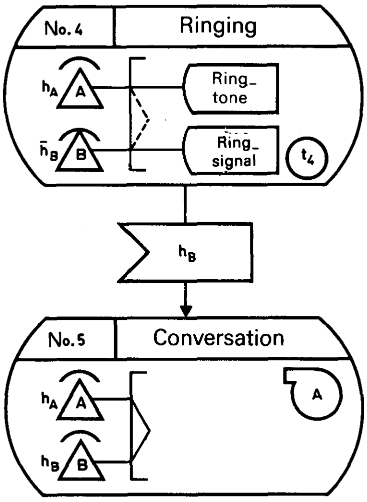
D.6.3.6.22.7 *Use of the block boundary symbol*

Pictorial elements shown outside the block boundary symbol imply elements that are not directly controlled by the given process, and pictorial elements shown within the block boundary symbol imply elements that are directly controlled by the given process. For example, the call process that is partially specified in Figure D-129 can connect or disconnect ring current, and start or stop timer T_4, but it cannot change either subscriber's handset condition.

In designing logic from an SDL specification with pictorial elements, only those pictorial elements shown inside the block boundary will affect the processing actions performed during transition sequences. The composite pictorial elements shown outside the block boundary are normally included in a state picture:

a) because they indicate input variables which must be monitored by the process during the given state; and/or

b) to improve the intelligibility of the diagram.



CCITT-34080

FIGURE D-129

Example of a transition between two states, where all processing actions are implied by the differences in the state pictures

D.6.3.6.22.8 *Task or output*

The interpretation of a processing action as either a task or an output sometimes appears to be arbitrary. In fact the decision to interpret the appearance or disappearance of a pictorial element within the functional block boundary as either a task or an output can only be resolved by consulting the process signal list.

D.6.3.6.22.9 *Use of the timer symbol*

Whether or not pictorial elements are used, the interruption of the supervised process upon expiry of a timer is always represented by an input.

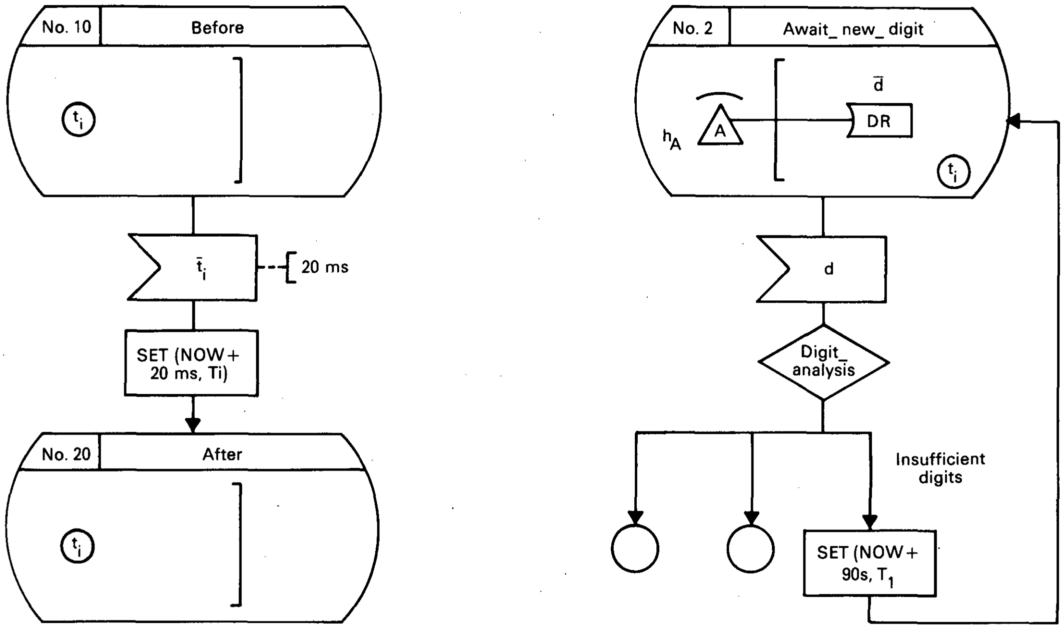
The presence of a timer symbol in a state picture implies that a timer is running during that state. Following the general principle stated in the Recommendations, the starting, stopping, restarting and expiry of timers is shown using pictorial elements in the following manner:

a) To show that a timer is started during a given transition, the timer symbol should appear in the state picture corresponding to the end of that transition but not in the state picture corresponding to the start of that transition.

b) To show that a timer has been stopped during a transition, the timer symbol should appear in the state picture corresponding to the start of that transition but not in the state picture corresponding to the end of that transition.

c) To show that a timer has been restarted during a transition, an explicit task symbol should be shown in that transition. (Two examples are shown in Figure D-130).

d) The expiry of a particular timer is shown by an input symbol associaed with a state in which the state picture includes the corresponding timer symbol. Of course more than one timer may concurrently supervise the same process if required, as shown in Figure D-131.



CCITT-75900

a) Restart of a timer following expiry of the timer

b) Restart of a timer when the timer has not yet expired

Note – Each timer T_i has two mutually exclusive conditions t_i and $\bar{t}_i$.

FIGURE D-130

Examples showing the restart of a timer

D.6.3.6.23 *Auxiliary documents*

To aid the reading and understanding of large process diagrams, auxiliary documents giving overviews are helpful. These documents are informal and serve only to give overviews and to be "lead-in" documents to the SDL diagrams. The users are free to find their own suitable syntax.

In this chapter examples are given for two types of auxiliary documents; the state overview diagram and the state/signal matrix.

D.6.3.6.23.1 *State overview diagram*

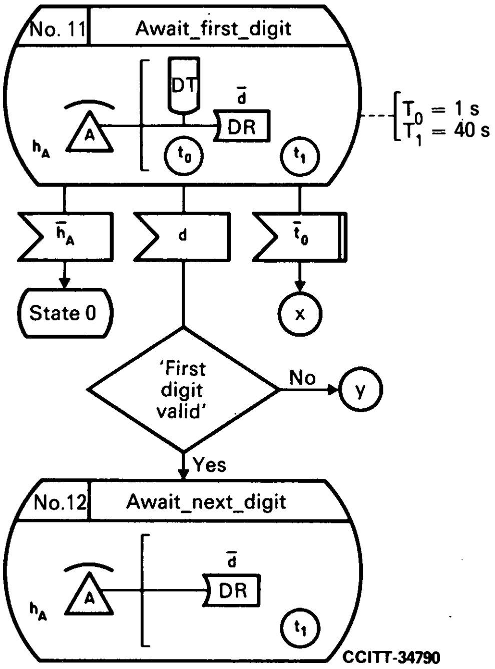
The intention with the state overview diagram is to give an overview of the states of a process and what transitions between these are possible. As the intention is to give an overview, unimportant states or transitions may be omitted.

The diagrams are composed of state symbols, directed arcs representing transitions, and optionally start and stop symbols.

The state symbol should contain the name of the referred state. The symbol may contain several state names and an asterisk (*) may be used as notation for all states.

To each of the directed arcs the name of the signal, or set of signals causing the transition can be given.

The use of the start and the stop symbol is optional.



Timer T_0 supervises the arrival of the first digit, whereupon dial tone is removed from the call and timer T_0 is stopped. Timer T_1 continues supervising the arrival of sufficient digits to permit routing of the call.

FIGURE D-131

Example of the use of two supervising timers in the same state

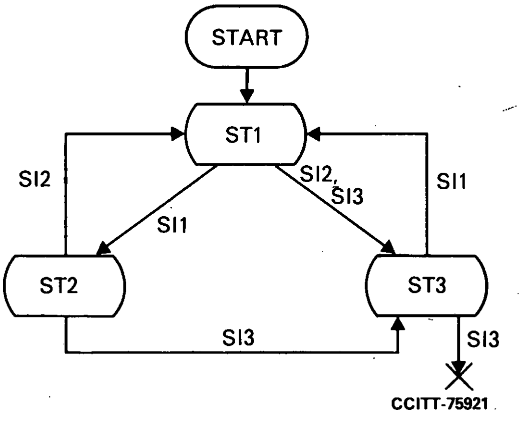


FIGURE D-132

Example of a state overview diagram

A state overview diagram for a process may be separated into several diagrams each covering different aspects, e.g. "normal case", fault-handling, etc.

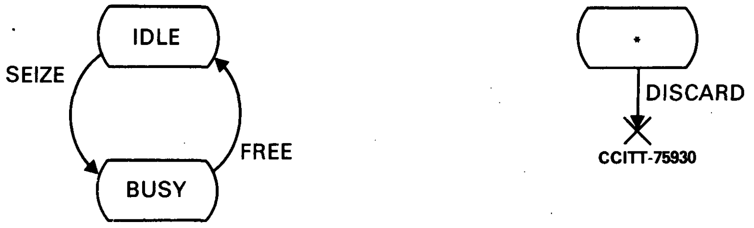


FIGURE D-133

Example of a separated state overview diagram

In order to be well-structured and easy to read, state overview diagrams should preferably:

- conform to the normal reading directions – top-down/left-right;
- be contained on one page.

D.6.3.6.23.2 *State/signal matrix*

The state/signal matrix is intended to be a "lead-in" document to a large process diagram. It gives references to where in the diagram a combination of a state and a signal is found.

The diagram consists of a two-dimensional matrix indexed on one axis by all the states of a process, and on the other by all valid input signals for a process. In each of the matrix elements a reference is given to where to find the combination given by the indices, if it exists.

STATES / SIGNALS	IDLE	BUSY	BLOCKED	------
OFFHOOK	P5	–	–	
ONHOOK	–	P6	–	
SEIZE	P7	P8	–	
BLOCK	P6	P6	–	
⋮				

CCITT-75940

FIGURE D-134

Example of state/signal matrix

The matrix may be separated into sub-parts contained on different pages. The references are the normal references used by the user in the documentation.

Preferably signals and states should be grouped together, so that each part of the matrix covers one aspect of the process behaviour, e.g. the "normal case", the "exception handling", the "maintenance part", etc. should all be found in different sub-groups.

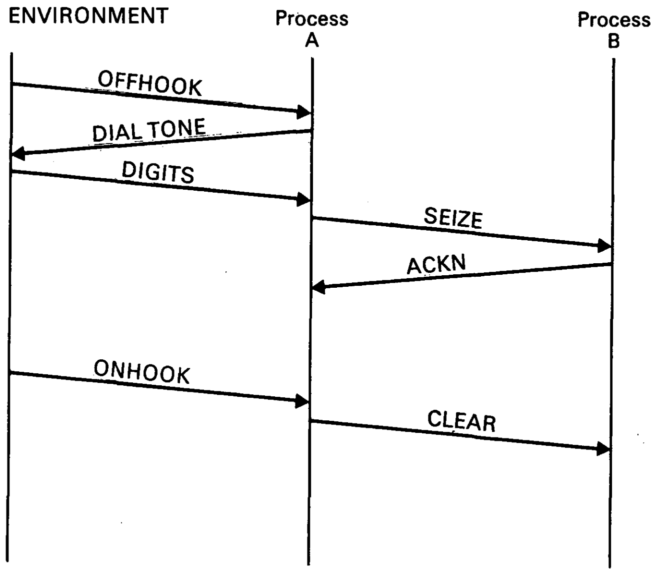
D.6.3.6.23.3 *Sequence charts*

The sequence chart may be provided as a supplement to the interaction diagram to show allowed sequences of signals interchanged between one or more processes and their environment.

The chart is intended to give an overview appreciation of the signal interchange between parts of the system. The complete signal interchange or only parts of it may be shown depending on what aspects are to be highlighted.

The column in the chart indicates the environment (usually placed at the edges .of the diagram) and the various processes or blocks.

Their interactions are represented by the set of diagonal directed lines, each one representing a signal or a set of signals. When the set of signals represented by a line is flowing in both directions it is suggested to have the line drawn horizontally (an example may be "connection set up") with arrowheads at both ends.



CCITT-75950

FIGURE D-135

Example of sequence chart

Each sequence may be annotated so that it is clear what set of information is interchanged. Each line is annotated with the information involved (signal names or procedures).

A decision symbol can be placed at the arrival of a line to indicate that the sequence following is valid if the indicated condition is true. Usually in this case the decision symbol appears several times indicating the different sequences resulting from the different values of the condition.

This diagram can show completely all the sequences of signals interchanged or only a meaningful subset.

A useful application of this diagram is the representation of the mutual interaction of subprocesses resulting from the partitioning of a process. In this case the environment of the subprocesses is the partitioned process environment.

D.7 *Guidelines for representing systems using SDL/PR*

Paragraph D.7 of the User Guidelines explains the textual phrase representation of SDL, named PR, which has a strong resemblance with a programming language.

The first section (§ D.7.1) contains general comments on SDL/PR; on its suitability for use as input to a machine, its similarity to a program, and its differences.

The second section (§ D.7.2) explains how SDL/PR is defined throughout the Recommendations, where the "Syntax Diagrams" are used.

The third section (§ D.7.3) is the most important from the user's perspective. It explains pragmatics about SDL use. In particular, the peculiarities and the expressive power of the textual phrase representation is considered, mentioning aspects such as the order of states, the ordering of transitions within a state, the multiplicity of states, the procedures, macros, labels and shorthand notations.

D.7.1 *Why PR?*

The textual phrase representation of SDL, SDL/PR, was devised during the 1977-1980 study period; in 1980 it was decided to attach its definition as an Annex to the Recommendations, since it was felt that some refinement was needed before it should be recommended. This refinement was accomplished in the following study period and now SDL/PR is one of the recommended concrete syntaxes of SDL.

At first, SDL/PR was intended to be used as an easy way to input SDL documents into a machine, since the GR form is more difficult to input. (Graphical peripherals are needed to handle it.) For this reason, emphasis was on a one-to-one mapping between the PR and GR form. The evolution in graphic terminals (increase of capabilities and decrease of cost) has since led to the acceptability of the GR form as suitable for input into a machine. This does not diminish the importance and the use of the PR form because some users find it more to their liking, especially those working with programming languages.

D.7.1.1 *How close to a program is SDL/PR?*

This evolution led to a looser correlation between GR and PR, so that we can still (easily) map one onto the other, but each language has its own peculiarities. At first glance, the PR form strongly resembles a programming language (see Figure D-136).

```
                 –
                 –
                 –
                 –
STATE aw_off_hook;
INPUT off_hook;
TASK 'activate charging';
TASK 'connect';
OUTPUT 'reset timer';
NEXTSTATE conversation.
                 –
                 –
                 –
                 –
```

FIGURE D-136

Example of SDL/PR

In fact, it all depends on what characterizes a text as a programming language.

If we assume a program is defined as a set of information interpretable by a machine, then not only PR but also GR are "programs". Restricting the definition to a set of information and directives interpretable and executable by a machine, we find the first difference: it is not essential that a PR representation should be executable by a machine (though it is not forbidden); what is essential is its capability of conveying precise information from one human being to another human being.

What may be considered as "wrong PR", if we look at it as a program (because of incompleteness of informal text), is perfectly valid PR if considered as a representation of the functional requirements of a system.

A further difference is in the "style" of a PR representation compared with the usual representation of a program.

Since PR is intended to aid communication between human beings, care has been taken to allow different layouts so that the PR layout can be used to guide the reader to focus on certain aspects which are considered more important than others. This is, of course, unimportant to a program which is supposed to be interpreted by a machine. The machine does not focus on any particular aspect, but has to consider equally the whole, nor is the machine trying to "understand" the program.

For its similarity to a program (which is more a psychological likeness since the GR form is "semantically" just as similar to a program as the PR is), the PR is preferred by some programmers who will probably use CHILL to implement the requirements expressed in PR. There is therefore a strong temptation to find a one-to-one mapping from PR to CHILL, so that the requirements expressed in PR can be automatically transformed into CHILL code. The reverse is also of interest because it would allow the derivation of a PR description from a CHILL program.

In § D.8, possible ways of mapping SDL to CHILL are illustrated.

D.7.2 *Metalanguage to describe the SDL/PR syntax: syntax diagrams*

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

A non-terminal symbol represents another syntax diagram with the same name. It is a shorthand symbol for more complex structures used in different places (structures that consist of terminal and non-terminal symbols again).

There is an ad hoc notation to insert comments.

Each symbol, that is, each diagram, has to have only one inlet and one outlet.

Flow lines are unidirectional and an arrow indicates the flow.

Terminal symbols are indicated by rectangles in which the vertical edges have been substituted with half circles. A non-terminal symbol is indicated by a rectangle.

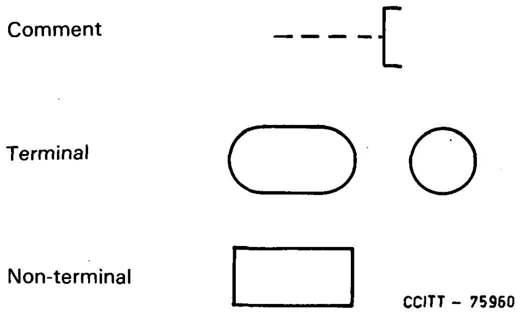


FIGURE D-137

Graphic symbols in syntax diagrams

D.7.3 *Using PR*

In SDL/GR the complete system is described in many documents, this makes it easier to handle the information. The Block Interaction Diagram represents the system structure and the communication among blocks and among processes via channels and signals, while the processes diagrams are represented in other documents.

In PR, the system structure and its communication can be represented by a splitting of the complete PR program of the system in different modules, each of them describing the structure of one or more processes. In this way, each module can be handled more easily.

Between these modules there can be a reference mechanism as explained in § D.5.

The PR version consists of a set of statements, each terminated by a ";" (semicolon). It begins with the statement

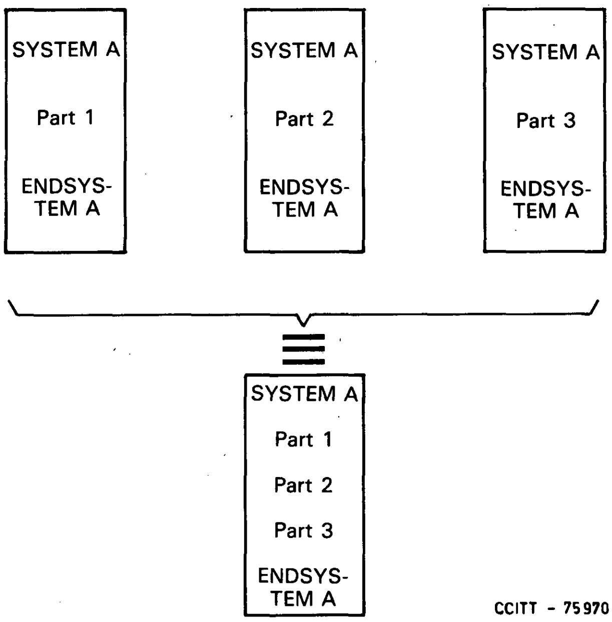
SYSTEM name;

and ends with the statement

ENDSYSTEM name;

In this closing statement the "name" is optional.

The PR representation does not consist of just a single set of statements embedded in the SYSTEM-ENDSYSTEM statements. We can have several sets each beginning with a SYSTEM statement and terminating with an ENDSYSTEM statement. They belong to the same SYSTEM representation if and only if they have the same name (Figure D-138). Each process module must be in only one set; it may not be split between two sets.



CCITT - 75970

FIGURE D-138

Separation of one system representation into three

We can group the statements contained between the SYSTEM and ENDSYSTEM in Block modules and in Signal, Channel, Data type, Macro and Procedure definitions, as can be seen in Figure D-139.

```
SYSTEM ......
    Signal definitions
    Channel definitions
    Data types definitions
    Macro definitions
    Procedure definitions

          [ BLOCK...   }
          [            } Block module
          [ ENDBLOCK   }


          [ BLOCK...   }
          [            } Block module
          [ ENDBLOCK   }


ENDSYSTEM ....
```

FIGURE D-139

Grouping of statements contained in a SYSTEM Module

Each block module is embedded between the BLOCK and ENDBLOCK statement. Like a system representation, a block representation may be split into two parts. In turn, we can consider each BLOCK module as a set of PROCESS modules plus signal, data, procedure, and macro definitions. Each process module is embedded between the PROCESS and ENDPROCESS statement. The PROCESS module cannot be further subdivided. This is illustrated in the examples of Figure D-140, where a system is divided into two parts, i.e. one part containing Block b1 with Process p2, Block b2 with Process p4 and Block b3 with Process p5 (Figure D-140 a)) and one part containing Block b1 with Process p1 and p3, and Block b3 with Process p6 (Figure D-140 b)). A global representation of these two parts is shown in Figure D-140 c).

```
SYSTEM a;
BLOCK b1;
  PROCESS p2;
  –
  –
  –
  ENDPROCESS p2;
ENDBLOCK b1;
BLOCK b2;
  PROCESS p4;
  –
  –
  –
  ENDPROCESS p4;
ENDBLOCK b2;
BLOCK b3;
  PROCESS p5;
  –
  –
  –
  ENDPROCESS p5;
ENDBLOCK b3;
ENDSYSTEM a;
```

a) Part 1

```
SYSTEM a;
  BLOCK b1; to
    PROCESS p1;
    –
    –
    –
    ENDPROCESS p1;
    PROCESS p3;
    –
    –
    –
    ENDPROCESS p3;
  ENDBLOCK b1;
  BLOCK b3;
    PROCESS p6;
    –
    –
    –
    ENDPROCESS p6;
  ENDBLOCK b3;
ENDSYSTEM a;
```

b) Part 2

```
SYSTEM a;
  BLOCK b1;
    PROCESS p1;
    –
    –
    –
    ENDPROCESS p1;
    PROCESS p2;
    –
    –
    –
    ENDPROCESS p2;
    PROCESS p3;
    –
    –
    –
    ENDPROCESS p3;
  ENDBLOCK b1;
  BLOCK b2;
    PROCESS p4;
    –
    –
    –
    ENDPROCESS p4;
  ENDBLOCK b2;
  BLOCK b3;
    PROCESS p5;
    –
    –
    ENDPROCESS p5;
    PROCESS p6;
    –
    –
    ENDPROCESS p6;
  ENDBLOCK b3;
ENDSYSTEM a;
```

c) Global

FIGURE D-140

Block representation

D.7.3.1 *Expressing structure in PR*

Useful documents in GR are the Block Tree, the Process Tree and the Channel Substructure Diagram. It is possible to obtain equivalent representations in PR, using blocks and channel substructures as explained in Recommendation Z.102.

In PR, the Block Tree, the Process Tree and the Block Interaction Diagram are mixed together. The obtained structure also contains the process statements, that is, it represents the complete system.

The direct correspondence with these documents is given by the list of subblocks, channels and signals that can be put in the block substructure part, while the structure of the system is given by the block nesting definitions.

Figure D-141 and Figure D-142 show the GR Block Tree and Process Tree which the PR examples of Figure D 143 refer to.

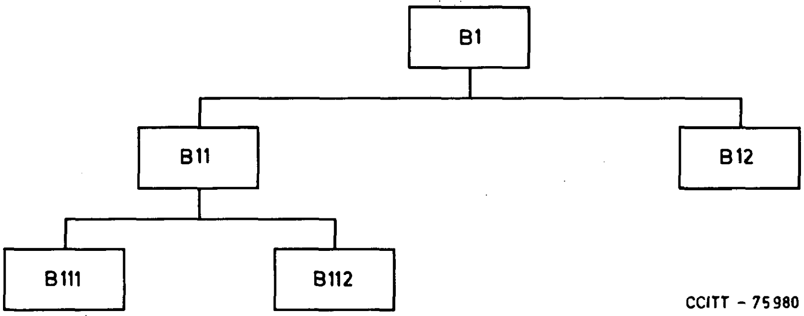


FIGURE D-141
Block tree

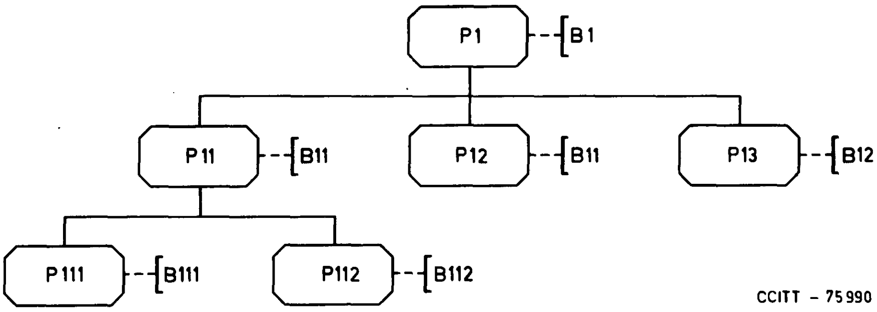


FIGURE D-142
Process Tree

```
SYSTEM J;
– – – –
BLOCK B1;
– – – –
SUBSTRUCTURE B1;
SUBBLOCKS: B11, B12;
– – – –
BLOCK B11;
– – – –
SUBSTRUCTURE B11;
SUBBLOCKS: B111, B112;
– – – –
BLOCK B111;
– – – –
ENDBLOCK B111;
BLOCK B112;
– – – –
ENDBLOCK B112;
ENDSUBSTRUCTURE B11;
ENDBLOCK B11;
BLOCK B12;
– – – –
ENDBLOCK B12;
ENDSUBSTRUCTURE B1;
ENDBLOCK B1;
ENDSYSTEM J;
```

FIGURE D-143

Example of the block nesting definitions

In PR, the processes and their structure are represented inside their blocks. There are two possible ways to show the process substructure: at the upper level block substructure or inside each upper level process. Figures D-144 and D-145 show these two possibilities.

```
SYSTEM s;
– – – –
BLOCK b1;
– – – –
PROCESS p1;
– – – –
ENDPROCESS p1;
SUBSTRUCTURE b1;
SUBBLOCKS: b11, b12;
– – – –
SUBSTRUCTURE p1;
  p11 in b11,
  p12 in b12,
ENDSUBSTRUCTURE p1;
BLOCK b11;
– – – –
PROCESS p11;
– – – –
ENDPROCESS p11;
SUBSTRUCTURE b11;
SUBBLOCKS: b111, b112;
– – – –
SUBSTRUCTURE p11;
  p111 in b111,
  p112 in b112,
ENDSUBSTRUCTURE p11;
BLOCK b111;
– – – –
PROCESS p111;
– – – –
ENDPROCESS p111;
ENDBLOCK b111;
BLOCK b112;
– – – –
PROCESS p112;
– – – –
ENDPROCESS p112;
ENDBLOCK b112;
ENDSUBSTRUCTURE b11;
ENDBLOCK b11;
BLOCK b12;
– – – –
PROCESS p12;
– – – –
ENDPROCESS p12;
ENDBLOCK b12;
ENDSUBSTRUCTURE b1;
ENDBLOCK b1;
ENDSYSTEM s;
```

FIGURE D-144

System representation with process substructure at block substructure level

```
SYSTEM s;
– – – –
BLOCK b1;
– – – –
PROCESS p1;
– – – –
SUBSTRUCTURE p1;
  p11 in b11,
  p12 in b12,
ENDSUBSTRUCTURE p1;
ENDPROCESS p1;
SUBSTRUCTURE b1;
SUBBLOCKS: b11, b12;
– – – –
BLOCK b11;
– – – –
PROCESS p11;
– – – –
SUBSTRUCTURE p11;
  p111 in b111;
  p112 in b112;
ENDSUBSTRUCTURE p11;
ENDPROCESS p11;
SUBSTRUCTURE b11;
SUBBLOCKS: b111, b112;
– – – –
BLOCK b111;
– – – –
PROCESS p111;
– – – –
ENDPROCESS p111;
ENDBLOCK b111;
BLOCK b112;
– – – –
PROCESS p112;
– – – –
ENDPROCESS p112;
ENDBLOCK b112;
ENDSUBSTRUCTURE b11;
ENDBLOCK b11;
BLOCK b12;
– – – –
PROCESS p12;
– – – –
ENDPROCESS p12;
ENDBLOCK b12;
ENDSUBSTRUCTURE b1;
ENDBLOCK b1;
ENDSYSTEM s;
```

FIGURE D-145

System representation with process substructure inside the process definition

The channels that interconnect the subblocks obtained by the partitioning are defined in the block substructure definition.

It is felt better to have the channel definition before the definition of the subblocks, because there are so many blocks contained in the system (see Figure D-146).

The signals of these channels are defined in the signal definition part.

Figure D-147 refers to the GR representation of Figure D-146.

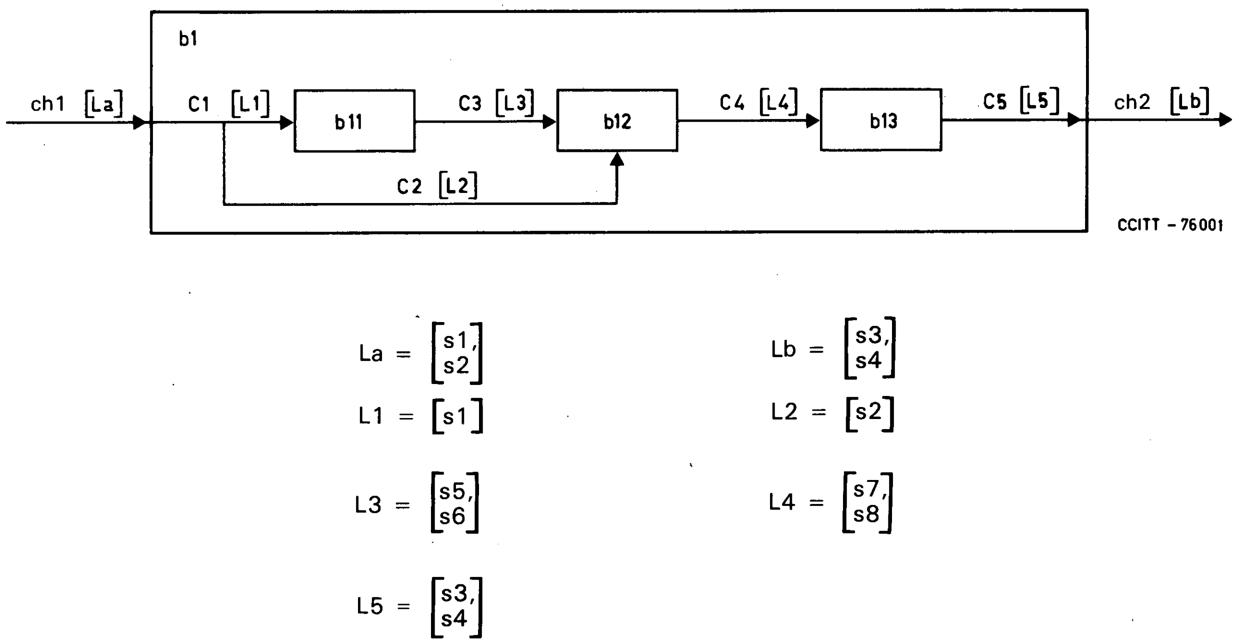


FIGURE D-146

A two-level system representation using the Block Interaction Diagram

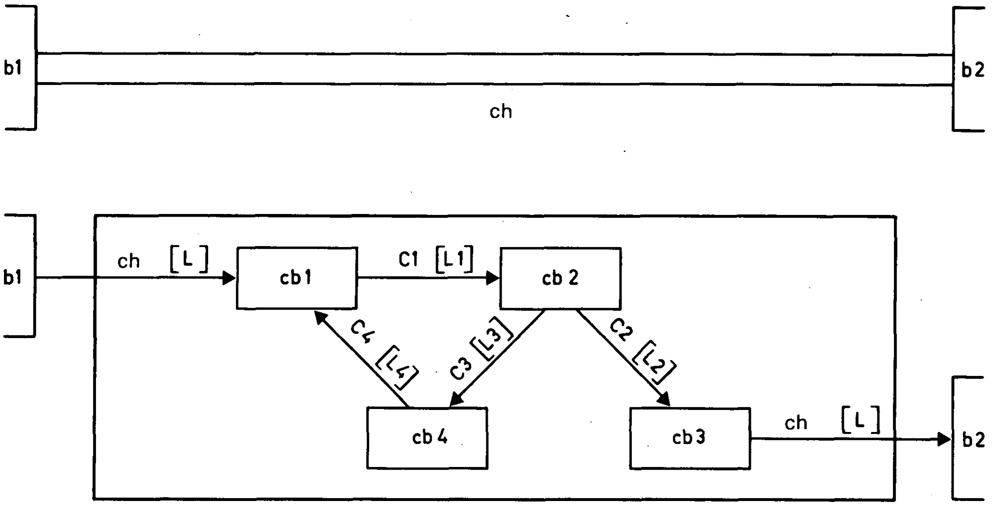
```
SYSTEM s;
CHANNEL ch1
  FROM ENV TO b1
  WITH s1, s2;
CHANNEL ch2
  FROM b1 TO ENV
  WITH s3, s4;
BLOCK b1;
– – – –
SUBSTRUCTURE b1;
SUBBLOCKS: b11, b12, b13;
CHANNELS: C1, C2, C3, C4, C5
  SPLIT ch1 INTO C1, C2;
  SPLIT ch2 INTO C5;
CHANNEL C1
  FROM ENV TO b11
  WITH s1;
CHANNEL C2
  FROM ENV TO b12
  WITH s2;
CHANNEL C3
  FROM b11 TO b12
  WITH s5, s6;
CHANNEL C4
  FROM b12 TO b13
  WITH s7; s8;
CHANNEL C5
  FROM b13 TO ENV
  WITH s3; s4;
    SIGNAL– – – –;
BLOCK b11;
– – – –
ENDBLOCK b11;
BLOCK b12;
– – – –
ENDBLOCK b12;
BLOCK b13;
– – – –
ENDBLOCK b13;
ENDSUBSTRUCTURE b1;
ENDSYSTEM s;
```

FIGURE D-147

Two-level system representations showing the use of channels

The channel definitions are either at system level (for the definition of channels between the blocks composing the system) or in the block substructure definition (for the definition of channels between the subblocks). In GR, the diagram describing the decomposition of a channel is the Channel Substructure Diagram; in PR, this corresponds to a description within the channel definition, and also the definition of the blocks and channels that define it at the lower level.

Figure D-148 shows an example of a channel between two blocks seen at two different levels (the second one lower and more detailed).



CCITT - 76011

$$L = \begin{bmatrix} s1, \\ s2, \\ s3, \\ s4 \end{bmatrix}$$

$$L1 = [s5]$$
$$L2 = [s6]$$
$$L3 = [s7]$$
$$L4 = [s8]$$

FIGURE D-148

Channel representation with blocks and channels

In the lower figure of this example, blocks b1 and b2 are not further decomposed, but the channel ch seen as a system at the higher level, shows its internal structure in the lower level representation. It is better to show the channel decomposition at the time when the system blocks cannot be further decomposed, otherwise the interaction between the blocks of the channels and respectively the subblocks of the blocks has to be shown. This increases the difficulty of representing the structure.

Figure D-149 shows the PR representation of the channel decomposition in Figure D-148.

```
SYSTEM s;
– – – –
CHANNEL ch
  FROM b1 TO b2
  WITH s1, s2, s3, s4;
SUBSTRUCTURE ch;
BLOCKS: cb1, cb2, cb3, cb4;
CHANNELS: C1, C2, C3, C4;
– – – –
INCOMING ch TO cb1;
OUTGOING ch FROM cb3;
CHANNEL C1
  FROM cb1 TO cb2
  WITH s5;
CHANNEL C2
  FROM cb2 TO cb3
  WITH s6;
CHANNEL C3
  FROM cb3 to cb4
  WITH s7;
CHANNEL C4
  FROM cb4 TO cb1
  WITH s8;
SIGNAL
– – – –;
BLOCK cb1;
– – – –
ENDBLOCK cb1;
BLOCK cb2;
– – – –
ENDBLOCK cb2;
BLOCK cb3;
– – – –
ENDBLOCK cb3;
BLOCK cb4;
– – – –
ENDBLOCK cb4;
ENDSUBSTRUCTURE ch;
BLOCK b1;
– – – –
ENDBLOCK b1;
BLOCK b2;
– – – –
ENDBLOCK b2;
ENDSYSTEM s;
```

FIGURE D-149

Example of a channel partitioning

In all of the examples presented here, it is possible to use the macro notation to describe the blocks, the processes, the channels somewhere else if they are too long or complex. In the place where the description is supposed to be, all that is needed is just a macro call. The macro definition then contains the information.

D.7.3.2 *Signal definition*

Signals can be defined at system level, block level, or in the internal part of a process definition. Signals defined at system level represent signals interchanged with the environment and between system blocks. Signals defined at block level represent signals interchanged between processes of the same block.

In Figure D-150, the signals SIG1, SIG2 are listed in a channel connecting blocks b1 and B2 or in a channel connecting one of these blocks with the environment. Signals s1, s2, s3 declared in block 1 are used by the processes belonging to this block. Signals s1, s2, s4, s5 declared in block B2 are different from those declared in b1 and they can be used by the processes of block B2.

In a structured system, there can be at block levels signals defined which represent signal interchange between blocks of the lower level (§ D.7.3.1). In the channel definition, there can be the definition of the signals of the lower level channels, which are generated by partitioning of the considered channel (§ D.7.3.1). The signal sets of the subchannels have to be disjointed.

```
SYSTEM a;
      .
  SIGNAL SIG1(ty1, ty2), SIG2(ty2, ty3);
      .
  BLOCK b1;
      .
      .
      .
    SIGNAL s1(t1,t2,t3),s2,s3(t4,t2);
      .
  ENDBLOCK b1;
  BLOCK B2;
      .
      .
      .
    SIGNAL s1(t1,t2,t3),s2,s4,s5(t4,t2);
      .
      .
  ENDBLOCK B2;
ENDSYSTEM a;
```

FIGURE D-150

Example of signal definitions at various levels

D.7.3.3 *Channel definition*

The channels can be defined at system levels, or, if the system is structured, also at block level. Finally, the channel can be defined within a channel definition. Channels defined at system level represent the channels between system blocks and between these blocks and the environment.

In Figure D-151, channels c1 and c2 are used by block b1 for its communication with the environment.

If the system is structured, each block can contain the definition of the channels that allow communication between the blocks generated by the partitioning of the considered block and between these blocks and the environment (§ D.7.3.1).

In the channel definition, there can be channel definitions of the lower level channels obtained with the decomposition of the lower level channel (§ D.7.3.1). The channel definition contains the list of the signals of that channel that have to be defined in the signal definition section.

```
SYSTEM a;
 .
 .
  CHANNEL    c1 FROM b1 TO ENV
             WITH s1,s2,s3
             REFINEMENT CHANNELS: c1.1,c1.2;
  CHANNEL    c2 FROM ENV TO b1
             WITH s4,s5
             REFINEMENT CHANNELS: c2.1,c2.2;
 .
 .
 .
ENDSYSTEM a;
```

FIGURE D-151

Example of channel definitions

D.7.3.4 *Data definition*

According to Recommendation Z.101, SDL has predefined data types that are available to every system. They are integer, real, character, string, Boolean, process instance identifier, time, duration. They do not need a declaration and can be used in a variable definition with their predefined names, i.e. INTEGER, REAL, CHARACTER, STRING, BOOLEAN, PID, TIME, DURATION.

The variables definition is in a process or in a procedure definition. Any variables that must be EXPORTED, IMPORTED, REVEALED or VIEWED must be declared with these properties.

Each data is owned by a process instance. Data are defined in the process definition (thus all the process instances of a process have copies of the data definition given at process level).

In case of revealing in the declare (DCL) section, it is only necessary to add the keyword REVEALED before the variable type name.

The VIEWED declaration is outside the DCL section and it is syntactically formed by the keyword VIEWED followed by the variable name, the type of this variable, and finally the identifier of the process that reveals it.

In Figure D-152, there is an example of a VIEWED and REVEALED variable "digit" declaration in processes p1 and p2 respectively. These belong to block b1. In this example, process p1 is the owner of the variable "digit" and process p2 can view it.

```
SYSTEM a;
      .
      .
  BLOCK b1;
      .
      .
  PROCESS p1;
      .
      .
  DCL
    REVEALED digit INT,
    counter, alarm number INT,
    a,b,c, BOOL;
      .
      .
      .
  ENDPROCESS p1;
  PROCESS p2;
  (   )
  DCL d,e,f, BOOL;
    VIEWED digit p1,
      .
      .
      .
    ENDPROCESS p2;
  ENDBLOCK b1;
ENDSYSTEM a;
```

FIGURE D-152

Example of visibility of data between processes of the same block

In Figure D-153, all the instances of the process p1 can view the variable “digit”, as it is declared as REVEALED and VIEWED.

```
SYSTEM a;
     .
     .
     .
BLOCK b1;
     .
     .
PROCESS p1;
     .
     .
     .
DCL
     REVEALED digit INT;
     .
     .
VIEWED digit INT p1;
     .
     .
     .
END PROCESS p1;
     .
     .
     .
     .
END BLOCK b1;
     .
     .
END SYSTEM a;
```

FIGURE D-153

Example of visibility between process instances of the same process definition

When it has to be indicated that the viewing operation is required, the keyword VIEW is used followed by the name of the variable to be viewed and the process instance identity owner of the variable. The variable name is separated from the process instance identity by a comma and both are enclosed by round parentheses. Omitting the view keyword causes the process instance to look for a variable having that name in its local data (Figure D-154).

```
SYSTEM s;
.......
 BLOCK b1;
  .......
  PROCESS p1;
  .......
  DCL
    REVEAL digit INT,
    .......
  ENDPROCESS p1;
  PROCESS p2;
  .......
  DCL
  t INT,
  VIEWED digit INT p1,
  .......
  TASK t:=5*VIEW(digit,p1);
    .......
  ENDPROCESS p2;
 ENDBLOCK b1;
ENDSYSTEM s;
```

FIGURE D-154

Example of use of a viewing variable

Variables which can be exported must have the attribute EXPORTED in their definitions in the exporting process.

The importing process declares the variables it intends to import in an imported section using the keyword IMPORTED followed by the names of the variables to be imported, their types and the identifier of the exporting process.

A variable can be declared as IMPORTED and EXPORTED at the same time. This allows a process instance to export a variable to the other instances of the same process and to import that variable from the other instances. In this case, care should be taken to avoid name clashes: the assignment

v:=IMPORT (v,PID)

will result in the replacement of one's v with the Pid instance "v".

As it is possible to see in the example in Figure D-155, every import operation requires the keyword IMPORT preceeding the variable name and the process instance identifier, the former separated from the latter by a comma and both bracketed by round parentheses.

If a variable is exported and revealed, the attribute REVEALED should be added to the EXPORTED attribute in the declaration (see Figure D-155). Remember that the revealing is valid only to process instances of the same block.

In the example of Figure D-155, the variable "counter" owned by the process p2 in block b2 is exported and revealed. It is exported for the process p1 and revealed for the process p3 in the same block b2.

```
SYSTEM a;
    .
    .
 BLOCK b1;
    .
    .
  PROCESS p1;
   DCL
  EXPORTED digit INT,
  IMPORTED counter INT BLOCK b2,p1;
    .
    .
  ENDPROCESS p1;
 ENDBLOCK b1;
 BLOCK b2;
    .
    .
  PROCESS p2;
   DCL
    EXPORTED,REVEALED counter INT,
    .
    .
    .
  ENDPROCESS p2;
  PROCESS p3;
    .
    .
   DCL
    VIEWED counter PROCESS p2,
    .
    .
    .
  ENDPROCESS p3;
 ENDBLOCK b2;
ENDSYSTEM a;
```

FIGURE D-155

Example of visibility of data between processes belonging to different blocks

D.7.3.5 *Macro definition*

The macro construct is a means to handle repetitiveness. In PR programs it is possible to have a macro cutting the stream of statements, replacing them by the macro call (§ D.7.3.7.8 and Figure D-156 b)).

The macro definition must be given at the beginning of the system, block, process, procedure definition (Figure D-156 a)), depending on whether it can be called from all processes, from those in a block, from only a process or from only a procedure. Note that in PR the macro always has one inlet and one outlet, so that it is necessary to use labels and joins to represent in PR a GR diagram with more than one inlet or outlet respectively.

The example in Figure D-156 represents a macro (Figure D-156 a)) with two outlets a and b. This means that in the main program (Figure D-156 b)) there are two corresponding labels a and b.

```
MACRO EXPANSION a;                   ...........
  INPUT register_alarm;              ...........
      .                              MACRO a;
  JOIN a;                            a: .........
  INPUT tone;                           .........
      .                              b: .........
      .                                 .........
  JOIN b;
ENDMACRO a;

a) Macro definition                  b) Macro call
```

FIGURE D-156

Example of use of a macro

D.7.3.6 *Procedure definition*

Procedures may be defined at various levels in the hierarchy of SDL constructs, i.e. at system, block, process or procedure level.

Depending upon where a procedure is defined, it is either visible to all processes and procedures in the system (definition at system level), to all processes and procedures of a block (definition at block level), or only to the process (procedure) into which it is defined.

The formal parameter definition starts with the keyword FPAR. The variables listed as formal parameters can have the attributes IN, IN/OUT, or SIGNAL if the corresponding actual parameter is a signal.

A parameter with the keyword IN is passed by value. With the keyword IN/OUT, a parameter is passed by reference (see also § D.7.3.7.9). A procedure has only one outlet, but there can be one or more RETURN statements. After the interpretation of the keyword RETURN, the next statement will be the one following the procedure call in the main program (Figure D-157).

The visibility schema is very close to the data type definition schema. The only exception is that there are no predefined procedures so that any procedure should be defined in such a way that its caller may see it. Figure D-157 shows an example of a procedure definition. In the first part of the example, the formal parameters are correct. In the second procedure definition statement, there is an error because the type of the formal parameters are not in the right order compared to the actual parameters in the procedure call.

```
. . . .
. . . .
SIGNAL sig1(int,bool,int),
. . . .
DCL number INTEGER, connected BOOLEAN;
. . . .
PROCEDURE proc1
FPAR
      SIGNAL sig,
      in num INTEGER,
      IN/OUT conn BOOLEAN;
. . . .
. . . .
. . . .
ENDPROCEDURE;
```

This procedure definition statement is correct according to the procedure call, but

```
PROCEDURE proc1
FPAR
    IN num INTEGER,
    SIGNAL sig,
    IN/OUT conn BOOLEAN;
```

is WRONG!!!!!!!!!!!!!!!!!!!!!!!!!!!

```
. . . .
. . . .
CALL proc1(sig1,number,conn);
. . . .
. . . .
```

FIGURE D-157

Examples of use of procedures

D.7.3.7 *Process definition*

The process definition is embedded, as seen before, between the statements PROCESS and ENDPROCESS. A name has to be associated to the PROCESS keyword. This name (having the implicit process qualification), with the name of the embedding block and the system name, forms the identifier of the process.

If there are any formal parameters, they must be declared after the process statement.. The keywords FPAR, as seen in Figure D-158, precedes the parameters.

The actual parameters are in the CREATE statement (see § D.7.3.7.1).

```
PROCESS process name:

FPAR

variable name type;
```

FIGURE D-158

Formal parameters definition

Instance numbers, which must be enclosed in brackets, are optional.

The instance number is a couple of integers separated by a comma, the first indicating how many process instances are present at system initialization, and the second indicating the maximum number of process instances that may exist during the system life cycle. If the first number is omitted, there is one instance at the system initialization. If all instance numbers are omitted, there is one process instance throughout all of the system life cycle (Figure D-159).

a) PROCESS p1(5,18);	there are five instances at the initialization and a maximum of 18 in the system life cycle
b) PROCESS p2(,5);	is equivalent to PROCESS p2(1,5);
c) PROCESS p3;	is equivalent to PROCESS p3(1,1);
d) Process mistake (5,3); WRONG!!!	

FIGURE D-159

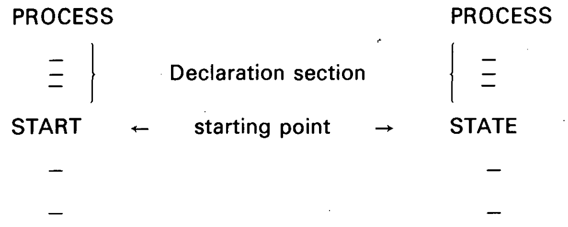
Examples of processes

Obviously the maximum number of instances (second number) should be equal to or greater than the number of instances existing at system initialization (Figure D-159 d)).

After the statement PROCESS, we must define the data owned by the process, and after these, the procedures and macros local to that process (see §§ D.7.3.5 and D.7.3.6).

The representation of the behaviour begins after these definitions. This is indicated either by a START statement followed by a transition string, or by a STATE statement.

If there is no START statement, each PROCESS instance starts its existence at the first STATE statement (Figure D-160 b)).



a)

Process *with* a START statement

b)

Process *without* a START statement

FIGURE D-160

Starting of a process

D.7.3.7.1 *Creation of processes*

A process instance can create other instances of the same process or of different processes of the same block, issuing a create action.

The CREATE statement may contain the list of the actual parameters within round brackets, as shown in Figure D-161.

```
SYSTEM s;
  BLOCK b;
    PROCESS p;
    .......
    .......
    DCL           a,c  INTEGER,
                  b    BOOLEAN,
           .......
           .......
             CREATE p1(a,b,c);
           .......
           ENDPROCESS p;
           .......
           PROCESS p1;
           FPAR
           .......
             digit INTEGER, connected BOOLEAN, number INTEGER;
           START
           .......
           ENDPROCESS p1;
         ENDBLOCK b;
       ENDSYSTEM s;
```

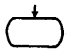
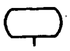
FIGURE D-161

Example of the creation of a process instance

D.7.3.7.2 *States and multiple appearances*

In PR, a state is represented by the keyword STATE followed by the state name.

A state ends at the following state statement (next STATE statement) or at the process end (ENDPROCESS or STOP statement).

Moreover in PR there are always distinct statements to indicate going into a state (NEXT-STATE ) or moving from a state (STATE ).

There is not a common statement indicating both as in GR.

Multiple appearances of states have the same explanation and relation to the semantic model as is given in § D.6 (GR).

The following are examples on the PR version of those given for GR in § D.6.3.6.5.

```
b:   NEXTSTATE state_1;
   STATE State_1;
     INPUT A;
     NEXTSTATE state_2;
     INPUT C;
     JOIN a;
   STATE State_2;
     INPUT C;
     JOIN b;
     INPUT B;
a: NEXTSTATE State_3;

   STATE State_3;
     INPUT D;
     JOIN b;
```

a) The complete diagram

```
     –
     –
   STATE State_1;
     INPUT A;
     NEXTSTATE STATE_2;
     INPUT C;
     NEXTSTATE State_3;
   STATE State_2;
     INPUT B;
     NEXTSTATE State_2;
     INPUT C;
     NEXTSTATE State_1;
   STATE State _3;
     INPUT D;
     NEXTSTATE State_1;
```

b) Diagram a) with main states and next states used as connectors to main state

FIGURE D-162

Examples of multiple appearances of a state (equivalent to Figure D-93)

D.7.3.7.3 *PR statement and use of data*

D.7.3.7.3.1 *Input statement*

The INPUT statement contains a signal list. Data items contained in signals are named using variable identifiers. The variable identifiers have to be of the type indicated in the signal definition, so their position is very important. These variable identifiers are contained within round brackets, and they are separated by commas (see Figure D-164). If one or more signal data items are discarded, the corresponding variables are missed, and this is represented by two consecutive commas (Figure D-163).

```
INPUT a(var1,var2,,var4);
```

Note – In this statement, the third data item of the signal a is discarded.

FIGURE D-163

Signal a as input with only 3 of its 4 data items defined

```
SIGNAL sig1 (INTEGER,BOOLEAN,INTEGER);
    DCL a INTEGER,b BOOLEAN,c INTEGER,
```

a) declarations

```
INPUT sig1(a,b,c);
```

b) a correct input

```
INPUT sig1(a,c,b);
```

c) an incorrect input

FIGURE D-164

INPUT statements

D.7.3.7.3.2 *Save statement*

The SAVE statement may have the list of signal names or an asterisk, if all the incoming signals not named in INPUT statements are saved. A simple example of use of SAVE is given in Figure D-165.

```
–
–
–
SAVE State _ 31;
  SAVE S;                          /* $S^1$ arrives, enters queue and remains in queue, R arrives and
                                   is immediately consumed. Transition to state _ 32 is triggered.
  INPUT R;

  NEXTSTATE State _ 32;
STATE State _ 32                   On arrival at state _ 32 '$S^1$' is immediately consumed and
                                   transition to next state is triggered. */
  INPUT S;
–
–
–
```

FIGURE D-165

Example of the use of save

D.7.3.7.3.3 *Output statement*

The OUTPUT statement contains the list of signals sent to other processes. For each data item contained in the signal, there can be a corresponding actual expression or value. If a value of one or more data items is undefined, it is omitted, and the empty position is represented by two consecutive commas (Figure D-166).

```
OUTPUT sig1(a,,c);
```

FIGURE D-166

Signal sig1 as output with only 2 of its 3 data items defined

The list of the actual values is put within round brackets. The value for each data item considered in the signal has to be of the defined type. Referring to the declarations in Figure D-164 a) for the input statement, a correct and an incorrect output is shown in Figure D-167.

```
    OUTPUT sig1(2,true,10)
a)      a correct output
    OUTPUT sig1(false,true,10)
b)      an incorrect output
```

FIGURE D-167

Output statements

D.7.3.7.3.4 *Task statement*

The task statement may contain one or more assignment statements, set of text statements and/or informal text. An informal text consists of a name and/or a phrase delimited by apostrophes. The statements or text are separated by commas (Figure D-168).

```
TASK a:=b;
TASK 'connect the subscriber';
TASK RESET(TIME);
TASK main_assignment, c:=d+e
TASK var1:=var2*var3,
     var4:=var5 MOD var6;
```

FIGURE D-168

Task statements

D.7.3.7.3.5 *DECISION statement*

The SDL PR decision is represented by the keyword DECISION followed by the decision name and/or expression or text string. The latter is a text (formal or informal) contained between " " (apostrophes).

The set of outgoing paths is delimited by the DECISION statement at the beginning and by the ENDDECISION statement at the end (see Figure D-169).

```
DECISION . . . .;
(result): . . . .;
(result): . . . .;
(result): , , , ,;
  ENDDECISION;
```

FIGURE D-169

Delimitation of a Decision

The results of a decision are indicated within round brackets, and they are represented by one or more text strings delimited by apostrophes, or by possible values obtained by the evaluation of the expression contained in the decision statement. Different results within the parentheses are separated by commas. The values are represented by constant expressions, or by ranges whose upper and lower bounds are constant expressions. The result values have to be of the type of the expression contained in the decision statement, as illustrated in the examples of Figures D-170, D-171 and D-172.

It is possible to indicate some results explicitly and to group all the other possible results using the keyword ELSE, as illustrated in, for instance, Figure D-170.

```
          DECISION 'x';

(2):                .  |  path 1
                    .  |

(6):                .  |  path 2
                    .  |

(9,10):             .  |  path 3
                    .  |

ELSE:               .  |  path 4
                    .  |

          ENDDECISION;
```

Note – x is an integer between 1 and 10.

Path 1 is selected with x=2.

Path 2 is selected with x=6.

Path 3 is selected with x=9 or x=10.

Path 4 is selected with x=1, =3, =4, =5, =7 or =8.

FIGURE D-170

Decision with ELSE

```
                                    DECISION 'Subscriber category';

('international', 'national'):        :  |
                                      :  |  path 1

('local')                 :           :  |
                                      :  |  path 2
```

FIGURE D-171

Example of DECISION

```
Example

DCL x INT,
  DECISION x;
(2+5)          : . . . . . . . . .;
(6+8,10+12)    : . . . . . . . . .;
ELSE           : . . . . . . . . .;
  ENDDECISION;
  DECISION a;
(true) : . . . . . . . .;
(false): . . . . . . . .;
  ENDDECISION;
  DECISION x;
    (10) : . . . . . . . . . .;
ELSE   : . . . . . . . . . .;
  ENDDECISION;
```

FIGURE D-172

Examples of the results of decisions

The ENDDECISION statement offers structuring capabilities as in structured programming as illustrated in Figure D-173.

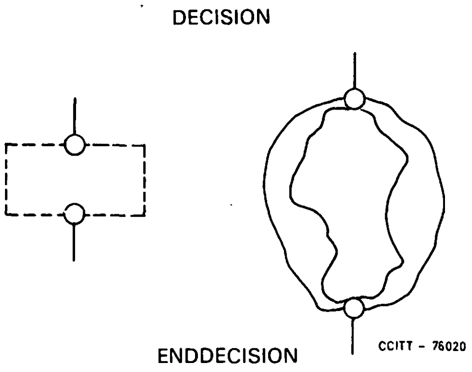
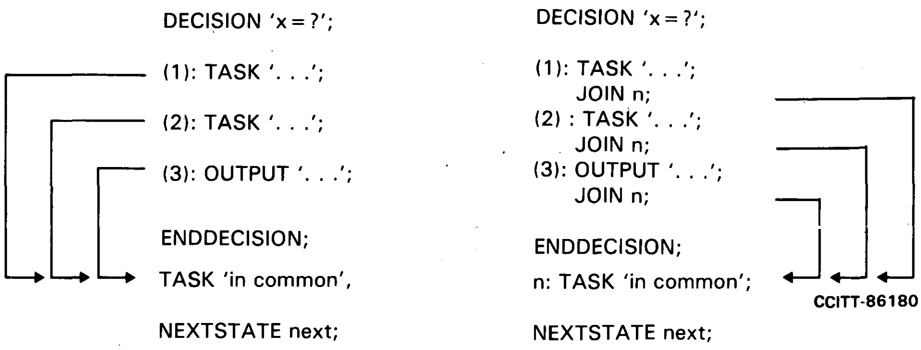


FIGURE D-173
Structuring capabilities in the DECISION

All paths terminate at the ENDDECISION statement. Those paths which have not been closed previously continue at the statement following the ENDDECISION, as shown the equivalent in Figures D-174 and D-175.

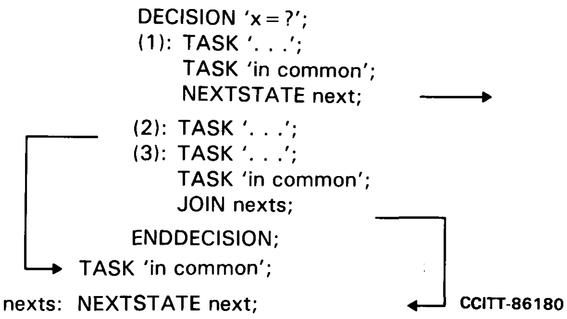


```
DECISION 'x=?';

(1): TASK '. . .';

(2): TASK '. . .';

(3): OUTPUT '. . .';

ENDDECISION;
TASK 'in common',

NEXTSTATE next;
```

```
DECISION 'x=?';

(1): TASK '. . .';
     JOIN n;
(2) : TASK '. . .';
     JOIN n;
(3): OUTPUT '. . .';
     JOIN n;

ENDDECISION;
n: TASK 'in common';

NEXTSTATE next;
```

FIGURE D-174

Branching from a DECISION



```
         DECISION 'x=?';
         (1): TASK '. . .';
              TASK 'in common';
              NEXTSTATE next;
         (2): TASK '. . .';
         (3): TASK '. . .';
              TASK 'in common';
              JOIN nexts;
         ENDDECISION;
       TASK 'in common';
nexts: NEXTSTATE next;
```

FIGURE D-175

Equivalent branching of the example shown in Figure D-174

The decision statement can be used to model the IF-THEN structure, the DO-WHILE and the LOOP-UNTIL structure.

The IF-THEN structure is shown in Figure D-176, the IF-THEN ELSE in Figure D-177, the DO-WHILE in Figure D-178, and the LOOP-UNTIL in Figure D-179.

```
BLOCK a;
  PROCESS b;
    STATE s1;
      INPUT i1;
      DECISION 'i1=?'
(0): TASK t1;
ELSE:;
      ENDDECISION;
  ENDPROCESS b;
ENDBLOCK a;
```

FIGURE D-176

IF THEN Structure

```
BLOCK a;
  PROCESS b;
    STATE s1;
      INPUT i1;
      DECISION 'i1=?'
(0): TASK t1;
ELSE: TASK t2;
      ENDDECISION;
  ENDPROCESS b;
ENDBLOCK a;
```

FIGURE D-177

IF THEN ELSE Structure

```
BLOCK b1;
  PROCESS p1;
    STATE s1;
      INPUT i1;
1:    DECISION 'b=?';
      (false): JOIN 11;
      ELSE : ;
      ENDDECISION;
      TASK t1;
      JOIN 1;
11:–
      –
      –
  ENDPROCESS p1;
ENDBLOCK b1;
```

FIGURE D-178

DO WHILE Structure

```
BLOCK b1;
  PROCESS p1;
    STATE s1;
      INPUT i1;
1   DECISION 'b=?';
      (false): JOIN 1;
      ELSE:;
      TASK t2;
      ENDDECISION;
  ENDPROCESS p1;
ENDBLOCK b1;
```

FIGURE D-179

LOOP. . .UNTIL. . .Structure

D.7.3.7.3.6 *Alternative statement*

When specifying a system, we may have situations where any of several behaviours would satisfy our requirements. We then want to indicate a number of permissible behaviours leaving the choice of one of them to the implementor.

The various alternatives are indicated by the keyword ALTERNATIVE with an associated expression or informal text.

The set of alternative behaviours is closed with the statement ENDALTERNATIVE; each alternative is identified by a name.

The global structure is the same as that of the DECISION; the difference is in the semantics. Whereas for DECISIONs one of the branches is selected according to the value of a data item changing in the process lifetime, for ALTERNATIVEs only a single branch exists in the process implementation so that it is selected before the implementation. An example is given in Figure D-180.

```
                ALTERNATIVE 'Alarm response';
                  ('first choice'): OUTPUT ring_bell;
                  ('second choice'): OUTPUT light_alarm;
                ENDALTERNATIVE;
At implementation time we could have
either
                OUTPUT ring_bell;
or
                OUTPUT light_alarm;
```

FIGURE D-180

Example of an ALTERNATIVE statement

The results of an ALTERNATIVE statement have the same syntax as the decision. Figure D-181 shows an example of this.

```
DCL a INTEGER 1:10;
ALTERNATIVE a;
(1,4) :...........;
(5,10):...........;
ENDALTERNATIVE;
```

FIGURE D-181

Example of the alternative statement

D.7.3.7.4 *State pictures*

Does not have a correspondence in PR.

D.7.3.7.5 *Time*

Time concept is used in SET and RESET statements. They can be the formal text in a PR TASK statement (see § D.7.3.7.3.4).

D.7.3.7.6 *Enabling condition*

In PR, the enabling condition is represented by the keyword PROVIDED followed by the condition to be evaluated. This keyword is attached to the INPUT statement. The condition attached to the INPUT is a Boolean expression. If the condition is true, the transition is enabled. If it is false, the signal is saved. As an input can only appear once per state, the possibility of having an input appearing twice, from the same state, each having a different enabling condition is not allowed.

The variables used in the expression attached to the enabling condition can be local or imported. They can not be viewed.

An example of the use of enabling conditions is shown in Figure D-182.

```
SYSTEM s;
  BLOCK b1;
  ........
    PROCESS p;
    ........
    DCL
      connected BOOLEAN,
    IMPORTED alarm BLOCK b2,p2;
    ........
    STATE s1;
      INPUT i1 PROVIDED connected;
      ........
      INPUT i2 PROVIDED IMPORT alarm,p2;
      ........
    ENDPROCESS p;
  ENDBLOCK b1;
  BLOCK b2;
  .......
    PROCESS p2;
    .......
    DCL
      EXPORTED alarm BOOLEAN;
      .......
    ENDPROCESS p2;
  .......
  ENDBLOCK b2;
ENDSYSTEM s;
```

FIGURE D-182

Examples of enabling conditions

D.7.3.7.7 *Continuous signals*

In PR, continuous signals are represented by the keyword : PROVIDED followed by the condition to be evaluated. Continuous signals differ syntactically from enabling conditions by not having an associated input statement. The condition is a Boolean expression on data visible to that process. If, at the evaluation time the condition is true, the transition is activated. If there are several continuous signals, the order of evaluation must be indicated. This is done by associating a priority to each condition using the keyword: PRIORITY, followed by an integer value. Continuous signals are defined as having lower priority than usual input signals. The condition to be evaluated may contain only local or imported variables. It may not contain any viewed variables (Figures D-183 and D-184).

```
SYSTEM s;
  ........
  BLOCK b1;
    ........
    PROCESS p1;
      ........
      DCL
        x INT;
      ........
      STATE s1;
      PROVIDED x=2 PRIORITY 0
      ........
      PROVIDED x=5 PRIORITY 1
      ........
      INPUT i1;
      ........
    ENDPROCESS p1;
  ENDBLOCK b1;
ENDSYSTEM s;
```

FIGURE D-183

Examples of continuous signals for local variables

```
SYSTEM s;
  ........
  BLOCK b1;
    ........
    PROCESS p1;
    ........
    DCL
      connected BOOL,
    IMPORTED alarm BLOCK b2,p2;
    ........
    STATE s1;
    PROVIDED connected PRIORITY 0;
    ........
    INPUT i1;
    ........
    PROVIDED IMPORT (alarm,p2) PRIORITY 1;
    ........
    ENDPROCESS p1;
  ENDBLOCK b1;
  BLOCK b2;
    ........
    PROCESS p2;
    ........
    DCL
      EXPORTED alarm BOOL;
    ........
    ENDPROCESS p2;
    ........
    ENDBLOCK b2;
    ........
  ENDSYSTEM s;
```

FIGURE D-184

Example of continuous signals for imported variables

D.7.3.7.8 *MACRO call*

The MACRO statement is: MACRO name;

The name associated to the MACRO keyword indicates a MACRO definition with that name (refer to § D.7.3.5). The MACRO statement can be inserted in any place of a PR representation but must come *after* the relevant MACRO definition.

To interpret the representation, the MACRO definition should replace every occurrence of the MACRO call statement, as the example of Figure D-185 indicates.

```
STATE a;                     STATE a;
                           | INPUT on_hook;
MACRO release;             |
                           | INPUT line_release;
INPUT digit;                 INPUT digit;
```

FIGURE D-185

Interpretation of a MACRO call statement

D.7.3.7.9 *PROCEDURE call*

The procedure call contains a list of actual parameters for the procedure. They consist of input and output signals visible from the procedure. *Note that all signals visible to the caller, and not declared as actual parameters in the procedure call, are automatically saved for the procedure lifetime.* In addition, the actual parameters contain the data visible to the caller that are made visible to the procedure.

Note also that the declaration of IN, or IN/OUT, is made in the procedure definition, so that it must not be repeated by the calling statement. Also, note that parameters with the attribute IN/OUT in the procedure definition correspond to variables owned directly or indirectly (when the caller is a procedure) by the caller. The data mentioned in the actual parameters are associated by order with the formal parameters. If a parameter is not given, it must be indicated by two consecutive commas. In this case the corresponding formal parameter has the value "undefined". An example is shown in Figure D-186.

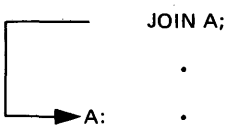
```
CALL proc1(sig1,,conn);
(see example in § D.7.3.6, i.e. Figure D-157
```

FIGURE D-186

Interpretation of a CALL to a procedure

D.7.3.7.10 *Labels (connectors)*

Labels are used as entry points associated to statements. This allows control to be transferred by means of a JOIN statement (Figure D-187).



```
JOIN A;        equivalent to a
   •           GO TO STATEMENT.
A: •
```

FIGURE D-187

Label

It is not possible to transfer control (and therefore associate labels) to the types of statement shown in Figure D-188.

SYSTEM,
BLOCK,
PROCESS,
STATE,
INPUT,
ENDMACRO,
ENDALTERNATIVE

ENDSYSTEM,
ENDBLOCK,
ENDPROCESS,
ENDDECISION,
SAVE,
ENDPROCEDURE,

FIGURE D-188

Non-allowable label points

Only one label can be associated with a statement. Labels are always *local* to a process. It is not possible to transfer control from one process to another by means of a label.

D.7.3.7.11 *Statement reachability*

Unless a transfer of control is indicated, statements are interpreted one after the other.

Statements transferring control are shown in Figures D-189 to D-193.

STATE:The control is transferred upon arrival of a signal to the INPUT or SAVE statement containing that signal name.

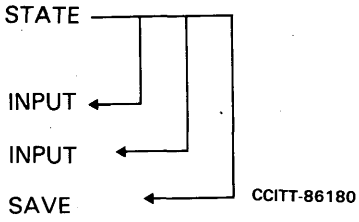


FIGURE D-189
Transfer of control in STATE

JOIN: The control is transferred to the statement having the label named in the JOIN; there should be one and only one statement having such a label.

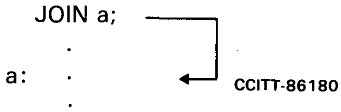
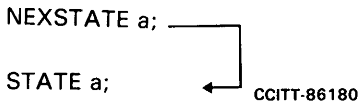


FIGURE D-190
Transfer of control in JOIN

NEXTSTATE: The control is transferred to the state having the name indicated in the NEXTSTATE statement.



```
NEXSTATE a;

STATE a;
```

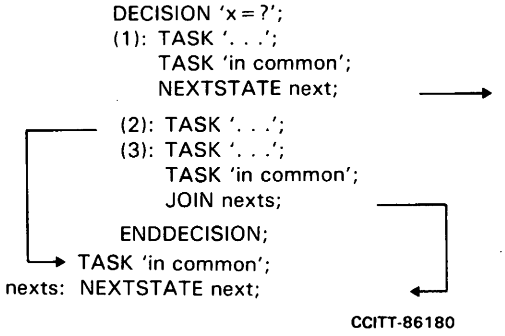
FIGURE D-191
Transfer of control in NEXTSTATE

STOP: The interpretation terminates; there is no transfer of control.

```
STOP;
```

FIGURE D-192
No transfer of control

Implicit transfer: is the same as previously explained for the decision branches.



```
          DECISION 'x=?';
          (1): TASK '. . .';
               TASK 'in common';
               NEXTSTATE next;
           (2): TASK '. . .';
           (3): TASK '. . .';
                TASK 'in common';
                JOIN nexts;
           ENDDECISION;
        TASK 'in common';
nexts:  NEXTSTATE next;
```

FIGURE D-193
Example of implicit transfer of control

D.7.3.7.12 *Use of divergence and convergence*

In PR, this is covered by labels also, as defined in § D.7.3.7.10.

D.7.3.7.13 *Comments*

Comments are inserted by means of the keyword: COMMENT.

The keyword can be inserted, as a statement, wherever a TASK statement can be inserted. In addition, the keyword can be associated with any other statement at the end of the statement:

```
STATE a COMMENT '. . . .';
```

With the exception of ENDstatements where the comment cannot be associated.

It is possible to add a CHILL comment (/* . . . . */) wherever a space (blank) may occur. Some examples of the use of this kind of comment are given in Figures D-194 and D-195.

```
DCL
a INTEGER, /* explanation of the use of variable a */
b BOOLEAN; /* explanation of the use of variable b */
```

FIGURE D-194

Use of CHILL comments in variable declaration

```
PROCEDURE ABC (IN a INTEGER, /* meaning of this formal parameter */,
IN/OUT b BOOLEAN; /* meaning of this formal parameter */);
```

FIGURE D-195

Use of CHILL comments in procedure definition

D.7.3.7.14 *Shorthand notations*

An * (asterisk) can be inserted in a STATE/INPUT/SAVE statement to indicate that all names applicable to that statement apply.

Example:

```
a) STATE *;                    b) STATE A,B,C,D,G;
```

Note – a) is equivalent to b) if A,B,C,D,G is the set of states defined in that process.

FIGURE D-196

ALL-states notation

```
a) INPUT*;                     b) INPUT I1, I3, I6;
```

Note – a) is equivalent to b) where I1, I3 and I6 are incoming signals to that process and *not declared as INPUT or SAVE in that state.* Only a single INPUT or SAVE with an * can be attached to a particular state.

FIGURE D-197

ALL-incoming-signals notation

```
a) SAVE *;                         b) SAVE I1, I3, I6;
```

Note – a) is equivalent to b) where I1, I3 and I6 are incoming signals to that process *not declared as INPUT in that state*. No other SAVE can be attached to the same state.

FIGURE D-198

ALL-saves notation

To a STATE *; we can associate a list of exceptions to exclude some states from the list. The names of states to be excluded are bound by [ ] (square brackets).

```
a) STATE * [A,C,D];                b) STATE B, E, F, G;
```

Note – a) and b) are equivalent if A,B,C,D,E,F,G are the states of the process.

FIGURE D-199

ALL-but-some state notation

A dash (–) can be inserted in a NEXTSTATE statement to indicate that the state from which the transition originates is the same as that where the transition terminates (Figure D-200).

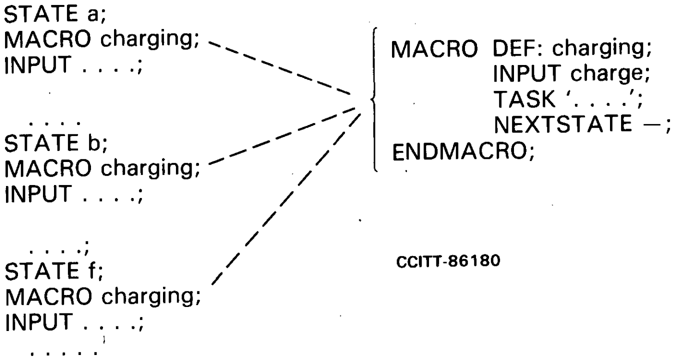
```
a)   STATE a;                b)   STATE a;
     INPUT i;                     INPUT i;

     NEXTSTATE –;                 NEXTSTATE a;
```

Note – a) and b) are equivalent.

FIGURE D-200

Identical originating and terminating state

This notation is particularly useful in cases where it is desired to perform a common set of actions in several states remaining after that in the same state, as shown in Figure D-201.



```
STATE a;
MACRO charging;              MACRO  DEF: charging;
INPUT . . . .;                      INPUT charge;
                                    TASK '. . . .';
  . . . .                           NEXTSTATE –;
STATE b;                     ENDMACRO;
MACRO charging;
INPUT . . . .;

  . . . .;
STATE f;
MACRO charging;
INPUT . . . .;
  . . . . .
```

FIGURE D-201
Macro definition

D.8 *Mappings*

This § describes some aspects of the mapping between SDL and CHILL (§ D.8.1), and the mapping between SDL/GR and SDL/PR (§ D.8.2).

D.8.1 *Mapping between SDL and CHILL*

In this § some possible ways of mapping SDL to CHILL are illustrated. This is done as an example and does not either pretend to be exhaustive or to suggest that any of these ways should be used in practice.

In fact the mapping should not only consider the available CHILL compiler and the target machine; because in general the mapping is a very complex intellectual activity and it is only through experience that designers/programmers can decide on a particular CHILL program structure to be used to implement a particular SDL representation. This also applies to the representation in SDL of the functions implemented by a CHILL program. A one-to-one mapping (if achievable) is not necessarily the best way of using SDL to represent the functions implemented in CHILL.

In this approach the overall structure of a mapping between a complete SDL diagram and a (incomplete) CHILL program is shown in Figure D-202.

Some examples of a mapping between constructs of the two languages are given in Figures D-203 to D-206. They cover the following SDL constructs:

- state and reception/save of signals; selection of a nextstate;
- output;
- join;
- decision.

The declaring module contains both the definition and the declaration of all the signals used in the transformed SDL diagram and all of the variables associated with these signals. All these variables are granted to the module representing the functional block of the SDL diagram.

```
Declaring: MODULE

          /*    CHILL module containing the signals and associated variables, used in the SDL diagram   */

                GRANT

          /*    granting of signals and variables                                                          */

                SIGNALS

          /*    signals definition                                                                         */

                SYNMODE (OR NEWMODE)

          /*    type definition                                                                            */

END Declaring;

Functional_Block: MODULE

          /*    module containing the procedural part of the SDL diagram                                   */

                SEIZE

          /*    seizing of all the signals and variables that can be received and sent from (to) this functional
                block                                                                                      */

          /*    data definition and declaration; such data is global to all the processes belonging to this
                module                                                                                     */

Process name:PROCESS ( );

          /*    local data definition and declaration                                                      */

                nextstate:=....;
                join:=none;

                DO FOR EVER;
                   state_loop:CASE nextstate OF

          /*    loop on the variable nextstate indicating the SDL state                                    */

                               (state_label1):RECEIVE CASE
                               (signal name1):
                                    .
                                    .
                                    .
                               (signal namen):

                ESAC state_loop;
                DO WHILE join/=none;
                     CASE join OF
                          (join_lab1): join: = none;
                                    .
                               .    .
                               .
                          (join_labm): join: = none;
                                    .
                                    .

                     ESAC;
                  OD;
               OD;
             END process_name;
END Functional_Block;
```

FIGURE D-202

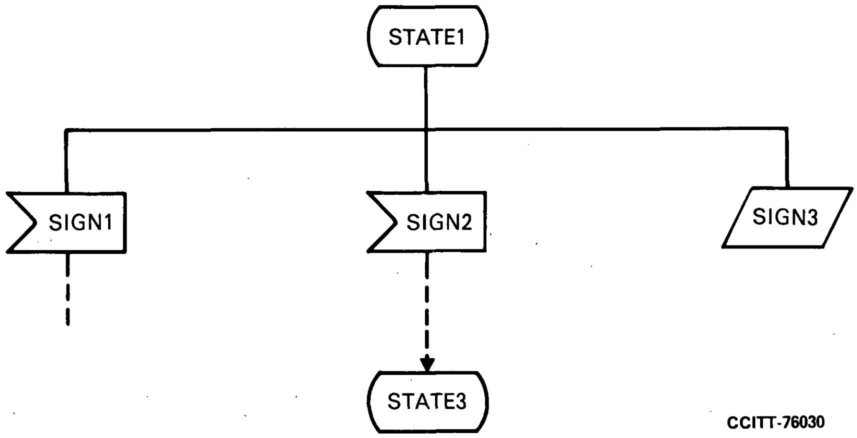
Example of a one-to-one mapping from SDL to CHILL

The functional block module represents the behaviour (procedural part) of the SDL processes.

In this translation schema, each SDL process is represented as an infinite loop; a variable named "nextstate" indicates the state to be examined, and a variable named "join" indicates possible join points determining common sets of statements.

A selection, by means of the case construct of CHILL, is made upon the value of nextstate; each entry of the case identifies an SDL state. In each entry a selection among the possible input signals is made. Each input signal determines the set of actions to be performed (the "transition path").

Each transition path ends with an assignment either to the variable "nextstate", directly determining the next state to be examined, or to the variable "join". A subsequent selection loop on the current value of the variable "join" allows every transition to terminate, in an SDL sense, and at the end assigns a value to the variable "nextstate".



a) SDL

```
STATE1 :
   RECEIVE CASE
      (SIGNAL1) : .............
                  .............
      (SIGNAL2) : .............
                  NEXTSTATE : = STATE3 ;
      ELSE GETOUT (LIST1)
ESAC STATE1 ;
```

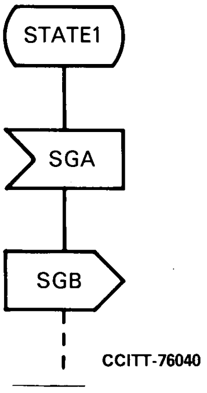
b) CHILL

FIGURE D-203

Examples of mapping STATE/INPUT/SAVE/NEXTSTATE

One of the major problems in relating SDL and CHILL is the different semantics of signal reception; in fact while CHILL doesn't consume (and therefore doesn't destroy) any signals unless they are received (persistent signals), SDL process consumes (and therefore destroys), all the signals received until there is a match with one of the inputs listed for that state. The semantics mismatch has been resolved by introducing the built-in-routine GETOUT, as an alternative (ELSE path) in the CHILL RECEIVE CASE construct, as shown in Figure D-203. The CHILL built-in-routine GETOUT, which knows (by parameters) the list of input and save signals, destroys the other signals available to the process when it is called.

After executing the GETOUT routine the state selector is set to repeat the loop for that state until a valid input signal is selected (or arrives if not already present).



a) SDL

```
STATE1 :
   RECEIVE CASE
      (SGA) : pi : = get_ instance_ value () ;
              send SGB to pi ;
              nextstate : = . . . . . . . . . ;
      ELSE    nextstate : = state1 ;
ESAC STATE1 ;
```

b) CHILL

FIGURE D-204

Example of mapping OUTPUT

For example, once the input signal SGA, in Figure D-204 has been recognized, the appropriate destination process instance for the signal SGB is selected and the signal SGB is sent.

Before sending the signal SGB it may be necessary to fill some information fields which are to be carried by the signal. This can either be done immediately before, or well in advance of sending the signal.

When a join point is met in the diagram (see Figure D-205), the appropriate value is assigned to the variable "JOIN". As explained in Figure D-202, a loop on the value of the variable "join" is performed to determine the next state to be examined. A join point can be seen, from the programming language point of view, as a "goto" construct; collecting all the join points together so that they can be examined which allows the entire skeleton program to be written without using gotos, thus making it easier to read.

An SDL decision has a direct translation into the case construct of CHILL, as shown in Figure D-206.

D.8.2 *Mapping between GR and PR*

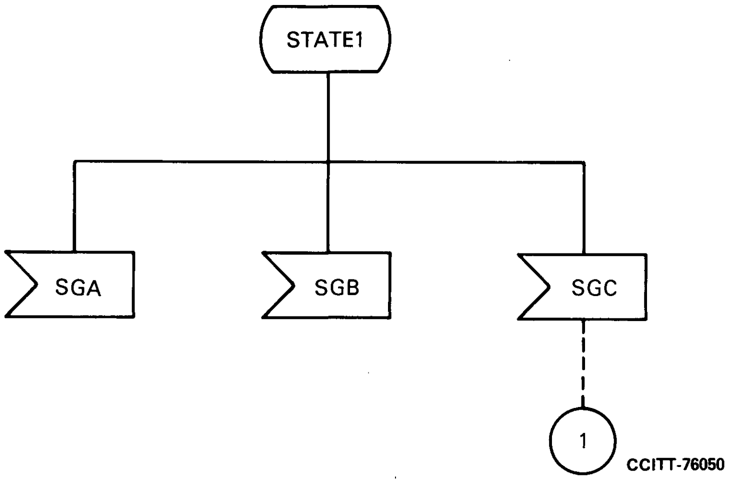
Some parts of a system can only be described using PR, e.g. data definition and signal definition. Therefore, systems represented using GR may need to be supplemented by some PR syntax. This means that systems represented using GR can always be mapped to PR, but a PR representation may not be completely representable in GR.

Figure D-207 shows the internal part of a block B given by a block interaction diagram and its correspondent parts in PR.

Figure D-208 shows an example of a simple process definition for process PR1. It is given in both GR and PR form. Only the parts having a graphical representation are shown.

For the block interaction diagram the GR constructs have a correlation with the PR constructs, as shown in Figure D-209.

Figure D-210 shows the corresponding correlation for the process diagram.



a) SDL

```
STATE1 :
   RECEIVE CASE
      (S1 in m) : case m.id of
                       (SGA) : ............ ;
                       (SGB) : ............ ;
                       (SGC) : ............ ; JOIN := 1 ;
                       ELSE nextstate := state 1 ;
                  esac ;
ESAC STATE1 ;
```

b) CHILL

FIGURE D-205

Example of mapping JOIN

D.9 *Application examples of SDL*

D.9.1 *Introduction*

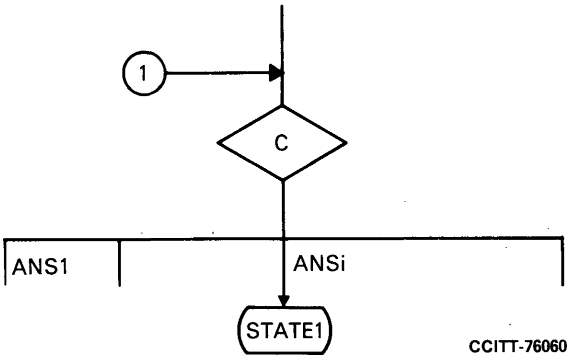
Paragraph D.9 contains three examples of using SDL. The examples are taken from the telecommunications application area, and make use of different subsets of SDL. An attempt was made to take realistic examples and cover as many SDL concepts as possible.

In each example a system is presented on a high level of abstraction, and only part of the system is elaborated in detail, the other parts are declared "undefined".

One essential requirement for a system definition is that all its different parts are tied together unambiguously into a complete whole (see § D.5). Since the recommendations do not provide language constructs to meet this requirement, such constructs has to be prepared within the scope of the examples. These referencing constructs are based on the syntax rules of SDL.

Since some of the SDL users probably prefer different kinds of referencing constructs, two main approaches are used in the examples. One approach is based on SDL/PR syntax, the other one uses comments in the graphical diagrams.

The examples are intended to show the use of SDL, and are not international specifications.



a) SDL

```
1 : CASE C OF
        (ANS1) : . . . . . . . . . . . . ;
        (ANS2) : . . . . . . . . . . . . ;
        .
        .
        .
        (ANSi) : . . . . . . . . . . . . ; nextstate : = state 1 ;
    ESAC 1 ;
```

b) CHILL

FIGURE D-206

Example of mapping DECISION

D.9.2 *Telephone switching system*

In this example only the basic SDL (Recommendation Z.101) is used.

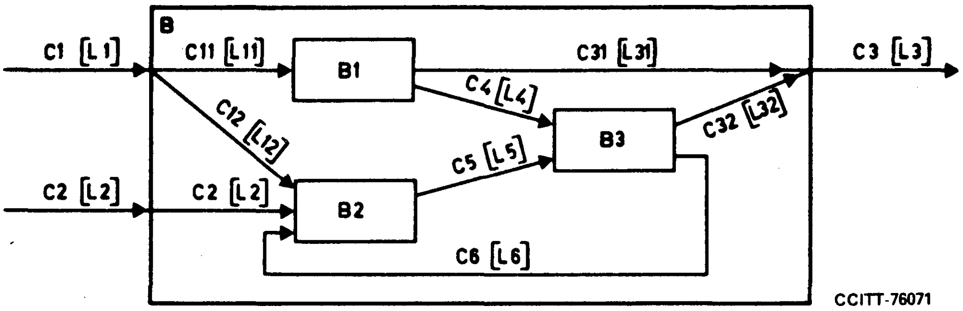
The system definition is presented in all the concrete syntactical forms of SDL in order to facilitate their comparison.

This example deals with a call handling process in a telephone switching system, and assumes a basic knowledge of telephony. In the first phase (call set-up) a connection is established between the caller subscriber (A) and the called subscriber (B), both belonging to the same exchange. In the second phase (conversation) the process is waiting for call terminating signals. In the last phase (call termination) the subscribers are disconnected and the process resumes its idle state.

D.9.2.1 *SDL/GR syntactical form in the two versions*

The normally used graphical syntactical form (transition-oriented version) does not make use of state pictures, while the other (state-oriented version) does. The block interaction diagram is common for both forms.

In the state-oriented version of the example no variable definitions are included and setting and resetting of timers is shown implicitly. Note further that nextstates are shown by a diminished size state symbol containing the state number alone as sufficient identification.



a) GR-syntax

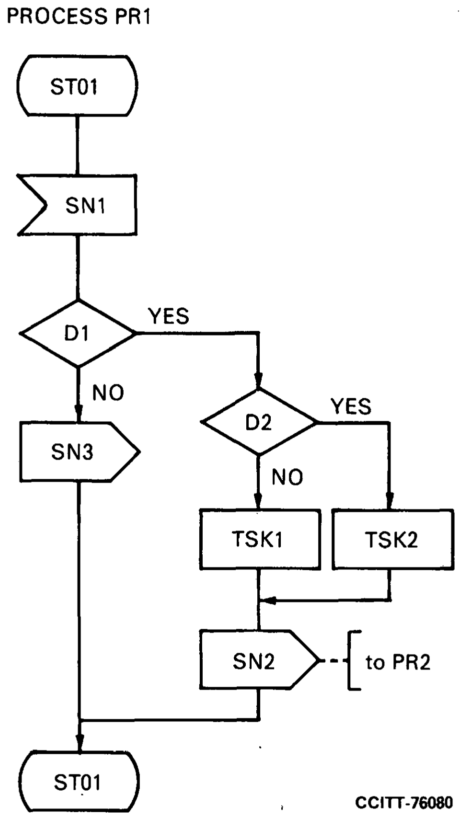
```
BLOCK B;

  SUBSTRUCTURE;
  SUBBLOCKS: B1, B2, B3;
  SPLIT C1 INTO C11, C12;
  SPLIT C2 INTO C2;
  SPLIT C3 INTO C31, C32;
  CHANNEL C11 FROM ENV TO B1 WITH /* L11 */;
  CHANNEL C12 FROM ENV TO B2 WITH /* L12 */;
  CHANNEL C2  FROM ENV TO B2 WITH /* L2 */;
  CHANNEL C31 FROM B1 TO ENV WITH /* L31 */;
  CHANNEL C32 FROM B3 TO ENV WITH /* L32 */;
  /* The new channels */
  CHANNEL C4 FROM B1 TO B3 WITH /* L4 */;
  CHANNEL C5 FROM B2 TO B3 WITH /* L5 */;
  CHANNEL C6 FROM B3 TO B2 WITH /* L6 */;
  ENDSUBSTRUCTURE;

ENDBLOCK B;
```

b) PR-syntax

Note – The PR program is not complete according to the syntax rules.

FIGURE D-207

Correlation between GR and PR for a block interaction diagram

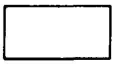
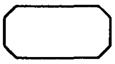
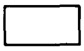


```
        PROCESS PR1 ;
          STATE ST01 ;
            INPUT SN1 ;
             DECISION D1 ;
             (NO):  OUTPUT SN3 ;
             (YES): DECISION D2 ;
                         (NO):  TASK TSK1 ;
                         (YES): TASK TSK2 ;
                         ENDDECISION ;
        OUTPUT SN2 TO PR2 ;
       ENDDECISION ;
     NEXTSTATE ST01 ;
    ENDPROCESS PR1 ;
```

Note – D1 and D2 are formally defined data. PR2 is data of type PID in the PR example, but can be the process name in the GR example.

FIGURE D-208

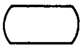
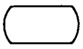
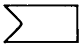
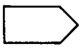
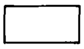
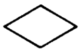
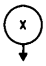
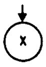
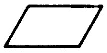
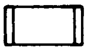
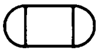
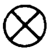
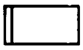
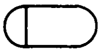
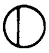
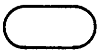
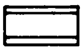
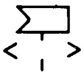
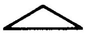
Correlation between GR and PR for a process diagram

CONCEPT	GR	PR
channel	—►—	CHANNEL
block		BLOCK ENDBLOCK
process		PROCESS ENDPROCESS
system environment	ENVIRONMENT	ENV
system		SYSTEM ENDSYSTEM
signal list	[ ]	
create	----►	CREATE
signal route	——►	
text extension	—[	
comment	/* */	/* */

CCITT-76090

FIGURE D-209

Correlation of constructs in GR and PR for symbols used in block interaction diagram

CONCEPT	GR	PR
state		STATE
nextstate		NEXTSTATE
input		INPUT
output		OUTPUT
task		TASK
decision		DECISION ENDDECISION
in-connector		x:
out-connector		JOIN x
text extension		
comment		COMMENT or /*......*/
merging transitions		JOIN x x:
all	*	*
all (except)	* [.......]	* [.......]
save		SAVE
procedure call		CALL
procedure start		PROCEDURE
procedure stop		RETURN
macro call		MACRO
macro inlet		MACRO EXPANSION
macro outlet		ENDMACRO
start		START
stop		STOP
create request		CREATE
continuous signal	< >	PROVIDED
enabling condition		INPUT... PROVIDED
option		ALTERNATIVE ENDALTERNATIVE

CCITT-76101

FIGURE D-210

Correlation of constructs in GR and PR within a process diagram

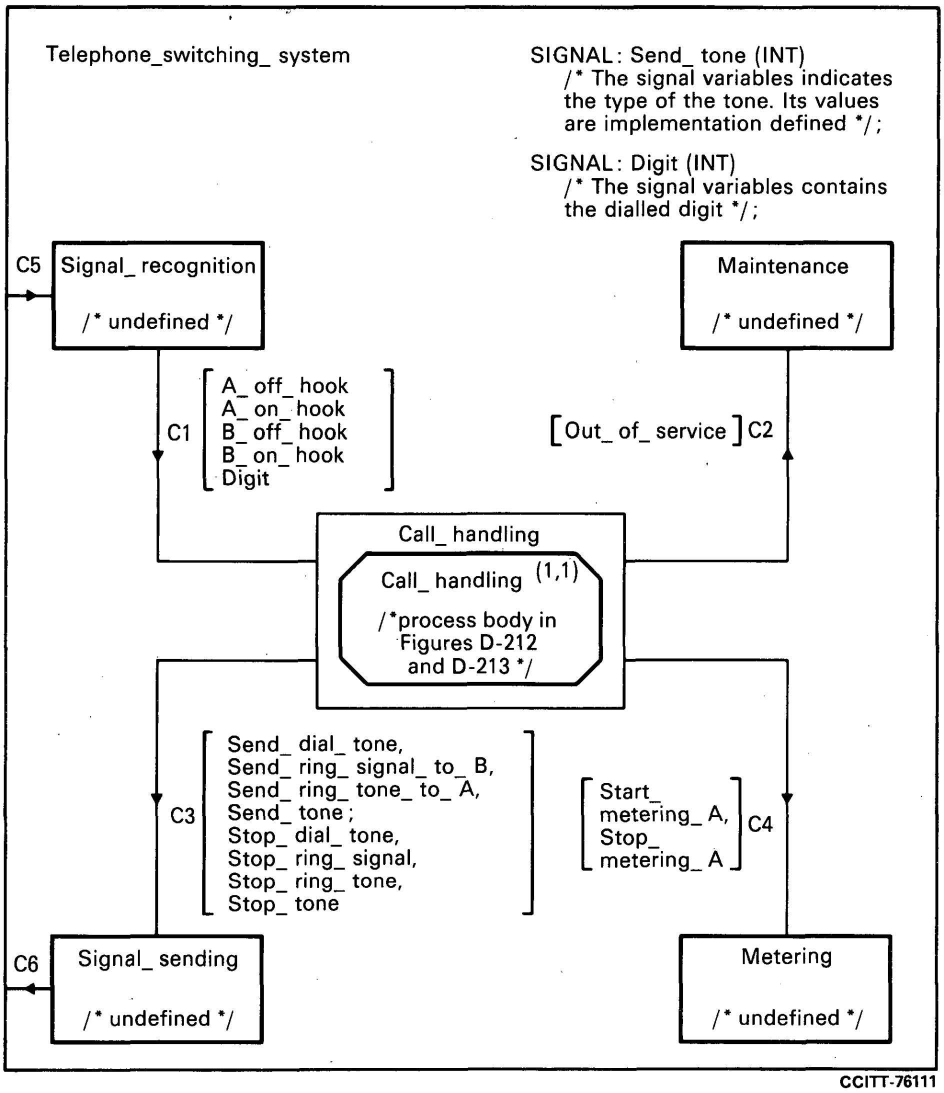


FIGURE D-211

Block interaction diagram for the system Telephone_ switching_ system

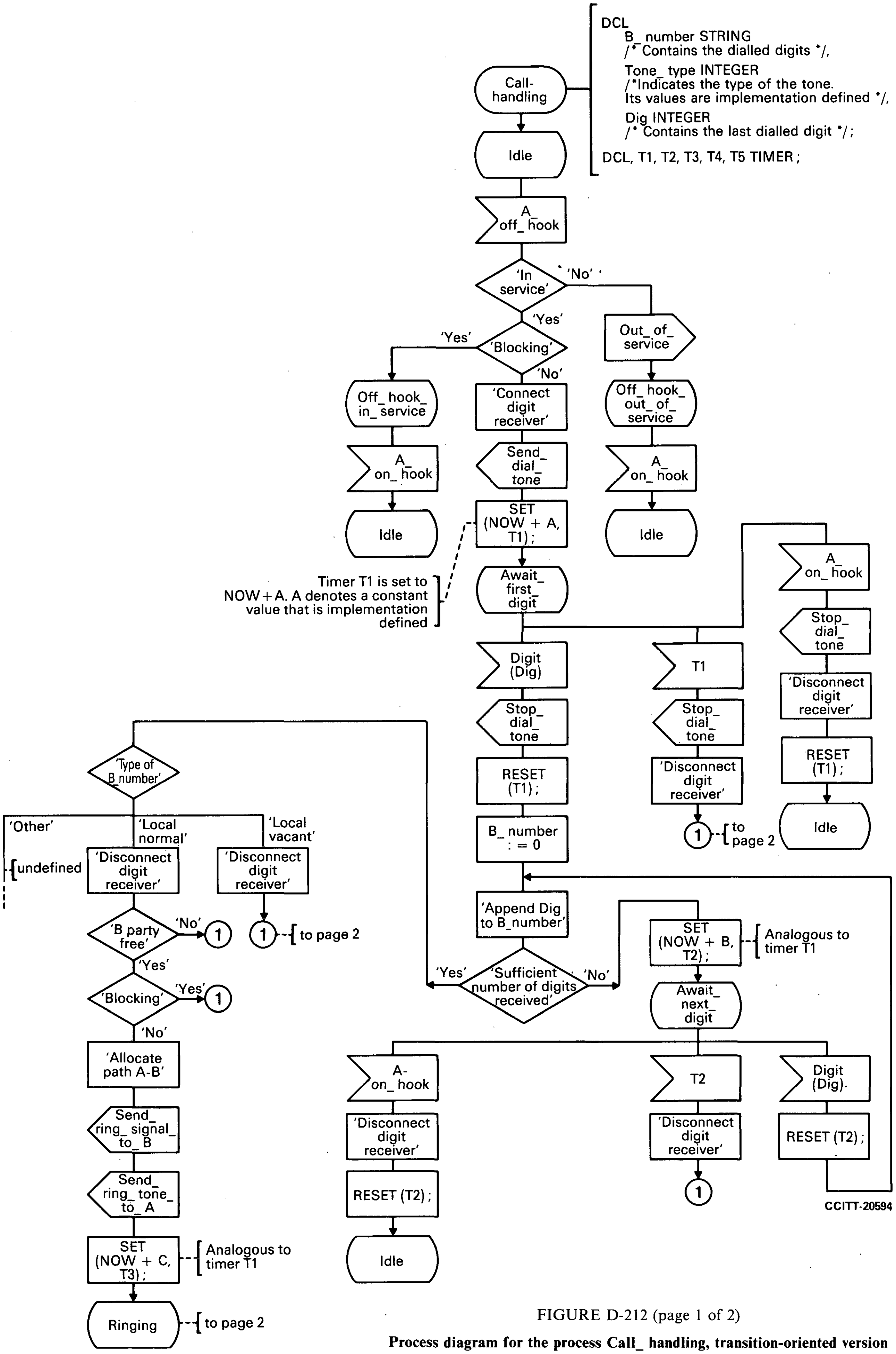


FIGURE D-212 (page 1 of 2)

Process diagram for the process Call_ handling, transition-oriented version

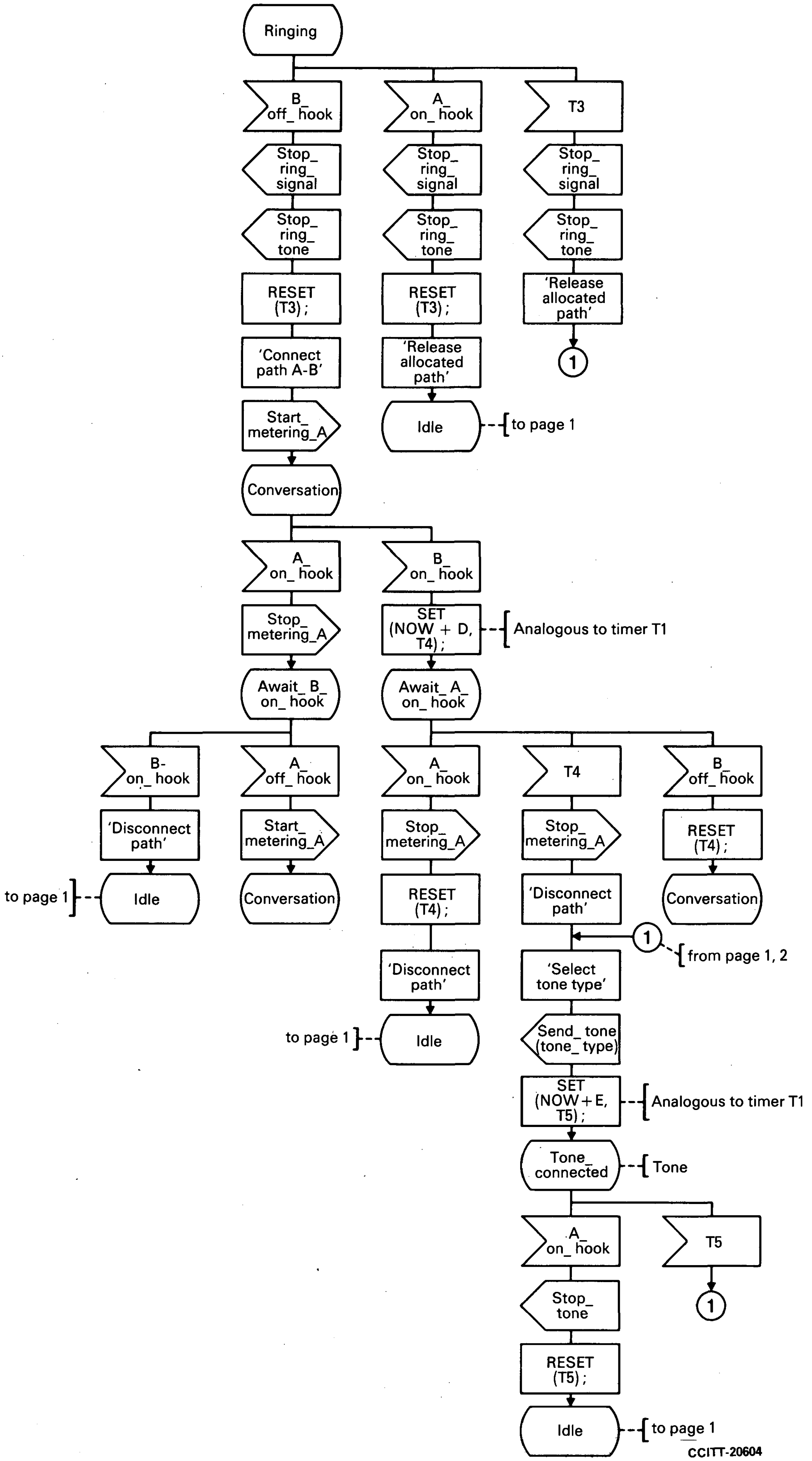


FIGURE D-212 (page 2 of 2)

Process diagram for the process Call_ handling, transition-oriented version

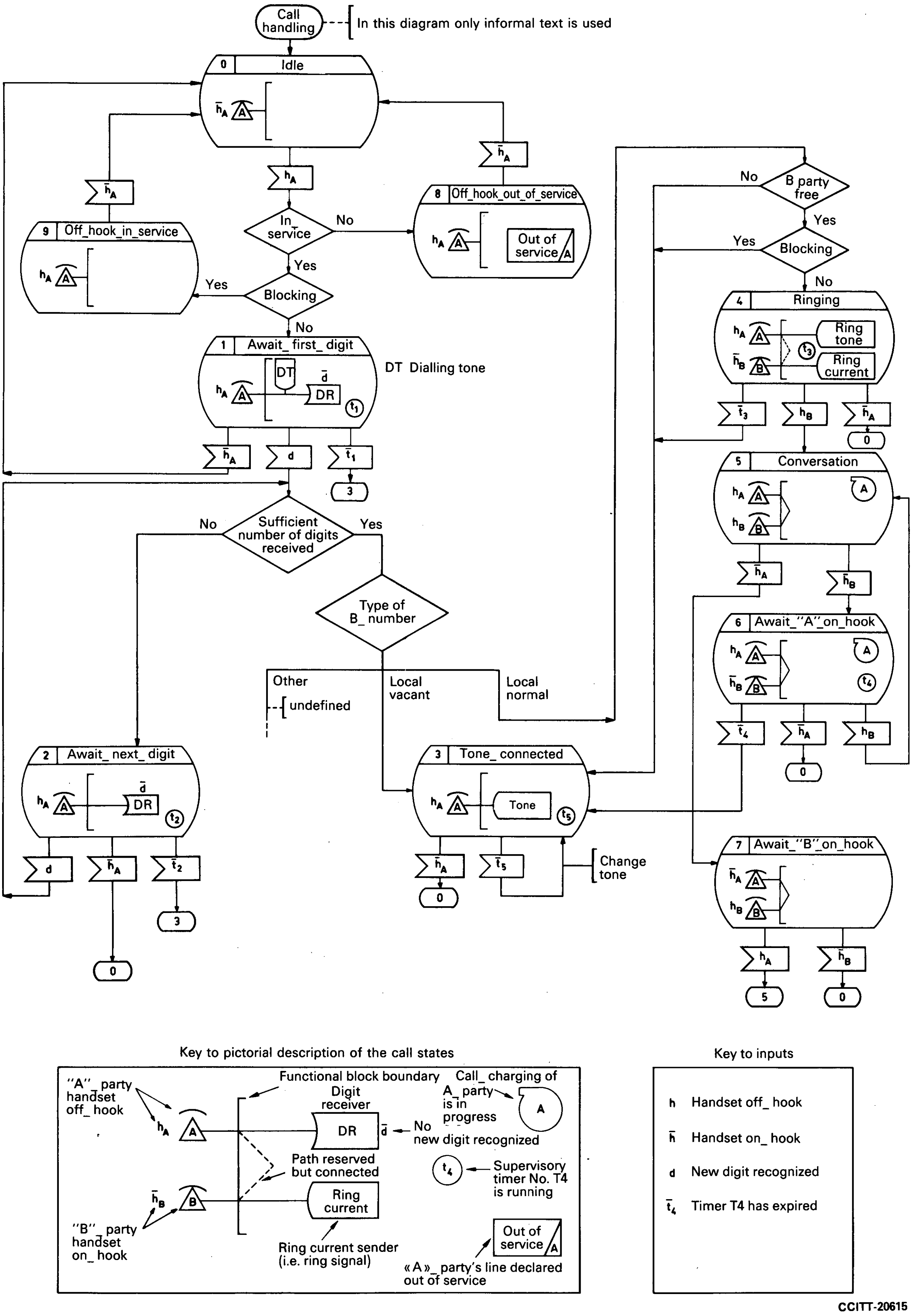


FIGURE D-213

Process diagram for the process Call_ handling, state-oriented version

D.9.2.2 *SDL/PR syntactical form*

```
SYSTEM Telephone_switching_system;
      CHANNEL C1 FROM Signal_recognition TO Call_handling
              WITH    A_off_hook,
                      A_on_hook,
                      B_off_hook,
                      B_on_hook,
                      Digit;
      CHANNEL C2 FROM Call_handling TO Maintenance
              WITH    Out_of_service;
      CHANNEL C3 FROM Call_handling TO Signal_ sending
              WITH    Send_dial_tone,
                      Send_ring_signal_to_B,
                      Send_ring_tone_to_A,
                      Send_tone,
                      Stop_dial_tone,
                      Stop_ring_signal,
                      Stop_ring_tone,
                      Stop_tone;
      CHANNEL C4 FROM Call_handling TO Metering
              WITH    Start_metering_A,
                      Stop_metering_A;
      CHANNEL C5 FROM ENV TO Signal_recognition
              WITH    /* undefined */;
      CHANNEL C6 FROM Signal_sending to ENV
              WITH    /* undefined */;
      SIGNAL Send_tone (INT)
              /* The signal variable indicates the type of the tone.
              Its values are implementation defined */;
      SIGNAL Digit (INT)
              /* The signal variable contains the dialled digit */;


BLOCK Signal_recognition /* undefined */;
ENDBLOCK;
BLOCK Signal_sending /* undefined */;
ENDBLOCK;
BLOCK Maintenance /* undefined */;
ENDBLOCK;
BLOCK Metering /* undefined */;
ENDBLOCK;
```

FIGURE D-214 (page 1 of 4)

System definition using SDL/PR-syntax

```
BLOCK Call_handling;
      PROCESS Call_handling (1,1);
          DCL B_number STRING
              /* contains the dialled digit */;
          DCL Tone_type INTEGER
              /* Indicates the type of the tone. Its values are implementation defined */;
          DCL Dig INTEGER
              /* contains the last dialled digit */;
          START; NEXTSTATE Idle;
          STATE Idle;
              INPUT A_off_hook;
                  DECISION 'In service';
                      ('No'):     OUTPUT Out_of_service;
                                  NEXTSTATE Off_hook_out _of_service;
                      ('Yes'):    DECISION 'Blocking';
                                      ('Yes'):    NEXTSTATE Off_hook _in_service;
                                      ('No'):     TASK 'Connect digit receiver';
                                                  OUTPUT Send_dial_tone;
                                                  TASK SET (NOW +A, T1);
                                                  /* Timer T1 is set to NOW +A. A denotes a constant value that is
                                                  implementation defined */
                                                  NEXTSTATE Await_first_digit;
                                  ENDDECISION;
                  ENDDECISION;
          STATE Off_hook_in_service;
              INPUT A_on_hook;
                  NEXTSTATE Idle;
          STATE Off_hook_out_of_service;
              INPUT A_on_hook;
                  NEXTSTATE Idle;
          STATE Await_first_digit;
              INPUT Digit (Dig);
                  OUTPUT Stop_dial_tone;
                  TASK RESET (T1),
                      B number:=0;
              10: TASK 'Append Dig to B_number';
                  DECISION 'Sufficient number of digits received';
                      ('No'):     TASK SET (NOW +B,T2);
                                  NEXTSTATE Await_next_digit;
                      ('Yes'):    DECISION 'Type of B_number';
                                      ('Local normal'): TASK 'Disconnect digit receiver';
                                          DECISION 'B party free';
                                              ('No'):     JOIN 1;
                                              ('Yes'):    DECISION 'Blocking';
                                                              ('Yes'):    JOIN 1;
                                                              ('No'):     TASK 'Allocate path A-B';
                                                                          OUTPUT Send_ring_signal_to_B;
                                                                          OUTPUT Send_ring_tone_to_A;
                                                                          TASK SET (NOW +C, T3);
                                                                          NEXTSTATE Ringing;
                                                          ENDDECISION;
                                          ENDDECISION;
                                      ('Local vacant'):           TASK 'Disconnect digit receiver';
                                                                  JOIN 1;
                                      ('Other'):                  /* undefined */;
                                  ENDDECISION;
                  ENDDECISION;
```

FIGURE D-214 (page 2 of 4)

System definition using SDL/PR-syntax

```
    INPUT T1;
        OUTPUT Stop_dial_tone;
        TASK 'Disconnect digit receiver';
        JOIN 1;
    INPUT A_on_hook;
        OUTPUT Stop_dial_tone;
        TASK 'Disconnect digit receiver';
        TASK RESET (T1);
        NEXTSTATE Idle;

STATE Await_next_digit;
    INPUT Digit (Dig);
        TASK RESET (T2);
        JOIN 10;
    INPUT T2;
        TASK 'Disconnect digit receiver';
        JOIN 1;
    INPUT A_on_hook;
        TASK 'Disconnect digit receiver';
        TASK RESET (T2);
        NEXTSTATE Idle;

STATE Ringing;
    INPUT B_off_hook;
        OUTPUT  Stop_ring_signal,
                Stop_ring_tone;
        TASK RESET (T3);
        TASK 'Connect path A-B';
        OUTPUT Start_metering_A;
        NEXTSTATE Conversation;
    INPUT A_on_hook;
        OUTPUT  Stop_ring_signal,
                Stop_ring_tone;
        TASK RESET (T3);
        TASK 'Release allocated path';
        NEXTSTATE Idle;
    INPUT T3;
        OUTPUT  Stop_ring_signal,
                Stop_ring_tone;
        TASK 'Release allocated path';
        JOIN 1;

STATE Conversation;
    INPUT A_on_hook;
        OUTPUT Stop_metering_A
        NEXTSTATE Await_B_on_hook;
    INPUT B_on_hook;
        TASK SET (NOW +D, T4);
        NEXTSTATE Await_A_on_hook;
```

FIGURE D-214 (page 3 of 4)

System definition using SDL/PR-syntax

```
    STATE Await_B_on_hook;
        INPUT A_off_hook;
            OUTPUT Start_metering_A;
            NEXTSTATE Conversation;
        INPUT B_on_hook;
            TASK 'Disconnect path';
            NEXTSTATE Idle;
    STATE Await_A_on_hook;
        INPUT A_on_hook
            OUTPUT Stop_metering_A;
            TASK RESET (T4);
            TASK 'Disconnect path';
            NEXTSTATE Idle;
        INPUT B_off_hook;
            TASK RESET (T4);
            NEXTSTATE Conversation;
        INPUT T4;
            OUTPUT Stop_metering_A;
            TASK 'Disconnect path';
        1:  TASK 'Select tone type';
            OUTPUT Send_tone (Tone_type);
            TASK SET (NOW +E, T5);
            NEXTSTATE Tone_connected;


    STATE Tone_connected;
        INPUT A_on_hook;
            OUTPUT Stop_tone;
            TASK RESET (T5);
            NEXTSTATE Idle;
        INPUT T5;
            JOIN 1;
ENDPROCESS Call_handling;
ENDBLOCK Call_handling;
ENDSYSTEM;
```

FIGURE D-214 (page 4 of 4)

System definition using SDL/PR-syntax

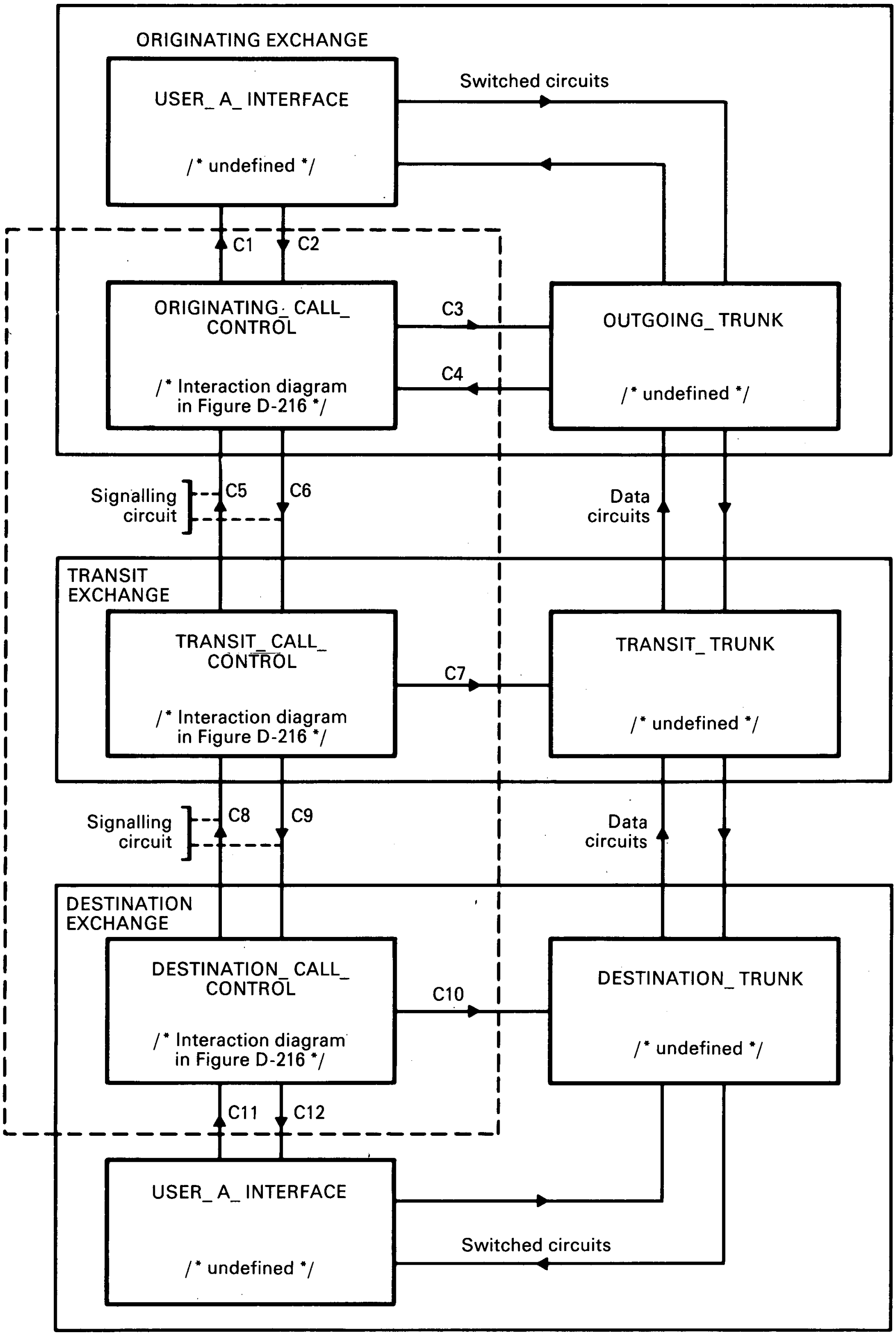
D.9.3 *Data user part of common channel signalling system*

The following example is based on Recommendation X.61. It is not necessarily equivalent to X.61 and is not intended as an international standard. The example illustrates the use of the procedure and option concepts which are defined in Recommendation Z.103.

In a circuit switched network which employs Common Channel Signalling, the signalling messages for a group of interexchange trunk circuits are conveyed by a centralized signalling network. Figure D-215 is an overview of a circuit switched data network comprising three exchanges. The data user part of the common channel signalling system is represented by the call control functional blocks in the originating, transit and destination exchanges. The overview diagram is not a formal part of the specification but it is included to explain the relationships between the signalling system and the other components of the data network. A Block Interaction Diagram of the signalling system is shown in Figure D-216.

The Originating, Transit and Destination processes (Figures D-217, D-218 and D-219) represent the functions necessary to establish, maintain and terminate a data connection between two user interfaces. During the call establishment phase, the destination exchange may implement one of two responses to the incoming call. These alternatives are defined by the use of the option symbol. Call acceptance by the called user results in a Ready for Data signal being sent to the originating process when the destination exchange has connected through.

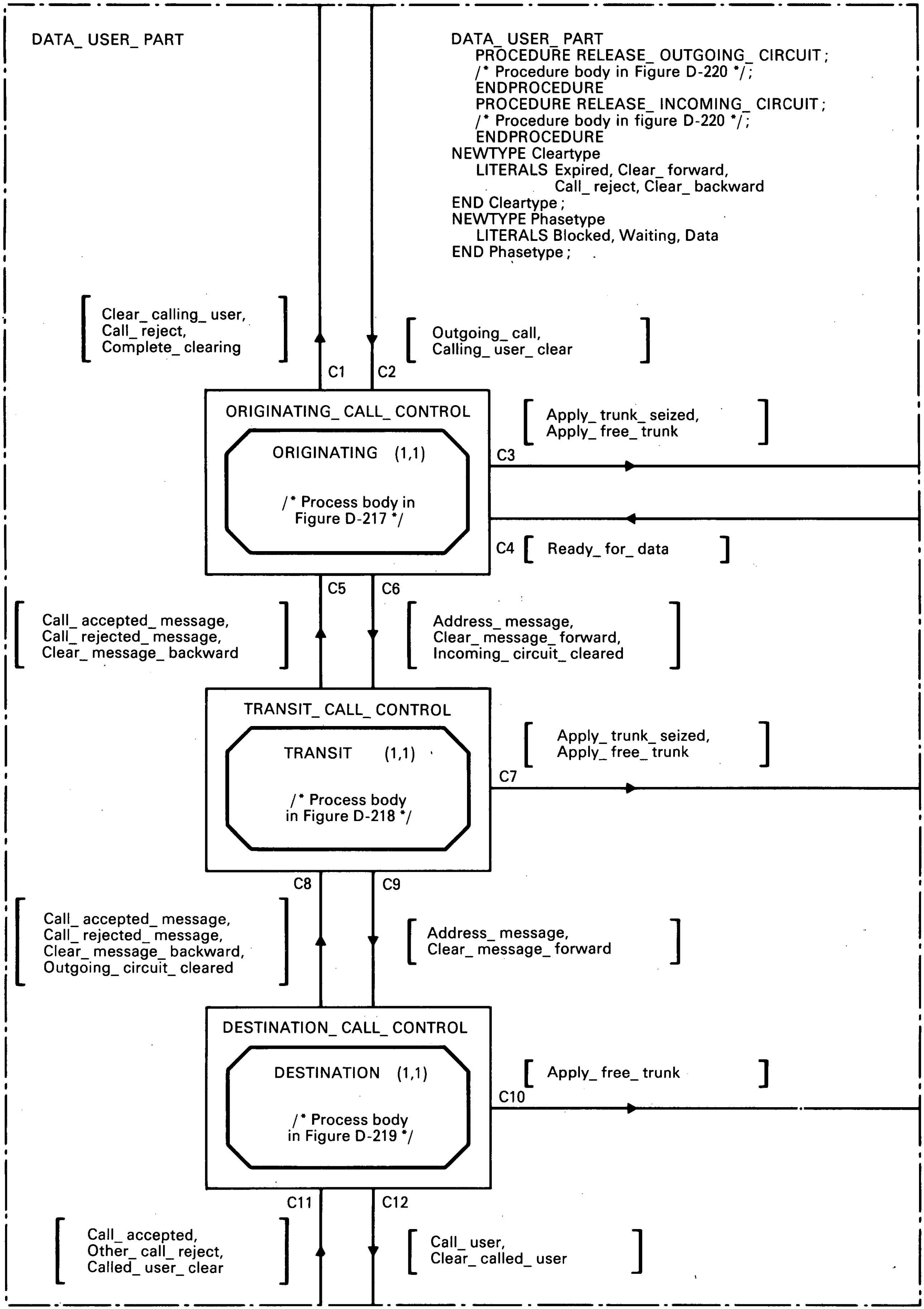
The Originating, Transit and Destination processes all perform similar functions to release the trunk circuits when the call is terminated. These functions are defined by the global procedures in Figures D-220 and D-221.



Note – Data User Part of common channel signalling system show in dotted boundary.

FIGURE D-215

Overview of circuit switched data network comprising three exchanges



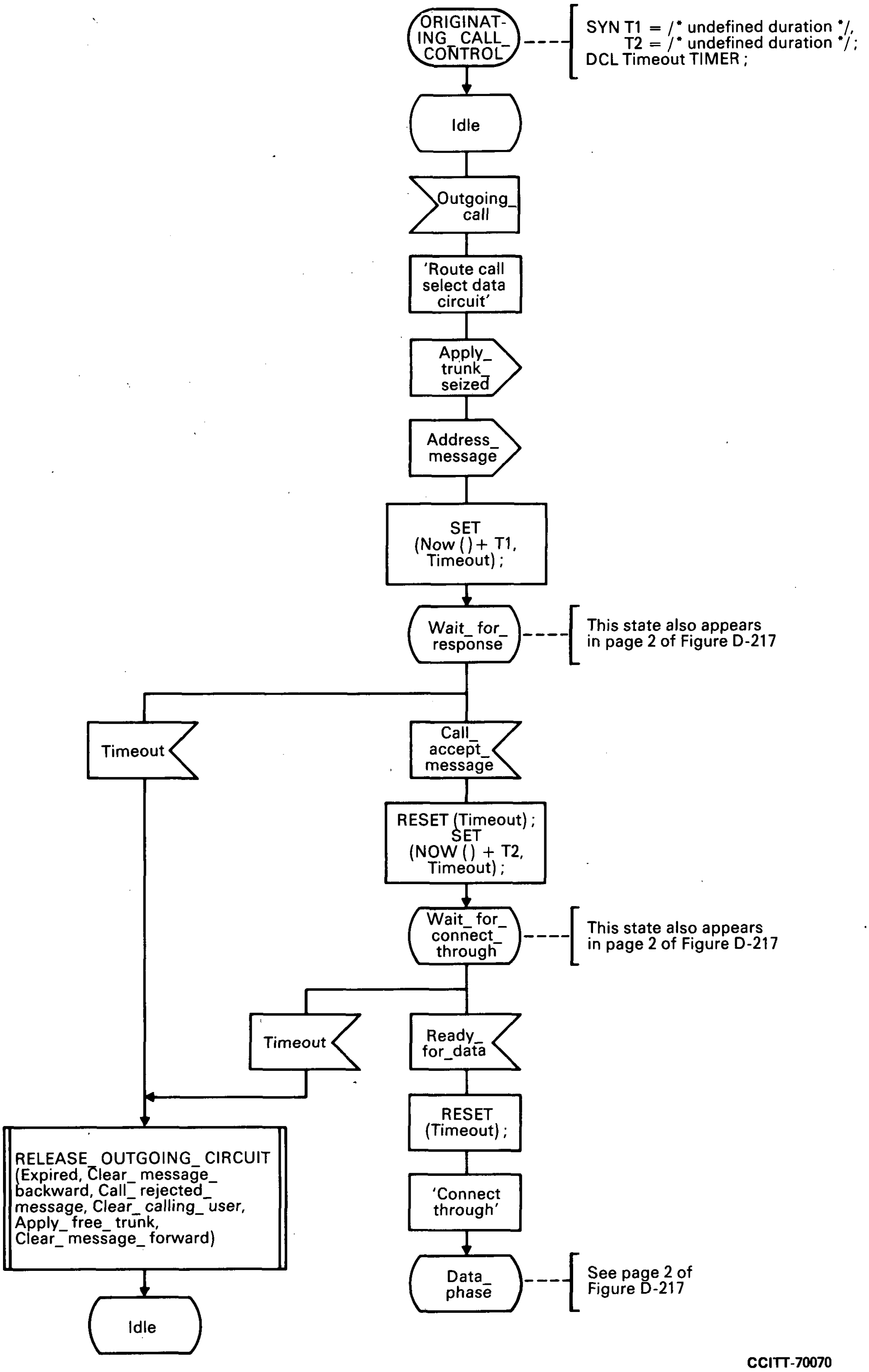
```
DATA_ USER_ PART
    PROCEDURE RELEASE_ OUTGOING_ CIRCUIT;
    /* Procedure body in Figure D-220 */;
    ENDPROCEDURE
    PROCEDURE RELEASE_ INCOMING_ CIRCUIT;
    /* Procedure body in figure D-220 */;
    ENDPROCEDURE
NEWTYPE Cleartype
    LITERALS Expired, Clear_ forward,
                Call_ reject, Clear_ backward
END Cleartype;
NEWTYPE Phasetype
    LITERALS Blocked, Waiting, Data
END Phasetype;
```

DATA_ USER_ PART
[Clear_ calling_ user, Call_ reject, Complete_ clearing]
C1
C2
[Outgoing_ call, Calling_ user_ clear]
ORIGINATING_ CALL_ CONTROL
ORIGINATING (1,1)
/* Process body in Figure D-217 */
[Apply_ trunk_ seized, Apply_ free_ trunk]
C3
C4 [Ready_ for_ data]
C5
C6
[Call_ accepted_ message, Call_ rejected_ message, Clear_ message_ backward]
[Address_ message, Clear_ message_ forward, Incoming_ circuit_ cleared]
TRANSIT_ CALL_ CONTROL
TRANSIT (1,1)
/* Process body in Figure D-218 */
[Apply_ trunk_ seized, Apply_ free_ trunk]
C7
C8
C9
[Call_ accepted_ message, Call_ rejected_ message, Clear_ message_ backward, Outgoing_ circuit_ cleared]
[Address_ message, Clear_ message_ forward]
DESTINATION_ CALL_ CONTROL
DESTINATION (1,1)
/* Process body in Figure D-219 */
[Apply_ free_ trunk]
C10
C11
C12
[Call_ accepted, Other_ call_ reject, Called_ user_ clear]
[Call_ user, Clear_ called_ user]

CCITT-70060

FIGURE D-216

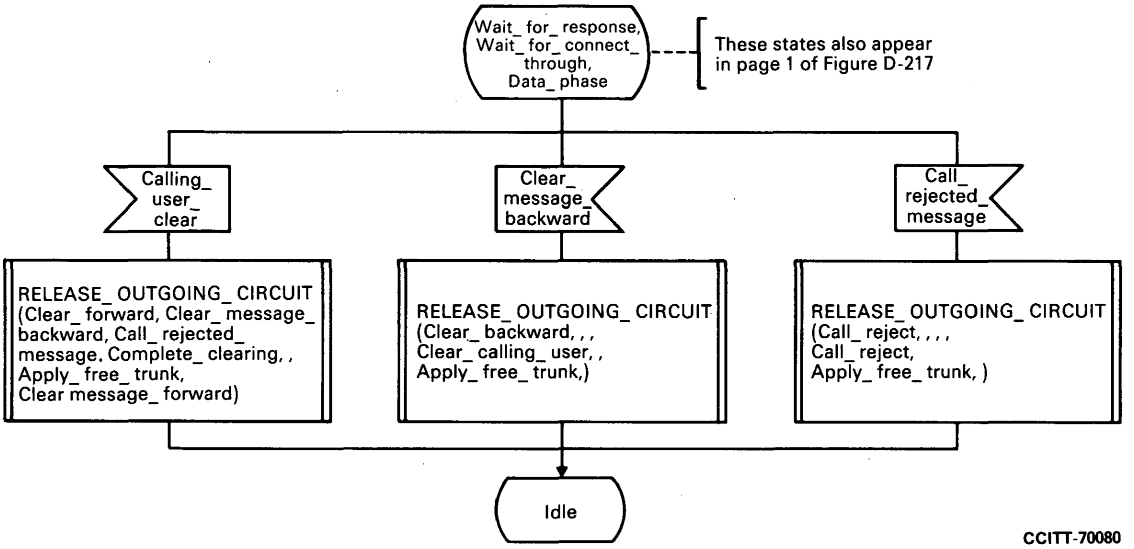
Block Interaction diagram for the system DATA_ USER_ PART


ORIGINAT-ING_CALL_CONTROL
SYN T1 = /* undefined duration */,
T2 = /* undefined duration */;
DCL Timeout TIMER ;
Idle
Outgoing_call
'Route call select data circuit'
Apply_trunk_seized
Address_message
SET (Now () + T1, Timeout) ;
Wait_for_response
This state also appears in page 2 of Figure D-217
Timeout
Call_accept_message
RESET (Timeout) ; SET (NOW () + T2, Timeout) ;
Wait_for_connect_through
This state also appears in page 2 of Figure D-217
Timeout
Ready_for_data
RESET (Timeout) ;
'Connect through'
Data_phase
See page 2 of Figure D-217
RELEASE_OUTGOING_CIRCUIT (Expired, Clear_message_backward, Call_rejected_message, Clear_calling_user, Apply_free_trunk, Clear_message_forward)
Idle

CCITT-70070

FIGURE D-217 (page 1 of 2)

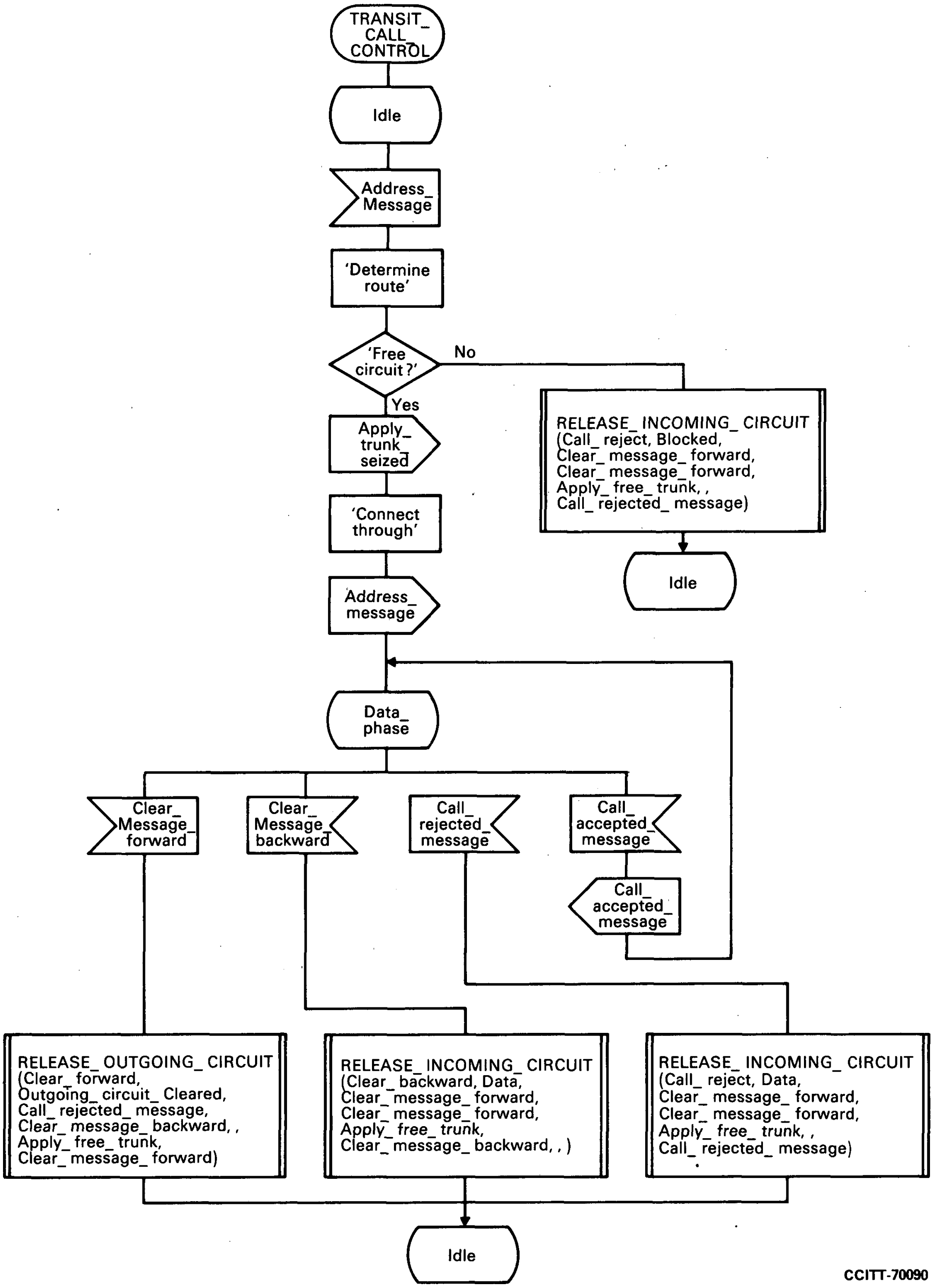
Process diagram for Originating_ Call_ Control


Wait_ for_ response,
Wait_ for_ connect_ through,
Data_ phase
These states also appear in page 1 of Figure D-217
Calling_ user_ clear
Clear_ message_ backward
Call_ rejected_ message
RELEASE_ OUTGOING_ CIRCUIT (Clear_ forward, Clear_ message_ backward, Call_ rejected_ message. Complete_ clearing, , Apply_ free_ trunk, Clear message_ forward)
RELEASE_ OUTGOING_ CIRCUIT (Clear_ backward, , , Clear_ calling_ user, , Apply_ free_ trunk,)
RELEASE_ OUTGOING_ CIRCUIT (Call_ reject, , , , Call_ reject, Apply_ free_ trunk,)
Idle

CCITT-70080

FIGURE D-217 (page 2 of 2)

Process diagram for Originating_ Call_ Control


TRANSIT_ CALL_ CONTROL
Idle
Address_ Message
'Determine route'
'Free circuit ?'
No
Yes
Apply_ trunk_ seized
'Connect through'
Address_ message
RELEASE_ INCOMING_ CIRCUIT (Call_ reject, Blocked, Clear_ message_ forward, Clear_ message_ forward, Apply_ free_ trunk, , Call_ rejected_ message)
Idle
Data_ phase
Clear_ Message_ forward
Clear_ Message_ backward
Call_ rejected_ message
Call_ accepted_ message
Call_ accepted_ message
RELEASE_ OUTGOING_ CIRCUIT (Clear_ forward, Outgoing_ circuit_ Cleared, Call_ rejected_ message, Clear_ message_ backward, , Apply_ free_ trunk, Clear_ message_ forward)
RELEASE_ INCOMING_ CIRCUIT (Clear_ backward, Data, Clear_ message_ forward, Clear_ message_ forward, Apply_ free_ trunk, Clear_ message_ backward, ,)
RELEASE_ INCOMING_ CIRCUIT (Call_ reject, Data, Clear_ message_ forward, Clear_ message_ forward, Apply_ free_ trunk, , Call_ rejected_ message)
Idle

CCITT-70090

FIGURE D-218

Process diagram for Transit_ Call_ Control

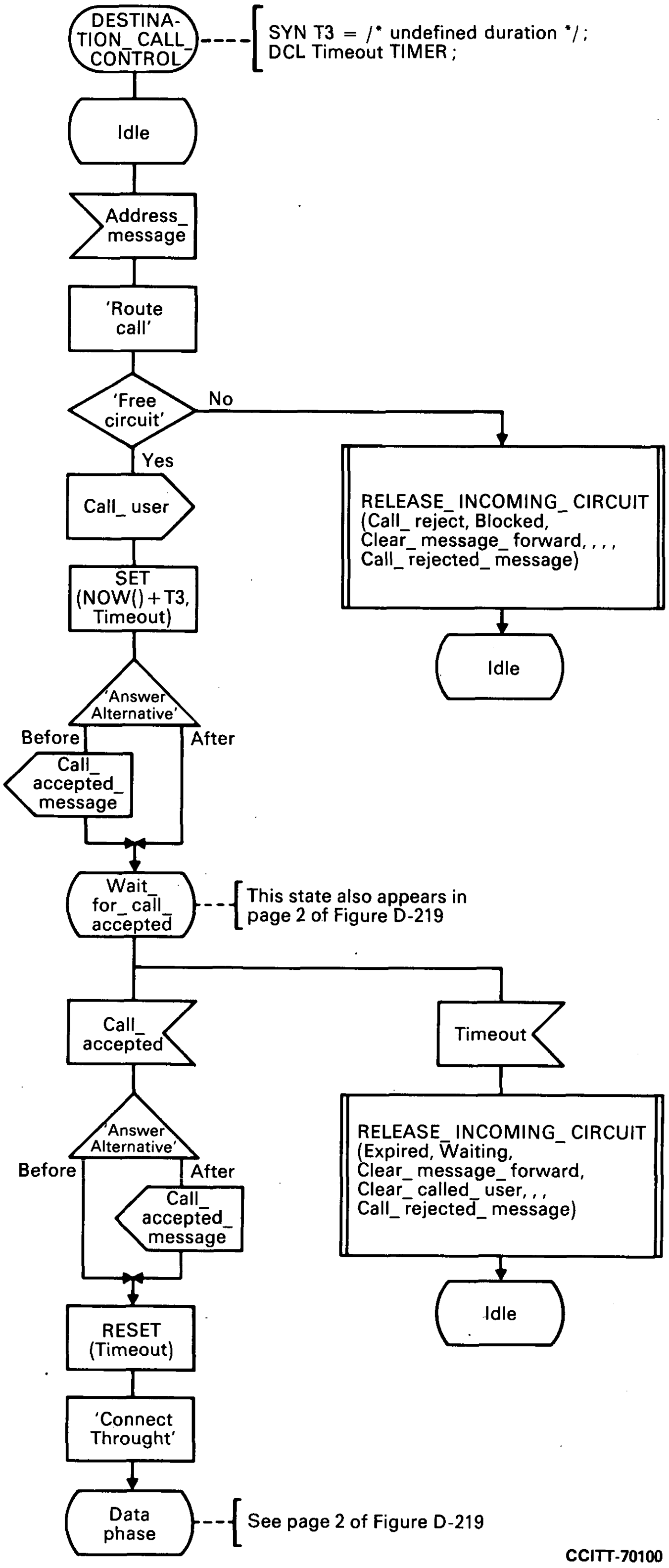

DESTINATION_CALL_CONTROL
SYN T3 = /* undefined duration */;
DCL Timeout TIMER;
Idle
Address_message
'Route call'
'Free circuit'
No
Yes
Call_user
RELEASE_INCOMING_CIRCUIT (Call_reject, Blocked, Clear_message_forward, , , , Call_rejected_message)
Idle
SET (NOW() + T3, Timeout)
'Answer Alternative'
Before
After
Call_accepted_message
Wait_for_call_accepted
This state also appears in page 2 of Figure D-219
Call_accepted
Timeout
'Answer Alternative'
Before
After
Call_accepted_message
RELEASE_INCOMING_CIRCUIT (Expired, Waiting, Clear_message_forward, Clear_called_user, , , Call_rejected_message)
Idle
RESET (Timeout)
'Connect Throught'
Data phase
See page 2 of Figure D-219
CCITT-70100

FIGURE D-219 (page 1 of 2)

Process diagram for Destination_ Call_ Control

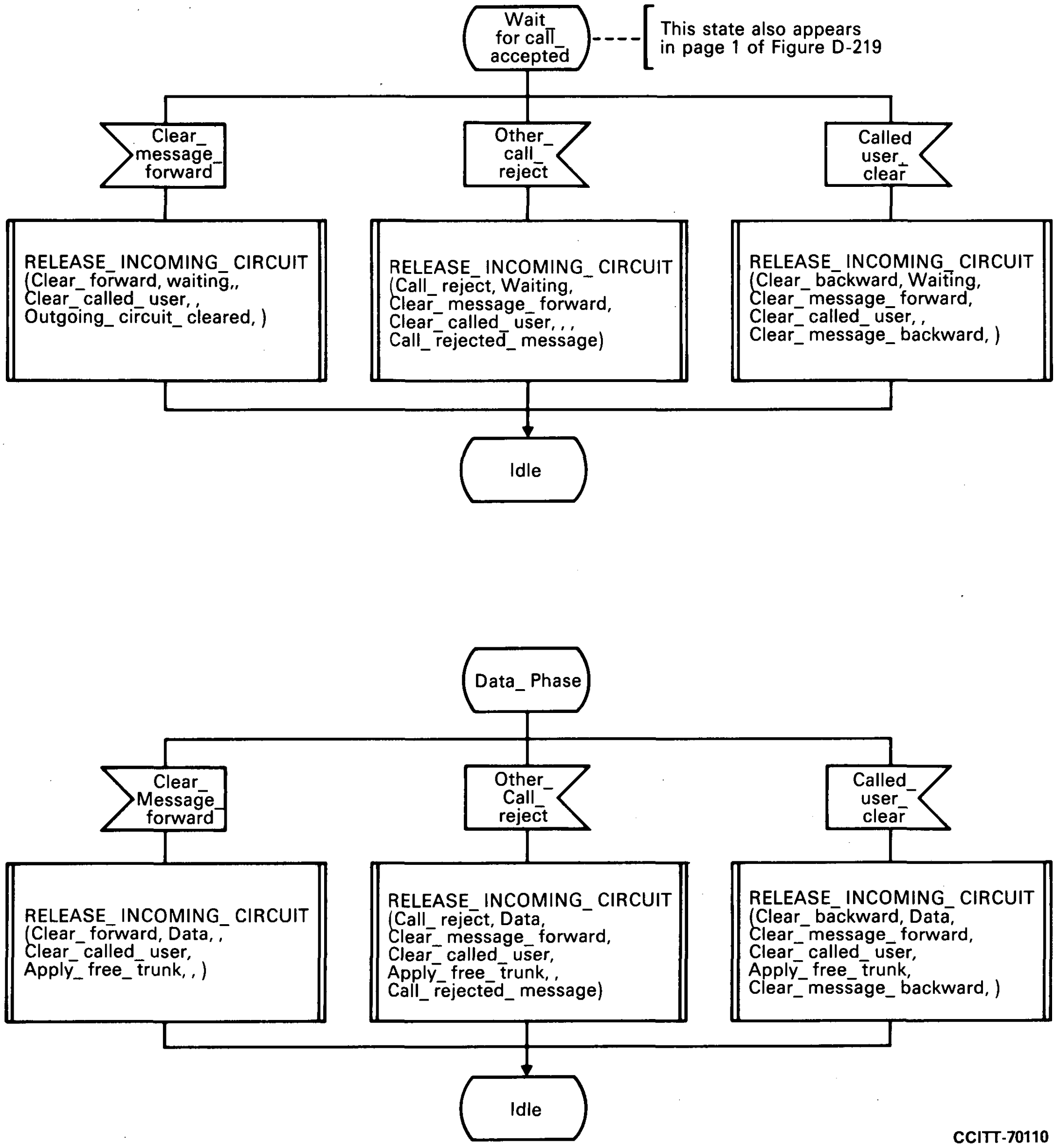

Wait_ for call_ accepted
This state also appears in page 1 of Figure D-219
Clear_ message_ forward
Other_ call_ reject
Called user_ clear
RELEASE_ INCOMING_ CIRCUIT (Clear_ forward, waiting,, Clear_ called_ user, , Outgoing_ circuit_ cleared,)
RELEASE_ INCOMING_ CIRCUIT (Call_ reject, Waiting, Clear_ message_ forward, Clear_ called_ user, , , Call_ rejected_ message)
RELEASE_ INCOMING_ CIRCUIT (Clear_ backward, Waiting, Clear_ message_ forward, Clear_ called_ user, , Clear_ message_ backward,)
Idle
Data_ Phase
Clear_ Message_ forward
Other_ Call_ reject
Called_ user_ clear
RELEASE_ INCOMING_ CIRCUIT (Clear_ forward, Data, , Clear_ called_ user, Apply_ free_ trunk, ,)
RELEASE_ INCOMING_ CIRCUIT (Call_ reject, Data, Clear_ message_ forward, Clear_ called_ user, Apply_ free_ trunk, , Call_ rejected_ message)
RELEASE_ INCOMING_ CIRCUIT (Clear_ backward, Data, Clear_ message_ forward, Clear_ called_ user, Apply_ free_ trunk, Clear_ message_ backward,)
Idle
CCITT-70110

FIGURE D-219 (page 2 of 2)

Process diagram for Destination_ Call_ Control

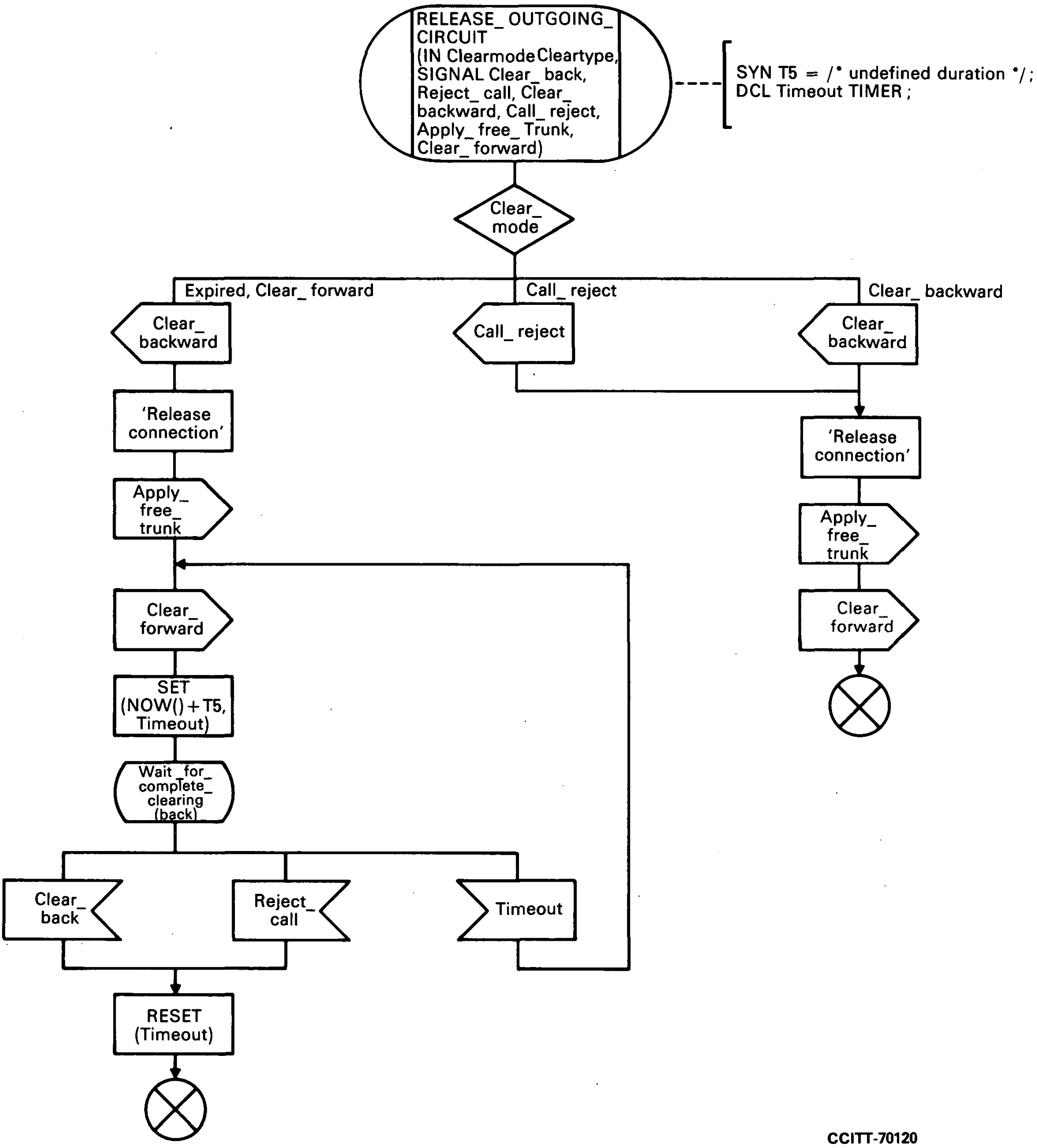


FIGURE D-220

Procedure diagram for Release_ Outgoing_ Circuit Procedure

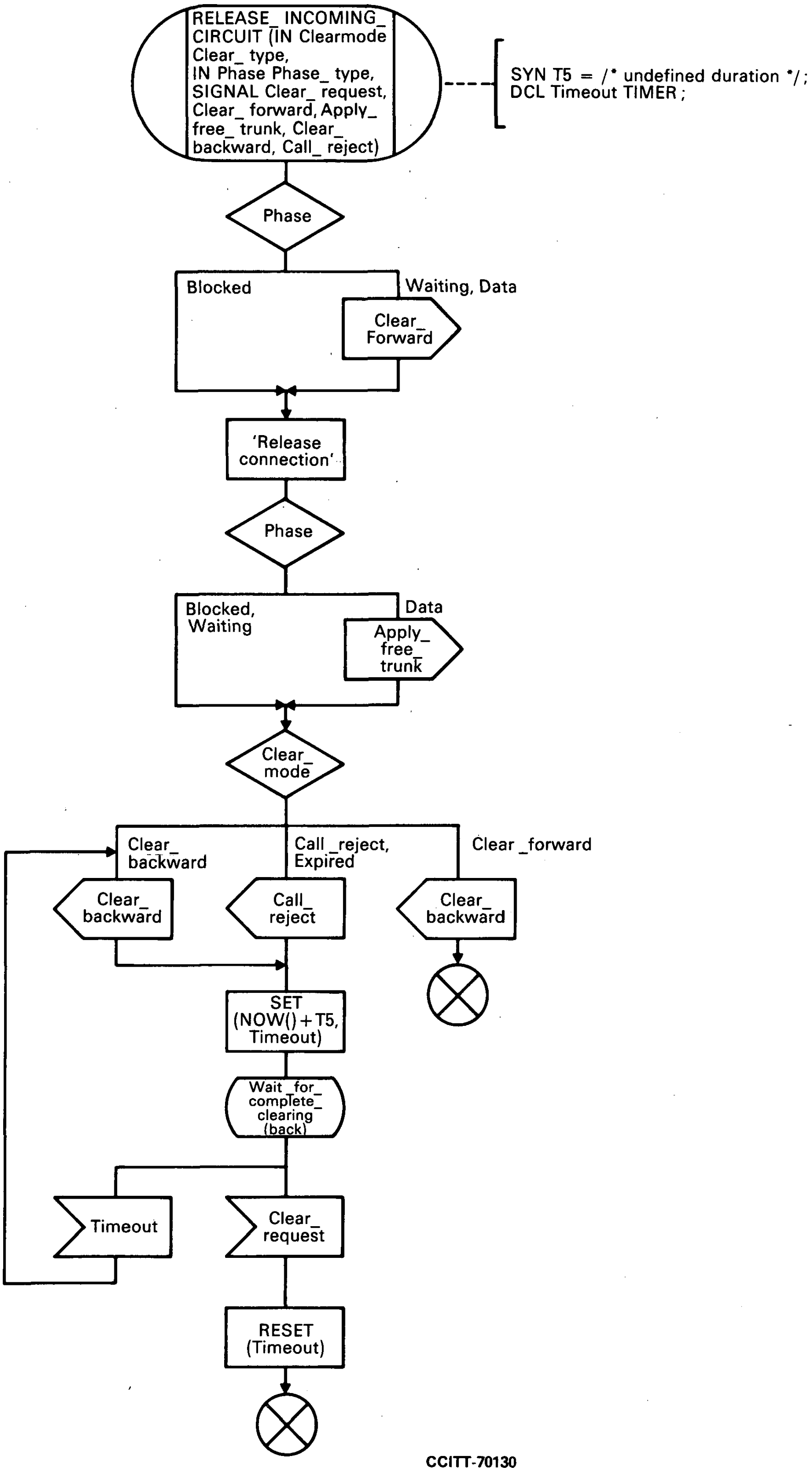

RELEASE_ INCOMING_ CIRCUIT (IN Clearmode Clear_ type, IN Phase Phase_ type, SIGNAL Clear_ request, Clear_ forward, Apply_ free_ trunk, Clear_ backward, Call_ reject)
SYN T5 = /* undefined duration */; DCL Timeout TIMER ;
Phase
Blocked
Waiting, Data
Clear_ Forward
'Release connection'
Phase
Blocked, Waiting
Data
Apply_ free_ trunk
Clear_ mode
Clear_ backward
Call _reject, Expired
Clear _forward
Clear_ backward
Call_ reject
Clear_ backward
SET (NOW() + T5, Timeout)
Wait _for_ complete_ clearing (back)
Timeout
Clear_ request
RESET (Timeout)
CCITT-70130

FIGURE D-221

Procedure diagram for Release_ Incoming_ Circuit Procedure

D.9.4 *Class 0 Transport Protocol*

The following example is based on the Transport Protocol Recommendation for Open Systems Interconnection. The example is a subset of the Transport Protocol which is not necessarily equivalent to the Recommendation, and is not intended as an international standard.

SDL following Recommendations Z.101 and Z.103 has been used. While data is employed, this is in a somewhat informal manner, without using Recommendation Z.104. Enumeration types of Z.104 have been simulated with INTEGER variables, with names being given to undefined constants. Operations on data have not been formally defined.

D.9.4.1 *Overview*

The Transport Layer of the OSI Reference Model provides a service to its users by providing communication channels (called Transport connections) between pairs of users. At each user connection point the Transport Layer is represented by a Transport Entity. Transport entities communicate with each other using a Transport Protocol and the services of the Network Layer. An overview diagram appears in Figure D-223. The example describes a Transport Entity which implements only the Class 0 Transport Protocol.

A Transport connection is established between two Transport entities. The model upon which the specification is based describes the behaviour of one entity independently, and may be applied to both ends of the connection. The Interaction Diagram, Figure D-224, shows a single block representing the Transport Entity, linked to the Environment by channels known as the Transport Service Access Point (TSAP) and Network Service Access Point (NSAP). The Transport Service User and the Network Layer are considered as the Environment.

The Transport Entity contains two processes. The first, the DIRECTOR, has a single instance which is always present (indicated by the numbers at the top right hand corner of the process symbol). The second process, the ENDPOINT, has only dynamically created instances, one of which is created for the lifetime of each attempt to establish a transport connection. If a signal 'T_CONNECT request' or 'N_CONNECT indication' is received by the DIRECTOR, the latter immediately creates a new instance of ENDPOINT to handle all subsequent signals associated with the connection attempt. Each instance of ENDPOINT has a property 'kind', with one of the values INITIATOR or RESPONDER, which determines the initial behaviour of the instance; the instance terminates itself when the connection is closed by one of the transport service users, or after an error.

The behaviour of DIRECTOR is given by the process diagram in Figure D-225. That of ENDPOINT is given by Figures D-226 to D-230. The specification of ENDPOINT makes use of SDL procedures for the connection establishment, data transfer, and termination phases.

D.9.4.2 *SDL specification of the system*

The system is specified by the following SDL-PR text. The text deviates from the Recommendations in that some parts are given in SDL-GR form with references to these given in the SDL-PR in the form of comments.

```
SYSTEM TRANSPORT_ENTITY;
          /*        the role of a TRANSPORT_ENTITY in a transport connection between two users is shown in the
                    overview diagram of Figure D-223. */
          /*        channels and associated signals are shown in the interaction diagram, Figure D-224 */
    SIGNAL /* input-list */
        T_CONN_REQ          (PID,STRING)    ,
        T_CONN_RSP          (PID,STRING)    ,
        T_CONN_RPLY                         ,
        T_DATA_REQ          (STRING)        ,
        T_DISC_REQ                          ,
        N_CONN_IND          (PID,STRING)    ,
        N_CONN_CFM          (PID,STRING)    ,
        N_CONN_RPLY                         ,
        N_DATA_IND          (STRING)        ,
        N_DISC_IND                          ,
        N_RESET_IND                         ;
            MACRO EXPANSION signals;
                T_CONN_REQ, T_CONN_RSP, T_CONN_RPLY, T_DATA_REQ, T_DISC_REQ, N_CONN_-
        IND, N_CONN_CFM, N_CONN_RPLY, N_DATA_IND, N_DISC_IND,
                N_RESET_IND,
            ENDMACRO signals;
    SIGNAL /* output-list */
        T_CONN_IND          (PID,STRING)    ,
        T_CONN_CFM          (PID,STRING)    ,
        T_DATA_IND          (STRING)        ,
        T_DISC_IND                          ,
        N_CONN_REQ          (PID,STRING)    ,
        N_CONN_RSP          (PID,STRING)    ,
        N_DATA_REQ          (STRING)        ,
        N_DISC_REQ                          ;
```

FIGURE D-222 (page 1 of 3)

System specification using SDL/PR

```
BLOCK TRANSPORT_ENTITY;
      PROCESS DIRECTOR (1,1);
          /* the single instance of this process controls the assignment of ENDPOINTs */
          DCL  conn_params STRING,
               conn_id PID ;
          /* process diagram, Figure D-225, gives process body */
      ENDPROCESS DIRECTOR;
          PROCESS ENDPOINT (0, );
               /* an instance of this process is created by DIRECTOR for each connection requested */
               FPAR
                      kind             INT /* values INITIATOR, RESPONDER */ ,
                      conn_params      STRING ,
                      conn_id          PID;
               DCL
                      ref              STRING ,
                      tc_id            PID ,
                      nc_id            PID ,
                      tpdu_size        INT   /* values                                   */ ,
                      cstat            INT   /* values DATA_TRANSFER, DISCONNECTED       */ ;
                          IMPORTED /* from tc_id via channel TSAP_in */
                      t_ind_flo        BOOL ,
                          IMPORTED /* from nc_id via channel NSAP_in */
                      n_req_flo        BOOL ,
                          EXPORTED /* to tc_id via channel TSAP_out */
                      t_req_flo        BOOL ,
                          EXPORTED /* to nc_id via channel NSAP_out */
                      n_ind_flo        BOOL ;

      PROCEDURE INITIATE_CONNECT;
          SIGNAL      MACRO signals;
          IN          conn_params      STRING ,
          IN          tc_id            PID ,
          IN/OUT      nc_id            PID ,
          IN/OUT      tpdu_size        INT ,
          IN/OUT      cstat          INT /* values DATA_TRANSFER, DISCONNECTED *
        DCL
          n_conn_params STRING ,
          prop_tpdu_size INT ,
          tpdu_flag       INT /* values CC, DR, ER, CR, DT, INV */ ,
          tpdu_data       STRING ,
          nbin            STRING ,
          nbout           STRING ,
          tbout           STRING ,
          taskr           INT /* values CLOSE, NC, TC, NC&TC, NONE */;
        /* procedure diagram, Figure D-227, gives procedure body */
      ENDPROCEDURE INITIATE_CONNECT;
      PROCEDURE RESPOND_CONNECT
          SIGNAL      MACRO signals;
          IN          conn_params      STRING ,
          IN/OUT      tc_id            PID ,
          IN          nc_id            PID ,
          IN/OUT      tpdu_size        INT ,
          IN/OUT      cstat INT /* values DATA_TRANSFER, DISCONNECTED */ ;
        DCL
          n_conn_params STRING ,
          prop_tpdu_size INT ,
          tpdu_flag       INT /* values CC, DR, ER, CR, DT, INV */ ,
          tpdu_data       STRING ,
          nbin            STRING ,
          nbout           STRING ,
          tbin            STRING ,
          tbout           STRING ,
          taskr           INT /* values CLOSE, NC, TC, NC&TC, NONE */
        /* procedure diagram, Figure D-228, gives procedure body */
      ENDPROCEDURE RESPOND_CONNECT;
```

FIGURE D-222 (page 2 of 3)

System specification using SDL/PR

```
          PROCEDURE DATA_PHASE;
              SIGNAL        MACRO signals;
              IN            ref                 STRING ,
              IN            tc_id               PID ,
              IN            nc_id               PID ,
              IN            tpdu_size           INT ,
              IN/OUT        cstat               INT /* values DATA_TRANSFER, DISCONNECTED */,
              IN/OUT IMPORTED /* from tc_id via channel TSAP_in */
              t_ind_flo                         BOOL ,
              IN/OUT IMPORTED /* from nc_id via channel NSAP_in */
              n_req_flo                         BOOL ,
              IN/OUT EXPORTED /* to tc_id via channel TSAP_out */
              t_req_flo                         BOOL ,
              IN/OUT EXPORTED /* to nc_id via channel NSAP_out */
              n_ind_flo                         BOOL ;
            DCL
              input_present                     BOOL ,
              output_present.                   BOOL ,
              tbin                              STRING ,
              nbin                              STRING ,
              tbout                             STRING ,
              nbout                             STRING ,
              output_segment                    STRING ,
              input_tsdu                        STRING ,
              tpdu_flag                         INT /* values CC, DR, ER, CR, DT, INV */ ,
              taskr                             INT /* values CLOSE, NC, TC, NC&TC, NONE */;
            /* procedure diagram, Figure D-229, gives procedure body */
          ENDPROCEDURE DATA_PHASE;
          PROCEDURE RELEASE_CONNECT
              SIGNAL        MACRO signals;

              IN            taskr               INT /* values CLOSE, NC, TC, NC&TC, NONE */,
              IN            tc_id               PID ,
              IN            nc_id               PID ,

              IN/OUT        cstat               INT /* values DATA_TRANSFER, DISCONNECTED */,
              IN/OUT        tpdu_flag           INT /* values CC, DR, ER, CR, DT, INV */ ,
              IN/OUT        last_tpdu           STRING;
            DCL
              nbin                              STRING ,
              nbout                             STRING ,
              tbout                             STRING ;
            /* procedure diagram, Figure D-230, gives procedure body */
          ENDPROCEDURE RELEASE_CONNECT;
        /* process diagram, Figure D-226, gives process body */
      ENDPROCESS ENDPOINT;
    ENDBLOCK TRANSPORT_ENTITY;
ENDSYSTEM TRANSPORT_ENTITY;
```

FIGURE D-222 (page 3 of 3)

System specification using SDL/PR

D.9.4.3 *Operations upon data items*

Operations upon data items are represented informally. These are:

alloc_ref: gives a unique Transport Protocol connection reference

form_t_conn_ind:
form_t_conn_cfm:
form_t_data_ind:
form_t_disc_ind:
form_n_conn_req:
form_n_data_req:
form_n_conn_rsp:
form_n_disc_req:
Each of these operations forms the set of data fields for the corresponding service primitive from the parameters supplied, together with background knowledge of the Transport entity (e.g. mapping between transport and network addresses in the case of n_conn_req)

conn_feasible: Transport entity knowledge at the calling TSAP indicates that the connection may be achieved (local resources available, called TSAP reachable, quality of service achievable) – boolean

conn_acceptable: the Transport entity at the called NSAP has resources to accept the network connection – boolean

conn_providable: Transport entity knowledge at the called end indicates that the transport connection may be achieved (called TSAP available, quality of service achieved) – boolean

tpdu-type: determines the content of n_data_ind (values: CR, CC, DT, DR, ER or INV – the last indicates a TPDU invalid for any reason)

prop_tpdu_size: initiator's proposal for value of tpdu_size

agree_tpdu_size: responder's agreed value for tpdu_size

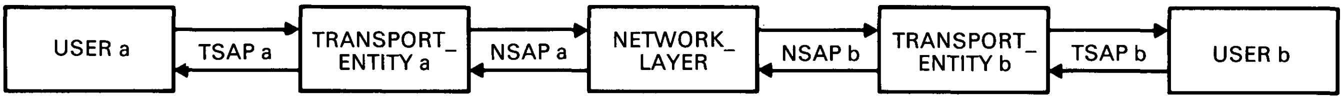
extract_tpdu_size: determines parameter value contained in CC TPDU

take_segment: takes next DT TPDU from TSDU

append_segment: adds current DT TPDU to partially completed TSDU

last_tpdu: current DT TPDU has End of TSDU mark set

The input values to operations are listed within brackets after the operation name.



CCITT-70140

FIGURE D-223

Overview diagram showing the role of a TRANSPORT_ ENTITY in a Transport Connection

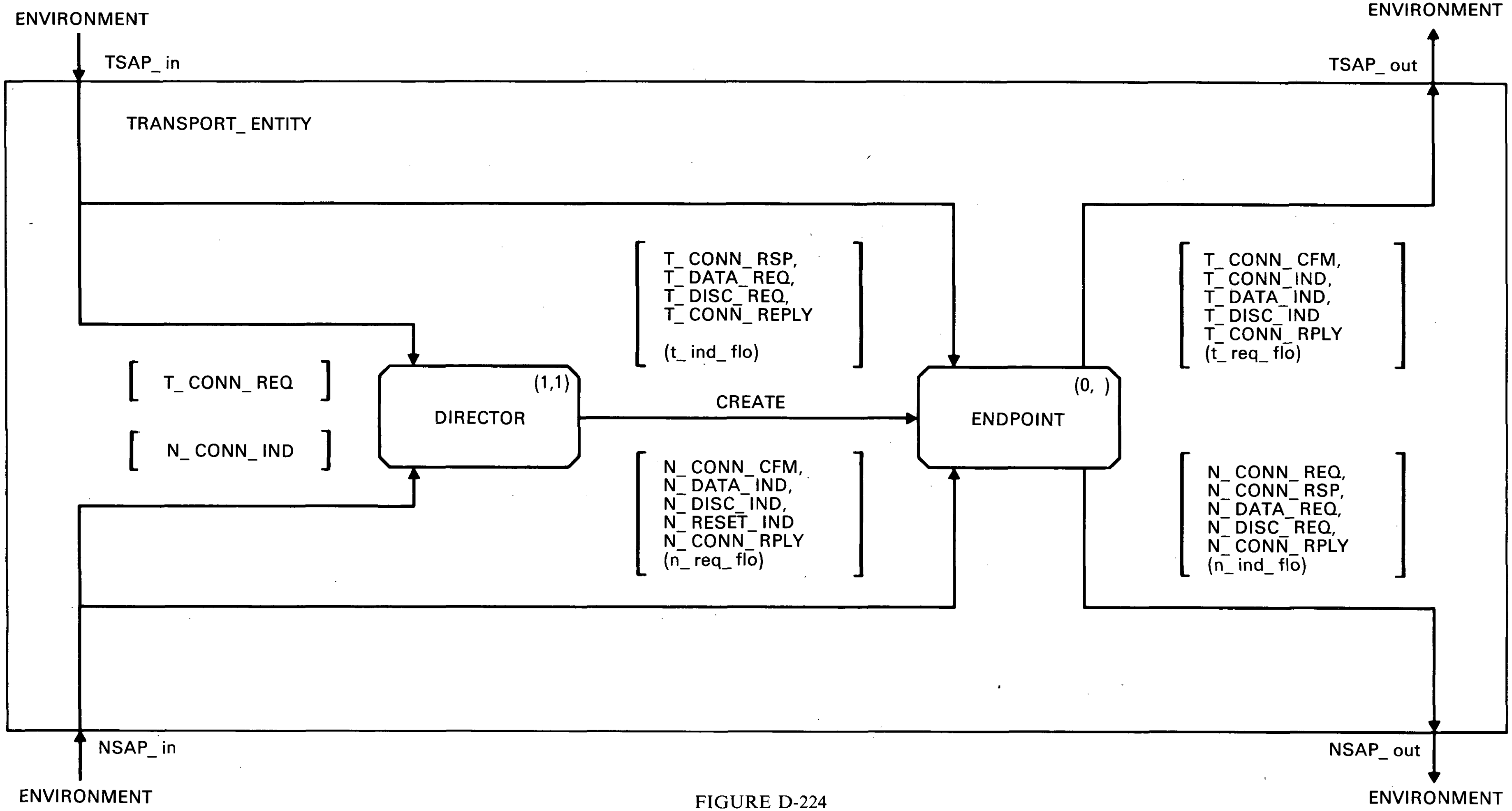


FIGURE D-224

Interaction diagram for TRANSPORT_ ENTITY block

CCITT-70145

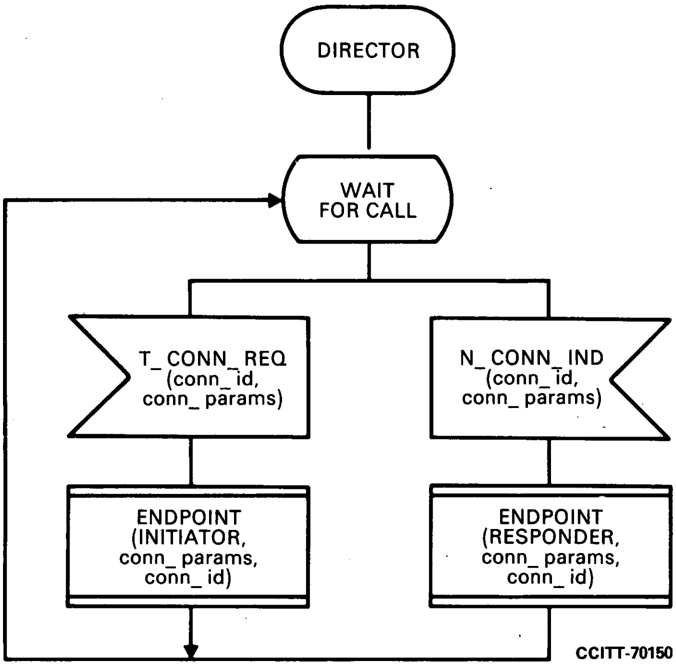

DIRECTOR
WAIT FOR CALL
T_CONN_REQ (conn_id, conn_params)
N_CONN_IND (conn_id, conn_params)
ENDPOINT (INITIATOR, conn_params, conn_id)
ENDPOINT (RESPONDER, conn_params, conn_id)
CCITT-70150

FIGURE D-225

Process diagram for DIRECTOR

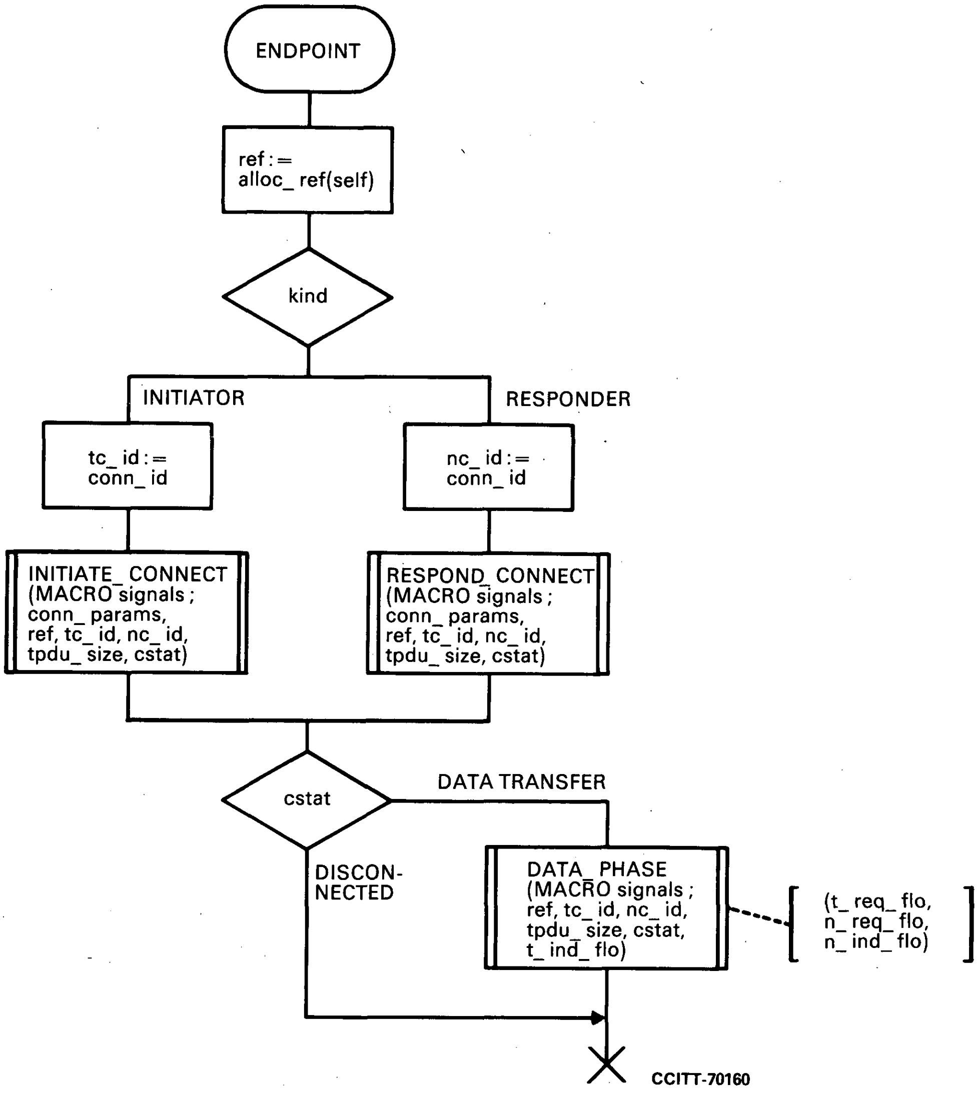


FIGURE D-226

Process diagram for ENDPOINT

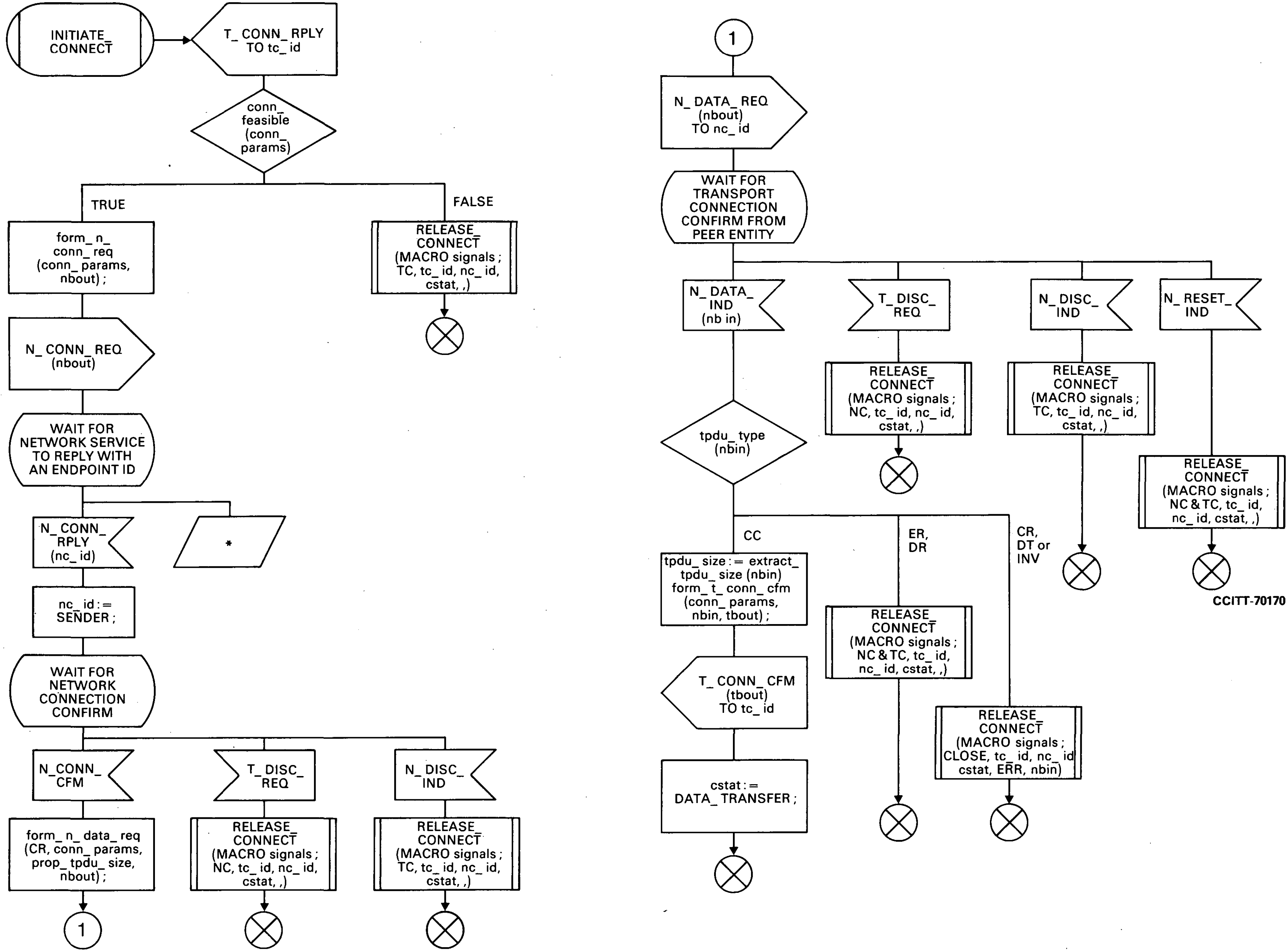

INITIATE_ CONNECT
T_ CONN_ RPLY TO tc_ id
conn_ feasible (conn_ params)
TRUE
FALSE
form_ n_ conn_ req (conn_ params, nbout) ;
RELEASE_ CONNECT (MACRO signals ; TC, tc_ id, nc_ id, cstat, ,)
N_ CONN_ REQ (nbout)
WAIT FOR NETWORK SERVICE TO REPLY WITH AN ENDPOINT ID
N_ CONN_ RPLY (nc_ id)
*
nc_ id := SENDER ;
WAIT FOR NETWORK CONNECTION CONFIRM
N_CONN_ CFM
T_ DISC_ REQ
N_ DISC_ IND
form_ n_ data_ req (CR, conn_ params, prop_ tpdu_ size, nbout) ;
RELEASE_ CONNECT (MACRO signals ; NC, tc_ id, nc_ id, cstat, ,)
RELEASE_ CONNECT (MACRO signals ; TC, tc_ id, nc_ id, cstat, ,)
1
1
N_ DATA_ REQ (nbout) TO nc_ id
WAIT FOR TRANSPORT CONNECTION CONFIRM FROM PEER ENTITY
N_ DATA_ IND (nb in)
T_ DISC_ REQ
N_ DISC_ IND
N_ RESET_ IND
RELEASE_ CONNECT (MACRO signals ; NC, tc_ id, nc_ id, cstat, ,)
RELEASE_ CONNECT (MACRO signals ; TC, tc_ id, nc_ id, cstat, ,)
RELEASE_ CONNECT (MACRO signals ; NC & TC, tc_ id, nc_ id, cstat, ,)
tpdu_ type (nbin)
CC
ER, DR
CR, DT or INV
tpdu_ size := extract_ tpdu_ size (nbin) form_ t_ conn_ cfm (conn_ params, nbin, tbout) ;
RELEASE_ CONNECT (MACRO signals ; NC & TC, tc_ id, nc_ id, cstat, ,)
T_ CONN_ CFM (tbout) TO tc_ id
RELEASE_ CONNECT (MACRO signals ; CLOSE, tc_ id, nc_ id cstat, ERR, nbin)
cstat := DATA_ TRANSFER ;
CCITT-70170

FIGURE D-227

Procedure diagram for INITIATE_ CONNECT

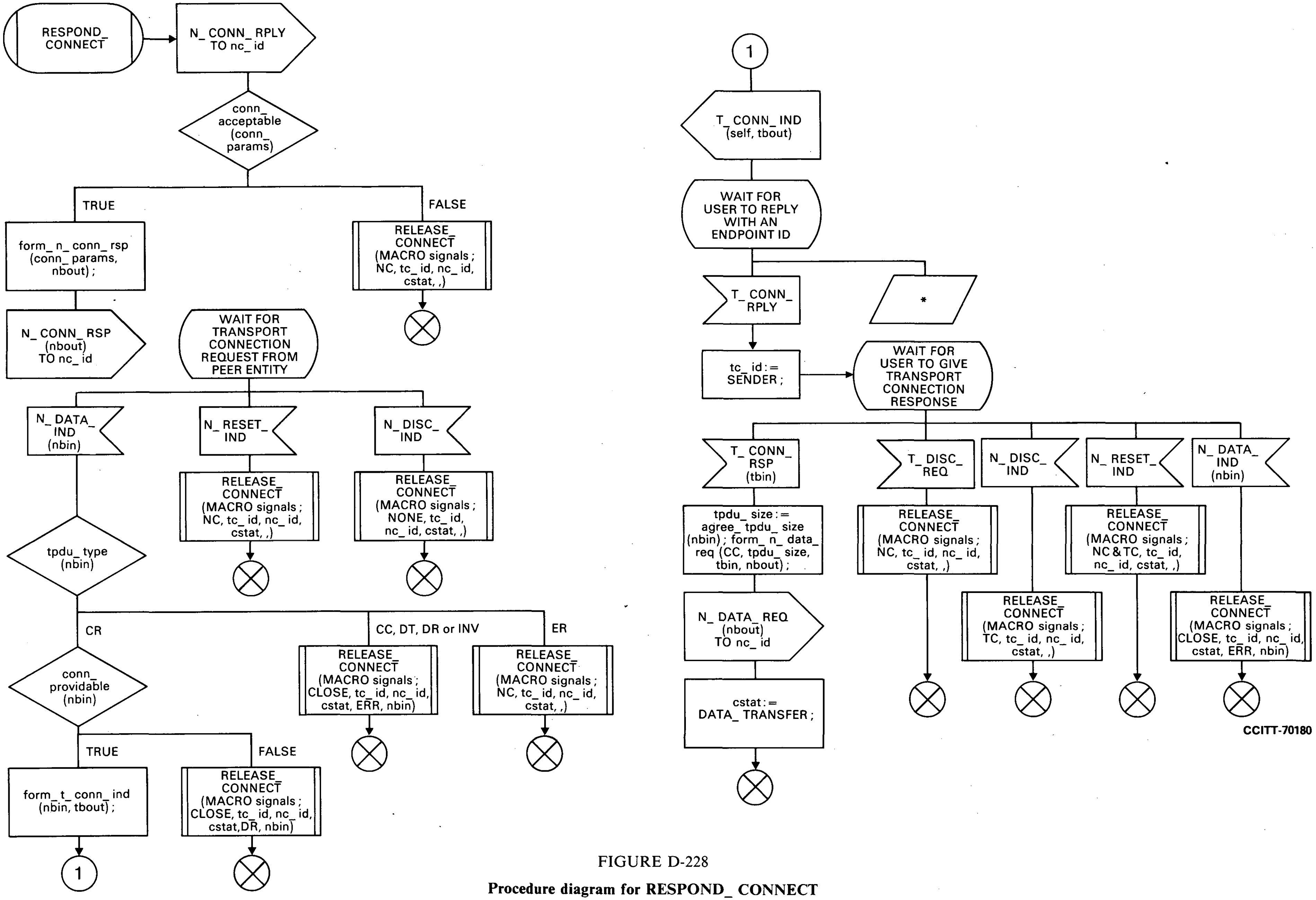

RESPOND_ CONNECT
N_ CONN_ RPLY TO nc_ id
conn_ acceptable (conn_ params)
TRUE
FALSE
form_ n_ conn_ rsp (conn_ params, nbout);
RELEASE_ CONNECT (MACRO signals; NC, tc_ id, nc_ id, cstat, ,)
N_ CONN_ RSP (nbout) TO nc_ id
WAIT FOR TRANSPORT CONNECTION REQUEST FROM PEER ENTITY
N_ DATA_ IND (nbin)
N_ RESET_ IND
N_ DISC_ IND
RELEASE_ CONNECT (MACRO signals; NC, tc_ id, nc_ id, cstat, ,)
RELEASE_ CONNECT (MACRO signals; NONE, tc_ id, nc_ id, cstat, ,)
tpdu_ type (nbin)
CR
CC, DT, DR or INV
ER
conn_ providable (nbin)
RELEASE_ CONNECT (MACRO signals; CLOSE, tc_ id, nc_ id, cstat, ERR, nbin)
RELEASE_ CONNECT (MACRO signals; NC, tc_ id, nc_ id, cstat, ,)
TRUE
FALSE
form_ t_ conn_ ind (nbin, tbout);
RELEASE_ CONNECT (MACRO signals; CLOSE, tc_ id, nc_ id, cstat,DR, nbin)
1
1
T_ CONN_ IND (self, tbout)
WAIT FOR USER TO REPLY WITH AN ENDPOINT ID
T_ CONN_ RPLY
*
tc_ id := SENDER;
WAIT FOR USER TO GIVE TRANSPORT CONNECTION RESPONSE
T_ CONN_ RSP (tbin)
T_ DISC_ REQ
N_ DISC_ IND
N_ RESET_ IND
N_ DATA_ IND (nbin)
tpdu_ size := agree_ tpdu_ size (nbin); form_ n_ data_ req (CC, tc_ id, tpdu_ size, tbin, nbout);
RELEASE_ CONNECT (MACRO signals; NC, tc_ id, nc_ id, cstat, ,)
RELEASE_ CONNECT (MACRO signals; NC & TC, tc_ id, nc_ id, cstat, ,)
N_ DATA_ REQ (nbout) TO nc_ id
RELEASE_ CONNECT (MACRO signals; TC, tc_ id, nc_ id, cstat, ,)
RELEASE_ CONNECT (MACRO signals; CLOSE, tc_ id, nc_ id, cstat, ERR, nbin)
cstat := DATA_ TRANSFER;
CCITT-70180

FIGURE D-228

Procedure diagram for RESPOND_ CONNECT

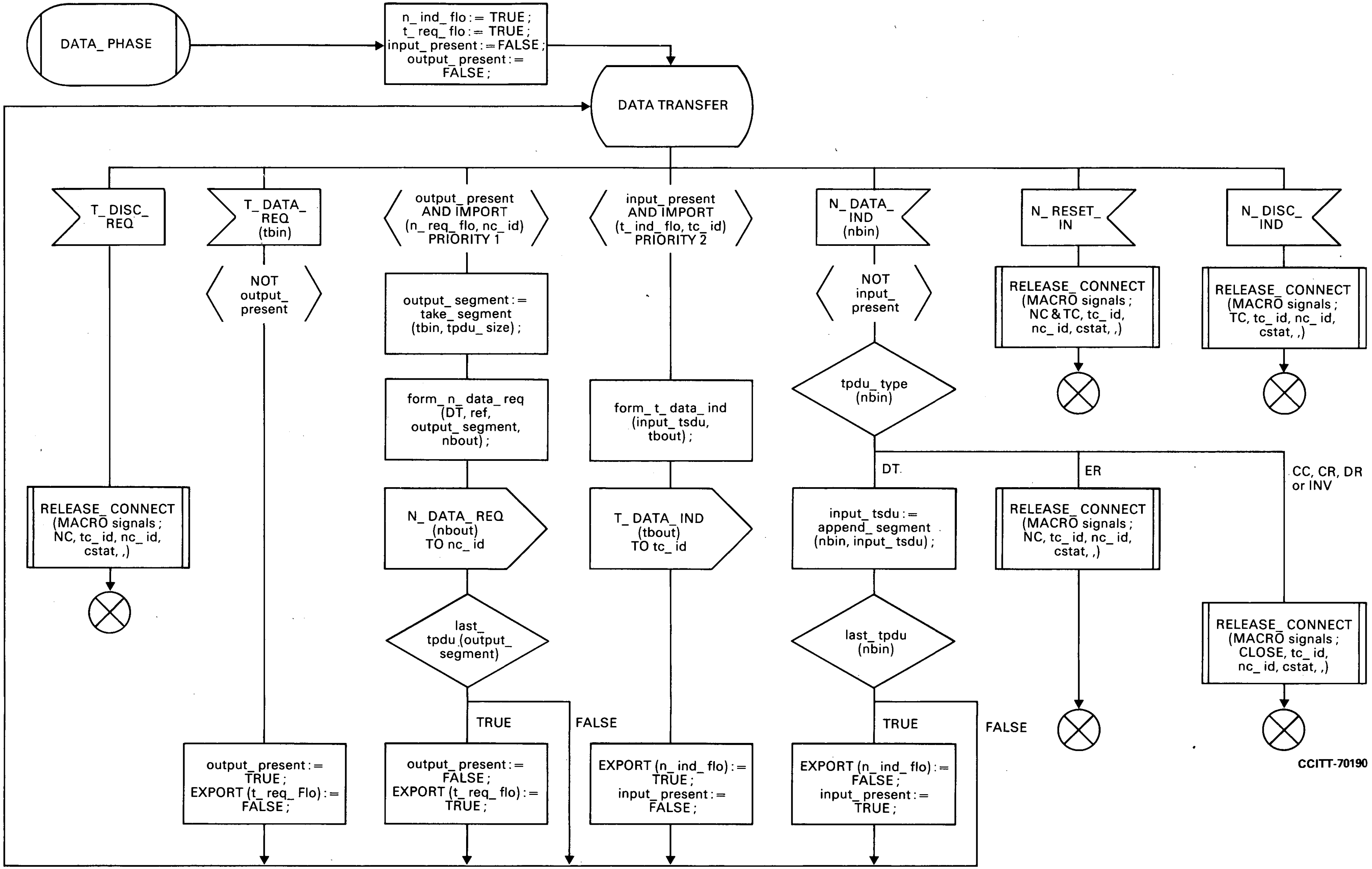

DATA_ PHASE
n_ ind_ flo := TRUE ;
t_ req_ flo := TRUE ;
input_ present := FALSE ;
output_ present := FALSE ;
DATA TRANSFER
T_ DISC_ REQ
T_ DATA_ REQ (tbin)
output_ present AND IMPORT (n_ req_ flo, nc_ id) PRIORITY 1
input_ present AND IMPORT (t_ ind_ flo, tc_ id) PRIORITY 2
N_ DATA_ IND (nbin)
N_ RESET_ IN
N_ DISC_ IND
NOT output_ present
output_ segment := take_ segment (tbin, tpdu_ size) ;
NOT input_ present
RELEASE_ CONNECT (MACRO signals ; NC & TC, tc_ id, nc_ id, cstat, ,)
RELEASE_ CONNECT (MACRO signals ; TC, tc_ id, nc_ id, cstat, ,)
tpdu_ type (nbin)
form_ n_ data_ req (DT, ref, output_ segment, nbout) ;
form_ t_ data_ ind (input_ tsdu, tbout) ;
DT
ER
CC, CR, DR or INV
RELEASE_ CONNECT (MACRO signals ; NC, tc_ id, nc_ id, cstat, ,)
N_ DATA_ REQ (nbout) TO nc_ id
T_ DATA_ IND (tbout) TO tc_ id
input_ tsdu := append_ segment (nbin, input_ tsdu) ;
RELEASE_ CONNECT (MACRO signals ; NC, tc_ id, nc_ id, cstat, ,)
last_ tpdu (output_ segment)
last_ tpdu (nbin)
RELEASE_ CONNECT (MACRO signals ; CLOSE, tc_ id, nc_ id, cstat, ,)
TRUE
FALSE
TRUE
FALSE
output_ present := TRUE ; EXPORT (t_ req_ Flo) := FALSE ;
output_ present := FALSE ; EXPORT (t_ req_ flo) := TRUE ;
EXPORT (n_ ind_ flo) := TRUE ; input_ present := FALSE ;
EXPORT (n_ ind_ flo) := FALSE ; input_ present := TRUE ;
CCITT-70190

FIGURE D-229

Procedure diagram for DATA_ PHASE

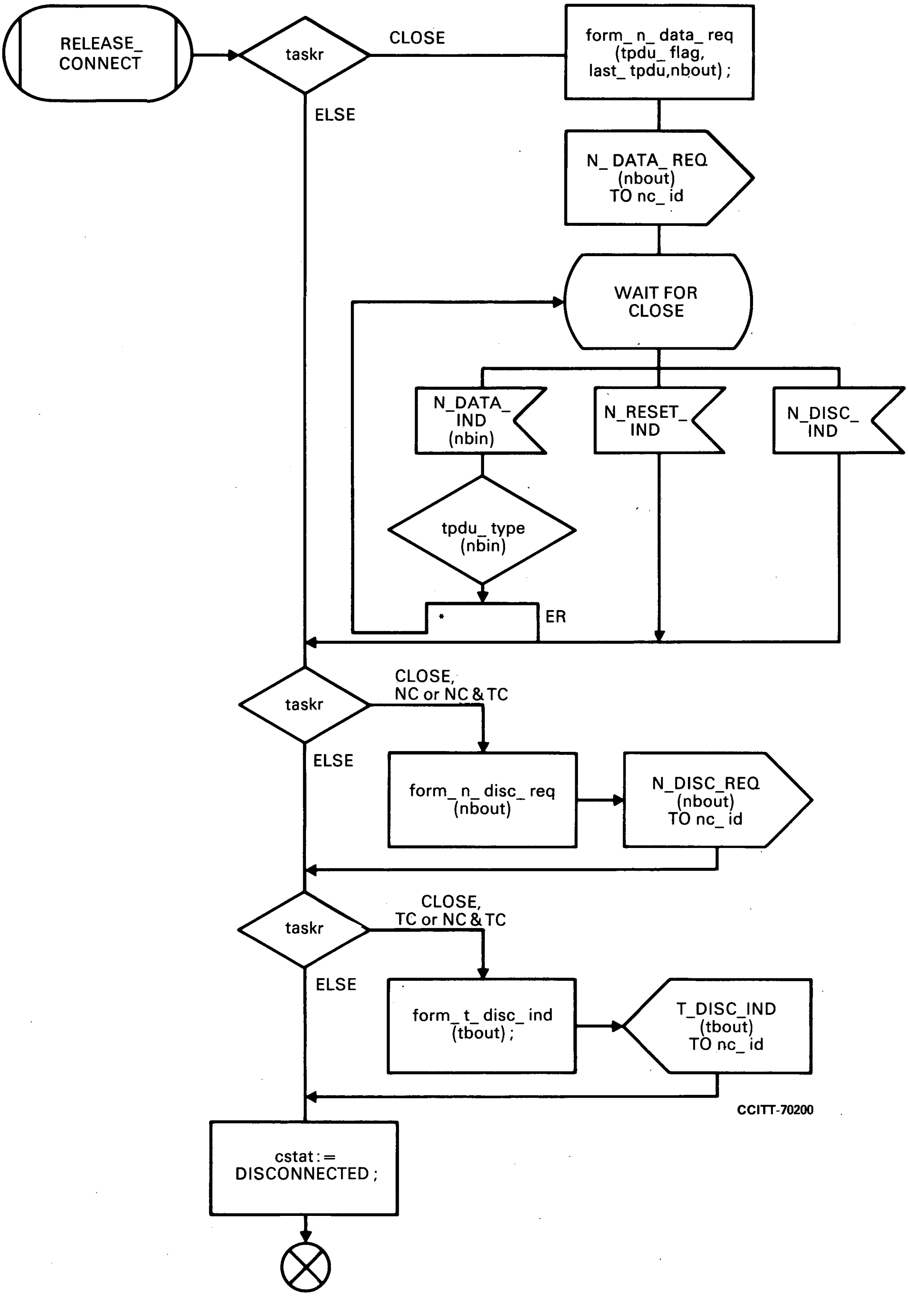


FIGURE D-230

Procedure diagram for RELEASE_ CONNECT

D.10 *Tools for SDL*

D.10.1 *Introduction*

This paragraph outlines a set of tools which may support SDL. These tools may support the generation of documents, SDL diagram (GR form) or listings (PR form), and/or the validation of SDL representations.

An exhaustive list of possible tools is not listed in these guidelines. The tools required are dependant on the methodology chosen by the user.

In principle SDL can be used without any tools. However, the inherent complexity of modern systems is such that the SDL representations are often complex. Therefore, automatic tools are needed to help manage the specification, design, and documentation of many systems. For instance, the complexity and cost of manually drawing, and possibly updating, the graphic documentation of a switching exchange would be significantly reduced by using suitable aids.

Due to the above considerations SDL has been designed so that tools can be effectively used to support its use.

D.10.2 *Classes of tools*

SDL tools may be classified by the activities carried out during the production of SDL documentation, for instance:

* Tools for input

According to the syntactic forms we have input aids for the SDL graphic, textual phrase, and pictorial forms.

* Tools for syntactic checking

These include syntax analyzers for each of the three syntaxes.

* Tools for document generation

Once SDL documents are stored within a machine, tools can access and reproduce them, possibly using several peripherals. These may use a syntactic form different from that used to input the document. In addition tools may be able to generate new types of documents derived from those originally entered.

* Tools for system modelling and analysis

The SDL documents representing a system can be used for abstracting a model of the system. Checks can be performed on that model. They can search for deadlocks, make a comparison between various models of the same system (e.g. between a specification and a description), or run a simulation of the system behaviour etc.

* Tools supporting code generation

Very detailed SDL representations may be used to help in the programming of software. Tools may be developed in such a way that a guided, semi-automatic, code sequence may be performed.

A specific, but useful, class of tools is represented by:

* SDL training tools

These may either stand alone or be integrated with other tools. Integration allows users of other functions to be given aid when required.

As SDL is used in many of the various phases of a systems life cycle, one can easily see a use of all types of tools in an integrated project environment.

D.10.3 *Document input*

No special requirements are needed for entering the textual phrase form of SDL, as the PR syntax is equivalent, from the input viewpoint, to any character string input. Therefore it may benefit by the same tools (string editors). The other two syntaxes however assume graphic handling capability.

It is obvious that whilst support for PR input can be beneficial, GR or PE input support is essential, if we plan to use these syntaxes as input media.

Input of GR/PE documents can be considered together as both syntaxes use graphics. What is needed is a graphic editor and a device to output diagrams in graphic form. A graphic input device is not required because it is possible to input a diagram over a predefined grid. Each symbol can be associated to a given string. For example it is possible to input a state by giving its position on a grid (couple of coordinates, square number etc.) and the type (state) as a character string.

A graphic input device can give immediate feedback to the user. However, the "grid method" would be faster and easier to implement.

A graphic editor is always required for functions such as the connection of two symbols, displacement of a set of symbols to another area of the page or to other pages, and concatenate deletion (deletion of a symbol implies the deletion of the connection to that symbol). As for the PR tools, the GR/PE input tools should be modelled on the SDL semantics/syntax. Therefore it should impede invalid connections, and prompt the user to fill up all of the uncompleted parts etc.

The tools are faced with several problems which derive from the physical constraints of graphics devices, such as "resolution". It is almost impossible to have a sufficient number of characters, which are readable, while also allowing the display of a reasonable number of symbols on the screen.

Solutions such as a zooming window, or scrolling, have to be considered, but they are not completely satisfactory. High resolution may not be considered a "must" if diagrams are copied by the user but it is very desirable if diagrams are produced directly by the user. For the same reason (requirement of an overview and of a certain amount of details) high resolution is desirable in the display of diagrams.

Tools for aiding the PR input can be helpful: they may prompt the user with expected PR keywords, pointing out that some closing keyword is still missing (such as enddecision, endstate etc.).

They can immediately format the PR according to the keywords received, insert delimiters automatically, and present to the user PR oriented function keys etc.

The implementation of such tools can be based on existing character string editors which can be extended to include the features mentioned above.

D.10.4 *Document verification*

Once documents are stored in a machine the next step is to verify them. They should first be verified individually, and then related diagrams should be combined and verified until the total system is verified.

If the input has been made through an SDL oriented tool a good amount of checking on each individual document could have already been done.

Errors deriving from "not possible" operations (e.g. inputs or saves following anything but a state) should all be detected and corrected during the input phase. The detection of some errors is however only possible at the completion of the input phase, both on a single document and, of course, in case of inconsistencies between documents.

Several SDL rules can be automatically checked. For instance the requirement that all outputs should have a corresponding input.

In the case of a multi-level representation, to a certain extent, consistency between levels can be checked.

The formal SDL model may be used to derive a collection of verification procedures.

D.10.5 *Document reproduction*

SDL documents stored in a machine must be able to be retrieved, displayed, and reproduced. Tools for all these activities are required. It may be useful to be able to retrieve only part, or subsets, of documents. The retrieval can be SDL oriented, e.g. "find all the processes sending" a given signal, or "in which states" is a certain action performed, and so on. Tools for displaying information are of particular interest when the information is to be displayed using the graphic syntax. The same observations as made for the input of documents in GR/PE syntaxes apply. The reproduction of documents is dependent on the type of document to be reproduced, on how this document has been stored and on the characteristics of the output peripheral. It may also be dependent on how it was entered. Users may wish for an output in a different syntax from the one used to enter the document.

The reproduction of documents is affected by the output peripheral constraints. For example a diagram can be too large to fit in a given space of paper, and therefore has to be chopped into pieces. Connectors have to be added and cross references should be inserted. It may be desirable to distinguish between an "addition" made by the tool and original input characteristics. Other physical constraints may impede the output of all the available information, e.g. a given symbol size is too small to contain all the associated text. Several strategies can be followed, perhaps chosen by the user. They include stretching the symbol, chopping the text, chopping it but adding full text as a footnote, putting the text beside the symbol... Tools allowing flexibility in output format are desirable: these features include different sizes of symbol, different output formats, vertical or horizontal display etc.

A document should always be able to be reproduced in exactly the same way as it was entered.

D.10.6 *Document generation*

Starting from the SDL documents entered by the users and stored in the machine, several other documents can be generated automatically, among these:

- the signal lists, organized per process, per block, or per system;
- the process interaction diagrams, showing the interactions and the following sequences of actions in communicating processes;
- the state overview diagrams, showing the process graph as a set of states connected by arcs representing the transitions;
- the cross reference table, organized per process, per block, or per system;
- the partitioning diagram, showing the structure of the blocks and levels;
- the system behaviour, as a response to sequences of environment actions;
- indexes: Once generated documents will have to be reproduced and the same considerations presented above are valid.

SDL documents entered in GR form can automatically be translated to the equivalent PR form and vice versa.

In order to produce a correct PR form all GR diagrams representing Processes of a Block have to be considered together with the associated GR Block Interaction Diagram.

The following considerations apply:

- the GR form contains visual information which cannot be translated into the PR form (it does not exist in PR). For example, the symbol coordinates are meaningless in the PR form;
- connectors linking flow lines on different pages can be eliminated.

The inverse translation, from PR to GR, is however much more complex and is unlikely to completely satisfy all the potential readers. Apart from identation, there is no subjective criteria to the elegance of a PR form, whilst a wide variety of subjective criteria exist for the GR form.

Due to the two dimensional representation of the GR form some labels inserted to cope with the sequential structure of the PR, can be deleted as a connection line is sufficient. From a PR form two different GR representations can be derived, namely the functional block interaction diagram and the process diagram (of course if the PR is representing a process alone, only the process diagram can be derived).

Usually the translation generates a model of the GR diagrams. This model contains all the information necessary for a tool to format and reproduce the diagram on a graphic device.

Note that two different tools translating some PR into GR may obtain two GR representations with different layouts. The GR representations so obtained are both correct provided they preserve the semantics expressed in the originating representation.

D.10.7 *System modelling and analysis*

SDL documents, independent of whether they specify or describe a system, are basically a model of that system.

This model, which is primarily intended to transfer information from one person to another, can also be interpreted by tools, checked for consistency, for completeness (this aspect may not be satisfied in cases of specification aiming at specifying only some parts of a system), and for correctness and compliance with the SDL rules (as described in the paragraph on document verification).

In addition, tools may be developed to use the model to simulate the systems functional behaviour. The simulator can interact with the environment and conclusions on the adherence of the model to the users expectations can be drawn.

If additional information is added to indicate the time consumed in executing each action, and to dimension the resources available (queues, instances etc.) the simulation can also study the capacity of the system.

Tools may be developed to create a model of the environment, starting from the system model, to create meaningful sequences to check the real system. Path analysis may detect deadlocks in the model.

The system model can also be used as on-line documentation. If appropriate links exist between the actual system and the documentation storage, a tool may be developed to trace system real time events on the model.

To achieve this, correlation should be provided between the physical events, as seen by the system, and the logical events dealt with by the SDL documentation. If the documentation is organized into several levels of abstraction the user can choose the level to be traced. This can be very useful as it allows users with different education and training to inspect the system activities.

Tools interpreting the SDL model can also be used to single out differences in behaviour of different models of the same system. It may also be used to compare different system descriptions (systems produced by different companies), or to compare the system specification with the description. In this way it is possible to see if a system description complies with the original specification.

D.10.8 *Code generation*

The provisions of a formally defined syntax and of a formal mathematical definition of SDL, make it possible to realize tools able to map the semantics of SDL representations to the semantics of programming languages. Such tools are possibly unable to provide complete implementation programs, but they may be very useful in providing at least a frame-work for the actual program.

Paragraph D.8.1 of this U.G. outlines an example of how the mapping between SDL and CHILL may be achieved.

D.10.9 *Training*

A complete training course on SDL has been developed; it consists of about two hundred pages of text and a collection of slides (about 200). The course covers all the aspects of the language providing at the same time examples and a few suggestions on the use of SDL.

The SDL Course is obtainable from the International Telecommunication Union, General Secretariat – Sales Section, Place des Nations, CH-1211 Genève 20 (Switzerland).

CCITT

SDL Symbols
Symboles LDS
Símbolos LED

Printed in Switzerland — ISBN 92-61-02241-3