INTERNATIONAL STANDARD

IEC 61499-1

First edition
2005-01

Function blocks –

Part 1: Architecture

Reference number
IEC 61499-1:2005(E)

Publication numbering

As from 1 January 1997 all IEC publications are issued with a designation in the 60000 series. For example, IEC 34-1 is now referred to as IEC 60034-1.

Consolidated editions

The IEC is now publishing consolidated versions of its publications. For example, edition numbers 1.0, 1.1 and 1.2 refer, respectively, to the base publication, the base publication incorporating amendment 1 and the base publication incorporating amendments 1 and 2.

Further information on IEC publications

The technical content of IEC publications is kept under constant review by the IEC, thus ensuring that the content reflects current technology. Information relating to this publication, including its validity, is available in the IEC Catalogue of publications (see below) in addition to new editions, amendments and corrigenda. Information on the subjects under consideration and work in progress undertaken by the technical committee which has prepared this publication, as well as the list of publications issued, is also available from the following:

- **IEC Web Site (www.iec.ch)**

- **Catalogue of IEC publications**

  The on-line catalogue on the IEC web site (www.iec.ch/searchpub) enables you to search by a variety of criteria including text searches, technical committees and date of publication. On-line information is also available on recently issued publications, withdrawn and replaced publications, as well as corrigenda.

- **IEC Just Published**

  This summary of recently issued publications (www.iec.ch/online_news/ justpub) is also available by email. Please contact the Customer Service Centre (see below) for further information.

- **Customer Service Centre**

  If you have any questions regarding this publication or need further assistance, please contact the Customer Service Centre:

  Email: custserv@iec.ch
  Tel: +41 22 919 02 11
  Fax: +41 22 919 03 00

INTERNATIONAL STANDARD

IEC 61499-1

First edition
2005-01

Function blocks –

Part 1: Architecture

International Electrotechnical Commission, 3, rue de Varembé, PO Box 131, CH-1211 Geneva 20, Switzerland
Telephone: +41 22 919 02 11 Telefax: +41 22 919 03 00 E-mail: inmail@iec.ch Web: www.iec.ch

Commission Electrotechnique Internationale
International Electrotechnical Commission
Международная Электротехническая Комиссия

PRICE CODE XE

For price, see current catalogue

CONTENTS

INTERNATIONAL ELECTROTECHNICAL COMMISSION

FUNCTION BLOCKS –

Part 1: Architecture

FOREWORD

1) The International Electrotechnical Commission (IEC) is a worldwide organization for standardization comprising all national electrotechnical committees (IEC National Committees). The object of IEC is to promote international co-operation on all questions concerning standardization in the electrical and electronic fields. To this end and in addition to other activities, IEC publishes International Standards, Technical Specifications, Technical Reports, Publicly Available Specifications (PAS) and Guides (hereafter referred to as "IEC Publication(s)"). Their preparation is entrusted to technical committees; any IEC National Committee interested in the subject dealt with may participate in this preparatory work. International, governmental and non-governmental organizations liaising with the IEC also participate in this preparation. IEC collaborates closely with the International Organization for Standardization (ISO) in accordance with conditions determined by agreement between the two organizations.
2) The formal decisions or agreements of IEC on technical matters express, as nearly as possible, an international consensus of opinion on the relevant subjects since each technical committee has representation from all interested IEC National Committees.
3) IEC Publications have the form of recommendations for international use and are accepted by IEC National Committees in that sense. While all reasonable efforts are made to ensure that the technical content of IEC Publications is accurate, IEC cannot be held responsible for the way in which they are used or for any misinterpretation by any end user.
4) In order to promote international uniformity, IEC National Committees undertake to apply IEC Publications transparently to the maximum extent possible in their national and regional publications. Any divergence between any IEC Publication and the corresponding national or regional publication shall be clearly indicated in the latter.
5) IEC provides no marking procedure to indicate its approval and cannot be rendered responsible for any equipment declared to be in conformity with an IEC Publication.
6) All users should ensure that they have the latest edition of this publication.
7) No liability shall attach to IEC or its directors, employees, servants or agents including individual experts and members of its technical committees and IEC National Committees for any personal injury, property damage or other damage of any nature whatsoever, whether direct or indirect, or for costs (including legal fees) and expenses arising out of the publication, use of, or reliance upon, this IEC Publication or any other IEC Publications.
8) Attention is drawn to the Normative references cited in this publication. Use of the referenced publications is indispensable for the correct application of this publication.
9) Attention is drawn to the possibility that some of the elements of this IEC Publication may be the subject of patent rights. IEC shall not be held responsible for identifying any or all such patent rights.

International Standard IEC 61499-1 has been prepared by IEC technical committee 65: Industrial-process measurement and control.

This standard cancels and replaces IEC/PAS 61499-1 published in 2000. This first edition constitutes a technical revision.

The following major technical changes have occurred between the PAS edition and this edition:

a) Syntax for network segments, links and parameters has been added in Clause B.3 to correspond to the system model of 4.1.
b) Syntax for parameters instead of constant data connections has been included for parameterization of function blocks, devices and resources in Clauses B.2 and B.3 for better consistency with IEC 61131-3.
c) The execution control model of 5.2.2.2 has been simplified and updated for consistency with modern models of state machine control.

The text of this standard is based on the following documents:

CDV	Report on voting
65/338/CDV	65/346/RVC

Full information on the voting for the approval of this standard can be found in the report on voting indicated in the above table.

This publication has been drafted in accordance with the ISO/IEC Directives, Part 2.

IEC 61499 consists of the following parts, under the general title *Function blocks:*

Part 1: Architecture

Part 2: Software tool requirements

Part 3: Tutorial information

Part 4: Rules for compliance profiles [1]

The committee has decided that the contents of this publication will remain unchanged until the maintenance result date indicated on the IEC web site under "http://webstore.iec.ch" in the data related to the specific publication. At this date, the publication will be

- reconfirmed;
- withdrawn;
- replaced by a revised edition, or
- amended.

A bilingual version of this standard may be issued at a later date.

[1] Under consideration.

INTRODUCTION

The IEC 61499 series consists of four Parts:

d) Part 1 (this part of IEC 61499) contains:

- general requirements, including scope, normative references, definitions, and reference models;
- rules for the declaration of *function block types*, and rules for the behavior of *instances* of the types so declared;
- rules for the use of function blocks in the *configuration* of distributed Industrial-Process Measurement and Control *Systems* (IPMCSs);
- rules for the use of function blocks in meeting the communication requirements of distributed IPMCSs;
- rules for the use of function blocks in the management of *applications*, *resources* and *devices* in distributed IPMCSs.

e) Part 2 defines requirements for *software tools* to support the following systems engineering tasks enumerated in Clause 1 of this part of IEC 61499:

- the specification of function block types;
- the functional specification of resource types and device types;
- the specification, analysis, and validation of distributed IPMCSs;
- the *configuration*, *implementation*, operation, and maintenance of distributed IPMCSs;
- the exchange of *information* among *software tools*.

f) Part 3 has the purpose of increasing the understanding, acceptance, and both generic and domain-specific applicability of IPMCS architectures and software tools meeting the requirements of the other Parts, by providing:

- answers to Frequently Asked Questions (FAQs) regarding the IEC 61499 series;
- examples of the use of IEC 61499 constructs to solve frequently encountered problems in control and automation engineering.

g) Part 4 defines rules for the development of *compliance profiles* which specify the features of IEC 61499-1 and IEC 61499-2 to be implemented in order to promote the following attributes of IEC 61499-based systems, devices and software tools:

- interoperability of devices from multiple suppliers;
- portability of software between software tools of multiple suppliers; and
- configurability of devices from multiple vendors by software tools of multiple suppliers.

FUNCTION BLOCKS –

Part 1: Architecture

1 Scope

This part of IEC 61499 defines a generic architecture and presents guidelines for the use of *function blocks* in distributed Industrial-Process Measurement and Control Systems (IPMCSs). This architecture is presented in terms of implementable reference *models*, textual syntax and graphical representations. These models, representations and syntax can be used for:

- the specification and standardization of *function block types*;
- the functional specification and standardization of system elements;
- the implementation independent specification, analysis, and validation of distributed IPMCSs;
- the *configuration*, *implementation*, operation, and maintenance of distributed IPMCSs;
- the exchange of *information* among *software tools* for the performance of the above *functions*.

NOTE 1 This part of IEC 61499 does not restrict or specify the functional capabilities of IPMCSs or their system elements, except as such capabilities are represented using the elements defined herein. IEC 61499-4 addresses the extent to which the elements defined in this part of IEC 61499 may be restricted by the functional capabilities of compliant systems, subsystems, and devices.

Part of the purpose of this part of IEC 61499 is to provide reference models for the use of function blocks in other standards dealing with the support of the system life cycle, including system planning, design, implementation, validation, operation and maintenance. The models given in this part of IEC 61499 are intended to be generic, domain independent and extensible to the definition and use of function blocks in other standards or for particular applications or application domains. It is intended that specifications written according to the rules given in this part of IEC 61499 be concise, implementable, complete, unambiguous, and consistent.

NOTE 2 The provisions of this part of IEC 61499 alone are not sufficient to ensure interoperability among devices of different vendors. Standards complying with this part of IEC 61499 may specify additional provisions to ensure such interoperability.

NOTE 3 Standards complying with this part of IEC 61499 may specify additional provisions to enable the performance of *system*, *device*, *resource* and *application* management *functions*.

2 Normative references

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

IEC 60050-351:1998, *International Electrotechnical Vocabulary (IEV) – Part 351: Automatic Control*

IEC 61131-3:2003, *Programmable controllers – Part 3: Programming languages*

ISO/IEC 7498-1:1994, *Information Technology – Open Systems Interconnection – Basic Reference Model: The Basic Model*

ISO/IEC 8824-1, *Information technology - Abstract Syntax Notation One (ASN.1): Specification of basic notation)*

ISO/IEC 8825-1, *Information technology - ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)*

ISO/IEC 10646, *Information technology - Universal Multiple-Octet Coded Character Set (UCS)*

ISO/IEC 10731, *Information technology - Open Systems Interconnection - Basic Reference Model - Conventions for the definition of OSI services*

ISO/AFNOR, *Dictionary of Computer Science*, 1989, ISBN 2-12-4869111-6

3 Terms and definitions

For the purposes of this document, the terms and definitions given in the ISO/AFNOR *Dictionary of computer science*, as well as the following apply.

NOTE Terms defined in this clause are *italicized* where they appear in the bodies of definitions.

3.1
acceptor
function block instance which provides a *socket adapter* of a defined *adapter interface type*

3.2
access path
association of a symbolic name with a *variable* for the purpose of open communication

3.3
adapter connection
connection from a *plug adapter* to a *socket adapter* of the same *adapter interface type*, which carries the flows of *data* and *events* defined by the adapter interface type

3.4
adapter interface type
type which consists of the definition of a set of *event inputs, event outputs, data inputs*, and *data outputs*, and whose *instances* are *plug adapters* and *socket adapters*

3.5
algorithm
finite set of well-defined rules for the solution of a problem in a finite number of *operations*

3.6
application
software functional unit that is specific to the solution of a problem in industrial-process measurement and control

NOTE An application may be distributed among *resources*, and may communicate with other applications.

3.7
attribute
property or characteristic of an *entity*, for instance, the version identifier of a *function block type* specification

3.8
basic function block type
function block type that cannot be decomposed into other function blocks and that utilizes an *execution control chart (ECC)* to control the *execution* of its *algorithms*

3.9
bidirectional transaction
transaction in which a request and possibly *data* are conveyed from an *requester* to a *responder*, and in which a response and possibly data are conveyed from the responder back to the requester

3.10
character
member of a set of elements that is used for the representation, organization, or control of *data*

[ISO/AFNOR: 1989]

3.11
communication connection
connection that utilizes the "communication mapping function" of one or more *resources* for the conveyance of *information*

3.12
communication function block
service interface function block that represents the *interface* between an *application* and the "communication mapping function" of a *resource*

3.13
communication function block type
function block type whose *instances* are *communication function blocks*

3.14
component function block
function block instance which is used in the specification of an *algorithm* of a *composite function block type*

NOTE A component function block can be of *basic*, *composite* or *service interface type*.

3.15
component subapplication
subapplication instance that is used in the specification of a *subapplication type*

3.16
composite function block type
function block type whose *algorithms* and the control of their *execution* are expressed entirely in terms of interconnected *component function blocks, events,* and *variables*

3.17
concurrent
pertaining to *algorithms* that are *executed* during a common period of time during which they may have to alternately share common *resources*

3.18
configuration (of a *system* or *device*)
selecting *functional units*, assigning their locations and defining their interconnections

3.19
configuration (of a programmable controller system)
language element corresponding to a *programmable controller system* as defined in IEC 61131-1

3.20
configuration parameter
parameter related to the *configuration* of a *system, device* or *resource*

3.21
confirm primitive
service primitive which represents an interaction in which a *resource* indicates completion of some *algorithm* previously *invoked* by an interaction represented by a *request primitive*

3.22
connection
association established between *functional units* for conveying *information*

[ISO/AFNOR: 1989]

3.23
critical region
operation or sequence of operations which is *executed* under the exclusive control of a locking object which is associated with the *data* on which the operations are performed

3.24
data
reinterpretable representation of *information* in a formalized manner suitable for communication, interpretation or processing

[ISO 2382-01.01.02]

3.25
data connection
association between two *function blocks* for the conveyance of *data*

3.26
data input
interface of a *function block* which receives *data* from a *data connection*

3.27
data output
interface of a *function block* which supplies *data* to a *data connection*

3.28
data type
set of values together with a set of permitted *operations*

[ISO 2382-15.04.01]

3.29
declaration
mechanism for establishing the definition of an *entity*

NOTE A declaration may involve attaching an *identifier* to the entity, and allocating *attributes* such as *data types* and *algorithms* to it.

3.30
device
independent physical *entity* capable of performing one or more specified *functions* in a particular context and delimited by its *interfaces*

NOTE A *programmable controller system* as defined in IEC 61131-1 is a *device*.

3.31
device management application
application whose primary function is the management of a multiple *resources* within a *device*

3.32
entity
particular thing, such as a person, place, *process*, object, concept, association, or *event*

3.33
event
instantaneous occurrence that is significant to scheduling the *execution* of an *algorithm*

NOTE The execution of an algorithm may make use of *variables* associated with an event.

3.34
event connection
association among *function blocks* for the conveyance of *events*

3.35
event input
interface of a *function block* which can receive *events* from an *event connection*

3.36
event output
interface of a *function block* which can issue *events* to an *event connection*

3.37
exception
event that causes suspension of normal *execution*

3.38
execution
process of carrying out a sequence of *operations* specified by an *algorithm*

NOTE The sequence of operations to be executed may vary from one *invocation* of a *function block instance* to another, depending on the rules specified by the function block's *algorithm* and the current values of *variables* in the function block's data structure.

3.39
Execution Control action
EC action
element associated with an *execution control state,* which identifies an *algorithm* to be *executed* and an *event* to be issued on completion of execution of the algorithm

3.40
Execution Control Chart
ECC
graphical or textual representation of the causal relationships among *events* at the *event inputs* and *event outputs* of a *function block* and the *execution* of the function block's *algorithms*, using *execution control states, execution control transitions,* and *execution control actions*

3.41
Execution Control initial state
EC initial state
execution control state that is active upon initialization of an *execution control chart*

3.42
Execution Control state
EC state
situation in which the behavior of a *basic function block* with respect to its *variables* is determined by the *algorithms* associated with a specified set of *execution control actions*

3.43
Execution Control transition
EC transition
means by which control passes from a predecessor *execution control state* to a successor *execution control state*

3.44
fault
abnormal condition that may cause a reduction in, or loss of, the capability of a *functional unit* to perform a required *function*

[IEC 61508-4:1998, 3.6.1]

3.45
function
specific purpose of an *entity* or its characteristic action

3.46
function block
function block instance
software functional unit comprising an individual, named copy of a data structure upon which associated *operations* may be performed as specified by a corresponding *function block type*

NOTE 1 Typical operations of a function block include modification of the values of the data in its associated data structure.

NOTE 2 The *function block instance* and its corresponding *function block type* defined in IEC 61131-3 are programming language elements with a different set of features.

3.47
function block network
network whose nodes are *function blocks* or *subapplications* and their *parameters* and whose branches are *data connections* and *event connections*

NOTE This is a generalization of the *function block diagram* defined in IEC 61131-3.

3.48
function block type
type whose *instances* are *function blocks*

NOTE Function block types include basic function block types, composite function block types, and service interface function block types

3.49
functional unit
entity of *hardware* or *software,* or both, capable of accomplishing a specified purpose

[ISO 2382-01.01.40]

3.50
hardware
physical equipment, as opposed to programs, procedures, rules and associated documentation

[ISO/AFNOR: 1989]

3.51
identifier
one or more *characters* used to name an *entity*

3.52
implementation
development phase in which the *hardware* and *software* of a *system* become operational

3.53
indication primitive
service primitive which represents an interaction in which a *resource* either: a) indicates that it has, on its own initiative, *invoked* some *algorithm*; or b) indicates that an *algorithm* has been invoked by a peer *application*

3.54
information
meaning that is currently assigned to *data* by means of the conventions applied to that data

[ISO/AFNOR: 1989]

3.55
input variable
variable whose value is supplied by a *data input*, and which may be used in one or more *operations* of a *function block*

NOTE An *input parameter* of a *function block*, as defined in IEC 61131-3, is an *input variable*.

3.56
instance
functional unit comprising an individual, named *entity* with the *attributes* of a defined *type*

3.57
instance name
identifier associated with and designating an *instance*

3.58
instantiation
creation of an *instance* of a specified *type*

3.59
interface
shared boundary between two *functional units,* defined by functional characteristics, signal characteristics, or other characteristics as appropriate

[IEV 351-11-19]

3.60
internal operation (of a *function block)*
o*peration* associated with an *algorithm* of a function block, with its *execution* control, or with the functional capabilities of the associated *resource*

3.61
internal variable
variable whose value is used or modified by one or more *operations* of a *function block* but is not supplied by a *data input* or to a *data output*

3.62
invocation
process of initiating the *execution* of the sequence of *operations* specified in an *algorithm*

3.63
link
design element describing the *connection* between a *device* and a *network segment*

3.64
literal
lexical unit that directly represents a value

3.65
management function block
function block whose primary *function* is the management of *applications* within a *resource*

3.66
management resource
resource whose primary *function* is the management of other *resources*

3.67
mapping
set of features or *attributes* having defined correspondence with the members of another set

3.68
message
ordered series of *characters* intended to convey *information*

[ISO 2382-16.02.01]

3.69
message sink
part of a communication *system* in which *messages* are considered to be received

[ISO 2382-16.02.03]

3.70
message source
part of a communication *system* from which *messages* are considered to originate

[ISO 2382-16.02.02]

3.71
model
representation of a real world process, *device*, or concept

3.72
multitasking
mode of operation that provides for the *concurrent execution* of two or more *algorithms*

3.73
network
arrangement of nodes and interconnecting branches

[ISO 2382-01.01.44]

3.74
operation
well-defined action that, when applied to any permissible combination of known *entities*, produces a new *entity*

[ISO 2382-02.10.01]

3.75
output variable
variable whose value is established by one or more *operations* of a *function block,* and is supplied to a *data output*

NOTE An *output parameter* of a *function block*, as defined in IEC 61131-3, is an *output variable*.

3.76
parameter
variable that is given a constant value for a specified *application* and that may denote the application

[ISO 2382-02.02.04]

3.77
plug
plug adapter
instance of an *adapter interface type* which provides a starting point for an *adapter connection* from a *provider* function block

3.78
provider
function block instance which provides a *plug adapter* of a defined *adapter interface type*

3.79
request primitive
service primitive which represents an interaction in which an *application* invokes some *algorithm* provided by a *service*

3.80
requester
functional unit which initiates a *transaction* via a *request primitive*

3.81
resource
functional unit which has independent control of its operation, and which provides various *services* to *applications,* including the scheduling and *execution* of *algorithms*

NOTE 1 The RESOURCE defined in IEC 61131-3 is a programming language element corresponding to the *resource* defined above.

NOTE 2 A *device* contains one or more resources.

3.82
resource management application
application whose primary function is the management of a single *resource*

3.83
responder
functional unit which concludes a transaction via a response primitive

3.84
response primitive
service primitive which represents an interaction in which an *application* indicates that it has completed some *algorithm* previously *invoked* by an interaction represented by an *indication primitive*

3.85
sample
sense and retain the instantaneous value of a *variable* for later use

3.86
scheduling function
function which selects *algorithms* or *operations* for *execution,* and initiates and terminates this execution

3.87
segment
physical partition of a *communication network*

3.88
service
functional capability of a *resource* which can be modeled by a sequence of *service primitives*

3.89
service interface function block
function block which provides one or more *services* to an *application*, based on a *mapping* of *service primitives* to the function block's *event inputs, event outputs, data inputs* and *data outputs*

3.90
service primitive
abstract, implementation-independent representation of an interaction between an *application* and a *resource*

3.91
socket
socket adapter
instance of an *adapter interface type* which provides an end point for an *adapter connection* to an *acceptor* function block

3.92
software
intellectual creation comprising the programs, procedures, rules, *configurations* and any associated documentation pertaining to the operation of a *system*

3.93
software tool
software that is used for the production, inspection or analysis of other software

3.94
subapplication instance
instance of a *subapplication type* inside an *application* or inside a subapplication type

NOTE A subapplication instance may be distributed among *resources*, i.e. its component function blocks or the content of its component subapplications may be assigned to different resources.

3.95
subapplication type
functional unit whose body consists of interconnected *component function blocks* or *component subapplications*

NOTE A subapplication type enables the creation of substructures of *applications* in the form of a self-similar hierarchy.

3.96
system
set of interrelated elements considered in a defined context as a whole and separated from its environment

[IEV-351-11-01]

NOTE 1 Such elements may be both material objects and concepts as well as the results thereof (for example forms of organisation, mathematical methods, and programming languages)

NOTE 2 The system is considered to be separated from the environment and other external systems by an imaginary surface, which can cut the links between them and the considered system.

3.97
transaction
unit of service in which a request and possibly *data* is conveyed from a *requester* to a *responder*, and in which a response and possibly data may also be conveyed from the responder back to the requester

3.98
type
software element which specifies the common *attributes* shared by all *instances* of the type

3.99
type name
identifier associated with and designating a *type*

3.100
unidirectional transaction
transaction in which a request and possibly *data* is conveyed from an *requester* to a *responder*, and in which a response is not conveyed from the responder back to the requester

3.101
variable
software entity that may take different values, one at a time

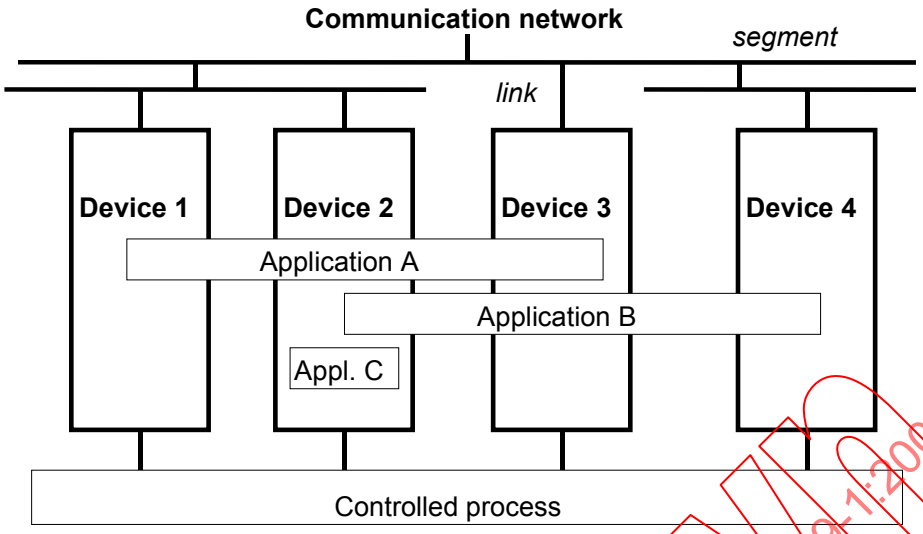
NOTE 1 The values of a variable are usually restricted to a certain *data type*.

NOTE 2 Variables may be classified as *input variables, output variables*, and *internal variables*.

4 Reference models

4.1 System model

For the purposes of this specification, an Industrial Process Measurement and Control *System* (IPMCS) is modeled, as shown in Figure 1, as a collection of *devices* interconnected and communicating with each other by means of a communication *network* consisting of *segments* and *links*. Devices are connected to network segments via links.



NOTE The controlled process is not part of the measurement and control system.

Figure 1 – System model

A *function* performed by the IPMCS is modeled as an *application* which may reside in a single device, such as application C in Figure 1, or may be distributed among several devices, such as applications A and B in Figure 1. For instance, an application may consist of one or more control loops in which the input sampling is performed in one device, control processing is performed in another, and output conversion in a third.

4.2 Device model

As illustrated in Figure 2, a *device* shall contain at least one *interface,* that is, process interface or communication interface, and can contain zero or more *resources*.

NOTE 1 A device is considered to be an *instance* of a corresponding device *type*, defined as specified in Clause 7.

NOTE 2 A device that contains no resources is considered to be functionally equivalent to a *resource* as defined in 4.3.

A "process interface" provides *a mapping* between the physical process (analog measurements, discrete I/O, etc.) and the resources. Information exchanged with the physical process is presented to the resource as *data* or *events*, or both.

Communication *interfaces* provide a mapping between resources and the information exchanged via a communication *network*. Services provided by communication interfaces may include:

- Presentation of communicated information to the resource as *data* or *events*, or both;
- Additional services to support programming, *configuration*, diagnostics, etc.

NOTE 3 Communication *links* may either be associated directly with a *device*, or with an instance of a specific *resource* type ("communication resource"), onto which part of the distributed application may or may not be mapped, depending on the resource type.

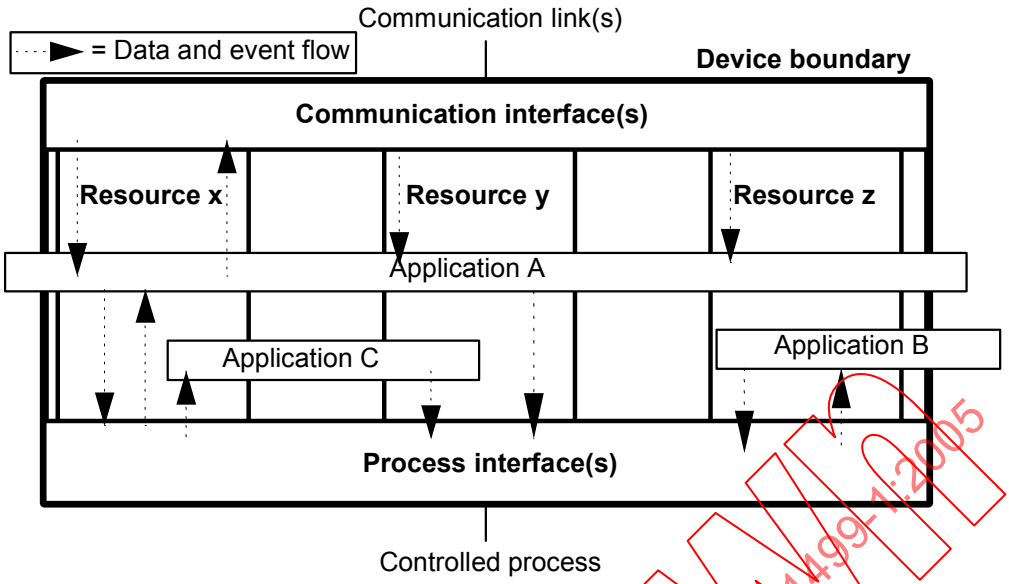


Figure 2 – Device model
(example: Device 2 from Figure 1)

4.3 Resource model

For the purposes of this part of IEC 61499, a *resource* is considered to be a *functional unit*, contained in a *device* which has independent control of its operation. It may be created, configured, parameterized, started up, deleted, etc., without affecting other resources within a device.

NOTE 1 A resource is considered to be an *instance* of a corresponding resource *type*, defined as specified in Clause 7.

NOTE 2 Although a resource has independent control of its operation, its operational states may need to be coordinated with those of other resources for the purposes of installation, test, etc.

The *functions* of a resource are to accept *data* and/or *events* from the process and/or communication *interfaces*, process the data and/or events, and to return data and/or events to the process and/or communication interfaces, as specified by the *applications* utilizing the resource.

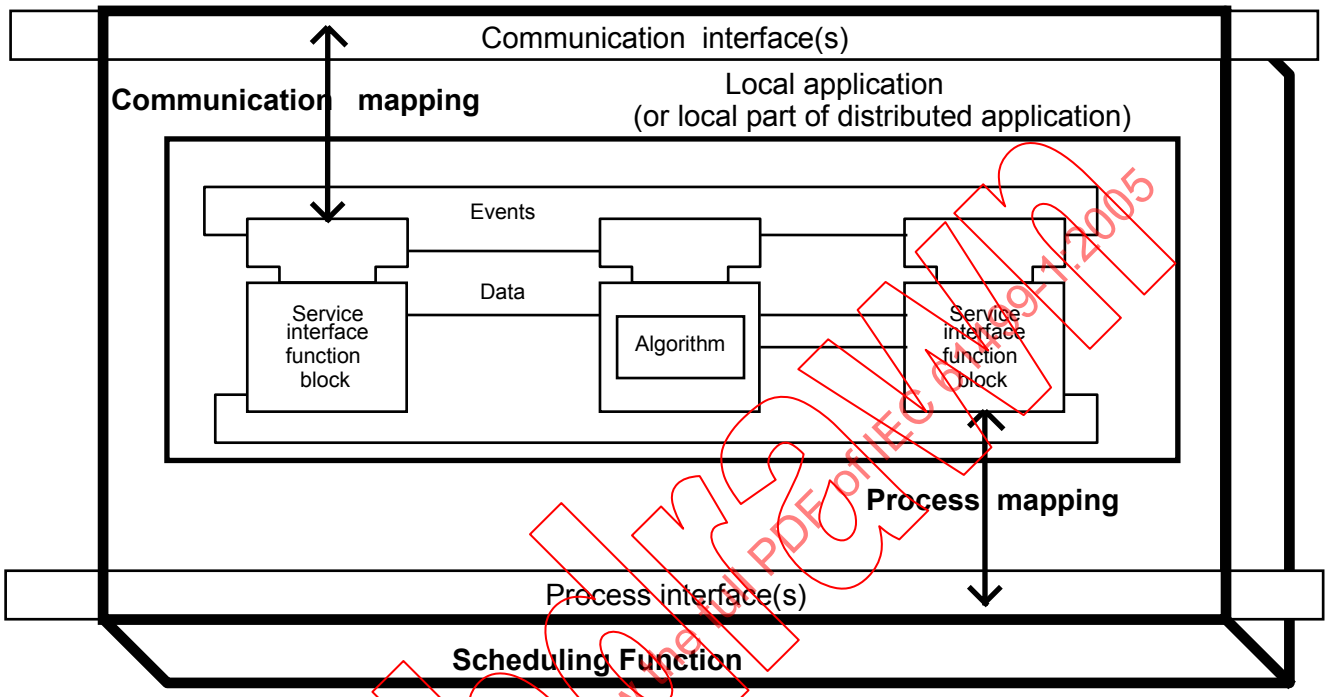
NOTE 3 Besides supporting the functions enumerated above, specific types of resources may represent the capability to implement interface functions such as process interfaces or lower layer communication services over communication links. Depending on the type of those resources, these services may or may not be the only ones they are able to provide.

NOTE 4 The consideration of other possible aspects of resources is beyond the scope of this part of IEC 61499.

As illustrated in Figure 3, a resource is modeled by the following:

- One or more "local applications" (or local parts of distributed applications). The *variables* and *events* handled in this part are *input* and *output variables* and events at *event inputs* and *event outputs* of *function blocks* that perform the *operations* needed by the application.
- A "process mapping" part whose function is to perform a *mapping* of *data* and *events* between *applications* and process *interface(s)*. As shown in Figure 3, this mapping may be modeled by *service interface function blocks* specialized for this purpose.
- A "communication mapping" part whose function is to perform a *mapping* of *data* and *events* between *applications* and *communication interfaces*. As shown in Figure 3, this mapping may be modeled by *service interface function blocks* specialized for this purpose.

- A scheduling *function* which effects the execution of, and data transfer between, the function blocks in the applications, according to the timing and sequence requirements determined by: 1) the occurrence of events; 2) function block interconnections; and 3) scheduling information such as periods and priorities. Means of achieving traditional scheduling functions, such as cyclic execution of a *function block network* are described in IEC 61499-3, Clause 3.



NOTE 1 This figure is illustrative only. Neither the graphical representation nor the location of function blocks is normative.

NOTE 2 Communication and process interfaces may be shared among resources.

Figure 3 – Resource model

4.4 Application model

For the purposes of this part of IEC 61499, an *application* consists of a *function block network,* whose nodes are *function blocks* or *subapplications* and their *parameters* and whose branches are *data connections* and *event connections*.

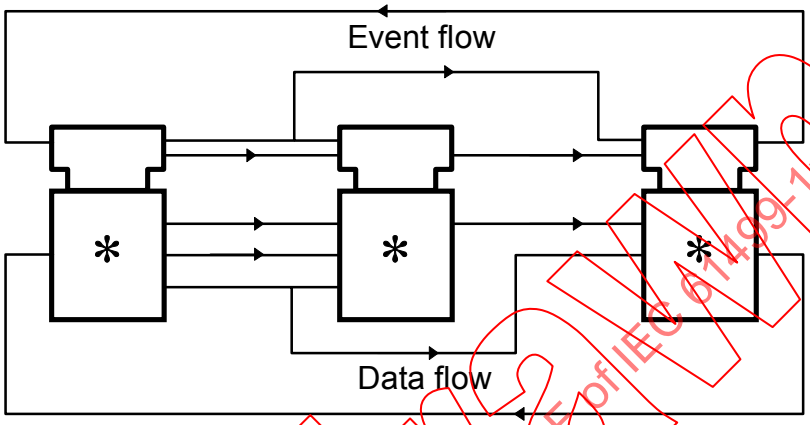
Subapplications are *instances* of *subapplication types*, which like applications consist of *function block networks*. Application names, subapplication and function block *instance names* may therefore be used to create a hierarchy of *identifiers* that can uniquely identify every *function block instance* in a *system*.

An application can be distributed among several *resources* in the same or different *devices*. A *resource* uses the causal relationships specified by the application to determine the appropriate responses to *events* which may arise from communication and process interfaces or from other functions of the resource. These responses may include:

- Scheduling and *execution* of *algorithms*
- Modification of *variables*
- Generation of additional events
- Interactions with communication and process interfaces

In the context of this part of IEC 61499, applications are defined by *function block networks* specifying event and data flow among *function block* or *subapplication instances,* as illustrated in Figure 4. The event flow determines the scheduling and *execution* by the associated resource of the *operations* specified by each function block's *algorithm(s),* according to the rules given in Clause 5.

Standards, components and systems complying with this part of IEC 61499 may utilize alternative means for scheduling of execution. Such alternative means shall be exactly specified using the elements defined in this part of IEC 61499.



NOTE 1 "*" represents function block or subapplication instances.

NOTE 2 This figure is illustrative only. The graphical representation is not normative.

Figure 4 – Application model

4.5 Function block model

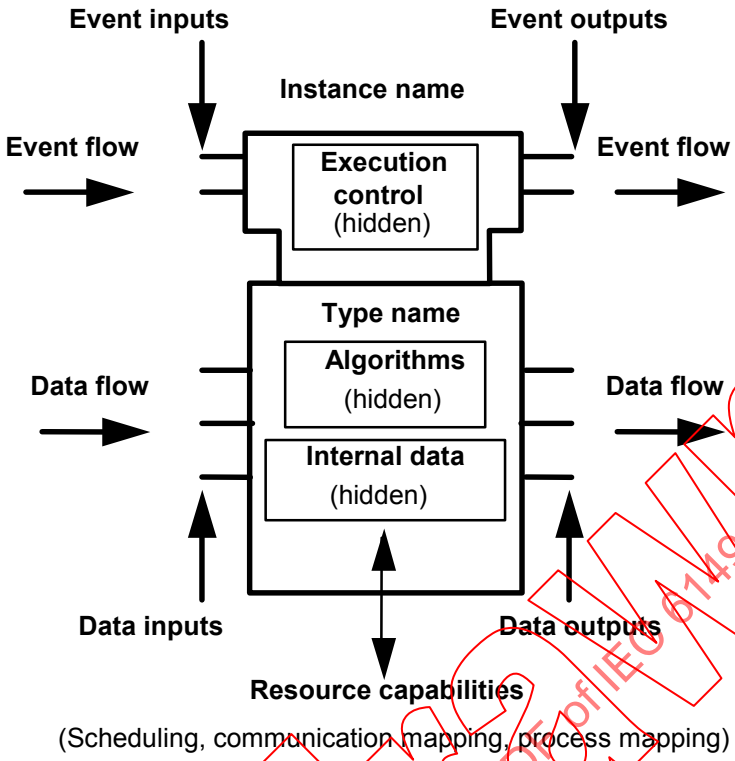
4.5.1 Characteristics of function block instances

A *function block (function block instance)* is a *functional unit* of *software* comprising an individual, named copy of the data structure specified by a *function block type*, which persists from one *invocation* of the function block to the next. The characteristics of function block instances are described in this subclause, and function block type specifications are described in 4.5.2.

A *function block instance* exhibits the following characteristic features as illustrated in Figure 5:

- its *type name* and *instance name*;
- a set of *event inputs,* each of which can receive *events* from an *event connection* which may affect the execution of one or more *algorithms*;
- a set of *event outputs,* each of which can issue *events* to an *event connection* depending on the execution of *algorithms* or on some other functional capability of the *resource* in which the function block is located;
- a set of *data inputs,* which may be *mapped* to corresponding *input variables*;
- a set of *data outputs,* which may be *mapped* to corresponding *output variables*;
- internal *data,* which may be *mapped* to a set of *internal variables*;
- functional characteristics which are determined by combining internal data or state information, or both, with a set of *algorithms*, functional capabilities of the associated *resource,* or both. These functional characteristics are defined in the function block's *type* specification.

NOTE 1 Internal state information may be represented by *internal variables* or by an internal representation of an execution control state machine.



NOTE This figure is illustrative only. The graphical representation is not normative.

Figure 5 – Characteristics of function blocks

The algorithms contained within a function block are in principle invisible from the outside of the function block, except as described formally or informally by the provider of the function block. Additionally, the function block may contain internal *variables* or state information, or both, which persist between invocations of the function block's algorithms, but which are not accessible by data flow connections from the outside of the function block.

NOTE 2 Access to internal variables and state information of function block instances may be provided by additional functional capabilities of the associated resource, as illustrated in 6.3.

Means for specifying the causal relationships among event inputs, event outputs, and execution of algorithms are defined in Clauses 5 and 6.

4.5.2 Function block type specifications

A *function block type* is a *software* element which specifies the characteristics of all *instances* of the type, including:

- Its *type name*.
- The number, names, type names and order of *event inputs* and *event outputs.*
- The number, names, *data type* and order of input, output and internal *variables*.

Mechanisms for the *declaration* of these characteristics are defined in 5.2.1.

In addition, the function block type specification defines the functionality of *instances* of the type. This functionality may be expressed as follows:

- For *basic function block types,* declaration mechanisms are provided in 5.2.1.3 for the specification of *algorithms*, which operate on the values of *input variables, output variables,* and *internal variables* to produce new values of *output variables* and *internal variables.* The associations among the *invocation* of algorithms and the occurrence of *events* at event inputs and outputs are expressed in terms of an *execution control chart* (ECC), using the declaration mechanisms defined in 5.2.1.4.
- The functionality of an *instance* of a *composite function block type* or a *subapplication type* is declared, using the mechanisms defined in 5.3.1 and 5.4.1 respectively, in terms of *data connections* and *event connections* among its *component function blocks* or subapplications and the event and data inputs and outputs of the composite function block or the subapplication.
- The functionality of an instance of a *service interface function block type* is described by a *mapping* of *service primitives* to *event inputs, event outputs, data inputs* and *data outputs*, using the declaration mechanisms defined in 6.1.
- Other means such as natural language text may be used for describing the functionality of a function block type; however, the specification of such means is beyond the scope of this part of IEC 61499.

4.5.3 Execution model for basic function blocks

As shown in Figure 6, the *execution* of *algorithms* for *basic function blocks* is invoked by the **execution control** portion of a *function block instance* in response to events at event inputs. This *invocation* takes the form of a request to the **scheduling function** of the associated *resource* to schedule the execution of the algorithm's *operations*. Upon completion of execution of an algorithm, the execution control generates zero or more events at *event outputs* as appropriate.

NOTE 1 *Events at event inputs* are provided by connection to *event outputs* of other function block instances or the same function block instance. Events at these event outputs may be generated by execution control as described above, or by the "communication mapping", "process mapping", "scheduling", or other functional capability of the *resource*.

NOTE 2 Execution control in composite function blocks is achieved via event flow within the function block body.

Figure 6 depicts the order of events and algorithm execution for the case in which a single event input, a single algorithm, and a single event output are associated. The relevant times in this diagram are defined as follows:

t1: Relevant input variable values (i.e., those associated with the event input by the WITH qualifier defined in 5.2.1.2) are made available.

t2: The event at the event input occurs.

t3: The execution control function notifies the resource scheduling function to schedule an algorithm for execution.

t4: Algorithm execution begins.

t5: The algorithm completes the establishment of values for the output variables associated with the event output by the WITH qualifier defined in 5.2.1.2.

t6: The resource scheduling function is notified that algorithm execution has ended.

t7: The scheduling function invokes the execution control function.

t8: The execution control function signals an event at the event output.

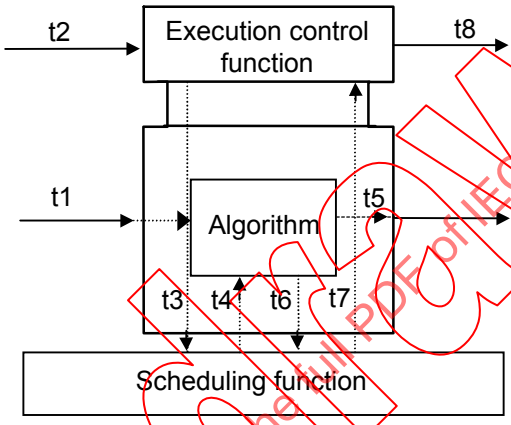
As shown in Figure 7, the significant timing delays in this case which are of interest in application design are:

T_{setup} = t2 – t1.

T_{start} = t4 – t2 (time from event at event input to beginning of algorithm execution).

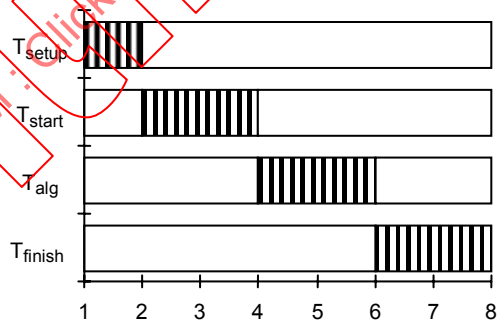
T_{alg} = t6 – t4 (algorithm execution time).

T_{finish} = t8 – t6 (time from end of algorithm execution to event at event output).



NOTE This figure is illustrative only. The graphical representation is not normative.

Figure 6 – Execution model



NOTE The axis labels 1, 2, etc. in the above figure correspond to the times t1, t2, etc. in Figure 6.

Figure 7 – Execution timing

Normative requirements for the specification of function block execution control in the general case (which includes the above case) are defined in Clause 5.

NOTE 3 Depending on the problem to be solved, various requirements may exist for the synchronization of the values of *input variables* with the *execution* of *algorithms*. Such requirements may include, for example:

– Assurance that the values of variables used by an algorithm remain stable during the execution of the algorithm.

– Assurance that the values of variables used by an algorithm correspond to the data present upon the occurrence of the event at the event input which caused the scheduling of the algorithm for execution.

– Assurance that the values of variables used by all algorithms scheduled for execution in a function block correspond to the data present upon the occurrence of the event at the event input which caused the scheduling of the first such algorithm for execution.

Users of this part of IEC 61499 should be aware that the results of algorithm execution may be unpredictable if such requirements are not met.

NOTE 4 *Resources* may need to schedule the *execution* of *algorithms* in a *multitasking* manner. The specification of attributes to facilitate such scheduling is described in Annex G.

4.6 Distribution model

As illustrated in Figure 8a), an *application* or *subapplication* can be distributed by allocating its *function block instances* to different *resources* in one or more *devices*. Since the internal details of a function block are hidden from any application or subapplication utilizing it, a function block shall form an atomic unit of distribution. That is, all the elements contained in a given function block instance shall be contained within the same resource.

The functional relationships among the function blocks of an application or subapplication shall not be affected by its distribution. However, in contrast to an application or subapplication confined to a single resource, the timing and reliability of communications functions will affect the timing and reliability of a distributed application or subapplication.

The following clauses apply when applications or subapplications are distributed among multiple resources:

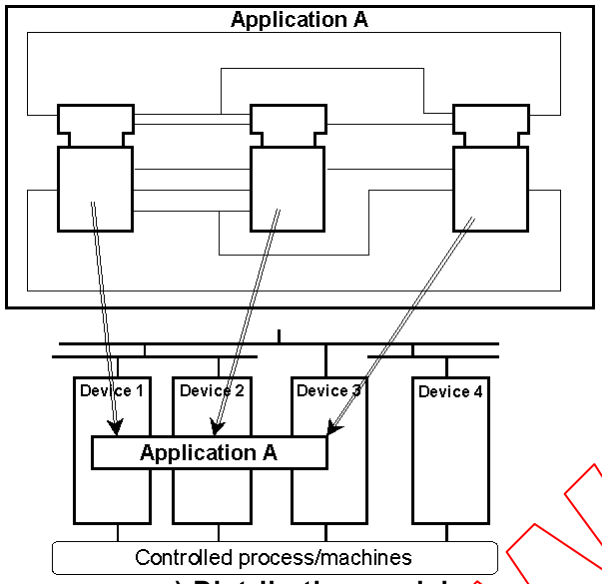
- Clause 6 defines the requirements for communication services to support distribution of applications or subapplications among multiple devices.
- Clause 7 defines the requirements for the case where multiple applications or subapplications are distributed among multiple resources and devices.

4.7 Management model

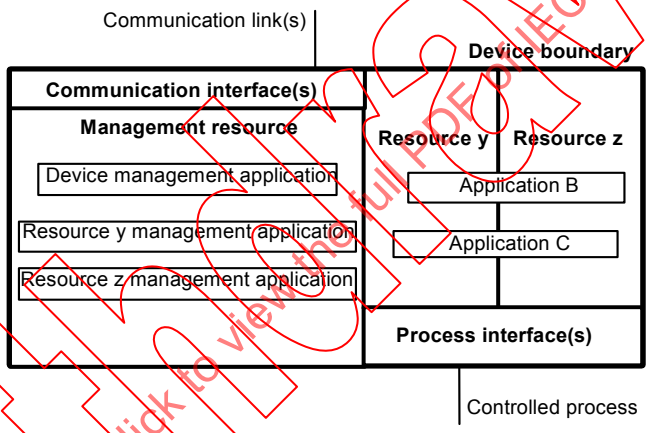
Figures 8b) and 8c) provide a schematic representation of the management of *resources* and *devices*. Figure 8b) illustrates a case in which a *management resource* provides shared facilities for management of other *resources* within a device, while Figure 8c) illustrates the distribution of management services among resources within a device. Management *applications* may be modeled using implementation-dependent *service interface function blocks* and *communication function blocks*.

NOTE 1 Subclause 6.3 defines *service interface function block types* for management of *applications,* and IEC 61499-2 provides examples of their usage.

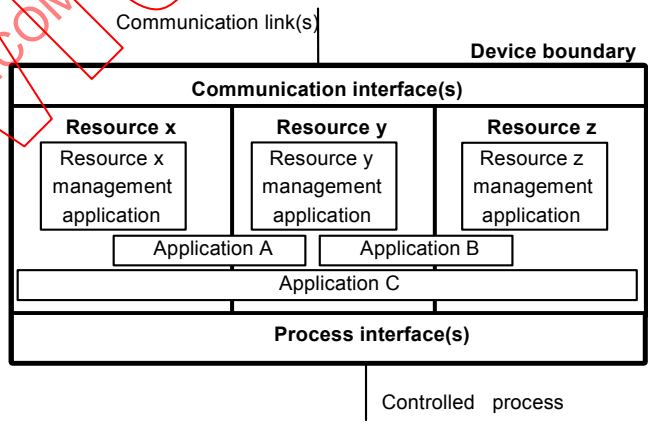
NOTE 2 *Management applications* may contain *service interface function block instances* representing *device* or *resource instances* for the purpose of querying or modifying device or resource *parameters*.



a) Distribution model



b) Shared management model



c) Distributed management model

Figure 8 – Distribution and management models

4.8 Operational state models

Any given *system* has to be designed, commissioned, operated and maintained. This is modeled through the concept of the system "life cycle". In turn, a system is composed of several *functional units* such as *devices, resources,* and *applications,* each of which has its own life cycle.

Different actions may have to be performed to support *functional units* at each step of the life cycle. To characterize which action can be done and maintain integrity of functional units, "operational states" shall be defined, for example, OPERATIONAL, CONFIGURABLE, LOADED, STOPPED, etc.

Each operational state of a functional unit specifies which actions are authorized, together with an expected behavior.

A system may be organized in such a way that certain functional units may possess or acquire the right to modify the operational states of other functional units.

Examples of the use of operational states are:

- A functional unit in a RUNNING state, i.e., in execution, may not be able to receive a download action.
- A distributed functional unit may need to maintain a consistent operational state across its components and develop a strategy to propagate changes of operational state through them.

Specific operational states for managed *function block instances* are defined in 6.3.3.

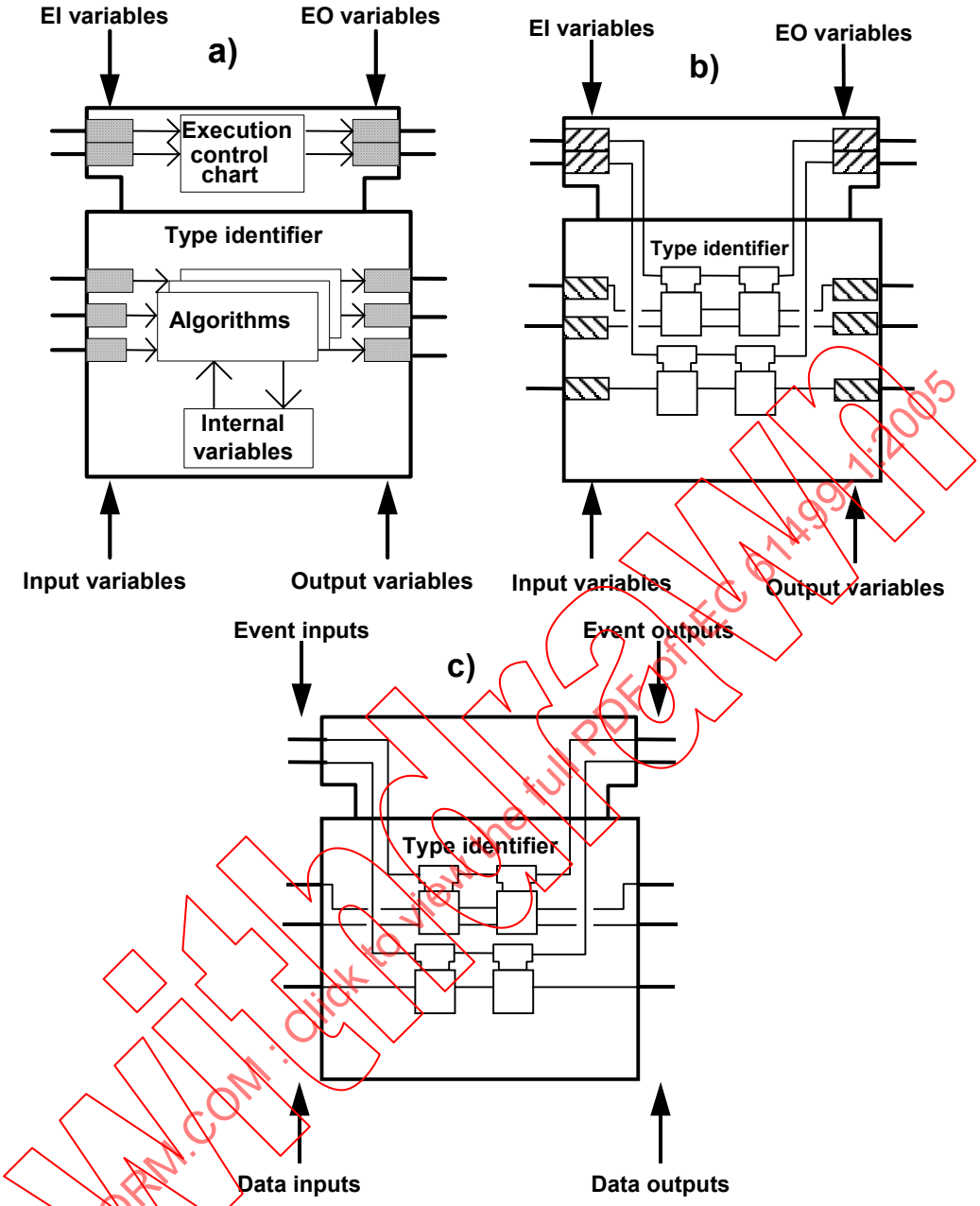
5 Specification of function block, subapplication and adapter interface types

5.1 Overview

As illustrated in Figure 9, this clause defines the means for the type specification of three kinds of blocks:

- Subclause 5.2 defines the means for specifying and determining the behavior of instances of *basic function block types,* as illustrated in Figure 9a). In this type of function block, execution control is specified by an *execution control chart (ECC),* and the *algorithms* to be executed are declared as specified in compliant Standards as defined in IEC 61499-4.
- Subclause 5.3 defines the means for specifying *composite function block types,* as illustrated in Figure 9b). In this type of function block, algorithms and their execution control are specified through event and data connections in one or more *function block networks.*
- Subclause 5.4 defines the means for specifying *subapplication types,* as illustrated in Figure 9c). In this type of block, algorithms and their execution control are specified as for composite function block types, but with the specific property that *component function blocks* of subapplications may be distributed among several *resources.* Subapplications may be nested, such that the body of a subapplication may also contain *component subapplications.*

Other means may be used for describing the behavior of instances of a function block type. The specification of such means is beyond the scope of this part of IEC 61499; therefore it is required that when such means are used, an unambiguous *mapping* shall be given between their terms and concepts and the corresponding terms and concepts of this part of IEC 61499.



a) Basic function block (5.2),

b) Composite function block (5.3),

c) Subapplication (5.4)

Figure 9 – Function block and subapplication types

NOTE This figure is illustrative only. The graphical representation is not normative.

5.2 Basic function blocks

5.2.1 Type declaration

5.2.1.1 General

A *basic function block* utilizes an *execution control chart (ECC)* to control the *execution* of its *algorithms*.

As illustrated in Figure 10, a *basic function block type* can be declared textually according to the syntax specified in Clause B.2 or graphically according to the following rules:

1. The function block *type name* is shown at the top center of the lower portion of the block.
2. The names and *type declarations* of *input variables* and *socket adapters* are shown at the left edge of the lower portion of the block.
3. The names and *type declarations* of *input variables* and *plug adapters* are shown at the right edge of the lower portion of the block.
4. The *interface* of the function block type to *events* is declared in the upper portion of the block as specified in 5.2.1.2.
5. The *algorithms* associated with the function block type are declared as specified in 5.2.1.3.
6. Control of the *execution* of the associated algorithms is declared as specified in 5.2.1.4.

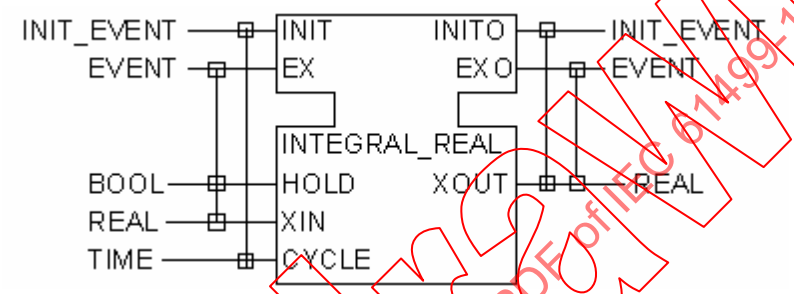


Figure 10 – Basic function block type declaration

NOTE 1 See Annex G for a textual declaration of this example.

NOTE 2 This example is illustrative only. Details of the specification are not normative.

5.2.1.2 Event interface declaration

As shown in Figure 10, the *interface* of a *basic function block type* to *events* can be declared textually according to the syntax given in Clause B.2, or graphically according to the following rules:

a) *Event interfaces* are located in a distinct area at the top of the block.

b) *Event input* names are shown at the left-hand side of the upper portion of the block.

c) *Event output* names are shown at the right-hand side of the upper portion of the block.

d) Event *types* are shown outside the block adjacent to their associated event inputs or outputs.

NOTE 1 If no event type is given for an event input or output, it is considered to be of the default type EVENT.

NOTE 2 An event output of type EVENT can be connected to an event input of any type, and an event input of type EVENT can receive an event of any type.

NOTE 3 An event output of any type other than EVENT can only be connected to an event input of the same type or of type EVENT.

NOTE 4 An event *type* is implicitly declared by its use in an event declaration.

As illustrated in Figure 10 and Annex G, the `WITH` qualifier or a graphical equivalent shall be used to specify an association among *input variables* or *output variables* and an *event* at the associated *event input* or *event output*, respectively.

Each *input variable* and *output variable* appears in zero or more `WITH` clauses or their graphical equivalents.

NOTE 5 This information may be used to determine the required communication *services* when *configuring* a distributed *application* as described in Clause 7.

NOTE 6 An input variable that does not appear in any `WITH` clause cannot be connected with an output variable of another function block. The values of such variables either remain at their declared initial values or are established by management commands such as WRITE, as described in 6.3.1.

NOTE 7 An output variable that does not appear in any WITH clause can be connected to an input variable of another function block or can be "read" by management commands such as READ, as described in 6.3.1.

NOTE 8 See 4.5.3 for an application of the WITH qualifier to the execution model of a basic function block.

5.2.1.3 Algorithm declaration

As shown in Annex G, *algorithms* associated with a *basic function block type* may be included in the function block type declaration according to the rules for declaration of the function block type specification given in Annex B. Other means may also be used for the specification of the identifiers and bodies of algorithms; however, the specification of such means is beyond the scope of this part of IEC 61499.

5.2.1.4 Declaration of algorithm execution control

The sequencing of algorithm invocations for *basic function block types* may be declared in the function block type specification. If the algorithms of a basic function block type are defined as specified in 5.2.1.3 (or otherwise identified), then the sequencing of algorithm invocation for such a function block can be in the form of an *Execution Control Chart (ECC)* consisting of *EC states, EC transitions*, and *EC actions*. These elements are represented and interpreted as follows:

a) The ECC is included in an *execution control* section of the function block type declaration, considered to reside in the upper portion of the block.

b) The ECC shall contain exactly one *EC initial state*, represented graphically as a double-outlined shape with an associated *identifier*. The EC initial state shall have no associated EC actions.

c) The ECC shall contain one or more *EC states*, represented graphically as single-outlined shapes, each with an associated *identifier*.

d) The ECC can utilize but not modify variables declared in the function block type specification.

e) An *EC state* can have zero or more associated *EC actions.* The association of the EC actions with the EC state can be expressed in graphical or textual form.

f) The *algorithm* (if any) associated with an EC action, and the *event* (if any) to be issued on completion of the algorithm, shall be expressed in graphical or textual form.

g) An *EC transition* is represented graphically or textually as a directed link from one EC state to another (or to the same state).

h) Each EC transition shall have an associated Boolean condition, equivalent to a Boolean expression utilizing one or more *event input variables, input variables, output variables,* or *internal variables* of the function block.

Figure 11 illustrates the elements of an ECC. Similar textual declarations using the syntax of Clause B.2 is given in Annex F.

NOTE 1 According to the model given in 5.2.2, the evaluation of an EC transition condition is disabled until the algorithms associated with its predecessor EC state have completed their execution. Therefore, for example, the 1 in the EC transitions following the EC states INIT and MAIN is equivalent in this case to the use of INITO and EXO, respectively.

NOTE 2 In this restricted domain, the same symbol (e.g., INIT) can be used to represent an EC state and algorithm name, since the referent of the symbol can be inferred easily from its usage.

NOTE 3 The text in *italics* is not part of the ECC.

NOTE 4 One-to-one association of events with algorithms, as illustrated in this figure, is frequently encountered but is not the only possible usage. See Table A.1 for examples of other usages, for example, the E_SPLIT block shows an association of two event outputs with one state but no algorithms; E_MERGE shows an association of one output event but no algorithms with two event inputs; E_DEMUX shows any of several algorithms associated with a single input event; etc.

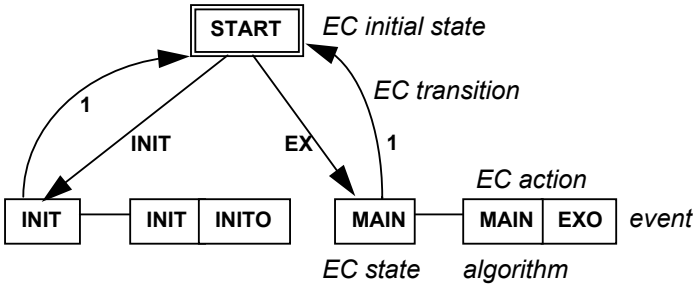


Figure 11 – ECC example

5.2.2 Behavior of instances

5.2.2.1 Initialization

Initialization of a basic function block *instance* by a *resource* shall be functionally equivalent to the following procedure:

a) The value of each *input*, *output*, and *internal variable* shall be initialized to the corresponding initial value given in the function block *type* specification. If no such initial value is defined, the value of the variable shall be initialized to the default initial value defined for the data type of the variable.

b) Any additional algorithm-specific initializations shall be performed; for example, all *initial steps* of IEC 61131-3 *Sequential Function Charts (SFCs)* shall be activated and all other *steps* shall be deactivated.

c) The *EC initial state* of the function block's *Execution Control Chart (ECC)* shall be activated, all other *EC states* shall be deactivated, and the ECC operation state machine defined in 5.2.2.2 shall be placed in its initial (`S0`) state.

NOTE The conditions under which a resource shall perform such initialization are implementation-dependent.

The function block *type* may also specify an initialization *algorithm* to be performed upon the occurrence of an appropriate event, for example the `INIT` algorithm shown in Figure 11. An *application* can then specify the conditions under which this algorithm is to be executed, for example by connecting an output of an instance of the `E_RESTART` type defined in Annex A to an appropriate event input, for example the `INIT` input shown in Figure 10.

5.2.2.2 Algorithm invocation

Execution of an *algorithm* associated with a *function block instance* is *invoked* by a request to the scheduling function of the *resource* to schedule the execution of the algorithm's *operations*.

NOTE 1 The operations performed by an algorithm may vary from one execution to the next due to changed internal states of the function block, even though the function block may have only a single algorithm and a single event input triggering its execution.

Algorithm invocation for an instance of a *basic function block type* shall be accomplished by the functional equivalent of the operation of its *execution control chart (ECC)*. The operation of the ECC shall exhibit the behavior defined by the state machine in Figure 12 and Table 1.

NOTE 2 It is a consequence of this model that an occurrence of an event at an event input will not cause a transition containing the event to clear, if the transition is not associated with the currently active state, i.e., if the event is not relevant in the given state. However, *sampling* of the input variables associated to the event by a WITH construct will occur in any case.

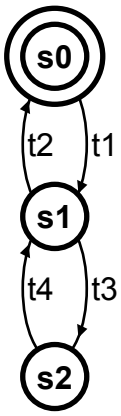


Figure 12 – ECC operation state machine

Table 1 – States and transitions of ECC operation state machine

State		Operations
s0		--
s1		evaluate transitions[c,e]
s2		perform actions[d,e]
Transition	**Condition**	**Operations**
t1	invoke ECC[a]	Sample inputs[b,e]
t2	no transition clears	
t3	a transition clears	
t4	actions completed	

a This transition is activated by the presence of an event at an event input.

b This operation consists of *sampling* the input variables associated with the current event input by a `WITH` declarations as described in 5.2.1.2.

c This operation consists of evaluating the conditions at all the EC transitions following the active EC state and clearing the first EC transition (if any) for which a `TRUE` condition is found. "Clearing the EC transition" consists of deactivating its predecessor EC state and activating its successor EC state. The order in which the transition conditions are evaluated corresponds to the order in which they are declared following the textual syntax defined in B.2.1, or equivalently in the XML syntax defined in IEC 61499-2.

d This operation consists of, for each *EC action* associated with the active *EC step*, executing the associated algorithm, if any, and issuing an event at the associated event output, if any. The order in which the actions are performed corresponds to the order in which they appear graphically from top to bottom, or to the order in which they are declared following the textual syntax defined in B.2.1, or equivalently in the XML syntax defined in IEC 61499-2.

e All operations performed from an occurrence of transition t1 to an occurrence of t2 shall be implemented as a *critical region* with a lock on the function block instance.

5.2.2.3 Algorithm execution

Algorithm *execution* in a basic function block shall consist of the execution of a finite sequence of *operations* determined by implementation-dependent rules appropriate to the language in which the algorithm is written, the *resource* in which it executes, and the domain to which it applies. Algorithm execution terminates after execution of the last operation in this sequence.

If an algorithm implements a state machine, repeated executions of the algorithm are necessary to recognize or perform state changes. Normally there is no association between those state changes and the completion of the algorithm. Such associations shall be created by the event output generation facilities described in 5.2.2.2.

5.3 Composite function blocks

5.3.1 Type specification

The declaration of *composite function block types* shall follow the rules given in 5.2.1 with the exception that *event inputs* and *event outputs* of the *component function blocks* can be interconnected with the event inputs and event outputs of the composite function block to represent the sequencing and causality of function block invocations. The following rules shall apply to this usage:

a) Each event input of the composite function block is connected to exactly one event input of exactly one component function block, or to exactly one event output of the composite function block, with the exception that the graphical shorthand for event splitting shown in Figure A.1 may be employed.

b) Each event input of a component function block is connected to no more than one event output of exactly one other component function block, or to no more than one event input of the composite function block, with the exception that the graphical shorthand for event merging shown in Figure A.1 may be employed.

c) Each event output of a component function block is connected to no more than one event input of exactly one other component function block, or to no more than one event output of the composite function block, with the exception that the graphical shorthand for event splitting shown in Figure A.1 may be employed.

d) Each event output of the composite function block is connected from exactly one event output of exactly one component function block, or from exactly one event input of the composite function block, with the exception that the graphical shorthand for event merging shown in Figure A.1 may be employed.

e) Use of the `WITH` qualifier in the declaration of event inputs of composite function block types is required. Use of the `WITH` qualifier may result in the *sampling* of the associated data inputs as in the case of basic or service interface function blocks.

NOTE 1 Software tools may provide means of elimination of redundant sampling in the implementation phase.

f) *Instances* of *subapplication types* as defined in 5.4 shall not be used in the specification of a composite function block type.

Data inputs and *data outputs* of the *component function blocks* can be interconnected with the data inputs and data outputs of the composite function block to represent the flow of data within the composite function block. The following rules shall apply to this usage:

1) Each data input of the composite function block can be connected to zero or more data inputs of zero or more component function blocks, or to zero or more data outputs of the composite function block, or both.

2) Each data input of a component function block can be connected to no more than one data output of exactly one other component function block, or to no more than one data input of the composite function block.

3) Each data output of a component function block can be connected to zero or more data inputs of zero or more component function blocks, or to zero or more data outputs of the composite function block, or both.

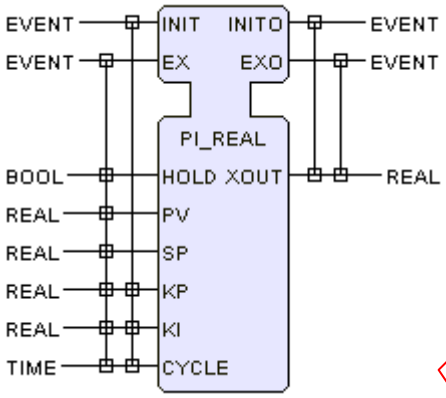
4) Each data output of the composite function block shall be connected from exactly one data output of exactly one component function block, or from exactly one data input of the composite function block.

NOTE 2 If an element declared in a `VAR_INPUT...END_VAR` or `VAR_OUTPUT...END_VAR` construct is associated with an input or output event, respectively, by a `WITH` construct, this will result in the creation of an associated input or output variable, respectively, as in the case of basic function block types. If such an element is not associated with an input or output event, then the associated data flow is passed directly to or from the component function blocks via the connections described above.

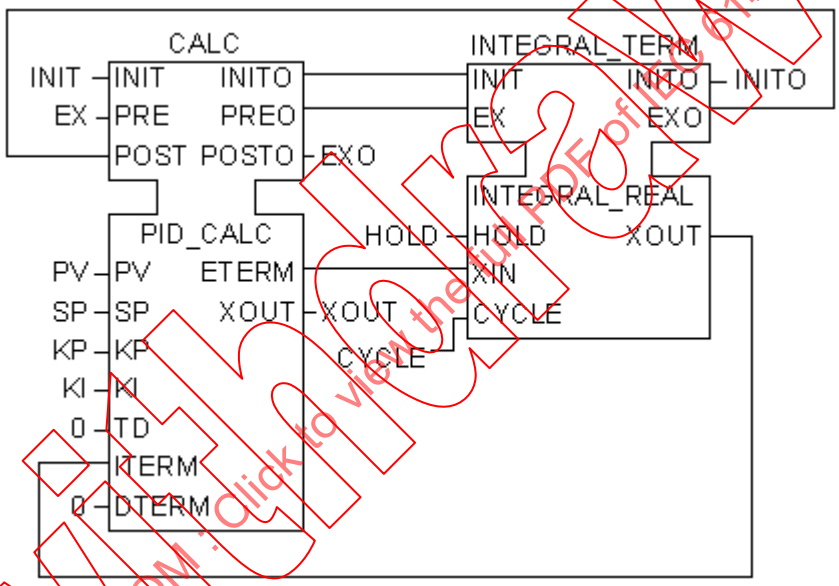
NOTE 3 The rules for interconnection of the event and variable inputs and outputs of *plugs* and *sockets* in the body of the composite function block are the same as for the interconnection of the inputs and outputs of the *component function blocks*. See 5.5 for further requirements regarding *adapter interfaces*.

Figure 13 illustrates the application of these rules to the example PI_REAL function block. Figure 13a) shows the graphical representation of the external interfaces and Figure 13b) shows the graphical construction of its body. Figure 14 shows the interfaces and execution control for the function block type PID_CALC used in the body of the PI_REAL example.

a)



b)



Key

a) External interfaces

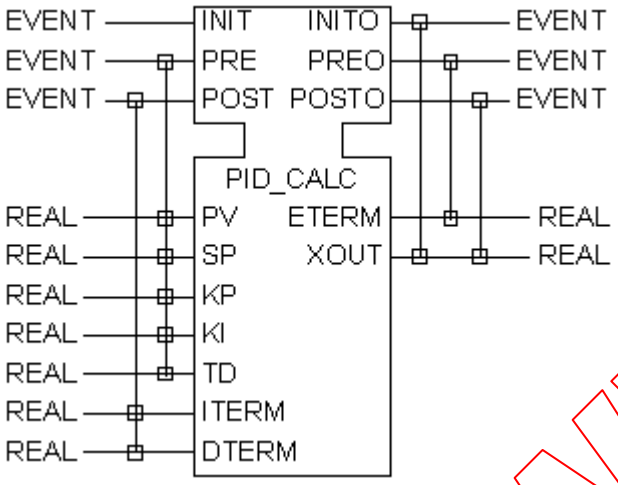
b) Graphical body

NOTE 1 A full textual declaration of this function block type is given in Annex F.

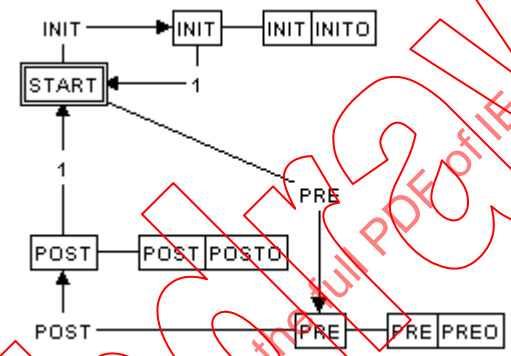
NOTE 2 This example is illustrative only. Details of the specification are not normative.

Figure 13 – Composite function block PI_REAL example

a)



b)



Key

a) External interfaces

b) Graphical body

NOTE This example is illustrative only. Details of the specification are not normative.

Figure 14 – Basic function block `PID_CALC` example

5.3.2 Behavior of instances

Invocation and *execution* of *component function blocks* in composite function blocks shall be accomplished as follows:

a) If an *event input* of the composite function block is connected to an *event output* of the block, occurrence of an *event* at the event input shall cause the generation of an event at the associated event output.

b) If an event input of the composite function block is connected to an event input of a component function block, occurrence of an event at the event input of the composite function block shall cause the scheduling of an invocation of the execution control function of the component function block, with an occurrence of an event at the associated event input of the component function block.

c) If an event output of a component function block is connected to an event input of a second component function block, occurrence of an event at the event output of the first block shall cause the scheduling of an invocation of the execution control function of the second block, with an occurrence of an event at the associated event input of the second block.

d) If an event output of a component function block is connected to an event output of the composite function block, occurrence of an event at the event output of the component block shall cause the generation of an event at the associated event output of the composite function block.

Initialization of instances of composite function blocks shall be equivalent to initialization of their component function blocks according to the provisions of 5.2.2.1.

5.4 Subapplications

5.4.1 Type specification

The declaration of *subapplication types* is similar to the declaration of *composite function block types* as defined in 5.2.1, with the exception that the delimiting keywords shall be SUBAPPLICATION..END_SUBAPPLICATION. The following rules shall apply to this usage:

a) The WITH qualifier is not used in the declaration of event inputs and event outputs of *subapplication types*.

b) Each event input of the subapplication shall be connected to exactly one event input of exactly one component function block or component subapplication, or to exactly one event output of the subapplication.

c) Each event input of a component function block or component subapplication is connected to no more than one event output of exactly one other component function block or component subapplication, or to no more than one event input of the subapplication.

d) Each event output of a component function block or component subapplication is connected to no more than one event input of exactly one other component function block or component subapplication, or to no more than one event output of the subapplication.

e) Each event output of the subapplication is connected from exactly one event output of exactly one component function block or component subapplication, or from exactly one event input of the subapplication.

NOTE 1 Component function blocks may include instances of the event processing blocks defined in Annex A, for example to "split" events using instances of the E_SPLIT block, to "merge" events using instances of the E_MERGE block, or for both cases, using the equivalent graphical shorthand.

Data inputs and *data outputs* of the *component function blocks* or *component subapplications* can be interconnected with the data inputs and data outputs of the subapplication to represent the flow of data within the subapplication. The following rules shall apply to this usage:

1) Each data input of the subapplication can be connected to zero or more data inputs of zero or more component function blocks or component subapplications, or to zero or more data outputs of the subapplication, or both.

2) Each data input of a component function block or component subapplication can be connected to no more than one data output of exactly one other component function block or component subapplication, or to no more than one data input of the subapplication.

3) Each data output of a component function block or component subapplication can be connected to zero or more data inputs of zero or more component function blocks or component subapplications, or to zero or more data outputs of the subapplication, or both.

4) Each data output of the subapplication shall be connected from exactly one data output of exactly one component function block or component subapplication, or from exactly one data input of the subapplication.

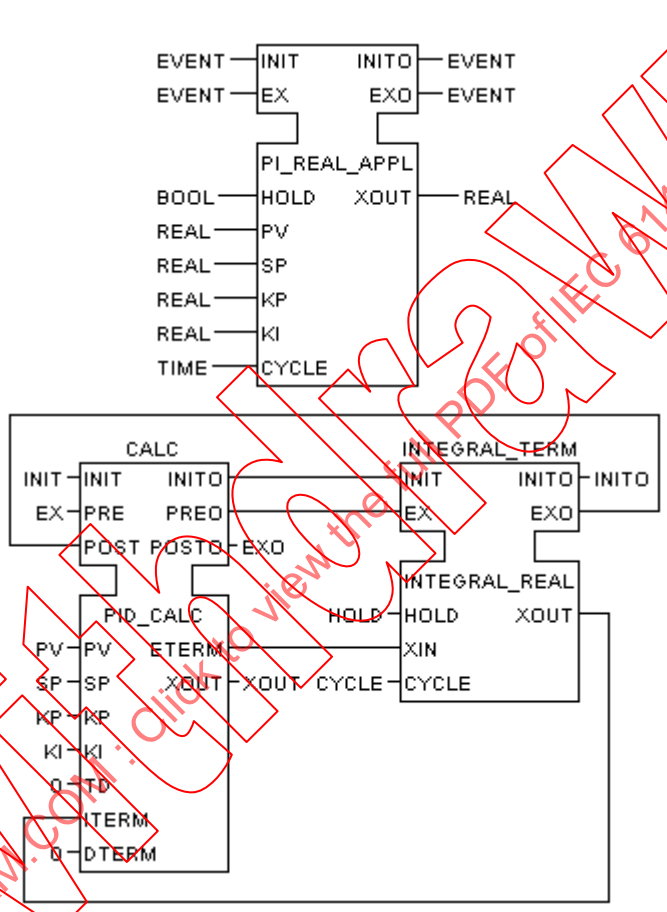
NOTE 2 Although the VAR_INPUT...END_VAR and VAR_OUTPUT...END_VAR constructs are used for the declaration of the data inputs and outputs of subapplication types, this does not result in the creation of input and output variables; the data flow is instead passed to the component function blocks or component subapplications via the connections described above.

NOTE 3 The rules for interconnection of the event and variable inputs and outputs of *plugs* and *sockets* in the body of the subapplication are the same as for the interconnection of the inputs and outputs of the *component function blocks*. See 5.5 for further requirements regarding *adapter interfaces*.

EXAMPLE Figure 15 illustrates the application of these rules to the example PI_REAL_APPL subapplication. Figure 15a) shows the graphical representation of its external interfaces and Figure 15b) shows the graphical construction of its body. The body of the PI_REAL_APPL subapplication example uses the function block type PID_CALC from the composite function block example in 5.3.1, which is shown in Figure 14.

a)

b)



Key

a) External interfaces

b) Graphical body

NOTE 1 A full textual declaration of this subapplication type is given in Annex F.

NOTE 2 This example is illustrative only. Details of the specification are not normative.

Figure 15 – Subapplication PI_REAL_APPL example

5.4.2 Behavior of instances

Invocation of the *operations* of *component function blocks* or *component subapplications* within subapplications shall be accomplished as follows:

a) If an *event input* of the subapplication is connected to an *event output* of the block, occurrence of an *event* at the event input shall cause the generation of an event at the associated event output.

b) If an event input of the subapplication is connected to an event input of a component function block or component subapplication, occurrence of an event at the event input of the subapplication shall cause the scheduling of an invocation of the execution control function of the component function block or component subapplication, with an occurrence of an event at the associated event input of the component function block or component subapplication.

c) If an event output of a component function block or component subapplication is connected to an event input of a second component function block or component subapplication, occurrence of an event at the event output of the first block shall cause the scheduling of an invocation of the execution control function of the second block, with an occurrence of an event at the associated event input of the second block.

d) If an event output of a component function block or component subapplication is connected to an event output of the subapplication, occurrence of an event at the event output of the component block shall cause the generation of an event at the associated event output of the subapplication.

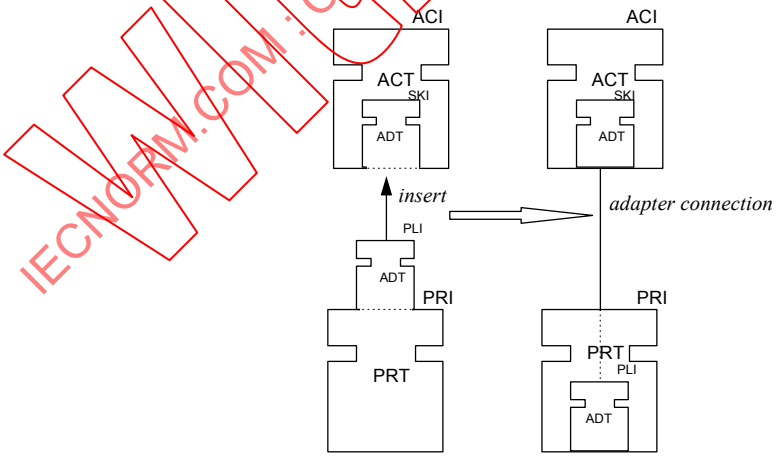
Since subapplications do not explicitly create variables, no specific initialization procedures are applicable to subapplication instances.

5.5 Adapter interfaces

5.5.1 General principles

Adapter interfaces can be used to provide a compact representation of a specified set of event and data flows. As illustrated in Figure 17, an *adapter interface type* provides a means for defining a subset (the *plug adapter)* of the *inputs* and *outputs* of a *provider* function block which can be inserted into a matching subset of corresponding *outputs* and *inputs* (the *socket adapter)* of an *acceptor* function block. Thus, the adapter interface represents the event and data paths by which the provider supplies a *service* to the acceptor, or vice versa, depending on the patterns of provider/acceptor interactions, which may be represented by sequences of *service primitives* as described in 6.1.3.

NOTE A given *function block type* may function as a *provider,* an *acceptor*, or both, or neither, and may contain more than one *plug* or *socket instance* of one or more *adapter interface types*.



Key

PRT – Provider type,
PRI – Provider instance
ACT – Acceptor type,
ACI – Acceptor instance
ADT – Adapter type,
PLI – Plug instance,
SKI – Socket instance

NOTE This figure is illustrative only. The graphical representation is not normative.

Figure 16 – Adapter interfaces – Conceptual model

5.5.2 Type specification

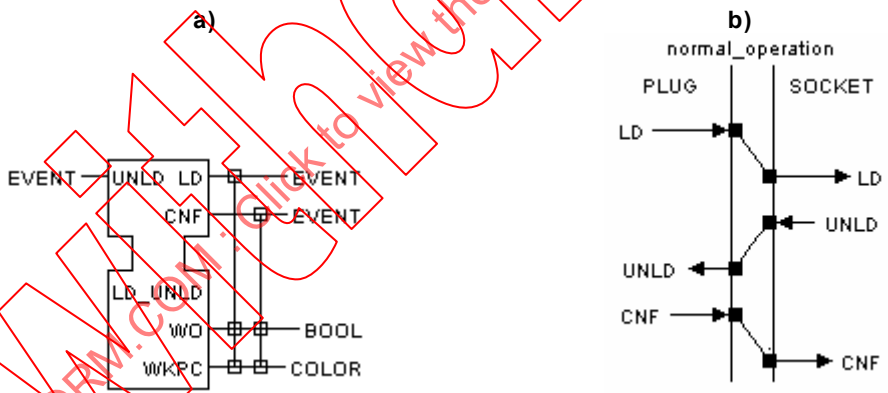
An *adapter interface type declaration* shall define only the *interface type* name and its contained *event* and *data interfaces.* These are defined graphically or textually in the same manner as the *type name, event interfaces* and *data interfaces* of a *basic function block type* as defined at the beginning of 5.2.1.1 and 5.2.1.2, with the exception that the keywords for beginning and ending the textual type declaration shall be `ADAPTER...END_ADAPTER`. Textual syntax for the declaration of adapter interfaces is given in Clause B.7.

EXAMPLE The adapter interface illustrated in Figure 17 represents the operation of transferring a workpiece from an "upstream" piece of transfer equipment represented by a *provider* of the *plug* adapter to a "downstream" piece of equipment represented by an *acceptor* with a corresponding *socket* adapter. As illustrated in Figure 17b), the typical operation of this interaction consists of the following sequence:

a) An event in the upstream equipment, for example, arrival of a workpiece at the unload position, causes a `LD` event, typically interpreted as a "load" command, to be transmitted to the downstream equipment. Associated with this event is a sensor value `WO`, indicating whether a workpiece is actually present for transfer, plus some measured property or set of properties of the workpiece, in this case its color.

b) A subsequent event in the downstream equipment, for example, completion of the load setup, causes an `UNLD` event, typically interpreted as a command to release the workpiece, to be sent to the upstream equipment.

c) Subsequently, a `CNF` event, typically interpreted as confirmation of the workpiece release, is passed from the upstream to the downstream equipment to complete the operation. At this point, the `WO` output is typically `FALSE` and the value of the `WKPC` output has no significance.



Key

a) Interface

b) Normal operation (see 6.1.3)

NOTE 1 A full textual declaration of this adapter type is given in Annex F.

NOTE 2 This example is illustrative only. Details of the specification are not normative.

Figure 17 – Adapter type declaration – graphical example

5.5.3 Usage

The usage of *adapter interface types* and *instances* shall be according to the following rules:

a) Adapter interface instances to be used as *plugs* in instances of a *function block type* shall be declared in its *type declaration* in a `PLUGS...END_PLUGS` block, declaring the *instance name* and *adapter interface type* of each plug. In the graphical representation of *function block types* and *instances*, plugs are shown as *output variables* with specialized textual or graphical indication that they are not ordinary output variables.

b) Adapter interface instances to be used as *sockets* in instances of a *function block type* shall be declared in its *type declaration* in a SOCKETS...END_SOCKETS block, declaring the *instance name* and *adapter interface type* of each socket. In the graphical representation of *function block types* and *instances*, sockets are shown as *input variables* with specialized textual or graphical indication that they are not ordinary input variables.

c) *Inputs* and *outputs* of a *plug* shall be used within its *function block type declaration* in the same manner as inputs and outputs of the function block.

d) *Inputs* and *outputs* of a *socket* shall be used within its *function block type declaration* in the same manner as *outputs* and *inputs* of the function block, respectively.

e) Insertion of *plugs* into *sockets* shall be specified in an ADAPTER_CONNECTIONS ... END_CONNECTIONS block in the *declaration* of the *application, subapplication, resource type, resource instance,* or *composite function block type* containing the respective *provider* and *acceptor* instances.

f) In the body of a *composite function block type* or *subapplication,* a *socket* is represented as a *function block* with the same inputs and outputs as the corresponding *adapter interface type*. Similarly, in this case, a *plug* is represented as a function block with the inputs and outputs of the corresponding adapter interface type reversed.

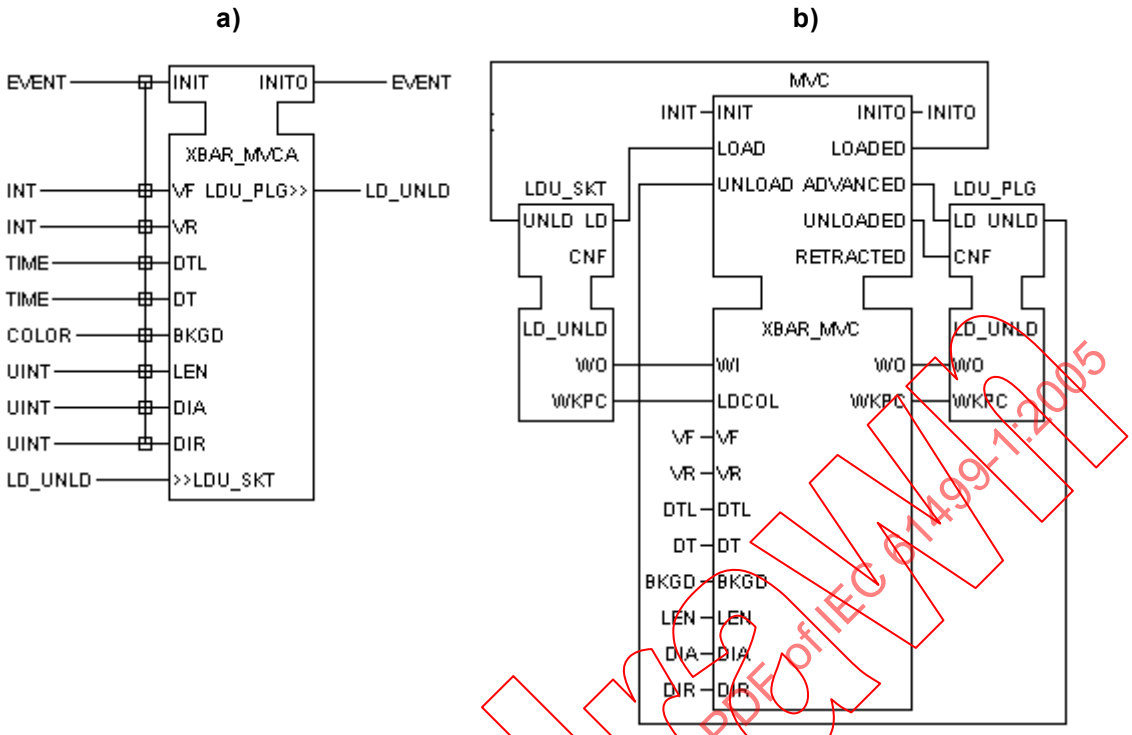
g) Insertion of plugs into sockets shall be subject to the following constraints:

- A plug can only be inserted into a socket of the same *adapter interface type.*
- A plug can only be inserted into zero or one socket at a time.
- A socket can only accept zero or one plug at a time.
- A plug can only be inserted in a socket if both are in the same *composite function block, resource*, *application* or *subapplication*.

NOTE A connection from a plug to a socket may be shown in an *application* or *subapplication* even though the corresponding function block instances may be *mapped* to separate *resources*. In this case, appropriate means, such as communication service interface function blocks as described in 6.2, shall be used to implement the corresponding transfer of events and data among resources.

Management function blocks as described in 6.3 may provide facilities for the dynamic creation, deletion, and querying of adapter connections.

EXAMPLE 1 An instance of the XBAR_MVCA type illustrated in Figure 18 acts as both a provider of a plug interface (LDU_PLG) and an acceptor with a socket interface (LDU_SKT). In so doing, it serves to abstract and encapsulate the interactions of an instance of the XBAR_MVC type with "upstream" and "downstream" functional units.



Key

a) Interface

b) Body

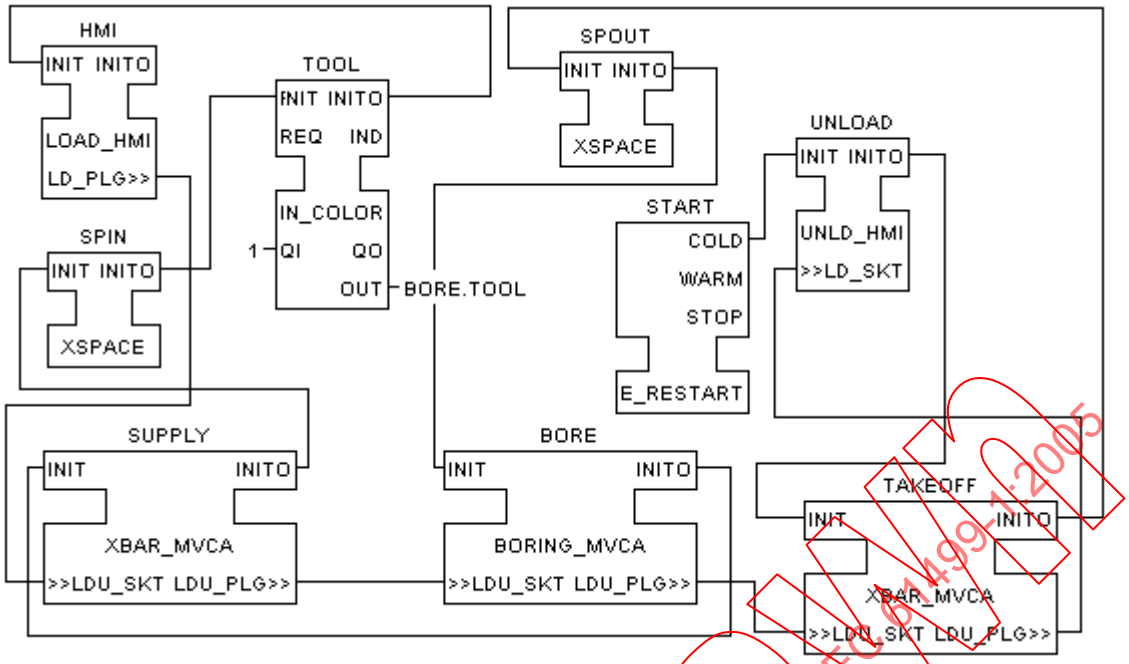
NOTE 1 A full textual declaration of this example is given in Annex F.

NOTE 2 This example is illustrative only. Details of the specification are not normative.

NOTE 3 Although this example presents only a composite type, *provider* and *acceptor* function block *types* may be either *basic* or *composite*.

Figure 18 – Illustration of provider and acceptor function block type declarations

EXAMPLE 2 Figure 19 illustrates a *resource configuration* containing two instances of the XBAR_MVCA type illustrated in Figure 18. The SUPPLY instance acts as an *acceptor* ("downstream unit") for the HMI block and a *provider* ("upstream unit") for the BORE block, while the TAKEOFF instance fulfills corresponding roles for the BORE and UNLOAD blocks, respectively.



NOTE 1 This example is illustrative only. Details of the specification are not normative.

NOTE 2 *Parameter* connections are omitted in this diagram for clarity.

NOTE 3 Type declarations for blocks other than the XBAR_MVCA type are not given in Annex F.

Figure 19 – Illustration of adapter connections

5.6 Exception and fault handling

Additional facilities for the prevention, recognition and handling of *exceptions* and *faults* may be provided by *resources*. Such capabilities may be modeled as *service interface function blocks*. The definition of specific function block types for prevention, recognition and handling of exceptions and faults is beyond the scope of this part of IEC 61499. However, `INIT-`, `CNF-` and `IND-` outputs of service interface function blocks, and the associated `STATUS` values, may be used to indicate the occurrence and type of exceptions and faults, as noted in 6.1.3.

6 Service interface function blocks

This clause defines general principles for the specification of *types* and the behavior of *instances* of *service interface function blocks*, and for two specific types of service interface function blocks, i.e., *communication function blocks* and *management function blocks*.

6.1 General principles

6.1.1 General

A *service interface function block* provides one or more *services* to an application, based on a *mapping* of *service primitives* to the function block's *event inputs, event outputs, data inputs* and *data outputs*.

The external interfaces of *service interface function block types* have the same general appearance as *basic function block types*. However, some inputs and outputs of service interface function block types have specialized semantics, and the behavior of *instances* of these types is defined through a specialized graphical notation for sequences of *service primitives.*

NOTE The specification of the internal operations of service interface function blocks is beyond the scope of this part of IEC 61499.

6.1.2 Type specification

Declaration of *service interface function block types* may use the standard *event inputs, event outputs, data inputs* and *data outputs* listed in Table 2, as appropriate to the particular service provided. When these are used, their semantics shall be as defined in this clause. The name of the function block *type* shall indicate the provided service.

EXAMPLE Figures 20a) and b) show examples of service interface function blocks in which the primary interaction is initiated by the *application* and by the *resource*, respectively.

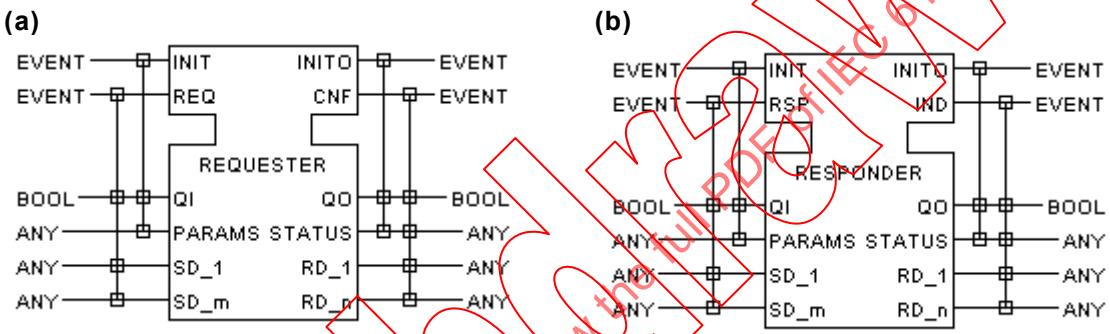
NOTE 1 Some services may provide both resource- and application-initiated interactions in the same service interface function block.

NOTE 2 Service interface types may also utilize inputs and outputs, including *plugs* and *sockets*, with names different from those given here; in such case their usage shall be defined in terms of appropriate sequences of service primitives.

Table 2 – Standard inputs and outputs for service interface function blocks

Event inputs
INIT This event input shall be *mapped* to a *request primitive* which requests an initialization of the service provided by the function block instance, e.g., local initialization of a *communication connection* or a process interface module.
REQ This event input shall be mapped to a *request primitive* of the service provided by the function block instance.
RSP This event input shall be mapped to a *response primitive* of the service provided by the function block instance.
Event outputs
INITO This event output shall be mapped to a *confirm primitive* which indicates completion of a service initialization procedure.
CNF This event output shall be mapped to a *confirm primitive* of the service provided by the function block instance.
IND This event output shall be mapped to an *indication primitive* of the service provided by the function block instance.
Data inputs
QI : **BOOL** This input represents a qualifier on the *service primitives* mapped to the *event inputs*. For instance, if this input is TRUE upon the occurrence of an INIT event, initialization of the service is requested; if it is FALSE, termination of the service is requested.
PARAMS : **ANY** This input contains one or more *parameters* associated with the service, typically as elements of an *instance* of a *structured data type*. When this input is present, the *function block type* specification shall define its *data type* and default initial value(s). NOTE 1 A service interface function block type specification may substitute one or more service parameter inputs for this input.
SD_1, ..., SD_m : ANY These inputs contain the data associated with *request* and *response primitives*. The *function block type* specification shall define the *data types* and default values of these inputs, and shall define their associations with event inputs in an event sequence diagram as illustrated in 6.1.3. NOTE 2 The function block type specification may define other names for these inputs.

Data outputs
QO : BOOL This variable represents a qualifier on the *service primitives* mapped to the *event outputs*. For instance, a TRUE value of this output upon the occurrence of an INITO event indicates successful initialization of the service; a FALSE value indicates unsuccessful initialization.
STATUS : ANY This output shall be of a *data type* appropriate to express the status of the service upon the occurrence of an event output. NOTE 3 A service specification may indicate that the value of this output is irrelevant for some situations, e.g., for INITO+, IND+ and CNF+ as described in 6.1.3.
RD_1, ..., RD_n : ANY These outputs contain the data associated with *confirm* and *indication primitives*. The function block *type specification* shall define the *data types* and initial values of these outputs, and shall define their associations with event outputs in an event sequence diagram as described in 6.1.3. NOTE 4 The function block type specification may define other names for these outputs.



Key

a) for application-initiated interactions

b) for resource-initiated interactions

NOTE 1 REQUESTER and RESPONDER represent the particular services provided by instances of the function block types.

NOTE 2 The *data types* of the SD_1,...,SD_n inputs and RD_1,...RD_m outputs will typically be fixed as some non-generic data type, for instance INT or WORD, in concrete implementations of the generic function block types illustrated here.

NOTE 3 See Annex C for a full textual declaration of the REQUESTER function block type.

Figure 20 – Example service interface function blocks

6.1.3 Behavior of instances

The behavior of *instances* of *service interface function blocks* shall be defined in the corresponding *function block type* specification. This specification can utilize the time-sequence diagrams described in ISO/IEC 10731. When such diagrams are used, such use shall be subject to the following rules:

a) The normal ISO/IEC 10731 semantics shall apply, that is:

- Time increases in the downward direction.
- Events which are sequentially related are linked together across or within resources.
- If there is no specific relationship between events, in that it is impossible to foresee which will occur first but both shall occur within a finite period of time, a tilde (~) or similar textual notation is used.

b) In the case where the service is represented by a single service interface function block, the diagram shall be partitioned by a single vertical line into two fields as illustrated in Figure 21:

- In the case where the service is provided primarily by an application-initiated interaction, the *application* shall be in the left-hand field and the *resource* in the right-hand field, as illustrated in Figure 21a).
- In the case where the service is provided primarily by a resource-initiated interaction, the *resource* shall be in the left-hand field and the *application* in the right-hand field, as illustrated in Figure 21b).

c) In the case where the service is represented by two or more service interface function blocks, the notation of Figures 4 and 5 of ISO/IEC 10731 shall be used, as illustrated in Clause E.2.

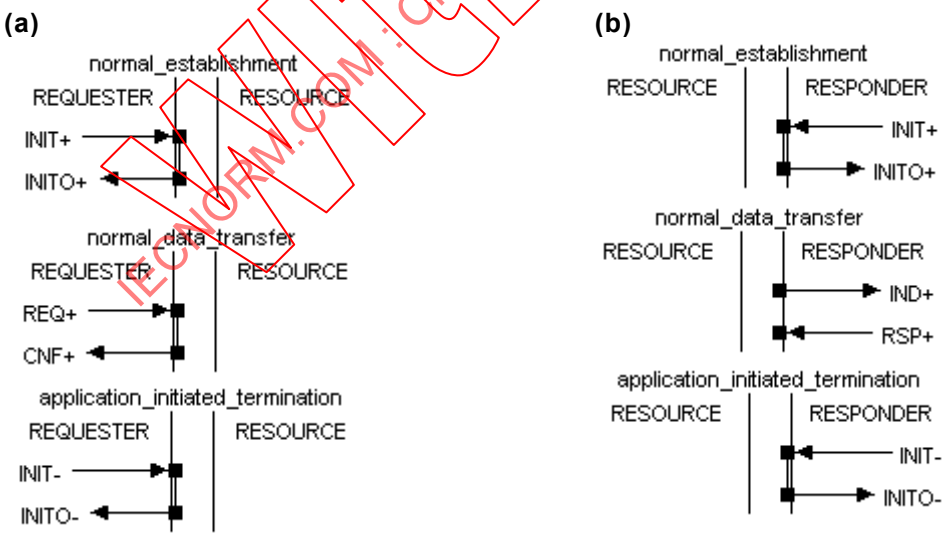
d) *Service primitives* shall be indicated by horizontal arrows as in ISO/IEC 10731. The name of the *event* representing the service primitive shall be written adjacent to the arrow, and means shall be provided to determine the names of the input and/or output *variables* representing the *data* associated with the primitive.

e) When a QI input is present in the function block type definition, the suffix "+" shall be used in conjunction with an *event input* name to indicate that the value of the QI input is TRUE upon the occurrence of the associated event, and the suffix "-" shall be used to indicate that it is FALSE.

f) When a QO output is present in the function block type definition, the suffix "+" shall be used in conjunction with an *event output* name to indicate that the value of the QO output is TRUE upon the occurrence of the associated event, and the suffix "-" shall be used to indicate that it is FALSE.

g) The standard semantics of asserted (+) and negated (-) events shall be as specified in Table 3.

Figure 21 illustrates normal sequences of service initiation, data transfer, and service termination. *Service interface function block type* specifications can utilize similar diagrams to specify all relevant sequences of service primitives and their associated data under both normal and abnormal conditions.



Key

a) Diagram for application-initiated (request/confirmation) interactions

b) Diagram for resource-initiated (indication/response) interactions

Figure 21 – Examples of time-sequence diagrams

Table 3 – Service primitive semantics

Primitive	Semantics
INIT+	Request for service establishment
INIT-	Request for service termination
INITO+	Indication of establishment of normal service
INITO-	Rejection of service establishment request or indication of service termination
REQ+	Normal request for service
REQ-	Disabled request for service
CNF+	Normal confirmation of service
CNF-	Indication of abnormal service condition
IND+	Indication of normal service arrival
IND-	Indication of abnormal service condition
RSP+	Normal response by application
RSP-	Abnormal response by application

6.2 Communication function blocks

6.2.1 Type specification

Communication function blocks provide *interfaces* between *applications* and the "communication mapping" functions of *resources* as defined in 4.3; hence, they are *service interface function blocks* as described in 6.1.

Like other service interface function blocks, a communication function block may be of either *basic* or *composite* type, as long its operation can be represented by a *mapping* of *service primitives* to the function block's *event inputs, event outputs, data inputs* and *data outputs*.

This subclause provides rules for the *declaration* of *communication function block types.* Subclause 6.2.2 provides rules for the behavior of *instances* of such function block types. Clause E.2 defines generic communication function block types for *unidirectional* and *bidirectional transactions*, and gives rules for the implementation-dependent customization of these types.

Declaration of *communication function block types* shall utilize the means defined in 6.1 for the declaration of *service interface function block types*, with the specialized semantics shown in Table 4 for *input* and *output variables*.

Table 4 – Variable semantics for communication function blocks

Variable	Semantics
PARAMS	This input provides *parameters* of the *communication connection* associated with the *communication function block instance*. This shall include means of identifying the communication protocol and communication connection, and may include other parameters of the communication connection such as timing constraints, etc.
SD_1,..., SD_m	These inputs represent *data* to be transferred along the *communication connection* specified by the PARAMS input upon the occurrence of a REQ+ or RSP+ *primitive*, as appropriate. [a]
STATUS	This output represents the status of the *communication connection*, for instance: - Normal completion of initiation, termination, or data transfer - Reasons for abnormal initiation, termination, or data transfer
RD_1,..., RD_n	These outputs represent *data* received along the *communication connection* specified by the PARAMS input upon the occurrence of an IND+ or CNF+ primitive, as appropriate. [a]
NOTE Communication function block type declarations may define constraints between RD_1,...,RD_n outputs and the SD_1,..., SD_m inputs of corresponding function block instances. For example. the number and types of the RD outputs may be constrained to match the number and types of the corresponding SD inputs.	
[a] *Communication function block type declarations* shall define the number and type of the SD_1,..., SD_m inputs and RD_1,...,RD_n outputs, and may assign them other names.	

6.2.2 Behavior of instances

As illustrated in Clause E.2, the behavior of *instances* of *communication function block types* shall be defined in the corresponding communication function block type *declaration*, utilizing the means specified for *service interface function blocks* in 6.1 with the specialized service primitive semantics given in Table 5. Such specification shall include *service primitive* sequences for:

- normal and abnormal establishment and release of *communication connections;*
- normal and abnormal data transfer.

Table 5 – Service primitive semantics for communication function blocks

Primitive	Semantics
INIT+	Request for communication connection establishment
INIT-	Request for communication connection release
INITO+	Indication of communication connection establishment
INITO-	Rejection of communication connection establishment request or indication of communication connection release
REQ+	Normal request for data transfer
REQ-	Disabled request for data transfer
CNF+	Normal confirmation of data transfer
CNF-	Indication of abnormal data transfer
IND+	Indication of normal data arrival
IND-	Indication of abnormal data arrival
RSP+	Normal response by application to data arrival
RSP-	Abnormal response by application to data arrival

6.3 Management function blocks

This subclause defines requirements and *function block types* for the management of *applications*, and the behaviors of function blocks under the control of *management function blocks*.

6.3.1 Requirements

Extending the functional requirements for "application management" in 8.3.2 of ISO/IEC 7498-1 to the distributed application model of this part of IEC 61499 indicates that *services* for management of resources and applications in IPMCSs should be able to perform the following *functions:*

a) In a *resource,* create, initialize, start, stop, delete, query the existence and *attributes* of, and provide notification of changes in availability and status of:
   - data types;
   - function block types and instances;
   - connections among function block instances.

b) In a *device,* create, initialize, start, stop, delete, query the existence and *attributes* of, and provide notification of changes in availability and status of *resources*.

NOTE 1 The provisions of this subclause are not intended to meet the requirements for *system management* addressed in ISO/IEC 7498-4 and ISO/IEC 10040, except as such requirements are addressed by the above listed functions.

NOTE 2 This subclause only deals with item (1) above, i.e., the management of *applications* in *resources*. A framework for device management is described in IEC 61499-2.

NOTE 3 The associations among *resources, applications,* and *function block instances* are defined in *system configurations* as described in 7.3.

NOTE 4 Starting and termination of a distributed *application* is performed by an appropriate *software tool*.

6.3.2 Type specification

Figure 22 illustrates the general form of *management function block types* whose *instances* meet the application management requirements defined above.

NOTE 1 In particular implementations, the type name (`MANAGER` in this example) may represent the type of the managed resource.

NOTE 2 For these function block types, the specific `CMD` and `OBJECT` inputs and `RESULT` output replace the generic `SD_1` and `SD_2` inputs and `RD_1` output described in 6.1.

NOTE 3 The `INIT` and `PARAMS` inputs and `INITO` output may or may not be present in a particular implementation.

NOTE 4 When present, the type and values of the `PARAMS` input are implementation-dependent parameters of the resource type.

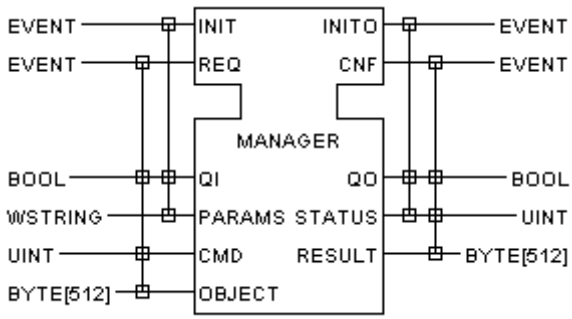
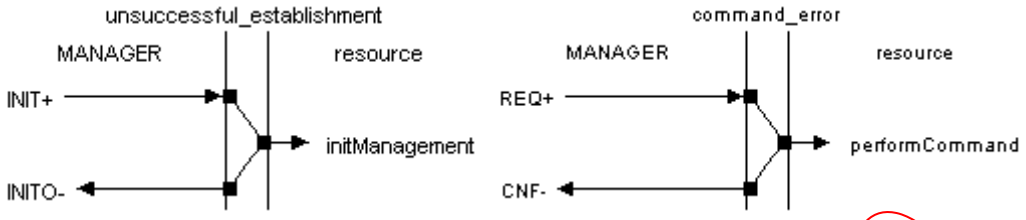


Figure 22 – Generic management function block type

The behavior of instances and input/output semantics of management function block types shall follow the rules given in 6.1 for *service interface function block types* with application-initiated interactions, with the additional behaviors shown in Figure 23 for unsuccessful service initiation and requests.



NOTE A full textual specification of this function block type, including all service sequences, is given in Annex F.

Figure 23 – Service primitive sequences for unsuccessful service

The management *operation* to be *executed* shall be expressed by the value of the CMD input of a management function block according to the semantics defined in Table 6.

Table 6 – CMD input values and semantics

Value	Command	Semantics
0	CREATE	Create specified object
1	DELETE	Delete specified object
2	START	Start specified object
3	STOP	Stop specified object
4	READ	Read data from access path
5	WRITE	Write data to access path
6	KILL	Make specified object unrunnable
7	QUERY	Request information on specified object

The values and corresponding semantics of the STATUS output of a management function block shall be as described in Table 7 to express the result of performing the specified command.

Table 7 – STATUS output values and semantics

Value	Status	Semantics
0	RDY	No errors
1	BAD_PARAMS	Invalid PARAMS input value
2	LOCAL_TERMINATION	Application-initiated termination
3	SYSTEM_TERMINATION	System-initiated termination
4	NOT_READY	Manager is not able to process the command
5	UNSUPPORTED_CMD	Requested command is not supported
6	UNSUPPORTED_TYPE	Requested object type is not supported
7	NO_SUCH_OBJECT	Referenced object does not exist
8	INVALID_OBJECT	Invalid object specification syntax
9	INVALID_OPERATION	Commanded operation is invalid for specified object
10	INVALID_STATE	Commanded operation is invalid for current object state
11	OVERFLOW	Previous transaction still pending

The actual lengths of the OBJECT input and RESULT output of management function block instances are **implementation-dependent**.

The OBJECT input shall specify the object to be operated on according to the CMD input, and the RESULT output shall contain a description of the object resulting from the operation if successful. The contents of these strings shall consist of implementation-dependent encodings of objects defined as non-terminal symbols in Annex B and referenced in Table 8.

NOTE 5 An example of XML-encoded OBJECT and RESULT strings is given in IEC 61499-3, Clause 5.

NOTE 6 The maximum allowable length of the OBJECT input and RESULT output is an implementation-dependent parameter; the value of 512 given in Figure 22 is illustrative.

Table 8 – Command syntax

CMD	OBJECT	RESULT
CREATE	type_declaration	data_type_name
	fb_type_declaration	fb_type_name
	fb_instance_definition	fb_instance_reference
	connection_definition	connection_start_point
	access_path_declaration	access_path_name
DELETE	data_type_name	data_type_name
	fb_type_name	fb_type_name
	fb_instance_reference	fb_instance_reference
	connection_definition	connection_definition
	access_path_name	access_path_name
START	fb_instance_reference	fb_instance_reference
	application_name	application_name
STOP	fb_instance_reference	fb_instance_reference
	application_name	application_name
KILL	fb_instance_reference	fb_instance_reference
QUERY	all_data_types	data_type_list
	all_fb_types	fb_type_list
	data_type_name	type_declaration
	fb_type_name	fb_type_declaration
	fb_instance_reference	fb_status
	connection_start_point	connection_end_points
	application_name	fb_instance_list
	access_path_name	access_path_declaration
READ	access_path_name	accessed_data
WRITE	access_path_data	access_path_name

NOTE 1 See Table 6 for the integer values of the CMD input corresponding to the commands listed above.

NOTE 2 The READ and WRITE commands are limited to a single access path at a time. Other standards may define more complex services, e.g., for multi-variable access, with appropriate service interfaces.

It shall be an error, resulting in a STATUS code of INVALID_OBJECT, if a CREATE command attempts to create:

- a *function block* whose *instance name* duplicates that of an existing function block within the same *resource*,
- a duplicate *connection*, or
- multiple connections to a *data input.*

The single exception to the above rule is that a CREATE command can replace a connection of a *parameter* to a *data input* with a new parameter connection.

It shall be an error, resulting in a STATUS code of UNSUPPORTED_TYPE, if a CREATE command attempts to create a function block instance or parameter of a *type* which is not known to the management function block.

It shall be an **error**, resulting in a STATUS code of INVALID_OPERATION, if a DELETE command attempts to delete a *function block type*, function block instance, *data type* or connection which is defined in the *type specification* of the managed *resource*.

The semantics of the START and STOP commands shall be as follows:

a) START and STOP of a *function block instance* shall be as defined in 6.3.3.

b) START and STOP of an *application* shall be equivalent to START and STOP, respectively, of all *function block instances* in the application contained within the managed *resource*.

c) STOP of a *management function block instance* shall be equivalent to STOP of all *function block instances* within the managed *resource.*

d) START of a *management function block instance* shall be equivalent to START of all *function block instances* within the managed *resource.* If the managed resource was previously stopped, this shall be followed by issuing of an event at the appropriate output of each instance of the E_RESTART function block type defined in Annex A. These events shall occur at the WARM outputs of the E_RESTART blocks if the resource was stopped due to a previous STOP command, and at the COLD outputs otherwise.

Specialized semantics for the QUERY command shall be as follows:

1) When the OBJECT input specifies an *event input, event output* or *data output,* the RESULT output shall contain zero or more opposite end points.
2) When the OBJECT input specifies a *data input,* the RESULT output shall list zero or one opposite end point.
3) When the OBJECT input specifies the name of an *application,* the RESULT output shall list the names of all function blocks in the application contained within the managed *resource*.

6.3.3 Behavior of managed function blocks

Function blocks that are under the control of a *management function block* shall exhibit operational behaviors equivalent to that shown in the state transition diagram of Figure 24, subject to the following rules:

a) The capitalized transition conditions in Figure 24 refer to a value of the CMD input, as specified in Table 6, of the management function block upon the occurrence of a REQ+ service primitive.

b) The command_error sequence of primitives for the MANAGER function block type shall occur, with the indicated value of the STATUS output as defined in Table 7, under the following conditions:

- UNSUPPORTED_CMD: No state exists in Figure 24 with a transition condition for the specified CMD value.
- INVALID_STATE: The currently active state does not have a transition condition for the specified CMD value.
- UNSUPPORTED_TYPE: The CMD value is CREATE, and the function block instance does not exist, but the function block type is unknown to the MANAGER instance, i.e., the guard condition **type_defined** is FALSE.
- INVALID_OPERATION: The CMD value is DELETE, and the function block instance is in the STOPPED or KILLED state, but the function block instance is *declared* in the *device* or *resource type* specification, i.e., the guard condition **is_deletable** is FALSE.

c) The normal_command_sequence of primitives shown for the MANAGER function block type shall follow a CMD+ service primitive under all other conditions, with a value of RDY for the STATUS output as defined in Table 7, and a corresponding value for the RESULT output as defined in Table 8.

d) The semantics of the actions shown in Figure 24 shall be as shown in Table 9 for managed *basic* and *service interface function blocks*.

e) The actions described in the previous rule apply recursively to all *component function blocks* of managed *composite function blocks*.

NOTE 1 The behaviors of function blocks that are not under the control of management function blocks are beyond the scope of this Standard.

NOTE 2 Specification of the behavior of managed function blocks under conditions of power loss and restoration is beyond the scope of this Standard. Such behavior may be specified by the manufacturer of a compliant device, for example by reference to an appropriate Standard.

NOTE 3 *Applications* may utilize *instances* of the E_RESTART block described in Annex A to generate events that can be used to trigger appropriate algorithms upon power loss and restoration.

NOTE 4 As described in 5.4.2, execution control in *subapplications* is entirely deferred to the execution control mechanisms of their component function blocks and component subapplications.

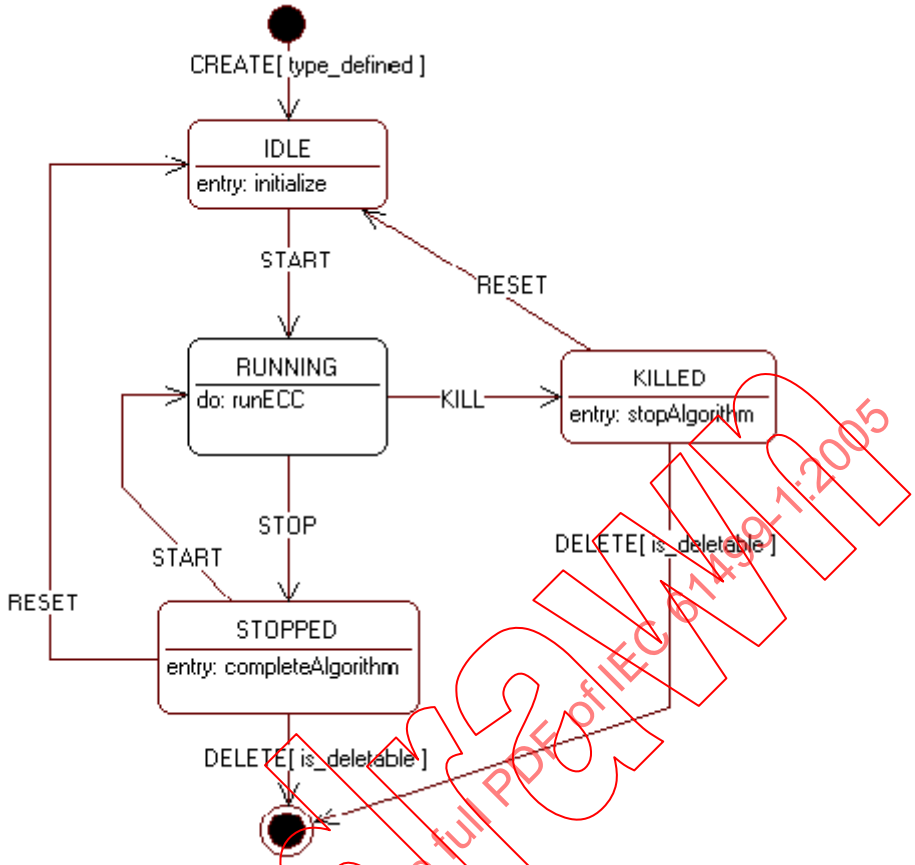


Figure 24 – Operational state machine of a managed function block

Table 9 – Semantics of actions in Figure 24

Action	Basic function blocks	Service interface function block
initialize	Initialize all variables as defined in 5.2.2.1.	
	Perform other initialization operations as defined in 5.2.2.1.	Place service in the proper state to respond correctly to an INIT+ primitive.
runECC	Enable operation of the ECC state machine defined in 5.2.2.2.	Enable invocation of service primitives by events at event inputs, and generation of events at event outputs.
completeAlgorithm	Allow the currently active algorithm (if any) without further generation of output events.	Allow the currently active service primitive to complete.
stopAlgorithm	Terminate the operations of the currently active algorithm (if any) immediately.	Terminate all operations of the service immediately.

7 Configuration of functional units and systems

7.1 Principles of configuration

This clause contains rules for the *configuration* of industrial-process measurement and control *systems* (IPMCSs) according to the following model:

a) An IPMCS consists of interconnected *devices.*

b) A *device* is an *instance* of a corresponding device *type*.

c) The functional capabilities of a *device type* are described in terms of its associated *resources.*

d) A *resource* is an *instance* of a corresponding resource *type*.

e) The functional capabilities of a *resource type* are described in terms of the *function block types* which can be *instantiated,* and the particular *function block instances* which exist, in all *instances* of the resource *type.*

The *configuration* of an IPMCS is thus considered to consist of the *configuration* of its associated *devices* and *applications,* including the allocation of *function block instances* in each *application* to the *resources* associated with the *devices*. This clause defines the following sets of rules to support this process:

- Rules for the functional specification of *types* of *resources* and *devices* are defined in 7.2.
- Rules for the *configuration* of an IPMCS in terms of its associated *devices* and *applications* are defined in 7.3.

7.2 Functional specification of resource and device types

7.2.1 Functional specification of resource types

The functional specification of a *resource type* includes:

- the resource *type name*;
- the *instance name, data type*, and initialization of each of the resource *parameters*;
- a declaration of the *data types* and *function block types* that each *instance* of the *resource type* is capable of *instantiating*;
- the instance names, types, and initial values of any function block instances that are always present in each instance of the resource type;
- any *data connections, adapter connections* and *event connections* that are always present in each instance of the resource type;
- any *access paths* that are always present in each instance of the resource type.

NOTE 1 Additional information may be supplied with resource type specifications, including:

– the maximum numbers of *data connections, adapter connections* and *event connections* that can exist in an instance of the resource type;

– the time (identified as "T_{alg}" in Figure 7) required for *execution* of each *algorithm* of function blocks of a specified type in an instance of the resource;

– the maximum number of instances of specified function block types that can exist in each instance of the resource;

– trade-offs among function block instances, for example, whether two instances of function block type "A" may be traded for one instance of type "B", etc.

NOTE 2 The functional specifications of a resource's communication and process *interfaces*, including the kind and degree of compliance to applicable standards, is beyond the scope of this part of IEC 61499 except as such interfaces are represented by *service interface function blocks.*

7.2.2 Functional specification of device types

The functional specification of a *device type* includes:

a) the device *type name*;

b) the *instance name, data type*, and initialization of each of the device *parameters*;

c) the instance name, type name, and initialization of each *function block instance* that is always present in each *instance* of the device type;

d) any *data connections, adapter connections* and *event connections* that are always present in each instance of the device type;

e) declarations of the *resource instances* which are present in each instance of the device type. Each such declaration shall contain:

   1) the resource instance name and type name;

   2) the instance name, type name, and initialization of each *function block instance* that is always present in the resource instance in each instance of the device type;

   3) any *data connections, adapter connections* and *event connections* that are always present in the resource instance in each instance of the device type;

   4) any *access paths* that are always present in the resource instance in each instance of the device type.

NOTE 1 Items (2), (3) and (4) above are considered to be in addition to the corresponding elements declared in the resource type specification as defined in 7.1.1.

NOTE 2 The functional specifications of a device's communication and process *interfaces*, including the kind and degree of compliance to applicable standards, is beyond the scope of this document except as such interfaces are represented by *service interface function blocks*.

NOTE 3 A device type can contain a function block network only when it is considered to consist of a single (undeclared) resource; in such a case the device type shall not contain any declarations of resource instances.

7.3 Configuration requirements

This subclause defines rules for the *configuration* of industrial process measurement and control *systems, devices, resources*, and *applications*.

7.3.1 Configuration of systems

The configuration of a *system* includes:

- the *name* of the system;
- the specification of each *application* in the system, as specified in 7.3.2;
- the configuration of each *device* and its associated *resources*, as specified in 7.3.3;
- the configuration of each *network segment* and its associated *links* to devices or resources, as specified in 7.3.4.

7.3.2 Specification of applications

The specification of an *application* consists of:

- its name in the form of an *identifier*;
- the *instance name, type name, data connections, event connections* and *adapter connections* of each *function block* and *subapplication* in the application.

It shall be an error if the name of an application is not unique within the scope of the *system*.

7.3.3 Configuration of devices and resources

The configuration of a *device* consists of:

a) the *instance name* and *type name* of the device;

b) configuration-specific values for the device *parameters*;

c) the *resource types* supported by the device *instance* in addition to those specified for the device *type*;

d) the *instance name* and *type name* of each *function block instance* that is present in the device instance in addition to those defined for the device type;

e) any *data connections, adapter connections* and *event connections* that are present in the device instance in addition to those defined for the device type;

f) the *resource types* supported by the device *instance* in addition to those specified for the device *type*;

g) the configuration of each of the *resources* in the device. These consist of any resource instances defined in the device *type* specification, plus any additional resources associated with the specific device *instance*.

NOTE A device instance can contain a function block network only when it is considered to consist of a single (undeclared) resource; in such a case the declaration of the device instance shall not contain any declarations of resource instances.

It shall be an **error** if the instance name of each device is not unique within the scope of the system.

The configuration of a *resource* consists of:

1) its *instance name* and *type name*;

2) the *data types* and *function block types* supported by the resource *instance;*

3) the *instance name, type name*, and initialization of each function block instance that is present in the resource instance;

4) any *data connections, event connections* and *adapter connections* that are present in the resource instance;

5) any *access paths* that are present in the resource instance.

Resource configuration is subject to the following rules:

- The name of a function block *instance* allocated to a *resource* from an application shall consist of the application name concatenated to the function block instance name within the application with a period (`"."`), for example `APP1.FB2`. If *subapplications* are instantiated in the application, the subapplication instance name (or names, if more than one subapplication is nested) shall be inserted in the concatenation before the respective function block instance name.
- Items (2), (3), (4) and (5) above are considered to be in addition to the corresponding elements declared in the device and resource type specifications as defined in subclauses 7.1.2 and 7.1.1, respectively.
- Items (3) and (4) include *function block instances, data connections, adapter connections* and *event connections* from those portions of *applications* allocated to the resource.
- Items (3) and (4) include *communication function blocks, data connections, event connections* and *adapter connections* as necessary to establish and maintain the data and event flows for any associated *applications*.

- The items in Item (3) may include the *mapping* of function block instances in the application to function block instances existing in the resource as a result of type definition as described in 7.1.1.
- It shall be an error if:
  - the instance name of a resource is not unique within the scope of the device containing it;
  - any function block instance in an application is not allocated to exactly one resource.

Automated means may be provided to meet the above requirements. Providers of such means shall either provide unambiguous rules by which their operation can be determined, or shall provide means by which the results of the application of such means can be examined and modified.

7.3.4 Configuration of network segments and links

The configuration of a *network segment* consists of:

- the *instance name* and *type name* of the segment;
- configuration-specific values for the *parameters* of the network segment.

It shall be an error if the *instance name* of each network segment is not unique within the scope of the *system*.

The configuration of a *link* consists of:

- the name of a *device* or the hierarchical name of a "communication *resource*" inside a device, and the name of the network segment to which the device or the resource is connected;
- configuration-specific values for the *parameters* of the link.

Annex A
(normative)

Event function blocks

Instances of the function block *types* shown in Table A.1 can be used for the generation and processing of *events* in *composite function blocks*; in *subapplications*; in the definition of *resource* and *device types*; and in the *configuration* of *applications*, *resources* and *devices*.

Those function block types shown in this Annex which utilize *execution control charts* are *basic function block types*. Where textual declarations of *algorithms* are given for these function block types, the language used is the Structured Text (ST) language defined in IEC 61131-3.

Reference implementations for some of the function block types in this Annex are given as *composite function block type* definitions. These implementations are normative only in the sense that the functional behaviors of compliant implementations shall be equivalent to those of the reference implementation, where the following considerations apply to the timing parameters defined in 4.5.3:

- The parameters **Tsetup**, **Tstart** and **Tfinish** are considered to be zero (0) for all *component function blocks* in the reference implementation.
- The parameter **Talg** is considered to be equal to the parameter `DT` for all instances of `E_DELAY` type used as *component function blocks* in the reference implementation, and to be zero (0) for all other component function blocks in the reference implementation.

All other function block types given in this Annex are *service interface function block types*.

NOTE Full textual specifications of all function block types shown in Table A.1 are given in Annex F.

Table A.1 – Event function blocks

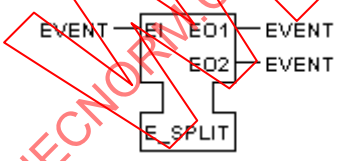
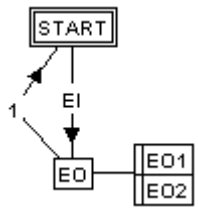
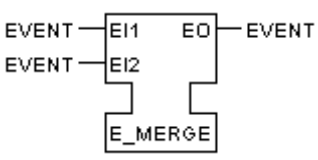
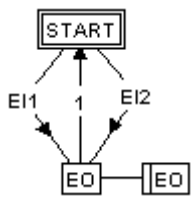
<table>
<tr><th>No.</th><th colspan="2">Description</th></tr>
<tr><th colspan="2">Interface</th><th>ECC/Algorithms/Service sequences</th></tr>
<tr><td>1</td><td colspan="2">Split an event</td></tr>
<tr><td colspan="2"> 
</td><td> 
</td></tr>
<tr><td colspan="3">The occurrence of an event at <code>EI</code> causes the occurrence of events at <code>EO1, EO2,...,EOn</code> (<code>n=2</code> in the above example).</td></tr>
<tr><td>2</td><td colspan="2">Merge (OR) of multiple events</td></tr>
<tr><td colspan="2"> 
</td><td> 
</td></tr>
<tr><td colspan="3">The occurrence of an event at any of the inputs <code>EI1, EI2,...,EIn</code> causes the occurrence of an event at <code>EO</code> (<code>n=2</code> in the above example).</td></tr>
</table>

Table A.1 *(continued)*

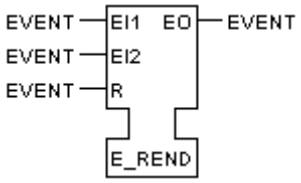
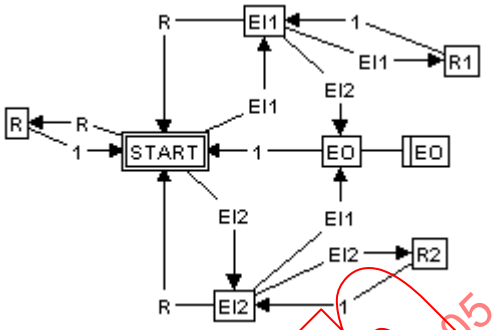
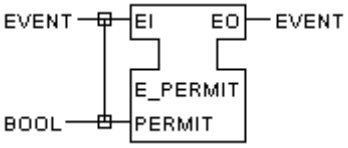
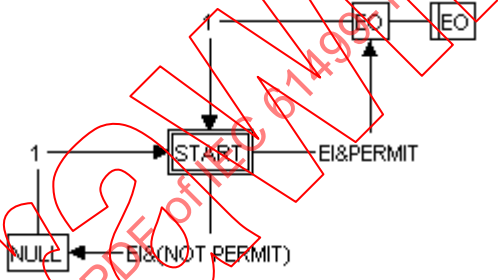
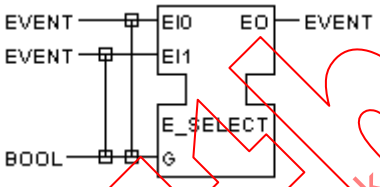
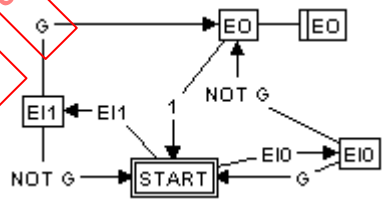
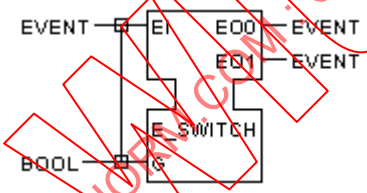
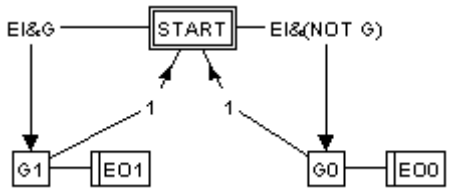
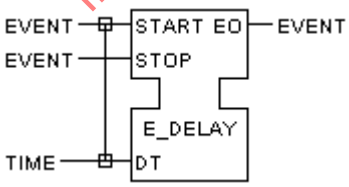
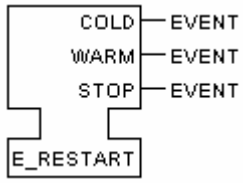
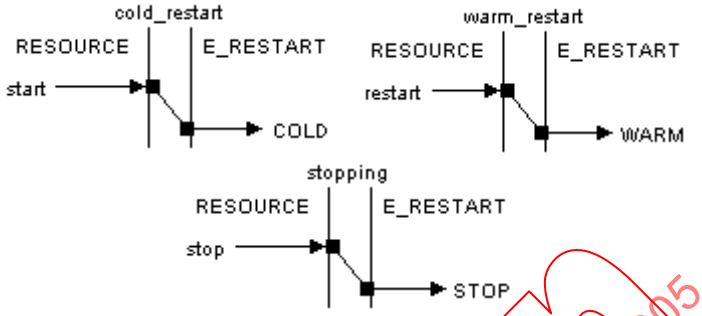
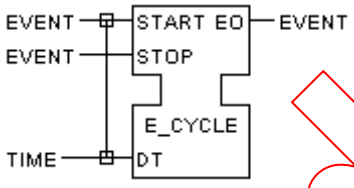
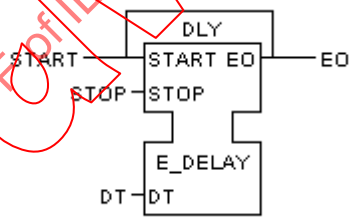
3	**Rendezvous of two events**
	
4	**Permissive propagation of an event**
	
5	**Selection between two events**
	
6	**Switching (demultiplexing) an event**
	
7	**Delayed propagation of an event**
	An event at EO is generated at a time interval DT after the occurrence of an event at the START input. The event delay is cancelled by an occurrence of an event at the STOP input. If multiple events occur at the START input before the occurrence of an event at EO, only a single event occurs at EO, at a time DT after the first event occurrence at the START input. No event delay will be initiated if an event occurs at the START input with a value of DT which is not greater than t#0s.

Table A.1 *(continued)*

8	**Generation of restart events**
	
1) An event is issued at the COLD output upon "cold restart" of the associated resource. 2) An event is issued at the WARM output upon "warm restart" of the associated resource. 3) An event is issued at the STOP output (if possible) prior to "stopping" of the associated resource. NOTE 1 See IEC 61131-3 for a discussion of "cold restart" and "warm restart".	

9	**Periodic (cyclic) generation of an event**
	
An event occurs at EO at an interval DT after the occurrence of an event at START, and at intervals of DT thereafter until the occurrence of an event at STOP.	

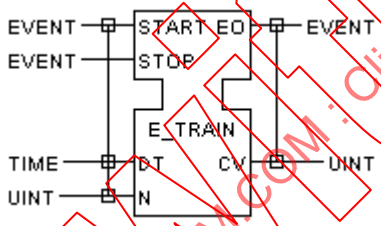
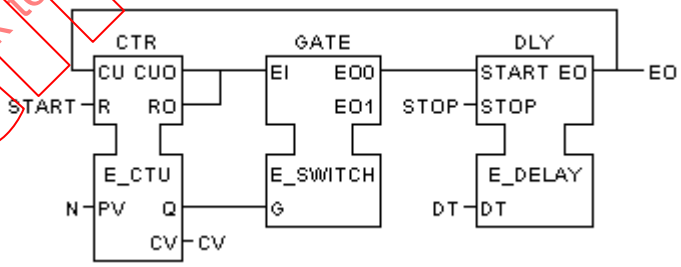
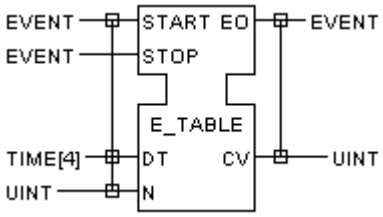
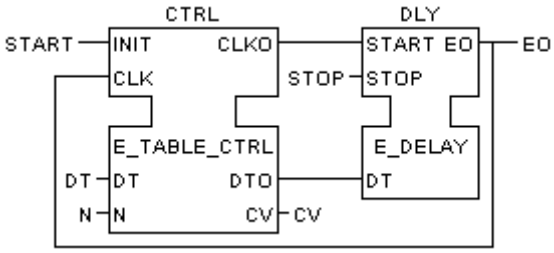
10	**Generation of a finite train of events**
	
NOTE See table entry #18 for a definition of the E_CTU type.	
An event occurs at EO at an interval DT after the occurrence of an event at START, and at intervals of DT thereafter, until N occurrences have been generated or an event occurs at the STOP input. NOTE 2 The count CV is reset whenever an event occurs at the START interface, but the delay does not restart unless it is already stopped. This behavior maintains the inter-EO interval when restarting the count.	

Table A.1 *(continued)*

<table>
<tr><td>11</td><td colspan="2">Generation of a finite train of events (table driven)</td></tr>
<tr><td colspan="2"> 
</td><td> 
</td></tr>
<tr><td colspan="3">An event occurs at EO at an interval DT[0] after the occurrence of an event at EI. A second event occurs at an interval DT[1] after the first, etc., until N occurrences have been generated or an event occurs at the STOP input. The current event count is maintained at the CV output.

NOTE 3 In this example implementation, N <= 4.

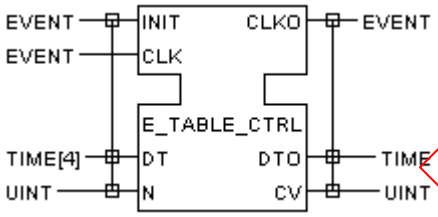
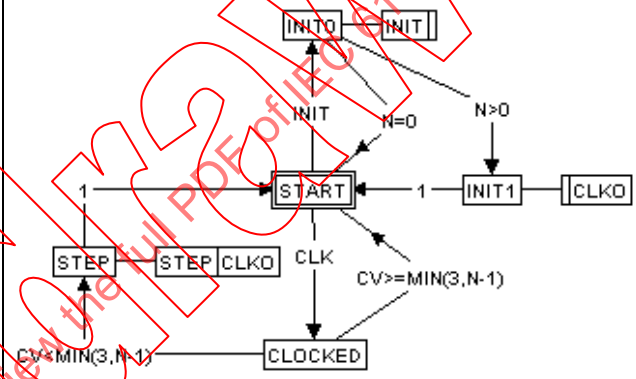
NOTE 4 Implementation using the E_TABLE_CTRL function block type illustrated below is not a normative requirement. Equivalent functionality may be implemented by various means.</td></tr>
<tr><td colspan="2"> 
</td><td> 
</td></tr>
<tr><td><pre>ALGORITHM INIT IN ST:
 CV := 0 ;
 DTO := DT[0];
END_ALGORITHM</pre></td><td><pre>ALGORITHM STEP IN ST:
 CV := CV+1;
 DTO := DT[CV];
END_ALGORITHM</pre></td></tr>
</table>

Table A.1 *(continued)*

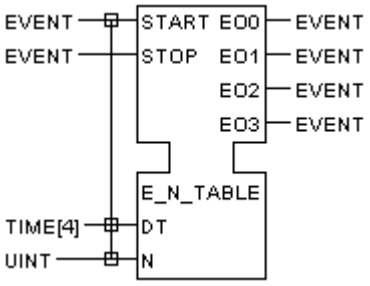
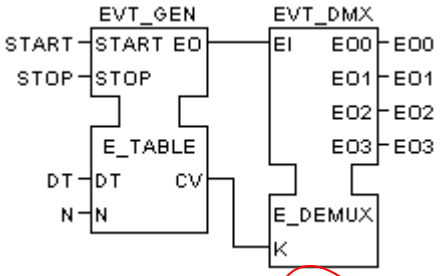
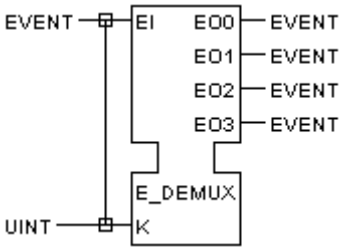
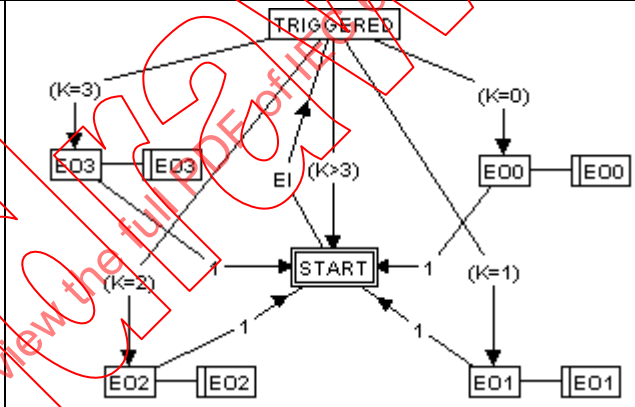
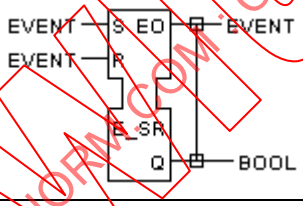
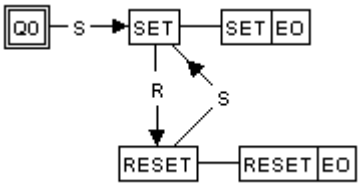
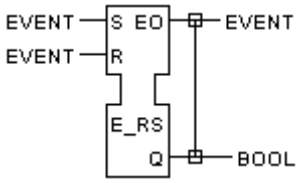
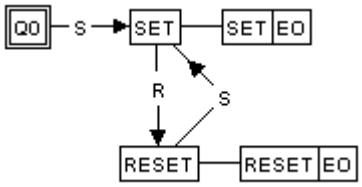
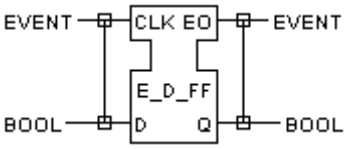
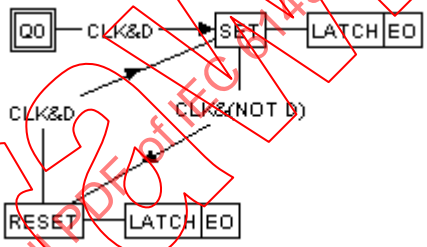
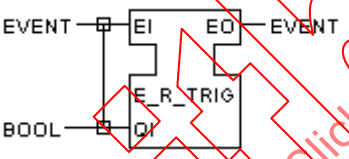
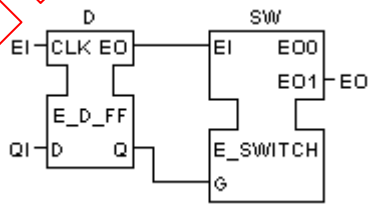
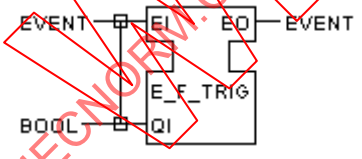
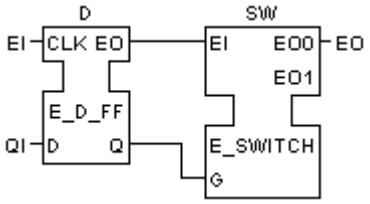
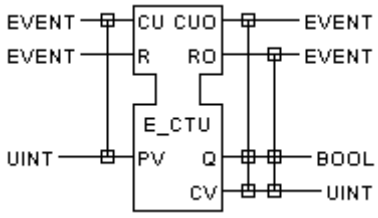
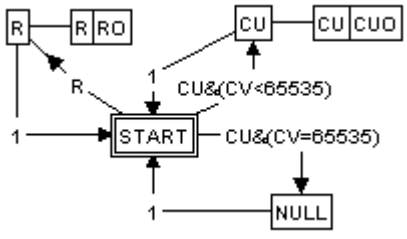
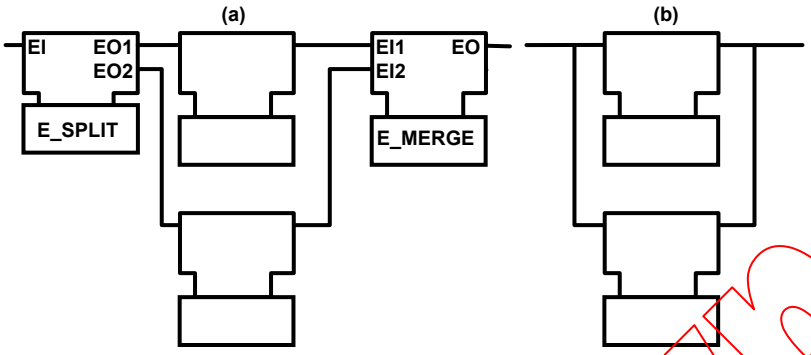
12	Generation of a finite train of separate events (table driven)
	
An event occurs at EO0 at an interval DT[0] after the occurrence of an event at EI. An event occurs at EO2 an interval DT[1] after the occurrence of the event at EO1, etc., until N occurrences have been generated or an event occurs at the STOP input. NOTE 5 In this example implementation, N <= 4. NOTE 6 Implementation using the E_DEMUX function block type illustrated below is not a normative requirement. Equivalent functionality may be implemented by various means.	
	
13	**Event-driven bistable**
	
The output Q is set to 1 (TRUE) upon the occurrence of an event at the S input, and is reset to 0 (FALSE) upon the occurrence of an event at the R input. An event is issued at the EO output when the value of Q changes.	
ALGORITHM SET IN ST : (* Set Q *) Q := TRUE ; END_ALGORITHM	ALGORITHM RESET IN ST : (* Reset Q *) Q := FALSE ; END_ALGORITHM

Table A.1 *(continued)*

14	Event-driven bistable
	
The output Q is set to 1 (TRUE) upon the occurrence of an event at the S input, and is reset to 0 (FALSE) upon the occurrence of an event at the R input. An event is issued at the EO output when the value of Q changes. NOTE 7 The implementation of this function block type is identical to E_SR. Both E_SR and E_RS are implemented for consistency with the SR and RS types of IC 61131-3, although there is no "dominance" of events as there would be for level-controlled R and S inputs.	
15	**D (Data latch) bistable**
 ALGORITHM LATCH IN ST : Q := D ; END_ALGORITHM	
16	**Boolean rising edge detection**
	
17	**Boolean falling edge detection**
	
18	**Event-driven Up Counter**
	
ALGORITHM R IN ST: (* Reset *) CV := 0 ; Q := 0 ; END_ALGORITHM	ALGORITHM CU IN ST: (* Count Up *) CV := CV+1; Q := (CV = PV); END_ALGORITHM

Graphical shorthand notations may be substituted for the E_SPLIT and E_MERGE blocks defined in Table A.1. For example, the shorthand (implicit) representation shown in Figure A.1(b) is equivalent to the explicit representation in Figure A.1(a).



Key

a) Explicit representation

b) Implicit representation

NOTE Irrelevant detail is suppressed in the above figure.

Figure A.1 – Event split and merge

Annex B
(normative)

Textual syntax

B.1 Syntax specification technique

The textual constructs in this Annex are specified in terms of a *syntax*, which specifies the allowable combinations of symbols which can be used to define a program; and a set of *semantics*, which specify the meanings of the symbol combinations defined by the syntax.

A syntax is defined by a set of *terminal symbols* to be utilized for program specification; a set of *non-terminal symbols* defined in terms of the terminal symbols; and a set of *production rules* specifying those definitions.

The terminal symbols for textual specifications of entities defined in this part of IEC 61499 consist of combinations of the characters in the "ISO-646 IRV" given as Table 1 – Row 00 of ISO/IEC 10646.

For the purposes of this part of IEC 61499, terminal textual symbols consist of the appropriate character string enclosed in paired single or double quotes. For example, a terminal symbol represented by the character string ABC can be represented by either `"ABC"` or `'ABC'`.

This allows the representation of strings containing either single or double quotes; for instance, a terminal symbol consisting of the double quote itself would be represented by `'"'`

A special terminal symbol utilized in this syntax is the "null string", that is, a string containing no characters. This is represented by the terminal symbol `NIL`

Non-terminal textual symbols are represented by strings of lower-case letters, numbers, and the underline character (_), beginning with a lower-case letter. For instance, the strings `nonterm1` and `non_term_2` are valid nonterminal symbols, while the strings `3nonterm` and `_nonterm4` are not.

The production rules given in this part of IEC 61499 form an *extended grammar* in which each rule has the form:

```
non_terminal_symbol ::= extended_structure
```

This rule can be read as:

"A `non_terminal_symbol` can consist of an `extended_structure`."

Extended structures can be constructed according to the following rules:

a) The null string, `NIL`, is an extended structure.

b) A terminal symbol is an extended structure.

c) A non-terminal symbol is an extended structure.

d) If `S` is an extended structure, then the following expressions are also extended structures:

- `(S)`, meaning `S` itself.
- `{S}`, *closure*, meaning zero or more concatenations of `S`.
- `[S]`, *option*, meaning zero or one occurrence of `S`.

e) If S1 and S2 are extended structures, then the following expressions are extended structures:

- S1 | S2, *alternation*, meaning a choice of S1 or S2.
- S1 S2, *concatenation*, meaning S1 followed by S2.

f) Concatenation *precedes* alternation, that is, S1 | S2 S3 is equivalent to S1 | (S2 S3), and S1 S2 | S3 is equivalent to (S1 S2) | S3.

Semantics are defined in this Specification by appropriate natural language text, accompanying the production rules, which references the descriptions provided in the appropriate clauses. Standard options available to the user and vendor are specified in these semantics.

In some cases it is more convenient to embed semantic information in an extended structure. In such cases, this information is delimited by paired angle brackets, for example, <semantic information>.

B.2 Function block and subapplication type specification

B.2.1 Function block type specification

The syntax defined in this subclause can be used for the textual specification of *function block types* according to the rules given in Clauses 5 and 6 of this part of IEC 61499.

SYNTAX:

```
fb_type_declaration ::=
     'FUNCTION_BLOCK' fb_type_name
       fb_interface_list
       [fb_internal_variable_list] <only for basic FB>
       [fb_instance_list] <only for composite FB>
       [plug_list]
       [socket_list]
       [fb_connection_list] <only for composite FB>
       [fb_ecc_declaration] <only for basic FB>
       {fb_algorithm_declaration} <only for basic FB>
       [fb_service_declaration] <only for service interface FB>
     'END_FUNCTION_BLOCK'

fb_interface_list ::=
     [event_input_list]
     [event_output_list]
     [input_variable_list]
     [output_variable_list]

event_input_list ::=
     'EVENT_INPUT'
       {event_input_declaration}
     'END_EVENT'

event_output_list ::=
     'EVENT_OUTPUT'
       {event_output_declaration}
     'END_EVENT'

event_input_declaration ::= event_input_name [ ':' event_type ]
     ['WITH' input_variable_name {',' input_variable_name}] ';'

event_output_declaration ::= event_output_name [ ':' event_type ]
     ['WITH' output_variable_name {',' output_variable_name}] ';'
```

```
input_variable_list ::=
      'VAR_INPUT' {input_var_declaration ';'} 'END_VAR'

output_variable_list ::=
     'VAR_OUTPUT' {output_var_declaration ';'} 'END_VAR'

fb_internal_variable_list ::=
     'VAR' {internal_var_declaration ';'} 'END_VAR'

input_var_declaration ::=
     input_variable_name {',' input_variable_name} ':' var_spec_init

output_var_declaration ::=
     output_variable_name {',' output_variable_name} ':' var_spec_init

internal_var_declaration ::=
     internal_variable_name {',' internal_variable_name}
       ':' var_spec_init

var_spec_init ::= located_var_spec_init <as specified in IEC 61131-3>

fb_instance_list ::= 'FBS'
     {fb_instance_definition ';'}
     'END_FBS'

fb_instance_definition ::= fb_instance_name ':' fb_type_name [parameters]

plug_list ::= 'PLUGS'
     {plug_name ':' adapter_type_name [parameters] ';'}
     'END_PLUGS'

socket_list ::= 'SOCKETS'
     {socket_name ':' adapter_type_name [parameters] ';'}
     'END_SOCKETS'

fb_connection_list ::= <may be empty, e.g. for basic FB>
     [event_conn_list]
     [data_conn_list]
     [adapter_conn_list]

event_conn_list ::=
     'EVENT_CONNECTIONS'
       {event_conn}
     'END_CONNECTIONS'

event_conn ::=
     ((fb_instance_name '.' event_output_name)
             | (plug_name '.' event_input_name))
       'TO' ((fb_instance_name '.' event_input_name)
             | (plug_name '.' event_output_name)) ';'
     | event_input_name 'TO'
              ((fb_instance_name '.' event_input_name)
             | (plug_name '.' event_output_name)) ';'
     | ((fb_instance_name '.' event_output_name)
             | (plug_name '.' event_input_name))
       'TO' event_output_name ';'

data_conn_list ::=
     'DATA_CONNECTIONS'
       {data_conn}
     'END_CONNECTIONS'
```

```
data_conn ::=
     ( fb_instance_name '.' output_variable_name
             | plug_name '.' input_variable_name
             | input_variable_name)
       'TO' ((fb_instance_name '.' input_variable_name)
             | (plug_name '.' output_variable_name)) ';'
     | ((fb_instance_name '.' output_variable_name)
             | (plug_name '.' input_variable_name))
       'TO' output_variable_name ';'


adapter_conn_list ::=
     'ADAPTER_CONNECTIONS'
       {adapter_conn}
     'END_CONNECTIONS'


adapter_conn ::=
     ((fb_instance_name '.' plug_name ) | socket_name)
     'TO' ((fb_instance_name '.' socket_name ) | plug_name) ';'


fb_ecc_declaration ::=
     'EC_STATES'
       {ec_state} <first state is initial state>
     'END_STATES'
     'EC_TRANSITIONS'
       {ec_transition}
     'END_TRANSITIONS'


ec_state ::= ec_state_name
     [':' ec_action {',' ec_action}] ';'


ec_action ::= algorithm_name | '->' event_output_name
     | algorithm_name '->' event_output_name


ec_transition      ::=
     ec_state_name
     'TO' ec_state_name
     ':=' ec_transition_condition ';'


ec_transition_condition ::= event_input_name
     | guard_condition
     | event_input_name ('&' | 'AND')
       (guard_variable | '(' guard_condition ')')


guard_variable ::= input_variable_name
     | output_variable_name
     | internal_variable_name
     <Shall be of BOOL type>


guard_condition ::= expression
     <as defined in IEC 61131-3>
     <Shall evaluate to a BOOL value>


fb_algorithm_declaration ::=
     'ALGORITHM' algorithm_name 'IN' language_type ':'
       algorithm_body
     'END_ALGORITHM'


algorithm_body ::= <as defined in compliant standards>


fb_service_declaration ::=
     'SERVICE' service_interface_name '/' service_interface_name
       {service_sequence}
     'END_SERVICE'
```

```
service_interface_name ::= fb_type_name | 'RESOURCE'

service_sequence ::=
     'SEQUENCE' sequence_name
       {service_transaction ';'}
     'END_SEQUENCE'

service_transaction ::=
       [input_service_primitive] '->' output_service_primitive
             {'->' output_service_primitive}

input_service_primitive ::= service_interface_name '.'
     ([plug_name '.'] event_input_name
        | socket_name '.' event_output_name)
     ['+' | '-']
     '(' [input_variable_name {',' input_variable_name}] ')'

output_service_primitive ::= service_interface_name '.' ('NULL' |
     ([plug_name '.'] event_output_name
        | socket_name '.' event_input_name)
     ['+' | '-']
     '(' [output_variable_name {',' output_variable_name}] ')')

algorithm_name ::= identifier

ec_state_name ::= identifier

event_input_name ::= identifier

event_output_name ::= identifier

event_type ::= identifier

fb_instance_name ::= identifier

fb_type_name ::= identifier

input_variable_name ::= identifier

internal_variable_name ::= identifier

language_type ::= identifier

output_variable_name ::= identifier

plug_name ::= identifier

sequence_name ::= identifier

socket_name ::= identifier
```

B.2.2 Subapplication type specification

The syntax defined in this subclause can be used for the textual specification of *subapplication types* according to the rules given in Clause 5.

The productions given in B.2.1 also apply to this subclause.

SYNTAX:

```
subapplication_type_declaration ::=
     'SUBAPPLICATION' subapp_type_name
          subapp_interface_list
          [fb_instance_list]
          [subapp_instance_list]
          [plug_list]
          [socket_list]
          [subapp_connection_list]
     'END_SUBAPPLICATION'


subapp_interface_list ::=
     [subapp_event_input_list]
     [subapp_event_output_list]
     [input_variable_list]
     [output_variable_list]
     subapp_event_input_list ::=
     'EVENT_INPUT'
       {subapp_event_input_declaration}
     'END_EVENT'


subapp_event_output_list ::=
     'EVENT_OUTPUT'
       {subapp_event_output_declaration}
     'END_EVENT'


subapp_event_input_declaration ::=
     event_input_name [ ':' event_type ] ';'


subapp_event_output_declaration ::=
     event_output_name [ ':' event_type ] ';'


subapp_instance_list ::= 'SUBAPPS'
     {subapp_instance_definition ';'}
     'END_SUBAPPS'


subapp_instance_definition ::= subapp_instance_name ':' subapp_type_name


subapp_connection_list ::=
     [subapp_event_conn_list]
     [subapp_data_conn_list]
     [adapter_conn_list]


subapp_event_conn_list ::=
     'EVENT_CONNECTIONS'
       {subapp_event_conn}
     'END_CONNECTIONS'


subapp_event_conn ::=
     (fb_subapp_name '.' event_output_name
       'TO' fb_subapp_name '.' event_input_name ';')
     | (event_input_name 'TO' fb_subapp_name '.' event_input_name ';')
     | (fb_subapp_name '.' event_output_name 'TO' event_output_name ';')


fb_subapp_name ::= fb_instance_name | subapp_instance_name


subapp_data_conn_list ::=
     'DATA_CONNECTIONS'
       {subapp_data_conn}
     'END_CONNECTIONS'
```

```
subapp_data_conn ::=
     ((( fb_subapp_name '.' output_variable_name )
       | input_variable_name )
       'TO' fb_subapp_name '.' input_variable_name ';')
     | ((fb_subapp_name '.' output_variable_name)
       'TO' output_variable_name ';')

subapp_type_name ::= identifier

subapp_instance_name ::= identifier
```

B.3 Configuration elements

The syntax defined in this subclause can be used for the textual specification of *resource types*, *device types*, *applications*, and *system configurations* according to the rules given in Clause 7.

The productions given in Clause B.2 also apply to this subclause.

SYNTAX:

```
application_configuration ::=
     'APPLICATION' application_name
       [fb_instance_list]
       [subapp_instance_list]
       [subapp_connection_list]
     'END_APPLICATION'

system_configuration ::= 'SYSTEM' system_name
      {application_configuration}
      device_configuration
      {device_configuration}
      [mappings]
      [segments]
      [links]
     'END_SYSTEM'

segments ::= 'SEGMENTS'
      segment
      {segment}
     'END_SEGMENTS'

segment ::= segment_name ':' segment_type_name [parameters] ';'

links ::= 'LINKS'
      link
      {link}
     'END_LINKS'

link ::= resource_hierarchy '=>' segment_name [parameters] ';'

parameters ::= '(' parameter {',' parameter} ')'

parameter ::= parameter_name ':='
     (constant | enumerated_value | array_initialization |
     structure_initialization) ';'
     <as defined in IEC 61131-3>
```

```
device_configuration ::=
     'DEVICE' device_name ':' device_type_name [parameters]
       [resource_type_list]
       {resource_configuration}
       [fb_instance_list]
       [devtype_connection_list]
     'END_DEVICE'

resource_type_list ::= 'RESOURCE_TYPES'
       {resource_type_name ';'}
     'END_RESOURCE_TYPES'

resource_configuration ::=
     'RESOURCE' resource_instance_name ':' resource_type_name [parameters]
       [fb_type_list]
       [fb_instance_list]
       [config_connection_list]
       [access_paths]
     'END_RESOURCE'

fb_type_list ::= 'FB_TYPES' {fb_type_name ';'} 'END_FB_TYPES'

config_connection_list ::=
     [config_event_conn_list]
     [config_data_conn_list]
     [config_adapter_conn_list]

config_event_conn_list ::= 'EVENT_CONNECTIONS'
       {config_event_conn}
     'END_CONNECTIONS'

config_event_conn ::= fb_instance_name '.' event_output_name
       'TO' fb_instance_name '.' event_input_name ';'

config_data_conn_list ::= 'DATA_CONNECTIONS'
       {config_data_conn}
     'END_CONNECTIONS'

config_data_conn ::=
     (fb_instance_name '.' output_variable_name)
       'TO' fb_instance_name '.' input_variable_name ';'

config_adapter_conn_list ::= 'ADAPTER_CONNECTIONS'
       {config_adapter_conn}
     'END_CONNECTIONS'

config_adapter_conn ::= fb_instance_name '.' plug_name
     'TO' fb_instance_name '.' socket_name ';'

access_paths ::= 'VAR_ACCESS'
     access_path_declaration {access_path_declaration}
     'END_VAR'

access_path_declaration ::=
     access_path_name ':' access_path [access_direction] ';'

access_path ::=
     fb_instance_reference {'.' fb_instance_name} '.' symbolic_variable
      <symbolic_variable is defined in IEC 61131-3, B.1.4>

fb_instance_reference ::= [app_hierarchy_name] fb_instance_name
```

```
app_hierarchy_name := application_name '.'{subapp_instance_name '.'}

access_direction ::= 'READ_ONLY' | 'READ_WRITE'
     <default is READ_ONLY>
     <READ_WRITE only applies to unconnected input variables or internal
     variables>

device_type_specification ::=
     'DEVICE_TYPE' device_type_name
       [input_variable_list]
       [resource_type_list] <if not given, defined by resource instances>
       {resource_instance}
       [fb_instance_list]
       [devtype_connection_list]
       [devtype_access_paths]
     'END_DEVICE_TYPE'

devtype_connection_list ::=
     [config_event_conn_list]
     [devtype_data_conn_list]
     [config_adapter_conn_list]

devtype_data_conn_list ::=
     'DATA_CONNECTIONS'
       {devtype_data_conn}
     'END_CONNECTIONS'

devtype_data_conn ::=
     (fb_instance_name '.' output_variable_name
           | input_variable_name | constant)
       'TO' fb_instance_name '.' input_variable_name ';'

devtype_access_paths ::= 'VAR_ACCESS'
     {devtype_access_path_declaration}
     'END_VAR'

devtype_access_path_declaration ::=
     access_path_name ':' devtype_access_path [access_direction] ';'

devtype_access_path ::=
     fb_instance_name ['.' fb_instance_name] '.' symbolic_variable
     <symbolic_variable is defined in IEC 61131-3, B.1.4>

resource_instance ::=
     'RESOURCE' resource_instance_name ':' resource_type_name
       [fb_instance_list]
       [devtype_connection_list]
       [devtype_access_paths]
     'END_RESOURCE'

resource_type_specification ::= 'RESOURCE_TYPE' resource_type_name
       [input_variable_list]
       [fb_type_list] <if not given, defined by function block instances>
       [fb_instance_list]
       devtype_connection_list
       [devtype_access_paths]
     'END_RESOURCE_TYPE'

mappings ::= 'MAPPINGS' mapping {mapping} 'END_MAPPINGS'

mapping ::= fb_instance_reference 'ON' fb_resource_reference ';'

fb_resource_reference = resource_hierarchy ['.' fb_instance_name]
```

```
resource_hierarchy ::= device_name ['.' resource_instance_name]

segment_name ::= identifier

segment_type_name ::= identifier

parameter_name ::= identifier

system_name ::= identifier

device_name::= identifier

device_type_name::= identifier

application_name ::= identifier

resource_instance_name::= identifier

resource_type_name::= identifier

access_path_name ::= identifier
```

B.4 Common elements

Where syntactic productions are not given for non-terminal symbols in this Annex, the syntactic productions and corresponding semantics given in Annex B of IEC 61131-3 shall apply.

B.5 Supporting productions for management commands

The syntax defined in this subclause is referenced in Table 8.

SYNTAX:

```
data_type_list ::= 'DATA_TYPES' {data_type_name ';'} 'END_DATA_TYPES'

connection_definition ::=
     connection_start_point ' ' connection_end_point

connection_start_point ::= fb_instance_reference '.' attachment_point

connection_end_points ::=
     connection_end_point {',' connection_end_point}

connection_end_point ::= fb_instance_reference '.' attachment_point

attachment_point ::= identifier

access_path_data ::= access_path_name ':' accessed_data

accessed_data ::= data_element {',' data_element}

data_element ::= constant | enumerated_value | structure_initialization
     | array_initialization

all_data_types ::= 'ALL_DATA_TYPES'
```

```
all_fb_types ::= 'ALL_FB_TYPES'

fb_status ::= 'IDLE' | 'RUNNING' | 'STOPPED' | 'KILLED'
```

B.6 Tagged data types

SYNTAX:

```
tagged_type_declaration ::=
     'TYPE'
       asn1_tag type_declaration ';'
       {asn1_tag type_declaration ';'}
     'END_TYPE'

asn1_tag ::= '[' ['APPLICATION' | 'PRIVATE'] (integer | hex_integer) ']'
```

SEMANTICS

These productions shall be used for the assignment of tags as defined in ISO/IEC 8824-1 to derived data types defined as specified in this Annex and in Annex E. As defined in ISO/IEC 8824-1, the class tags `APPLICATION` and `PRIVATE` shall be used except for types to be used only in context-specific tagging.

B.7 Adapter interface types

SYNTAX:

```
adapter_type_declaration ::=
     'ADAPTER' adapter_type_name
          fb_interface_list
     'END_ADAPTER'

adapter_type_name ::= identifier
```

SEMANTICS: See 5.5.

Annex C
(informative)

Object models

C.1 Model notation

This Annex presents object models for some of the classes which may be used in Engineering Support Systems (ESS) to support the design, implementation, commissioning and operation of Industrial-Process Measurement and Control Systems (IPMCSs) constructed according to the architecture defined in this part of IEC 61499.

The notation used in this Annex is the Unified Modeling Language (UML). References to extensive documentation of this notation can be found on the Internet at the Uniform Resource Locator (URL) **http://www.omg.org/uml/**.

C.2 ESS Models

C.2.1 General

Figure C.1 presents an overview of the major classes in the ESS (Engineering Support System) for an industrial-process measurement and control system (IPMCS), and their correspondence to the classes of objects in the IPMCS. Descriptions of the classes in Figure C.1 are given in Table C.1.

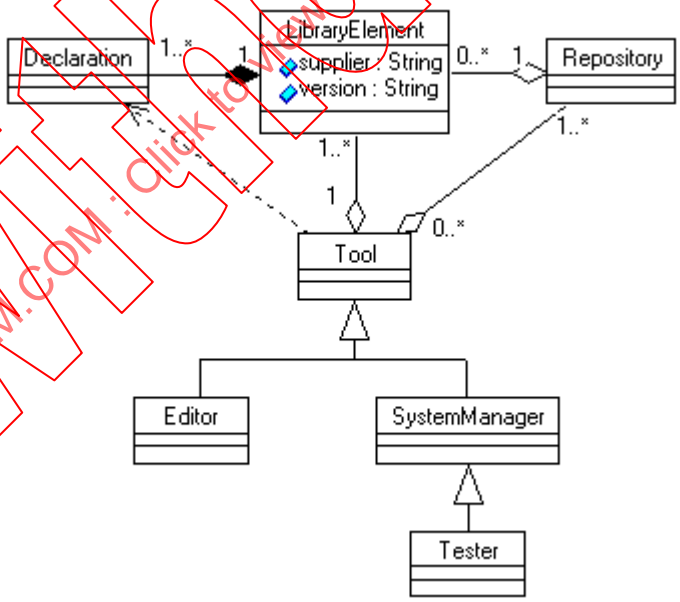


Figure C.1 – ESS overview

Table C.1 – ESS class descriptions

Declaration	This is the abstract superclass for *declarations*.
Editor	Instances of this class provide the editing functions on *declarations* necessary to support the EDIT use case.
LibraryElement	This is the abstract superclass of objects which may be stored in repositories and which may be imported and exported in the textual syntax defined in Annex B, or the XML syntax defined in IEC 61499-2. Such objects have supplier (vendor, programmer, etc.) and version (version number, date, etc.) attributes to assist in management, in addition to a name (inherited from NamedDeclaration – see C.2.3) as a key attribute.
Repository	Instances of this class provide persistent storage and retrieval of library elements. They may also provide version control services.
SystemManager	Instances of this class provide the functions necessary to support the INSTALL and OPERATE use cases.
Tester	This class extends the capabilities of the SystemManager class to support the operations of the TEST use case.
Tool	This class models the generic behaviors of *software tools* for engineering support of IPMCSs.

C.2.2 Library elements

The subclasses of LibraryElement are shown in Figure C.2. The syntactic production in Annex B corresponding to each subclass is listed in Table C.2.

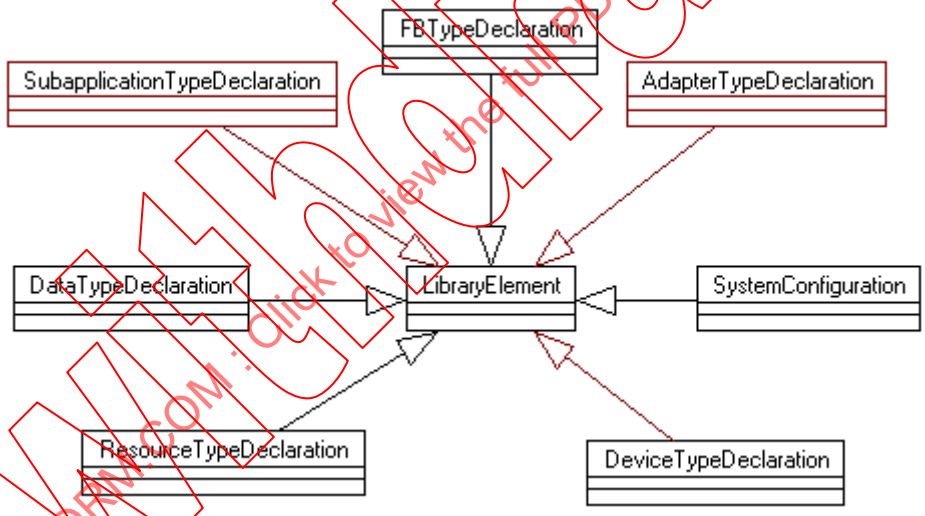


Figure C.2 – Library elements

Table C.2 – Syntactic productions for library elements

Class	Syntactic production
DataTypeDeclaration	type_declaration
FBTypeDeclaration	fb_type_declaration
AdapterTypeDeclaration	adapter_type_declaration
SubapplicationTypeDeclaration	subapp_type_declaration
ResourceTypeDeclaration	resource_type_specification
DeviceTypeDeclaration	device_type_specification
SystemConfiguration	system_configuration

C.2.3 Declarations

Figure C.3 shows the class hierarchy of *declarations* which may be manipulated by *software tools*. The syntactic productions in Annex B corresponding to each of these subclasses are listed in Table C.3.

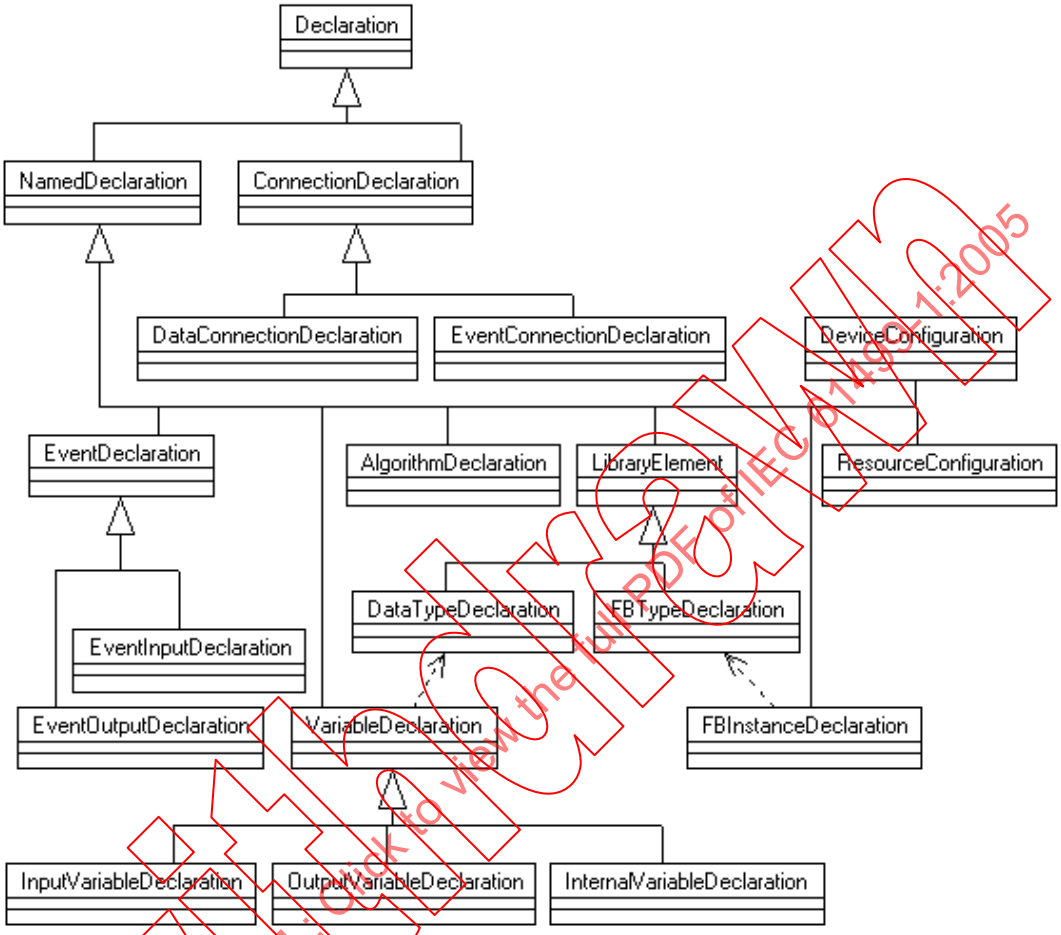


Figure C.3 – Declarations

Table C.3 – Syntactic productions for declarations

Class	Syntactic production
AlgorithmDeclaration	fb_algorithm_declaration
DataConnectionDeclaration	data_conn
DeviceConfiguration	device_configuration
EventConnectionDeclaration	event_conn
EventInputDeclaration	event_input_declaration
EventOutputDeclaration	event_output_declaration
FBInstanceDeclaration	fb_instance_definition
InputVariableDeclaration	input_var_declaration
InternalVariableDeclaration	internal_var_declaration
OutputVariableDeclaration	output_var_declaration
ResourceConfiguration	resource_instance

C.2.4 Function block network declarations

Figure C.4 shows the relationships among the elements of *function block network declarations*. See C.2.3 for definitions of the aggregated classes in this diagram.

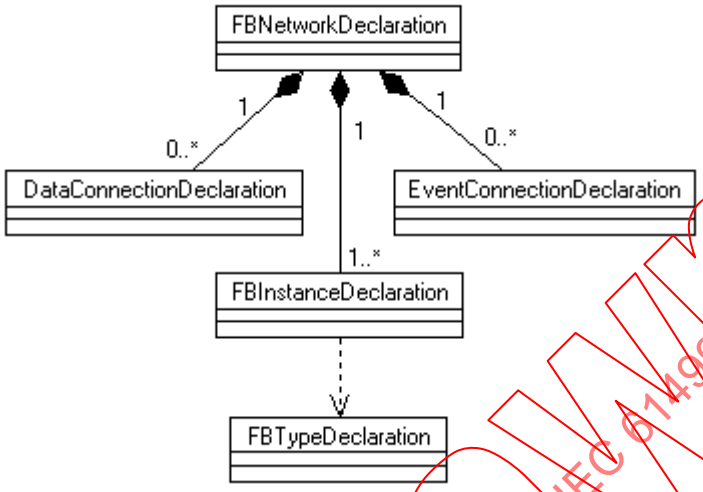


Figure C.4 – Function block network declarations

C.2.5 Function block type declarations

Figure C.5 shows the relationships among the elements of *function block type declarations*. Syntactic productions for the classes EventInputDeclaration, EventOutputDeclaration, InputVariableDeclaration, OutputVariableDeclaration, InternalVariableDeclaration, and the component classes of FBNetworkDeclaration are given in Table C.3. The syntactic productions `fb_ecc_declaration` and `fb_service_declaration` in Clause B.2 correspond to classes ECCDeclaration and ServiceDeclaration, respectively.

NOTE 1 Declarations of *subapplications* are represented by instances of the class BasicFBTypeDeclaration which contain no event `WITH` data associations.

NOTE 2 NamedDeclaration is the abstract superclass of declarations which have names, for example, *type names* or *instance names*.

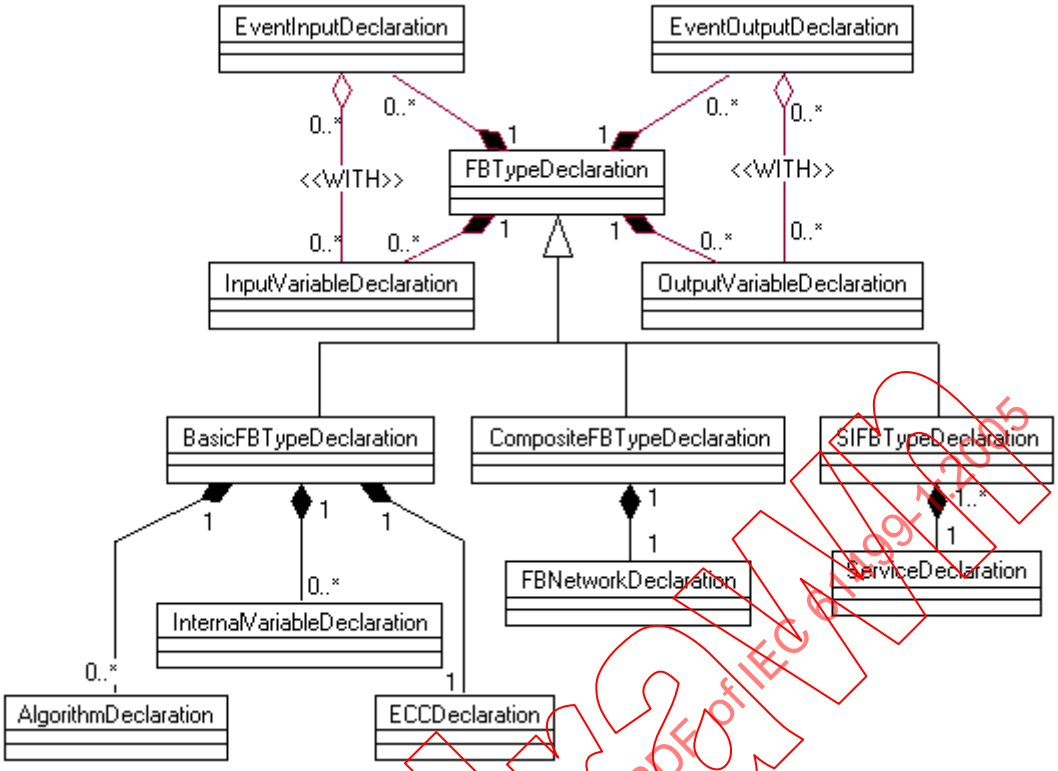


Figure C.5 – Function block type declarations

C.3 IPMCS models

Figure C.6 presents an overview of the major classes in the industrial-process measurement and control system (IPMCS). Descriptions of the classes in Figure C.6 and their corresponding objects in the Engineering Support System (ESS) are given in Table C.4.

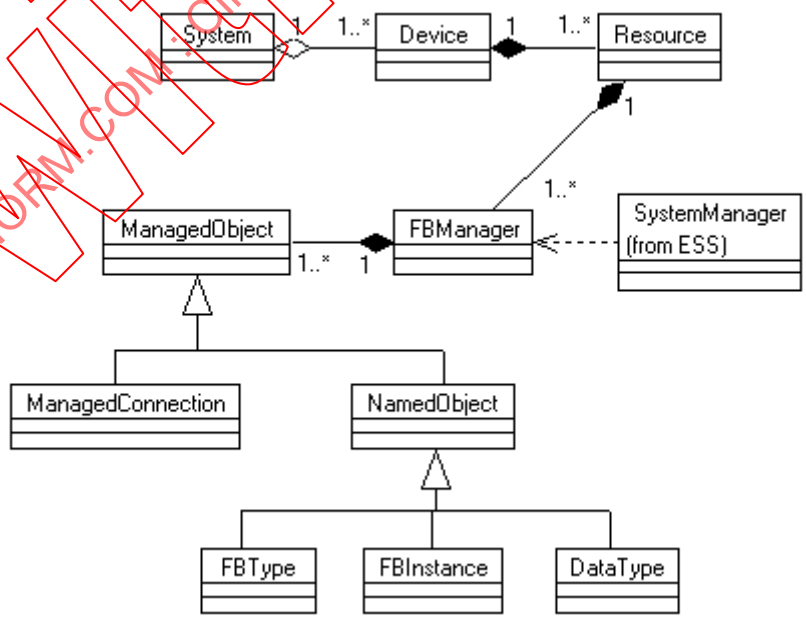


Figure C.6 – IPMCS overview

Table C.4 – IPMCS classes

IPMCS class	Description	Corresponding ESS class
DataType	An instance of this class is a *data type*.	**DataTypeDeclaration**
Device	An instance of this class represents a *device*.	**DeviceConfiguration**
FBInstance	An instance of this class is a *function block instance*.	**FBInstanceDeclaration**
FBManager	An instance of this class provides the management services defined in Clause 6.	**SystemManager**
FBType	An instance of this class is a *function block type*.	**FBTypeDeclaration**
ManagedConnection	Instances of this class can be accessed by an instance of the **FBManager** class using the source and destination combination as a unique key.	**ConnectionDeclaration**
ManagedObject	This is the abstract superclass of objects which are managed by an instance of the **FBManager** class. Such objects may have supplier (vendor, programmer, etc.) and version (version number, date, etc.) attributes to assist in management.	none
NamedObject	This is the abstract superclass of objects which can be accessed by name by an instance of the **FBManager** class.	**NamedDeclaration**
Resource	An instance of this class represents a *resource*.	**ResourceConfiguration**
System	An instance of this class represents an Industrial-Process Measurement and Control *System* (IPMCS).	**SystemConfiguration**

Figure C.7 shows the relationships among the elements of a *function block instance* and its associated *function block type*.

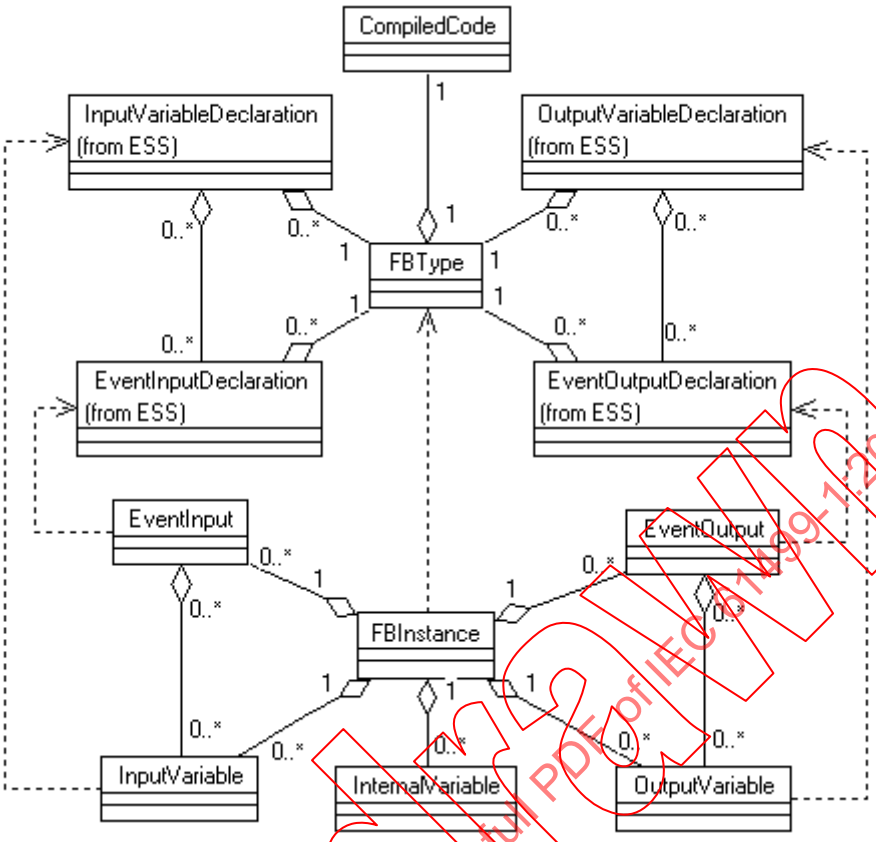


Figure C.7 – Function block types and instances

Annex D
(informative)

Relationship to IEC 61131-3

D.1 General

Functions and *function blocks* as defined in IEC 61131-3 can be used for the *declaration* of *algorithms* in *basic function block types* as specified in Clause 5. Clause D.2 defines rules for the conversion of IEC 61131-3 *functions* and *function block types* into *simple function block types* so that they can be used in the specification of *applications* and *resource types*. Clause D.3 defines event-driven versions of IEC 61131-3 functions and function blocks for the same uses.

D.2 "Simple" function blocks

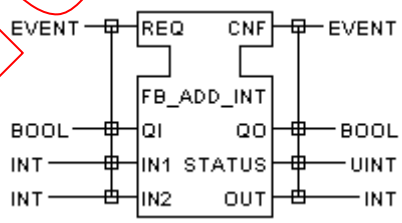
As illustrated in Figure D.1, IEC 61131-3 functions and function blocks can be converted to "simple" function blocks according to the following rules:

a) Simple function blocks are represented as *service interface function blocks* for application-initiated interactions as shown in Figure 21(a).

b) The *type name* of the simple function block type is the name of the converted IEC 61131-3 function or function block type with the prefix `FB_` (for instance, `FB_ADD_INT` in Figure D.1).

c) The *input* and *output variables* and their corresponding *data types* are the same as the corresponding input and output variables of the converted IEC 61131-3 function or function block type.

d) The `INIT` event input and `INITO` event output are used with simple function block types that have been converted from IEC 61131-3 *function block types*, and are not used with simple function block types that have been converted from IEC 61131-3 *functions*.



NOTE A complete textual declaration of this function block type is given in Annex F.

Figure D.1 – Example of a "simple" function block type

The behavior of *instances* of simple function block *types* is according to the following rules:

1) Initialization is as specified in 2.4.2 of IEC 61131-3 for *variables*, and as specified in 2.6 of IEC 61131-3 for Sequential Function Chart (SFC) elements.

2) The occurrence of an INIT+ service primitive is equivalent to "cold restart" initialization as defined in the above mentioned subclauses of IEC 61131-3, followed by an INITO+ service primitive with a STATUS value of zero (0).

3) The occurrence of an INIT– or REQ– service primitive has no effect except to cause an INITO– or CNF– service primitive, respectively, with a STATUS value of one (1).

4) The occurrence of a REQ+ service primitive causes the *execution* of the *algorithm* specified in the function block body, according to the rules given in IEC 61131-3 for the language in which the algorithm is programmed.

5) Successful execution of the algorithm in response to a REQ+ primitive results in a CNF+ primitive with a STATUS value of zero (0).

6) If an error occurs during the execution of the algorithm, the result is a CNF– primitive with a STATUS value determined according to Table D.1.

Table D.1 – Semantics of STATUS values

Value	Semantics
0	Normal operation
1	INIT- or REQ- propagation
2	Type conversion error
3	Numerical result exceeds range for data type
4	Division by zero
5	Selector (K) out of range for MUX function
6	Invalid character position specified
7	Result exceeds maximum string length
8	Simultaneously true, non-prioritized transitions in a selection divergence
9	Action control contention error
10	Return from function without value assigned
11	Iteration fails to terminate

D.3 Event-driven functions and function blocks

IEC 61131-3 *functions* can be converted into function blocks for efficient use in event-driven systems according to the rules given in Clause D.2 with the following modifications:

a) The *type name* of the event-driven function block type is the same as the name of the converted IEC 61131-3 function with the additional prefix E_, for example, E_ADD_INT.

b) A CNF+ or CNF– primitive does not follow execution of the algorithm unless such execution results in a changed value of the function output.

NOTE If "daisy-chaining" of CNF outputs to REQ inputs is used to implement a sequence of calculations, then the sequence will stop at the first point where an output value does not change.

In general, since IEC 61131-3 *function blocks* have internal state information, such blocks should be specially converted for use in event-driven systems. For instance, the E_DELAY function block shown in Table A.1 can be used for many of the delay functions provided by the timer function blocks in IEC 61131-3. An example of a conversion of the standard IEC 61131-3 CTU function block is given as Feature 18 of Table A.1.

D.4 Compliance with IEC 61131-3

Implementations of this Specification shall comply with the requirements of 1.5.1, 2.1, 2.2, 2.3 and 2.4 and the associated elements of Annex B of IEC 61131-3, for the syntax and semantics of textual representation of common elements, with the exceptions and extensions noted below.

Where syntactic productions are not given for non-terminal symbols in Annex B of this part of IEC 61499, the corresponding syntactic productions given in Annex B of IEC 61131-3 shall apply.

D.5 Exceptions

Implementations of this Specification shall not utilize the *directly represented variable* notation defined in 2.4.1.1 of IEC 61131-3 and related features in other subclauses. However, a *literal* of `STRING` or `WSTRING` type, containing a string whose syntax and semantics correspond to the directly represented variable notation, may be used as a *parameter* of a *service interface function block* which provides access to the corresponding variable.

Annex E
(informative)

Information exchange

NOTE The contents of this Annex may be considered normative in that other Standards may specify that compliance to its provisions is required, and industrial-process measurement and control *systems* and *devices* that comply with its provisions may claim such compliance.

E.1 Use of application layer facilities

Subclause 7.1.3.2 of ISO/IEC 7498-1 identifies a number of facilities provided by *application-entities* (i.e., *entities* in the *application layer*) to enable *application-processes* to exchange information. To provide these facilities, the application-entities use *application-protocols* and *presentation services.* These facilities are provided by the *communication mapping function* of *resources*. This Clause discusses the ways in which communication function blocks may use these facilities, when provided by appropriate application-entities.

NOTE 1 See ISO/IEC 7498-1 for definitions of terms used in this Clause but not defined in this part of IEC 61499.

NOTE 2 A *resource* is an "application-process" as defined in ISO/IEC 7498-1.

NOTE 3 Many of the facilities listed below are not provided by application-entities of industrial-process measurement and control systems (IPMCSs). In this case, the communication function blocks should implement equivalent facilities to provide the required services.

NOTE 4 In particular, presentation services are often not provided by IPMCS application-entities. Therefore, in order to facilitate implementation of these services by communication function blocks, this part of IEC 61499 defines transfer syntaxes for both information transfer and application management in Clause E.3.

0) information transfer,c) synchronization of cooperating applications: Communication function blocks utilize the information transfer facilities provided by application-entities to provide the synchronization represented by the `REQ`, `CNF`, `IND`, and `RSP` events and to transfer the data represented by the `SD` inputs and `RD` outputs.

a) identification of the intended communications partners, b) determination of the acceptable quality of service, d) agreement on responsibility for error recovery, e) agreement on security aspects g) identification of abstract syntax: These facilities may be used during service initialization as represented by the `INIT` and `INITO` events, using elements of the `PARAMS` data structure as necessary.

f) selection of mode of dialog: These facilities may be used by the specific function block types , for example, by a `SUBSCRIBER` to assure that it is interacting properly with a `PUBLISHER`.

NOTE The numbering above corresponds top that of ISO/IEC 7498-1, subclause 7.1.3.2.

E.2 Communication function block types

E.2.1 General

This subclause defines generic *communication function block types* for *unidirectional* and *bidirectional transactions.* Implementation-dependent customizations of these types should adhere to the following rules:

a) The implementation shall specify the data types and semantics of values of the data inputs and data outputs of each such function block type.

b) The implementation shall specify the treatment of abnormal data transfer.

c) The implementation shall specify any differences between the behavior of instances of such function block types and the behaviors specified in this clause.

E.2.2 Function blocks for unidirectional transactions

Figures E.1 to E.4 provide type declarations and typical service primitive sequences of function blocks which provide *unidirectional transactions* over a *communication connection*. Such a connection consists of one *instance* of `PUBLISH` and one or more instances of `SUBSCRIBE` type.

NOTE 1 Full textual specifications of these function block types are not given in Annex F.

NOTE 2 The data types and semantics of the `PARAMS` input and `STATUS` output are implementation-dependent.

NOTE 3 The number (m) and types of the received data `RD_1, . . . , RD_m` correspond to the number and types of the transmitted data `SD_1, . . . , SD_m`.

NOTE 4 The means by which communication connections are set up are beyond the scope of this part of IEC 61499.

NOTE 5 Data transfer may be required in order to determine whether the required constraints on `RD_1, . . . , RD_m` are met per Note 3.

NOTE 6 The transfer syntaxes defined in E.3 may be used to make the determination described above.

NOTE 7 Treatment of abnormal data transfer is implementation-dependent.

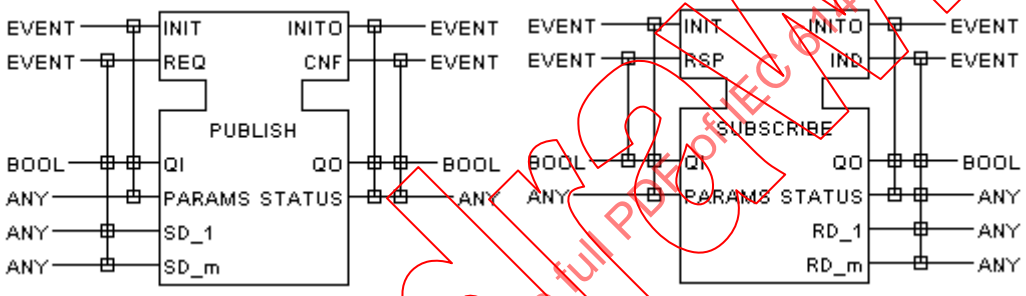


Figure E.1 – Type specifications for unidirectional transactions

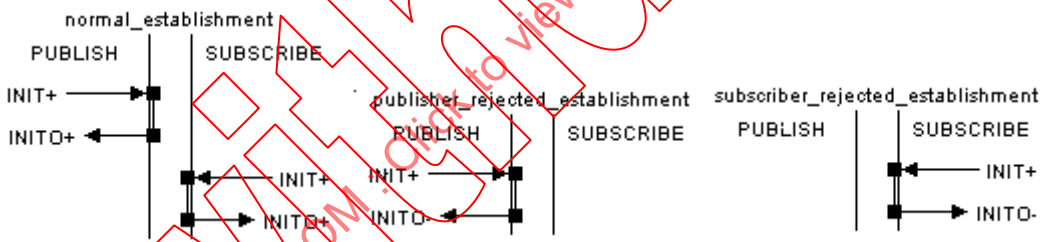


Figure E.2 – Connection establishment for unidirectional transactions

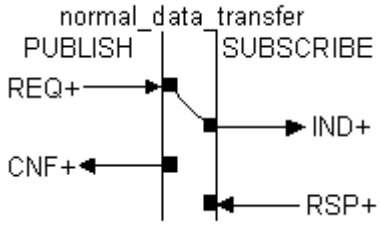


Figure E.3 – Normal unidirectional data transfer

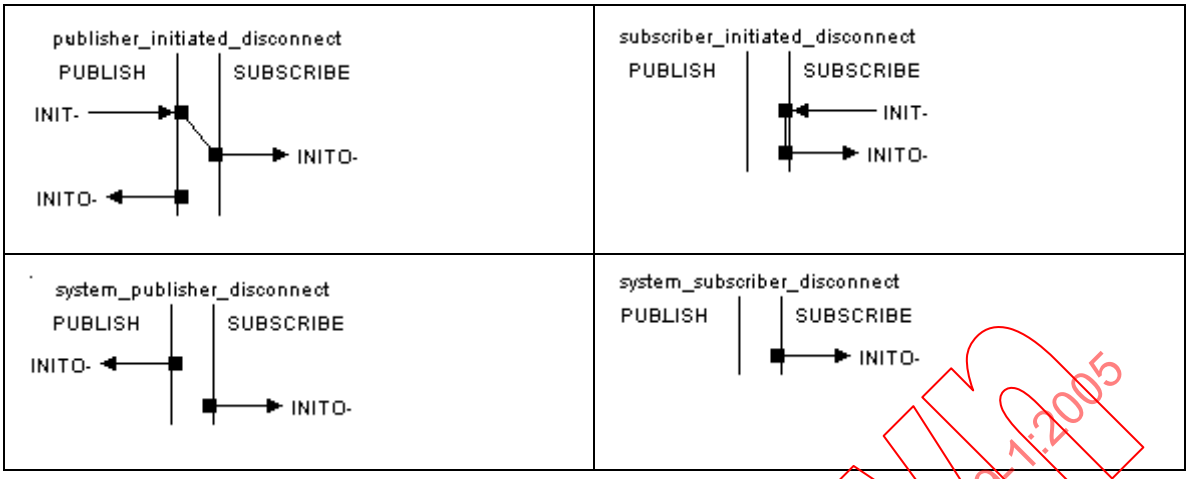


Figure E.4 – Connection release in unidirectional data transfer

E.2.3 Function blocks for bidirectional transactions

Figures E.5 through E.8 provide type declarations and service primitive sequences of function blocks which provide *bidirectional transactions* over *a communication connection*. Such a connection consists of one instance of CLIENT type and one instance of SERVER type.

NOTE 1 Full textual specifications of these function block types are not given in Annex F.

NOTE 2 The data types and semantics of the `PARAMS` input and `STATUS` output are implementation-dependent.

NOTE 3 The number (`m`) and types of the received data `RD_1, . . . , RD_m` correspond to the number and types of the transmitted data `SD_1, . . . , SD_m`.

NOTE 4 The number (`n`) and types of the received data `RD_1, . . . , RD_n` correspond to the number and types of the transmitted data `SD_1, . . . , SD_n`.

NOTE 5 Data transfer may be required in order to determine whether the required constraints on `RD_1, . . . , RD_m` and `RD_1, . . . , RD_n` are met per Note 3 of Figure E.5.

NOTE 6 The transfer syntaxes defined in Clause E.3 may be used to make the determination described above.

NOTE 7 Treatment of abnormal data transfer is implementation-dependent.

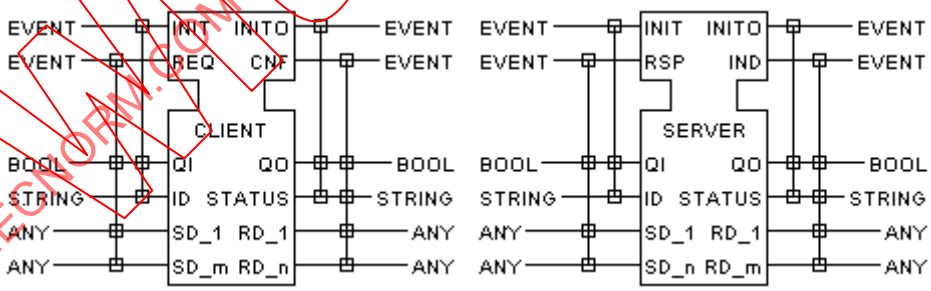


Figure E.5 – Type specifications for bidirectional transactions