IEC 61499-1

Edition 2.0 2012-11

INTERNATIONAL STANDARD

NORME INTERNATIONALE

colour inside

Function blocks – Part 1: Architecture

Blocs fonctionnels – Partie 1: Architecture

IEC 61499-1:2012

IEC Central Office
3, rue de Varembé
CH-1211 Geneva 20
Switzerland

Tel.: +41 22 919 02 11
Fax: +41 22 919 03 00
info@iec.ch
www.iec.ch

About the IEC

The International Electrotechnical Commission (IEC) is the leading global organization that prepares and publishes International Standards for all electrical, electronic and related technologies.

About IEC publications

The technical content of IEC publications is kept under constant review by the IEC. Please make sure that you have the latest edition, a corrigenda or an amendment might have been published.

Useful links:

IEC publications search - www.iec.ch/searchpub

The advanced search enables you to find IEC publications by a variety of criteria (reference number, text, technical committee,…).
It also gives information on projects, replaced and withdrawn publications.

IEC Just Published - webstore.iec.ch/justpublished

Stay up to date on all new IEC publications. Just Published details all new publications released. Available on-line and also once a month by email.

Electropedia - www.electropedia.org

The world's leading online dictionary of electronic and electrical terms containing more than 30 000 terms and definitions in English and French, with equivalent terms in additional languages. Also known as the International Electrotechnical Vocabulary (IEV) on-line.

Customer Service Centre - webstore.iec.ch/csc

If you wish to give us your feedback on this publication or need further assistance, please contact the Customer Service Centre: csc@iec.ch.

A propos de la CEI

La Commission Electrotechnique Internationale (CEI) est la première organisation mondiale qui élabore et publie des Normes internationales pour tout ce qui a trait à l'électricité, à l'électronique et aux technologies apparentées.

A propos des publications CEI

Le contenu technique des publications de la CEI est constamment revu. Veuillez vous assurer que vous possédez l'édition la plus récente, un corrigendum ou amendement peut avoir été publié.

Liens utiles:

Recherche de publications CEI - www.iec.ch/searchpub

La recherche avancée vous permet de trouver des publications CEI en utilisant différents critères (numéro de référence, texte, comité d'études,…).
Elle donne aussi des informations sur les projets et les publications remplacées ou retirées.

Just Published CEI - webstore.iec.ch/justpublished

Restez informé sur les nouvelles publications de la CEI. Just Published détaille les nouvelles publications parues. Disponible en ligne et aussi une fois par mois par email.

Electropedia - www.electropedia.org

Le premier dictionnaire en ligne au monde de termes électroniques et électriques. Il contient plus de 30 000 termes et définitions en anglais et en français, ainsi que les termes équivalents dans les langues additionnelles. Egalement appelé Vocabulaire Electrotechnique International (VEI) en ligne.

Service Clients - webstore.iec.ch/csc

Si vous désirez nous donner des commentaires sur cette publication ou si vous avez des questions contactez-nous: csc@iec.ch.

IEC 61499-1

Edition 2.0 2012-11

INTERNATIONAL STANDARD

NORME INTERNATIONALE

colour inside

Function blocks – Part 1: Architecture

Blocs fonctionnels – Partie 1: Architecture

INTERNATIONAL ELECTROTECHNICAL COMMISSION

COMMISSION ELECTROTECHNIQUE INTERNATIONALE

PRICE CODE
CODE PRIX XF

ICS 25.040; 35.240.50

ISBN 978-2-83220-481-8

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 itself does not provide any attestation of conformity. Independent certification bodies provide conformity assessment services and, in some areas, access to IEC marks of conformity. IEC is not responsible for any services carried out by independent certification bodies.
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 subcommittee 65B: Measurement and control devices, of IEC technical committee 65: Industrial-process measurement, control and automation.

This second edition cancels and replaces the first edition published in 2005. This edition constitutes a technical revision.

This edition includes the following significant technical changes with respect to the previous edition:

- *Execution control* in basic function blocks (5.2) has been clarified and extended:
  - Dynamic and static parts of the EC transition condition are clearly delineated by using the `ec_transition_event[guard_condition]` syntax of the Unified Modeling Language (UML) (5.2.1.3, B.2.1).
  - The terminology "crossing of an EC transition" (3.10) is used preferentially to "clearing" to avoid the misinterpretation that the entire transition condition corresponds to a Boolean variable that can be "cleared."

  - Operation of the ECC state machine in 5.2.2.2 has been clarified and made more rigorous.
  - Event and data outputs of adapter instances (plugs and sockets) can be used in EC transition conditions, and event inputs of adapter instances can be used as EC action outputs.
- *Temporary variables* (3.97) can be declared (B.2.1) and used in algorithms of basic function blocks.
- *Service sequences* (6.1.3) can now be defined for basic and composite function block types and adapter types, as well as service interface types.
- The syntax for *mapping* of FB instances from applications to resources has been simplified (Clause B.3).
- Syntax for definition of *segment types* (7.2.3) for network segments of system configurations has been added (Clause B.3).
- Function block types for interoperation with programmable controllers are defined (Clause D.6).
- The READ/WRITE management commands (Table 8) now apply only to *parameters*.

The text of this part of IEC 61499 is based on the following documents:

FDIS	Report on voting
65B/845/FDIS	65B/855/RVD

Full information on the voting for the approval of this standard can be found in the report on voting indicated in the above table (when voting is completed).

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

A list of all parts of the IEC 61499 series can be found, under the general title *Function blocks*, on the IEC website.

Terms used throughout this International Standard that have been defined in Clause 3 appear in *italics*.

The committee has decided that the contents of this publication will remain unchanged until the stability 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.

IMPORTANT – The 'colour inside' logo on the cover page of this publication indicates that it contains colours which are considered to be useful for the correct understanding of its contents. Users should therefore print this document using a colour printer.

INTRODUCTION

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

- Part 1 (this document) 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.
- Part 2 defines requirements for *software tools* to support the following systems engineering tasks:
  - 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*.
- Part 3 (Tutorial information) has been withdrawn due to the widespread current availability of tutorial and educational materials regarding IEC 61499. However, an updated 2nd Edition of Part 3 may be developed in the future.
- 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*.

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 standard may be restricted by the functional capabilities of compliant systems, subsystems, and devices.

Part of the purpose of this standard 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 standard 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 standard be concise, implementable, complete, unambiguous, and consistent.

NOTE 1 The provisions of this standard alone are not sufficient to ensure interoperability among devices of different vendors. Standards complying with this part of IEC 61499 can specify additional provisions to ensure such interoperability.

NOTE 2 Standards complying with this part of IEC 61499 can specify additional provisions to enable the performance of *system*, *device*, *resource* and *application* management *functions*.

2 Normative references

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

IEC 61131-1, *Programmable controllers – Part 1: General*

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

IEC/ISO 7498-1:1994, *Information technology – Open systems interconnection – Basic reference model: The basic model*

ISO/IEC 8824-1:2008, *Information technology – Abstract Syntax Notation One (ASN.1): Specification of basic notation*

ISO/IEC 10646:2003, *Information technology – Universal Multiple-Octet Coded Character Set (UCS)*

3 Terms and definitions

For the purposes of this document, the following terms and definitions apply.

NOTE Terms defined in Clause 3 are *italicized* where they appear in definitions and Notes to entry of other terms as well as throughout the body of the document.

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

3.2
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*

3.3
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*

3.4
algorithm
finite set of well-defined rules for the solution of a problem in a finite number of *operations*

3.5
application
software functional unit that is specific to the solution of a problem in industrial-process measurement and control

Note 1 to entry: An application can be distributed among *resources*, and might communicate with other applications.

3.6
attribute
property or characteristic of an *entity*, for instance, the version identifier of a *function block type* specification

3.7
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*

3.8
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

3.9
character
member of a set of elements that is used for the representation, organization, or control of *data*

3.10
crossing
clearing
<of an EC transition> *operation* by means of which control is passed from the predecessor *EC state* of an *EC transition* to its successor EC state

Note 1 to entry: This operation consists of de-activation of the predecessor EC state, followed by activation of the successor EC state.

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 1 to entry: 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 parameter
parameter related to the *configuration* of a *system, device* or *resource*

3.20
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*

3.21
connection
association established between *functional units* for conveying *information*

3.22
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

3.23
data
reinterpretable representation of *information* in a formalized manner suitable for communication, interpretation or processing

3.24
data connection
association between two *function blocks* for the conveyance of *data*

3.25
data input
interface of a *function block* which receives *data* from a *data connection*

3.26
data output
interface of a *function block* which supplies *data* to a *data connection*

3.27
data type
set of values together with a set of permitted *operations*

3.28
declaration
mechanism for establishing the definition of an *entity*

Note 1 to entry: A declaration can involve attaching an *identifier* to the entity, and allocating *attributes* such as *data types* and *algorithms* to it.

3.29
device
independent physical *entity* capable of performing one or more specified *functions* in a particular context and delimited by its *interfaces*

Note 1 to entry: A *programmable controller system* as defined in IEC 61131-1 is a *device*.

3.30
device management application
application whose primary function is the management of multiple *resources* within a *device*

3.31
entity
particular thing, such as a person, place, *process*, object, concept, association, or *event*

3.32
event
instantaneous occurrence that is significant to scheduling the *execution* of an *algorithm*

Note 1 to entry: The execution of an algorithm may make use of *variables* associated with an event.

3.33
event connection
association among *function blocks* for the conveyance of *events*

3.34
event input
interface of a *function block* which can receive *events* from an *event connection*

3.35
event output
interface of a *function block* which can issue *events* to an *event connection*

3.36
exception
event that causes suspension of normal *execution*

3.37
execution
process of carrying out a sequence of *operations* specified by an *algorithm*

Note 1 to entry: 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.

3.38
execution control action
EC action
element associated with an *execution control state,* which identifies an *algorithm* to be *executed,* an *event* to be issued, or both

Note 1 to entry: Timing of algorithm execution and event issuance are addressed in 5.2.2.

3.39
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*

3.40
execution control initial state
EC initial state
execution control state that is active upon initialization of an *execution control chart*

3.41
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*

3.42
execution control transition
EC transition
means by which control passes from a predecessor *execution control state* to a successor *execution control state*

3.43
fault
abnormal condition that may cause a reduction in, or loss of, the capability of a *functional unit* to perform a required *function*

3.44
function
specific purpose of an *entity* or its characteristic action

3.45
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 to entry: Typical operations of a function block include modification of the values of the data in its associated data structure.

Note 2 to entry: 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.

3.46
function block network
network whose nodes are *function blocks* or *subapplications* and their *parameters* and whose branches are *data connections* and *event connections*

Note 1 to entry: This is a generalization of the *function block diagram* defined in IEC 61131-3.

3.47
function block type
type whose *instances* are *function blocks*

Note 1 to entry: Function block types include basic function block types, composite function block types, and service interface function block types

3.48
functional unit
entity of *hardware* or *software,* or both, capable of accomplishing a specified purpose

3.49
hardware
physical equipment, as opposed to programs, procedures, rules and associated documentation

3.50
identifier
one or more *characters* used to name an *entity*

3.51
implementation
development phase in which the *hardware* and *software* of a *system* become operational

3.52
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*

3.53
information
meaning that is currently assigned to *data* by means of the conventions applied to that data

3.54
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 1 to entry: An *input parameter* of a *function block*, as defined in IEC 61131-3, is an *input variable*.

3.55
instance
functional unit comprising an individual, named *entity* with the *attributes* of a defined *type*

3.56
instance name
identifier associated with and designating an *instance*

3.57
instantiation
creation of an *instance* of a specified *type*

3.58
interface
shared boundary between two *functional units,* defined by functional characteristics, signal characteristics, or other characteristics, as appropriate

3.59
internal operation
<of a *function block*> *operation* associated with an *algorithm* of a function block, with its *execution* control, or with the functional capabilities of the associated *resource*

3.60
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*

3.61
invocation
process of initiating the *execution* of the sequence of *operations* specified in an *algorithm*

3.62
link
design element describing the *connection* between a *device* and a *network segment*

3.63
literal
lexical unit that directly represents a value

3.64
management function block
function block whose primary *function* is the management of *applications* within a *resource*

3.65
management resource
resource whose primary *function* is the management of other *resources*

3.66
mapping
set of features or *attributes* having defined correspondence with the members of another set

3.67
message
ordered series of *characters* intended to convey *information*

3.68
message sink
part of a communication *system* in which *messages* are considered to be received

3.69
message source
part of a communication *system* from which *messages* are considered to originate

3.70
model
mathematical or physical representation of a system or a process

3.71
multitasking
mode of operation that provides for the *concurrent execution* of two or more *algorithms*

3.72
network
arrangement of nodes and interconnecting branches

3.73
operation
well-defined action that, when applied to any permissible combination of known *entities*, produces a new *entity*

3.74
output variable
variable whose value is established by one or more *operations* of a *function block,* and is supplied to a *data output*

Note 1 to entry: An *output parameter* of a *function block*, as defined in IEC 61131-3, is an *output variable.*

3.75
parameter
variable that is given a constant value for a specified *application* and that may denote the application

3.76
plug
plug adapter
instance of an *adapter interface type* which provides a starting point for an *adapter connection* from a *provider* function block

3.77
provider
function block instance which provides a *plug adapter* of a defined *adapter interface type*

3.78
request primitive
service primitive which represents an interaction in which an *application* invokes some *algorithm* provided by a *service*

3.79
requester
functional unit which initiates a *transaction* via a *request primitive*

3.80
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 to entry: The RESOURCE defined in IEC 61131-3:2003, 1.3.66 is a programming language element corresponding to the *resource* defined above.

Note 2 to entry: A *device* contains one or more resources.

3.81
resource management application
application whose primary function is the management of a single *resource*

3.82
responder
functional unit which concludes a *transaction* via a *response primitive*

3.83
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*

3.84
sample, verb
to sense and retain the instantaneous value of a *variable* for later use

3.85
scheduling function
function which selects *algorithms* or *operations* for *execution,* and initiates and terminates such execution

3.86
segment
physical partition of a *communication network*

3.87
service
functional capability of a *resource* which can be modeled by a sequence of *service primitives*

3.88
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*

3.89
service primitive
abstract, implementation-independent representation of an interaction between an *application* and a *resource*

3.90
service sequence diagram
diagram representing a sequence of *service primitives*

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 1 to entry: 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 1 to entry: 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

Note 1 to entry: Such elements may be both material objects and concepts as well as the results thereof (e.g. forms of organisation, mathematical methods, and programming languages).

Note 2 to entry: 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
temporary variable
variable whose value is initialized, used and possibly modified during *execution* of an *algorithm*; that is not visible outside the body of the algorithm, and whose value does not persist from one execution of the algorithm to the next

3.98
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

3.99
type
software element which specifies the common *attributes* shared by all *instances* of the type

3.100
type name
identifier associated with and designating a *type*

3.101
unidirectional transaction
transaction in which a request and possibly *data* is/are conveyed from an *requester* to a *responder*, and in which a response is not conveyed from the responder back to the requester

3.102
variable
software entity that may take different values, one at a time

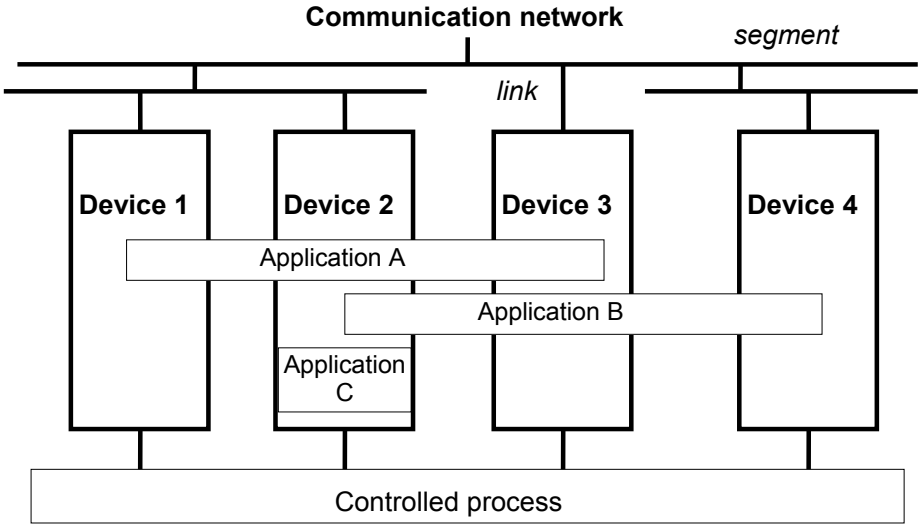
Note 1 to entry: The values of a variable are usually restricted to a certain *data type*.

Note 2 to entry: Variables may be classified as *input variables, output variables*, *internal variables* and *temporary variables*.

4 Reference models

4.1 System model

For the purposes of IEC 61499, 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 7.2.2.

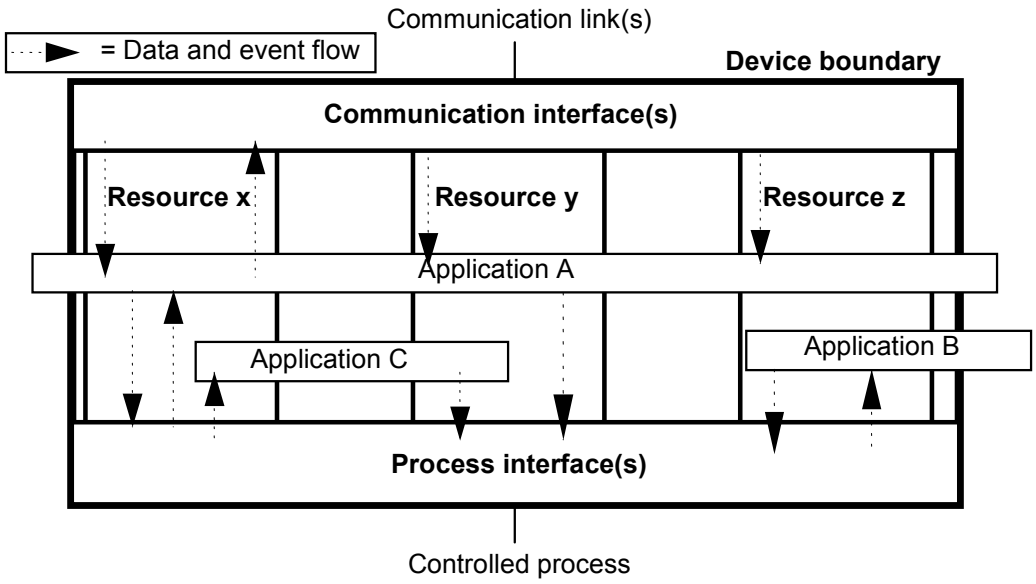
NOTE 2 A device that contains no resources is considered to be functionally equivalent to a *resource* as defined in 4.3.

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.

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.



NOTE This figure shows a possible internal structure of Device 2 from Figure 1.

Figure 2 – Device model

4.3 Resource model

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

NOTE 1 A resource is considered to be an *instance* of a corresponding resource *type*, defined as specified in 7.2.1.

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

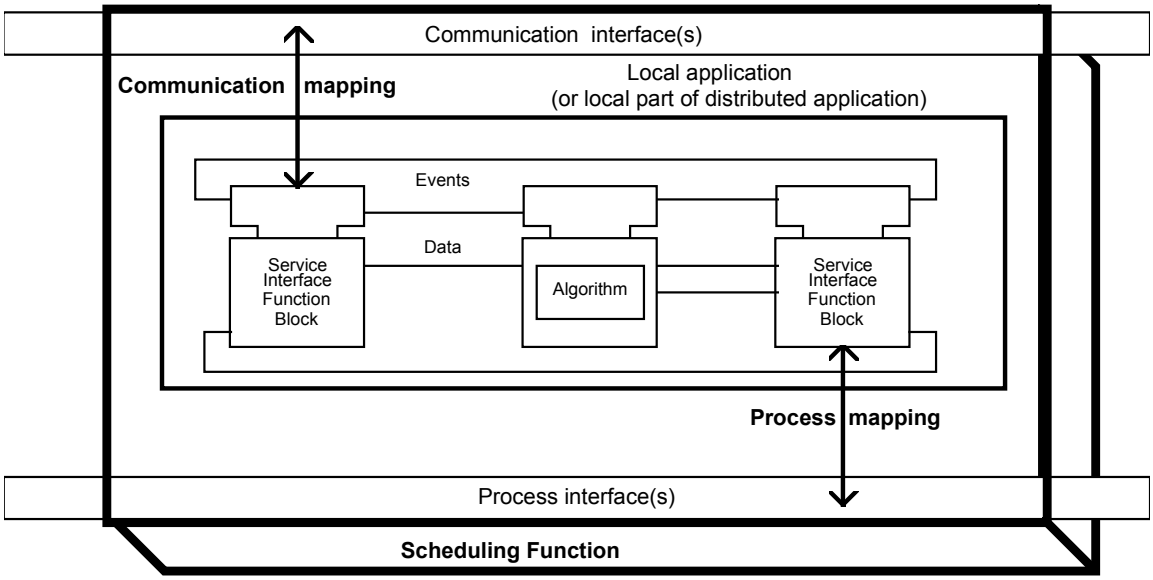
The *functions* of a resource are to accept *data* and/or *events* from the process and/or communication *interfaces*, process the data and/or events, and to return data and/or events to the process and/or communication interfaces, as specified by the *applications* utilizing the resource.

NOTE 3 Besides supporting the functions enumerated above, specific types of resources might 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 might or might 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 standard.

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:
  a) the occurrence of events;
  b) function block interconnections; and
  c) scheduling information such as periods and priorities.



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 can be shared among resources.

Figure 3 – Resource model

4.4 Application model

For the purposes of this document, 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*.

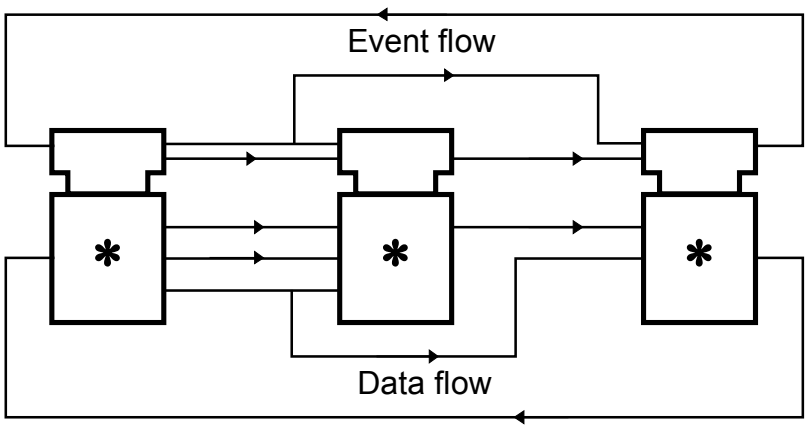
Subapplications are *instances* of *subapplication types*, which like applications consist of *function block networks*. Application names, subapplication and function block *instance names* may therefore be used to create a hierarchy of *identifiers* that can uniquely identify every *function block 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 document, 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 5.2.2.

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



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

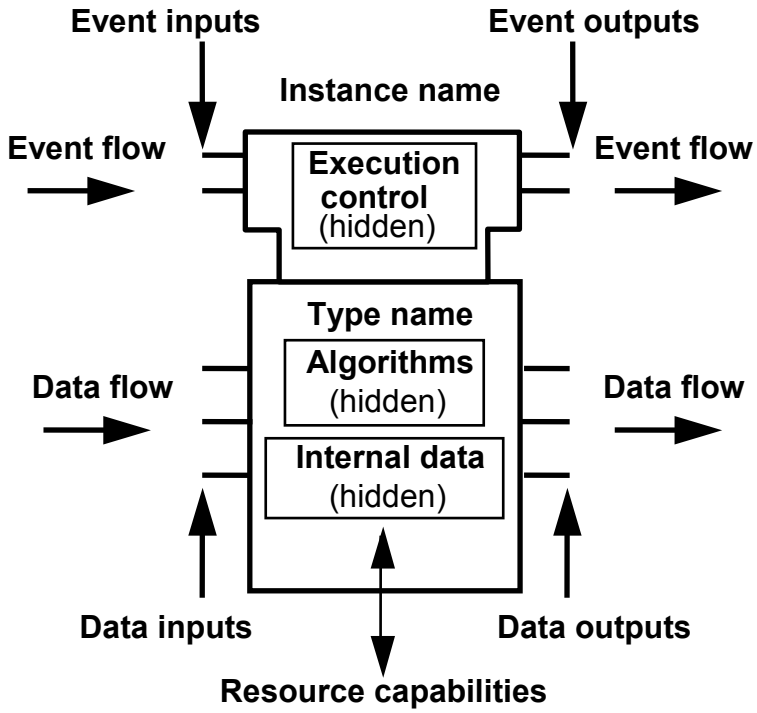
4.5.1 Characteristics of function block instances

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 4.5.1, 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 Internal state information can 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.

Access to internal variables and state information of function block instances may be provided by additional functional capabilities of the associated resource.

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 standard.

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.

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 1 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:

t_1: 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;

t_2: the event at the event input occurs;

t_3: the execution control function notifies the resource scheduling function to schedule an algorithm for execution;

t_4: algorithm execution begins;

t_5: 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;

t_6: the resource scheduling function is notified that algorithm execution has ended;

t_7: the scheduling function invokes the execution control function;

t_8: the execution control function signals an event at the event output.

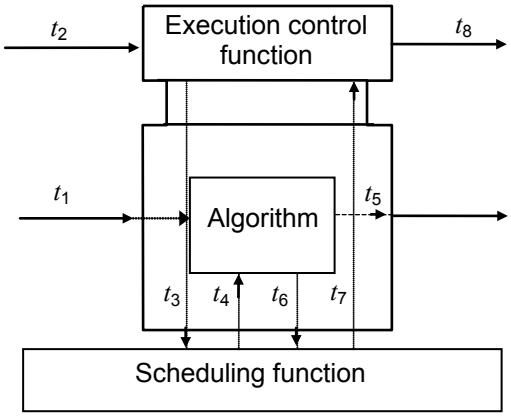
As shown in Figure 7, the significant timing delays in this case which are of interest in application design are:

$T_{setup} = t_2 - t_1$

$T_{start} = t_4 - t_2$ (time from event at event input to beginning of algorithm execution)

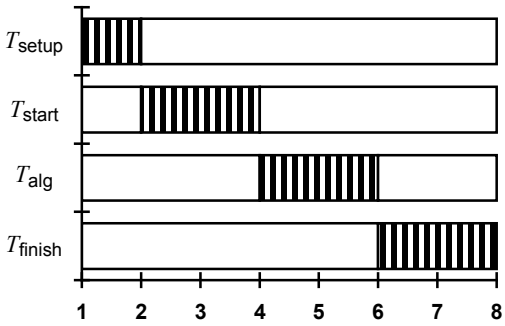
$T_{alg} = t_6 - t_4$ (algorithm execution time)

$T_{finish} = t_8 - t_6$ (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,... in the above figure correspond to the times t_1, t_2,... in Figure 6.

Figure 7 – Execution timing

Specific requirements for the graphical representation of *function block types* are given in 5.2.1.1.

NOTE 2 Depending on the problem to be solved, various requirements might exist for the synchronization of the values of *input variables* with the *execution* of *algorithms in order to ensure predictability of the results of algorithm execution*. Such requirements could 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.

NOTE 3 *Resources* might 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 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* might contain *service interface function block instances* representing *device* or *resource instances* for the purpose of querying or modifying device or resource *parameters*.

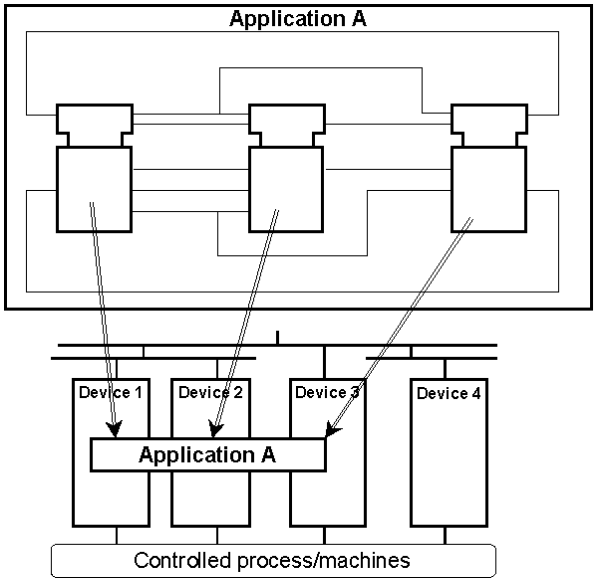


Figure 8a – Distribution model

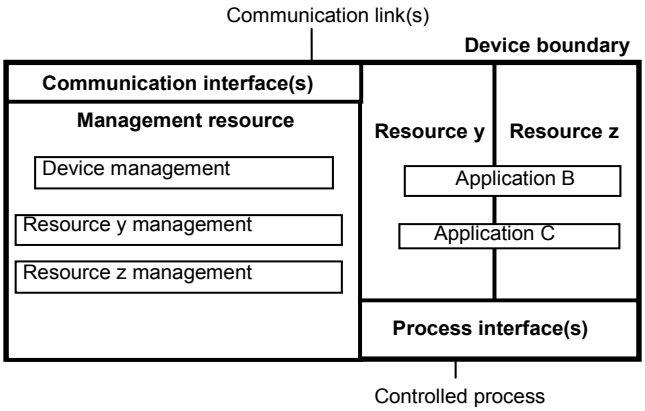


Figure 8b – Shared management model

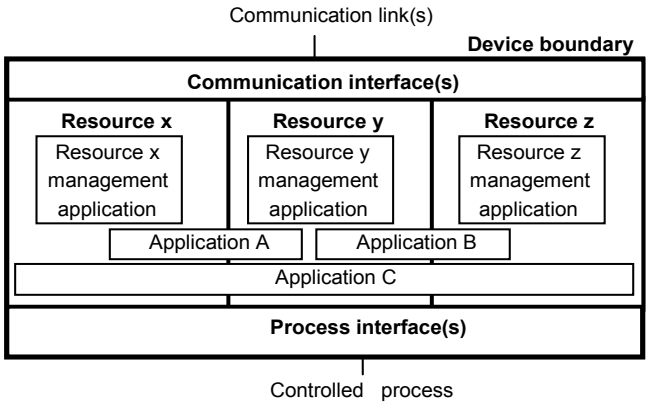


Figure 8c – Distributed management model

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" should be defined, e.g., 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 of modifying 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 6.3.2.

5 Specification of function block, subapplication and adapter interface types

5.1 Overview

As illustrated in Figure 9, Clause 5 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 standard; 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 standard.

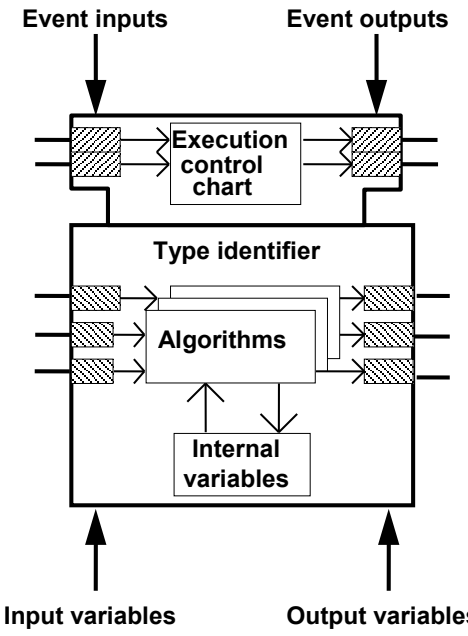


Figure 9a – Basic function block (5.2)

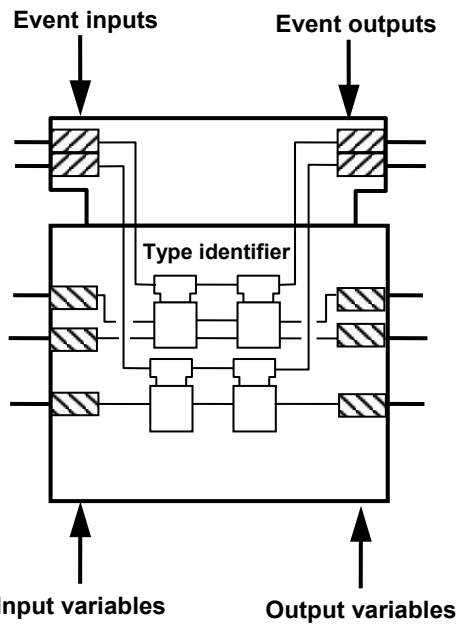
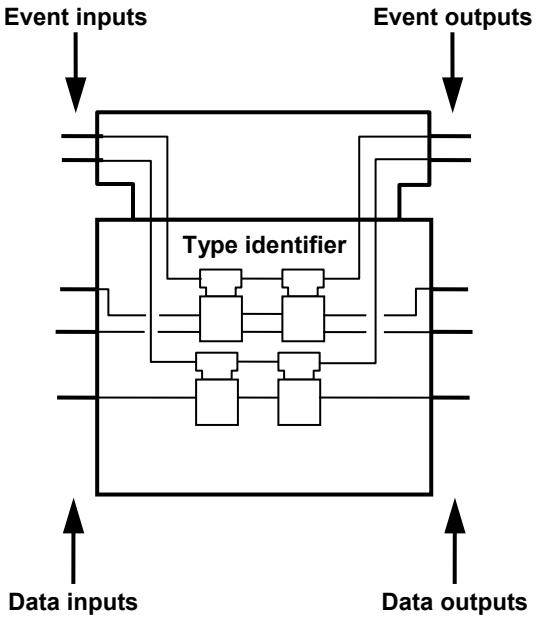


Figure 9b – Composite function block (5.3)



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

Figure 9c – **Subapplications** (5.4)

Figure 9 – Function block and subapplication types

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:

a) the function block *type name* is shown at the top center of the lower portion of the block;

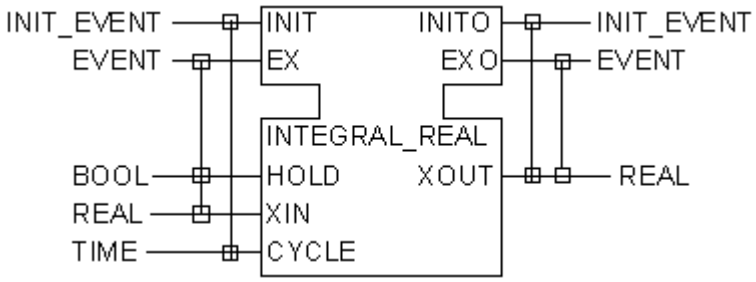
b) the names and *type declarations* of *input variables* and *socket adapters* are shown at the left edge of the lower portion of the block;

c) the names and *type declarations* of *output variables* and *plug adapters* are shown at the right edge of the lower portion of the block;

d) 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;

e) the *algorithms* associated with the function block type are declared as specified in 5.2.1.3;

f) control of the *execution* of the associated algorithms is declared as specified in 5.2.1.4.



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

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

Figure 10 – Basic function block type declaration

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 F, 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 can 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 6.3.2.

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 6.3.2.

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

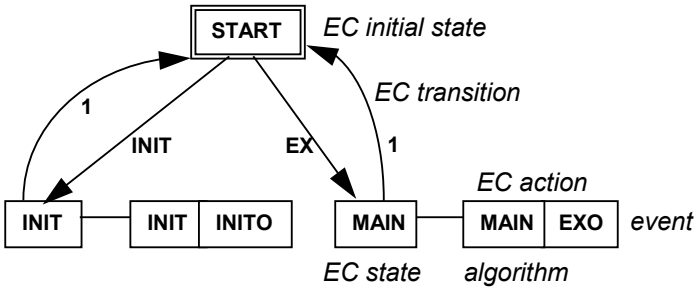


Figure 11 – ECC example

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 performs 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 can 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 be crossed, 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.

5.2.2.3 Algorithm declaration

As shown in Annex F, *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 standard.

The declaration of an algorithm may include the declaration of temporary variables that:

- are only visible in the body of the algorithm;
- are initialized upon each invocation of the algorithm;
- may be used and modified during execution of the algorithm; and
- do not have values that persist between executions of the algorithm.

5.2.2.4 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 transition condition, containing a reference to an *event*, a *guard condition*, or both, expressed in the syntax defined for the non-terminal `ec_transition_condition` in B.2.1.

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

NOTE 1 The notation `1` (one), illustrated in Figure 11, is considered to be equivalent to `[TRUE]` representing a transition condition with no associated event and a guard condition that is always `TRUE`.

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: 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.

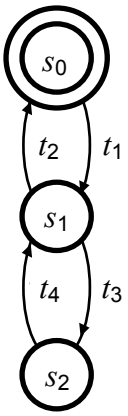


Figure 12 – ECC operation state machine

Table 1 – States and transitions of ECC operation state machine

State		Operations
s_0		--
s_1		evaluate transitions[c,e]
s_2		perform actions[d,e]
Transition	**Condition**	**Operations**
t_1	an input event occurs[a]	Sample inputs[b,e]
t_2	no transition is crossed	
t_3	a transition is crossed	
t_4	actions completed	

a The resource shall ensure that no more than one input event occurs at any given instant in time.

b This operation consists of *sampling* (or its functional equivalent) of the input variables associated with the current input event by a WITH declaration as described in 5.2.1.2.

c This operation consists of evaluating the transition conditions at the EC transitions following the active EC state and crossing the first EC transition (if any) for which a TRUE guard_condition as defined in B.2.1 is found, according to the following rules:

1 "Crossing the EC transition" shall consist of deactivating its predecessor EC state and activating its successor EC state.

2 The order in which the transition conditions are evaluated shall correspond to the order in which the transitions are declared as defined in B.2.1, or equivalently in the XML syntax defined in IEC 61499-2.

3 The guard_condition of a transition condition containing only an `event_input_name` shall have the default value TRUE.

4 If state s_1 was entered via t_1, only transition conditions associated with the current input event via its event_input_name as defined in B.2.1, or transition conditions with no event associations, shall be evaluated.

5 If state s_1 was entered via t_4, only transition conditions with no event associations shall be evaluated.

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 t_1 to an occurrence of t_2 shall be implemented as a *critical region* with a lock on the function block instance.

5.2.2.5 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 have to 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, or 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:

- 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.
- 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.
- 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.
- 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 1 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 2 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 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.

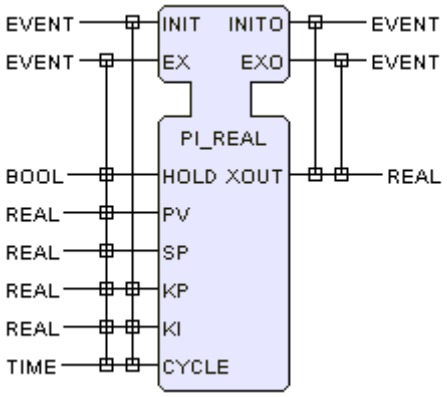


Figure 13a – External interface

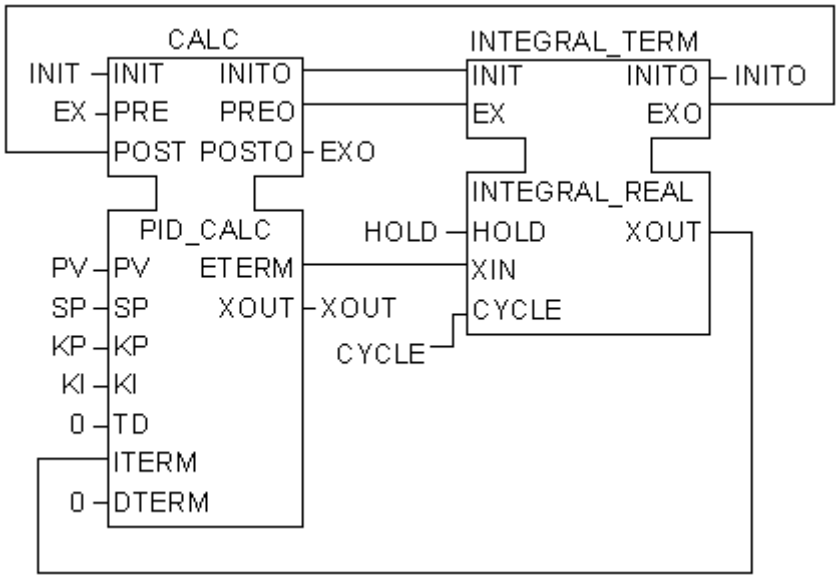


Figure 13b – Graphical body

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

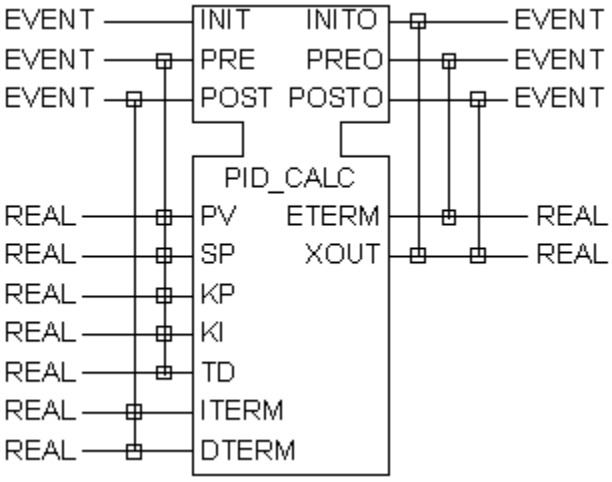


Figure 14a – External interface

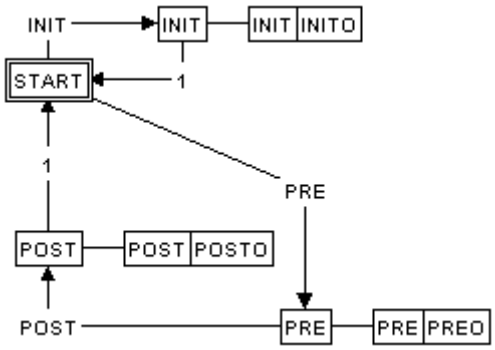


Figure 14b – Execution control

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 5.3.1, 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 can 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:

- 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.
- 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.
- 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.
- 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.

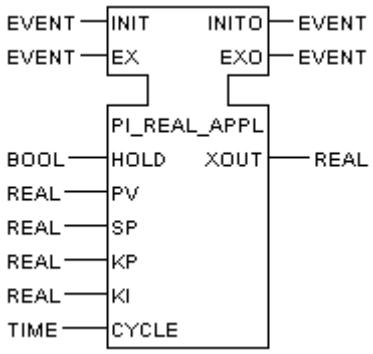


Figure 15a – External interface

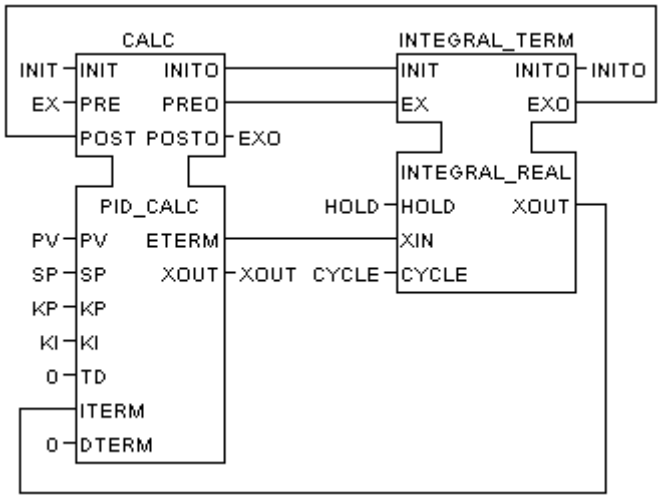


Figure 15b – Graphical body

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.

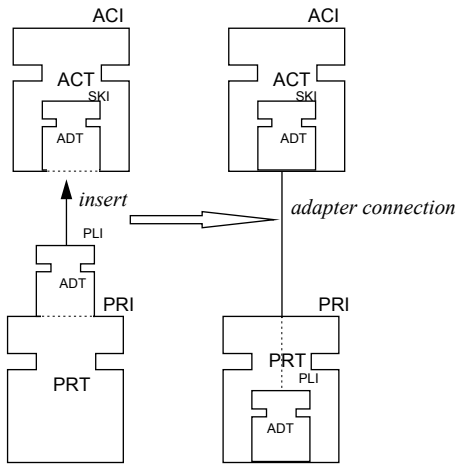
Since subapplications do not explicitly create variables, no specific initialization procedures are applicable to subapplication instances.

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 16, 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* might 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 in 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, e.g., 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, e.g., 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.

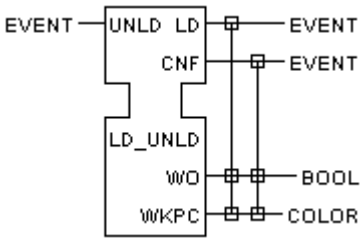


Figure 17a – Interface

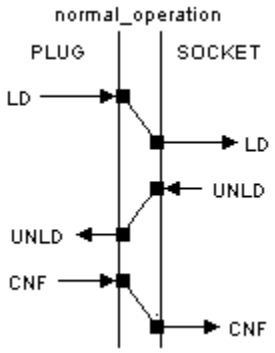


Figure 17b – Service sequence

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.

NOTE 3 See 6.1.2 for an explanation of service sequences.

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 to show 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 to show 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:

1) a plug can only be inserted into a socket of the same *adapter interface type;*

2) a plug can only be inserted into zero or one socket at a time;

3) a socket can only accept zero or one plug at a time;

4) a plug can only be inserted in a socket if both are in the same *composite function block, resource*, *application* or *subapplication*.

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.

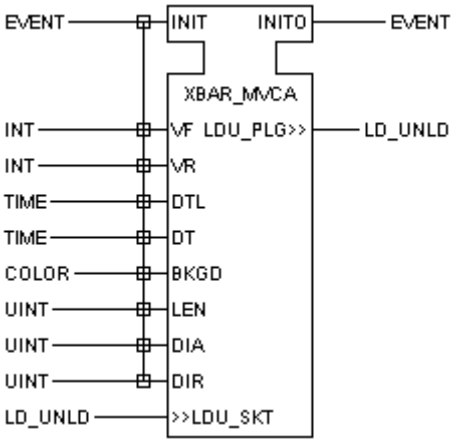


Figure 18a – Interface

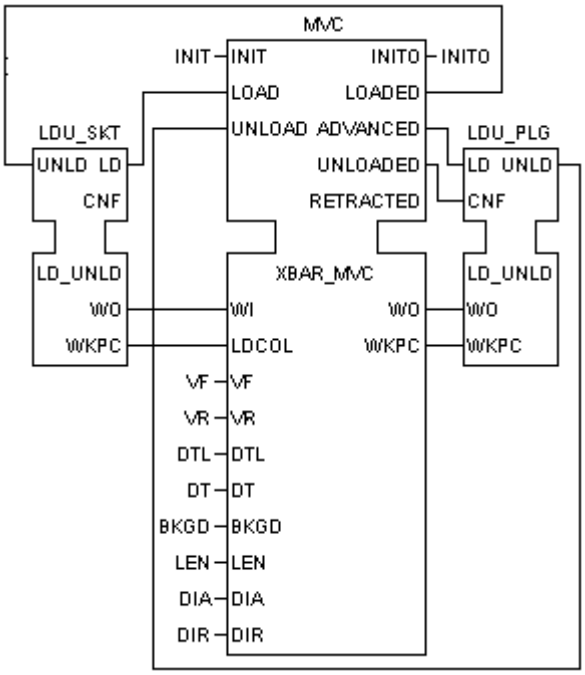


Figure 18b – Body

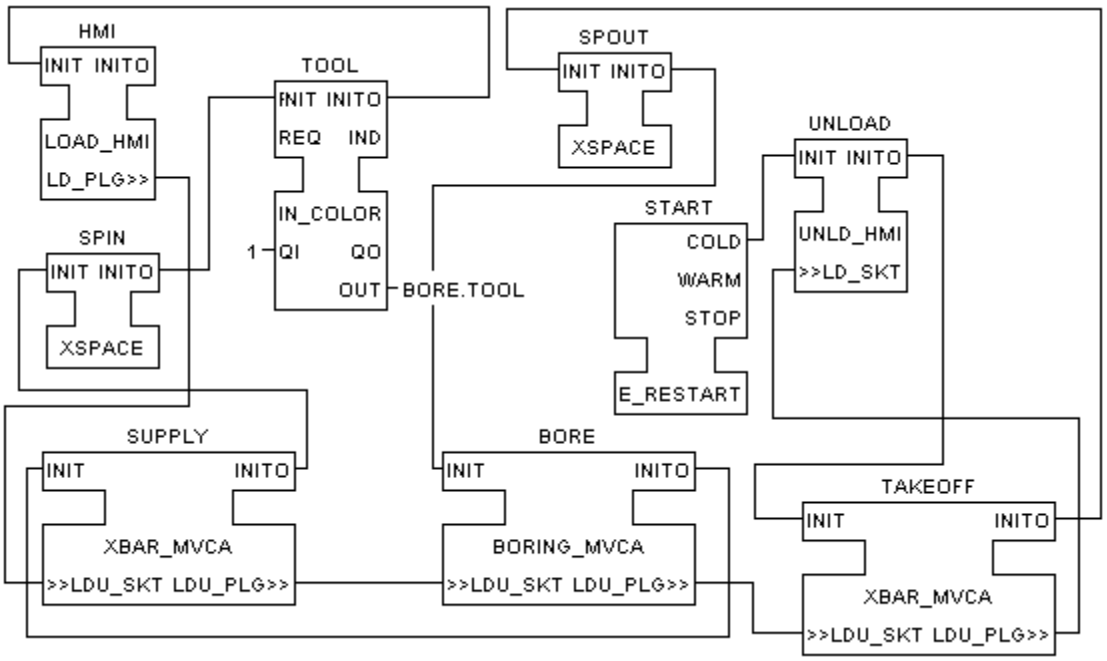
NOTE 1 A full textual declaration of this example is given in Annex F.

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* can 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 standard. 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

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 standard.

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 6.1.2. The name of the function block *type* shall indicate the provided service.

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

NOTE 1 Services can provide both resource- and application-initiated interactions in the same service interface function block.

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

Table 2 – Standard inputs and outputs for service interface function blocks (1 of 2)

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). 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. The function block type specification may define other names for these inputs.

Table 2 (2 of 2)

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. A service specification may indicate that the value of this output is irrelevant for some situations,for instance, 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. The function block type specification may define other names for these outputs.

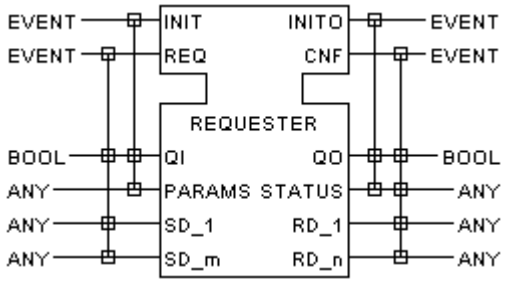


Figure 20a – Application-initiated interactions

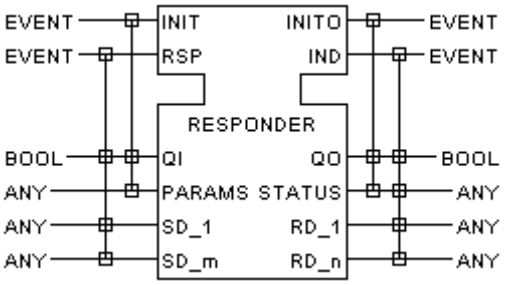


Figure 20b – Resource-initiated interactions

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 F 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, which can utilize *service sequence diagrams* subject to the following rules:

a) The following semantics shall apply:
   1) Time increases in the downward direction.
   2) Events which are sequentially related are linked together across or within resources.
   3) 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:

   1) 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.

   2) 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 illustrated in E.2.2 and E.2.3 can be used.

d) *Service primitives* shall be indicated by horizontal arrows. 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.

NOTE Sequence diagrams can also be used to document the externally observable behaviors of basic and composite function block types.

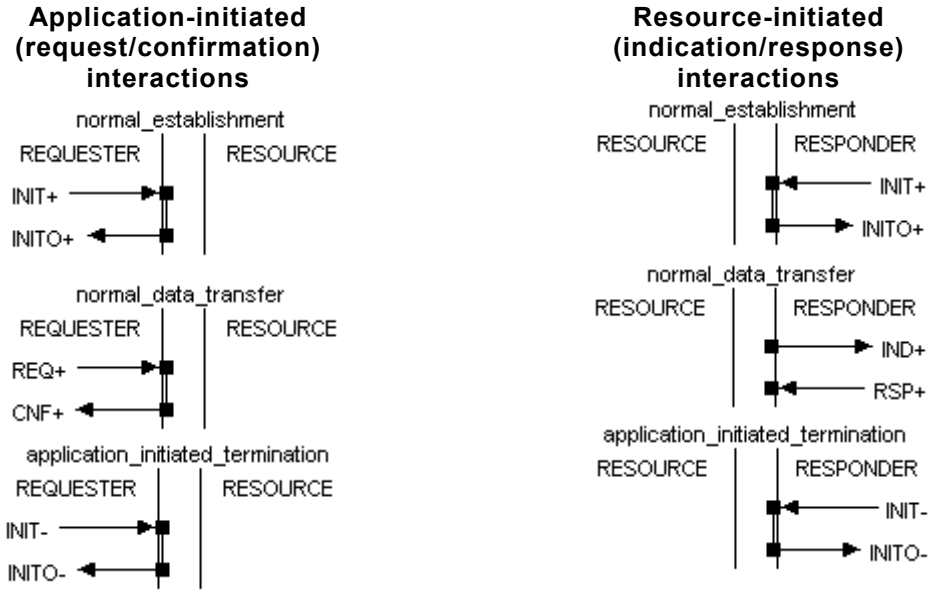


Figure 21 – Example service sequence diagrams

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.* 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 can 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 might be constrained to match the number and types of the corresponding SD inputs.	
[a] *Communication function block type declarations* define the number and type of the SD_1,..., SD_m inputs and RD_1,...,RD_n outputs, and can 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

6.3.1 Requirements

Extending the functional requirements for "application management" in subclause 8.3.2 of ISO/IEC 7498-1:1994 to the distributed application model of this standard 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:
   1) data types
   2) function block types and instances
   3) 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 standard 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 standard only deals with item a) 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) might 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 might or might 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.

NOTE 5 A full textual specification of this function block type, including all service sequences, is given in Annex F.

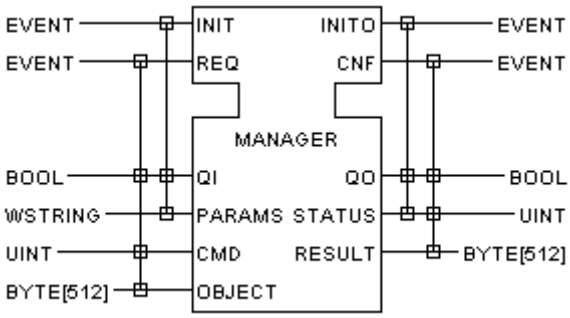
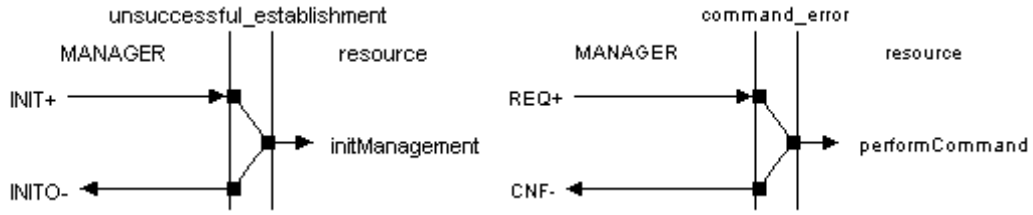


Figure 22 – Generic management function block type

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 parameter data
5	WRITE	Write parameter data
6	KILL	Make specified object unrunnable
7	QUERY	Request information on specified object
8	RESET	Reset 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 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
DELETE	data_type_name	data_type_name
	fb_type_name	fb_type_name
	fb_instance_reference	fb_instance_reference
	connection_definition	connection_definition
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
READ	parameter_reference	parameter
WRITE	referenced_parameter	parameter_reference
RESET	fb_instance_reference	fb_status
NOTE See Table 6 for the integer values of the CMD input corresponding to the commands listed above.		

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:

- START and STOP of a *function block instance* shall be as defined in 6.3.2;
- 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*;
- STOP of a *management function block instance* shall be equivalent to STOP of all *function block instances* within the managed *resource*;
- 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:

- when the OBJECT input specifies an *event input, event output* or *data output,* the RESULT output shall contain zero or more opposite end points;
- when the OBJECT input specifies a *data input,* the RESULT output shall list zero or one opposite end point;
- 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:

   1) UNSUPPORTED_CMD: No state exists in Figure 24 with a transition condition for the specified CMD value;

   2) INVALID_STATE: The currently active state does not have a transition condition for the specified CMD value;

   3) 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;

   4) 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 can be specified by the manufacturer of a compliant device, for example by reference to an appropriate standard.

NOTE 3 *Applications* can 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.

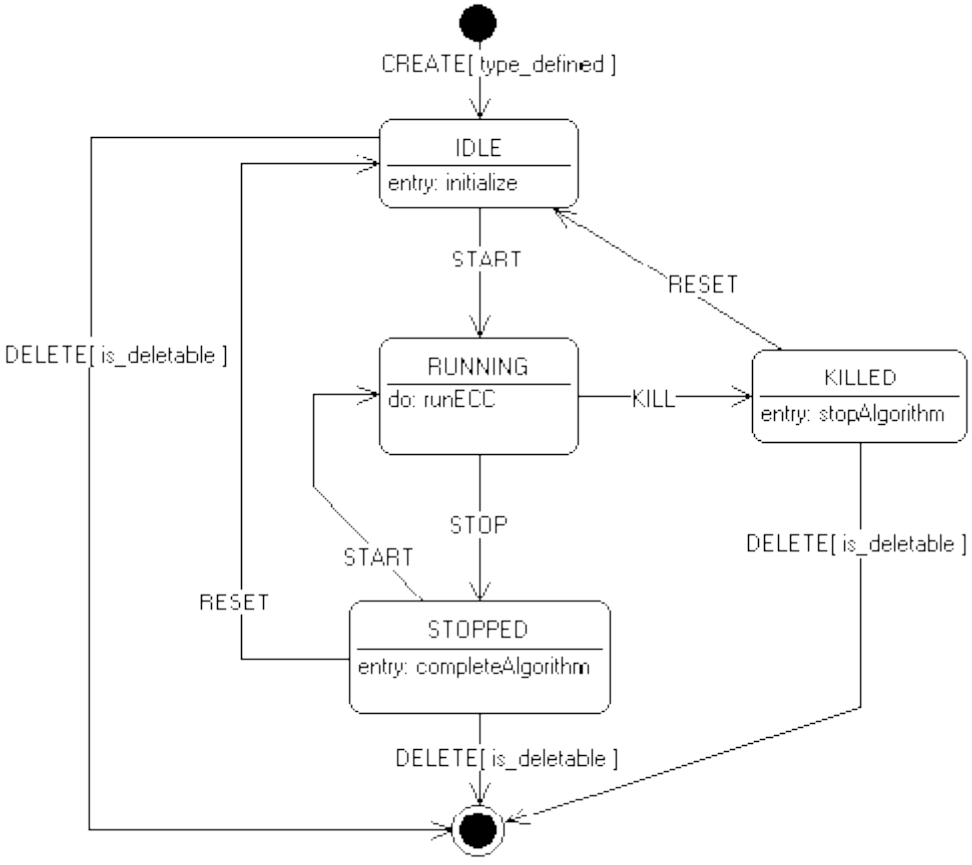


Figure 24 – Operational state machine of a managed function block

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

Clause 7 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*. Clause 7 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, device and segment 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.

NOTE 1 Additional information can 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, e.g., whether two instances of function block type "A" can 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 standard 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.

NOTE 1 Items (2) and (3) above are considered to be in addition to the corresponding elements declared in the resource type specification as defined in 7.2.1.

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 standard 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 does not contain any declarations of resource instances.

7.2.3 Functional specification of segment types

The functional specification of a *segment type* includes:

- the segment *type name*;
- the *instance name, data type*, and initialization of each of the segment *parameters*.

7.3 Configuration requirements

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:

- the *instance name* and *type name* of the device;
- configuration-specific values for the device *parameters*;
- the *resource types* supported by the device *instance* in addition to those specified for the device *type*;
- 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;
- any *data connections, adapter connections* and *event connections* that are present in the device instance in addition to those defined for the device type;
- the *resource types* supported by the device *instance* in addition to those specified for the device *type*;
- 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 does 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:

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

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

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

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

Resource configuration is subject to the following rules:

– Items b), c), and d) above are considered to be in addition to the corresponding elements declared in the device and resource type specifications as defined in 7.2.2 and 7.2.1, respectively.

– Items c) and d) include *function block instances, data connections, adapter connections* and *event connections* from those portions of *applications* allocated to the resource.

– Items c) and d) 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 c) 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 7.2.1.
– It shall be an **error** if the instance name of a resource is not unique within the scope of the device containing it, or if 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*, or if the declared values of the segment parameters are inconsistent with the declaration (if any) of the *segment type* defined in 7.2.3.

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 Annex A 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 Annex A 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 T_{setup}, T_{start} and T_{finish} are considered to be zero (0) for all *component function blocks* in the reference implementation.
- The parameter T_{alg} 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 Annex A 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 (1 of 6)

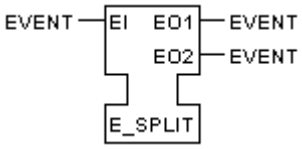
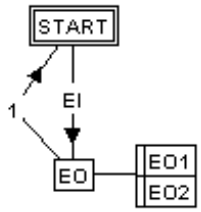
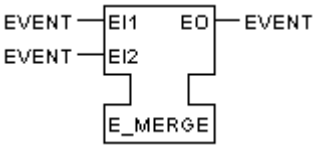
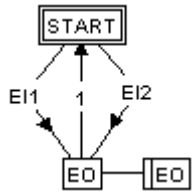
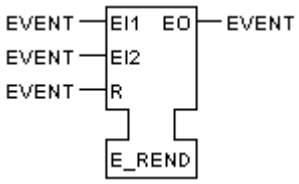
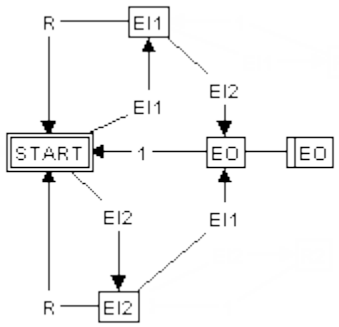
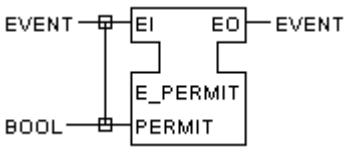
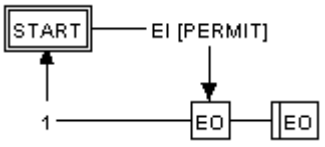
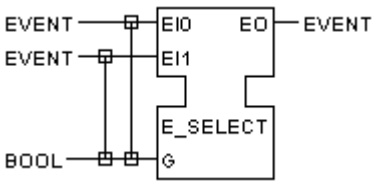
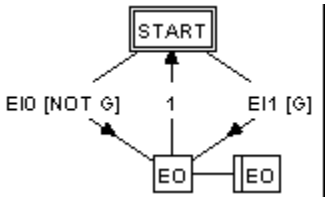
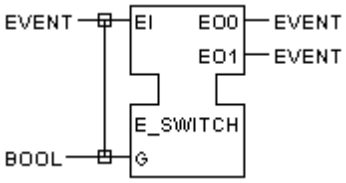
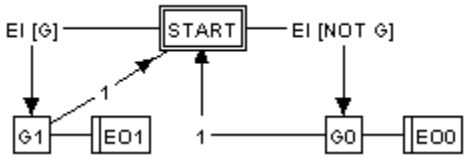
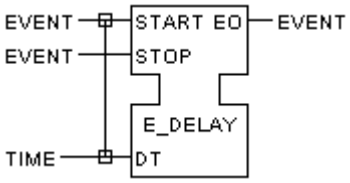
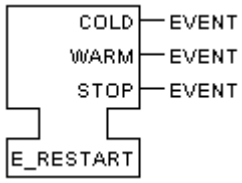
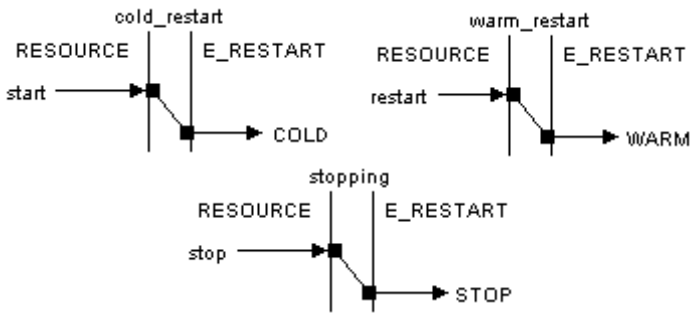
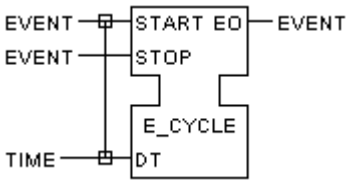
<table>
<tr><th>No.</th><th colspan="2">Description</th></tr>
<tr><th colspan="2">Interface</th><th>ECC/Algorithms/Service sequences</th></tr>
<tr><td>1</td><td colspan="2">Split an event</td></tr>
<tr><td colspan="2"> 
</td><td> 
</td></tr>
<tr><td colspan="3">The occurrence of an event at EI causes the occurrence of events at EO1, EO2,...,EOn (n=2 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 EI1, EI2,...,EIn causes the occurrence of an event at EO (n=2 in the above example).</td></tr>
<tr><td>3</td><td colspan="2">Rendezvous of two events</td></tr>
<tr><td colspan="2"> 
</td><td> 
</td></tr>
<tr><td>4</td><td colspan="2">Permissive propagation of an event</td></tr>
<tr><td colspan="2"> 
</td><td> 
</td></tr>
<tr><td>5</td><td colspan="2">Selection between two events</td></tr>
<tr><td colspan="2"> 
</td><td> 
</td></tr>
</table>

Table A.1 (2 of 6)

<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>6</td><td colspan="2">Switching (demultiplexing) an event</td></tr>
<tr><td colspan="2"> 
</td><td> 
</td></tr>
<tr><td>7</td><td colspan="2">Delayed propagation of an event</td></tr>
<tr><td colspan="2"> 
</td><td>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.</td></tr>
<tr><td>8</td><td colspan="2">Generation of restart events</td></tr>
<tr><td colspan="2"> 
</td><td> 
</td></tr>
<tr><td colspan="3">a) An event is issued at the COLD output upon "cold restart" of the associated resource.
b) An event is issued at the WARM output upon "warm restart" of the associated resource.
c) An event is issued at the STOP output (if possible) prior to "stopping" of the associated resource.
NOTE 1 See IEC 61131-1 for a discussion of "cold restart" and "warm restart".</td></tr>
<tr><td>9</td><td colspan="2">Periodic (cyclic) generation of an event</td></tr>
<tr><td colspan="2"> 

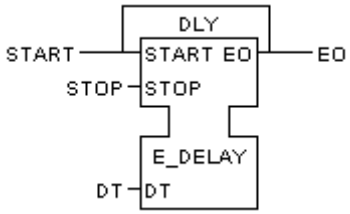
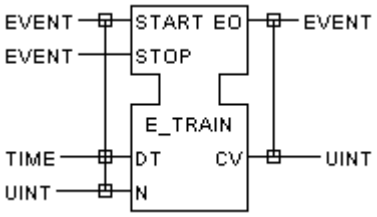
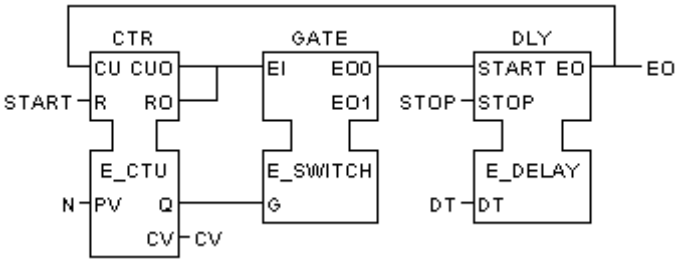
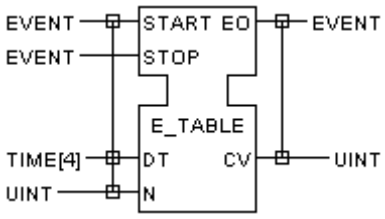
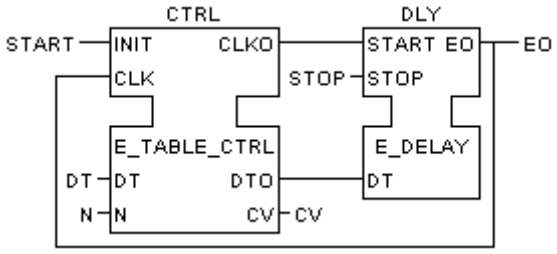
An event occurs at EO at an interval DT after the occurrence of an event at START, and at intervals of DT thereafter until the occurrence of an event at STOP.</td><td> 
</td></tr>
</table>

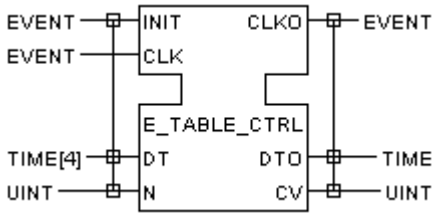
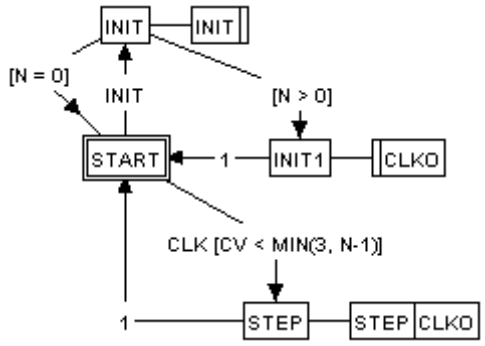
Table A.1 (3 of 6)

<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>10</td><td colspan="2">Generation of a finite train of events</td></tr>
<tr><td colspan="2"> 
</td><td> 

NOTE 2 See table entry 18 for a definition of the E_CTU type.</td></tr>
<tr><td colspan="3">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 3 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.</td></tr>
<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 START. 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 4 In this example implementation, N <= 4.

NOTE 5 Implementation using the E_TABLE_CTRL function block type illustrated below is not a normative requirement. Equivalent functionality can be implemented by various means.</td></tr>
<tr><td colspan="2"> 
</td><td> 
</td></tr>
<tr><td>

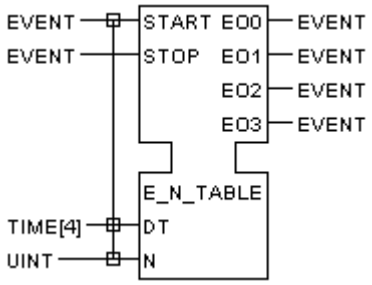
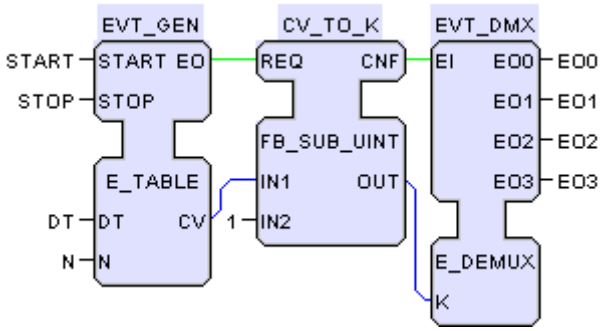
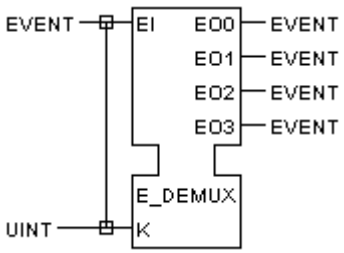
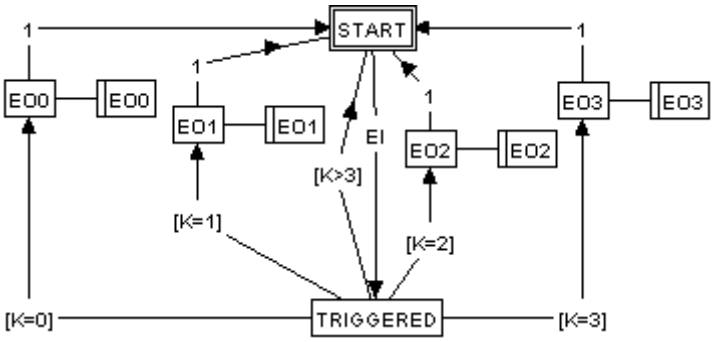
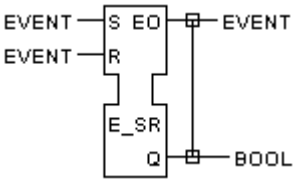
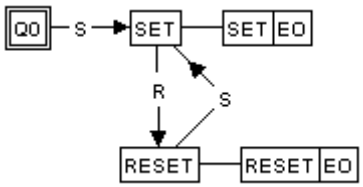
```
ALGORITHM INIT IN ST:
  CV:= 0;
  DTO:= DT[0];
END_ALGORITHM
```

</td><td>

```
ALGORITHM STEP IN ST:
  CV:= CV+1;
  DTO:= DT[CV];
END_ALGORITHM
```

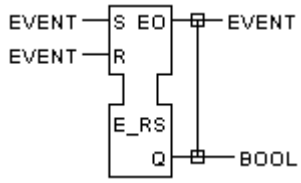
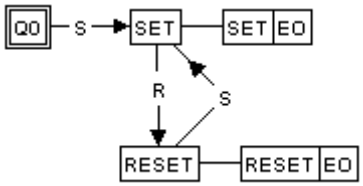
</td></tr>
</table>

Table A.1 (4 of 6)

No.	Description	
	Interface	**ECC/Algorithms/Service sequences**
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 `START`. An event occurs at `EO1` an interval `DT[1]` after the occurrence of the event at `EO0`, etc., until `N` occurrences have been generated or an event occurs at the `STOP` input. NOTE 6 In this example implementation, `N <= 4`. NOTE 7 Implementation using the `E_DEMUX` function block type illustrated below is not a normative requirement. Equivalent functionality can 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 (5 of 6)

<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>14</td><td colspan="2">Event-driven bistable</td></tr>
<tr><td colspan="2"> 
</td><td> 
</td></tr>
<tr><td colspan="3">The output <code>Q</code> is set to <code>1 (TRUE)</code> upon the occurrence of an event at the <code>S</code> input, and is reset to <code>0 (FALSE)</code> upon the occurrence of an event at the <code>R</code> input. An event is issued at the <code>EO</code> output when the value of <code>Q</code> changes.

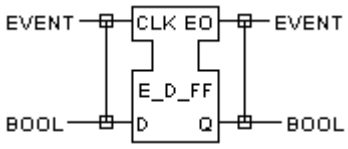
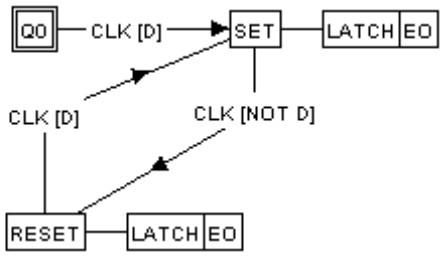
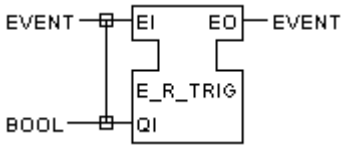
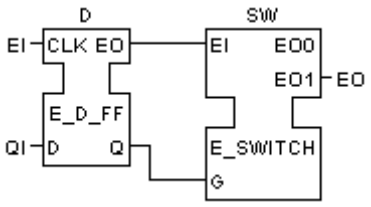
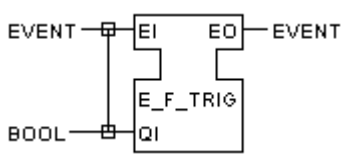
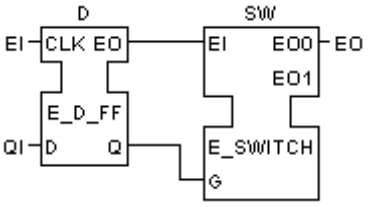
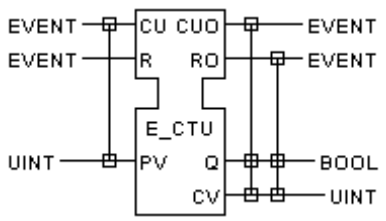
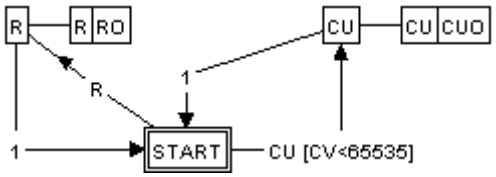
NOTE 8 The implementation of this function block type is identical to <code>E_SR</code>. Both <code>E_SR</code> and <code>E_RS</code> are implemented for consistency with the <code>SR</code> and <code>RS</code> types of IEC 61131-3, although there is no "dominance" of events as there would be for level-controlled <code>R</code> and <code>S</code> inputs.</td></tr>
<tr><td>15</td><td colspan="2">D (Data latch) bistable</td></tr>
<tr><td colspan="2"> 
</td><td> 
</td></tr>
<tr><td colspan="2"></td><td><pre>ALGORITHM LATCH IN ST:
  Q:= D;
END_ALGORITHM</pre></td></tr>
<tr><td>16</td><td colspan="2">Boolean rising edge detection</td></tr>
<tr><td colspan="2"> 
</td><td> 
</td></tr>
<tr><td>17</td><td colspan="2">Boolean falling edge detection</td></tr>
<tr><td colspan="2"> 
</td><td> 
</td></tr>
</table>

Table A.1 (6 of 6)

<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>18</td><td colspan="2">Event-driven up counter</td></tr>
<tr><td colspan="2"> 
</td><td> 
</td></tr>
<tr><td colspan="2"><pre>ALGORITHM R IN ST: (* Reset *)
  CV:= 0;
  Q:= 0;
END_ALGORITHM</pre></td><td><pre>ALGORITHM CU IN ST: (* Count Up *)
  CV:= CV + 1;
  Q:= (CV >= PV);
END_ALGORITHM</pre></td></tr>
</table>

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.1b is equivalent to the explicit representation in Figure A.1a.

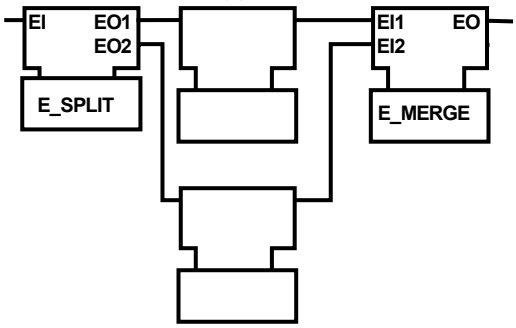


Figure A.1a – Explicit representation

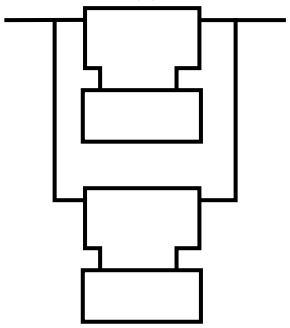


Figure A.1b – Implicit representation

NOTE Irrelevant details are 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 Annex B 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 standard consist of combinations of the characters in the character set given as Table 2 – Row 00 of the "Basic Latin to CJK Compatibility" table linked to Clause 33 defined in ISO/IEC 10646:2003.

For the purposes of this standard, 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 standard 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 standard 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 B.2.1 can be used for the textual specification of *function block types* according to the rules given in Clauses 5 and 6 of this standard.

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]
     '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::= event_conn_source 'TO' event_conn_destination ';'


event_conn_source::= ([plug_name '.'] event_input_name)
     | ((fb_instance_name | socket_name) '.' event_output_name)

event_conn_destination::= ([plug_name '.'] event_output_name)
     | ((fb_instance_name | socket_name) '.' event_input_name)

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

data_conn::= data_conn_source 'TO' data_conn_destination ';'
```

```
data_conn_source::= ([plug_name '.'] input_variable_name)
     | ((fb_instance_name | socket_name) '.' output_variable_name)

data_conn_destination::= ([plug_name '.'] output_variable_name)
     | ((fb_instance_name | socket_name) '.' input_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 | ('->' ec_action_output)
     | (algorithm_name '->' ec_action_output)


ec_action_output:= ([plug_name '.'] event_output_name)
     | (socket_name '.' event_input_name)

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

ec_transition_condition::= '1'
     | ec_transition_event | '[' guard_condition ']'
     | ec_transition_event '[' guard_condition ']'


ec_transition_event::= ([plug_name '.'] event_input_name)
     | (socket_name '.' event_output_name)

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

ec_expression_operand::=
     ([(plug_name | socket_name) '.'] input_variable_name)
     | ([(plug_name | socket_name) '.'] output_variable_name)
     | internal_variable_name
     | constant

fb_algorithm_declaration::=
     'ALGORITHM' algorithm_name 'IN' language_type ':'
     [temp_var_decls]
     algorithm_body
     'END_ALGORITHM'

temp_var_decls::= <as defined in IEC 61131-3>
```

```
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 5.4.1.

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::= subapp_event_source 'TO' subapp_event_destination ';'

subapp_event_source:: = ([plug_name '.'] event_input_name)
     | ((fb_subapp_name | socket_name) '.' event_output_name
```

```
subapp_event_destination::= ([plug_name '.'] event_output_name)
     | ((fb_subapp_name | socket_name) '.' event_input_name)

fb_subapp_name::= fb_instance_name | subapp_instance_name

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

subapp_data_conn::= subapp_data_source 'TO' subapp_data_destination ';'

subapp_data_source::= ([plug_name '.'] input_variable_name)
     | ((fb_subapp_name | socket_name) '.' output_variable_name)

subapp_data_destination::= ([plug_name '.'] output_variable_name)
     | ((fb_subapp_name | socket_name) '.' input_variable_name)

subapp_type_name::= identifier

subapp_instance_name::= identifier
```

B.3 Configuration elements

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

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

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]
     [config_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]
     '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 | input_variable_name)
     'TO'
     (fb_instance_name | resource_instance_name)'.' input_variable_name ';'
     <resource_instance_name only applies to connections within device_type or
     device_configuration declarations>

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 ';'

fb_instance_reference::= [app_hierarchy_name] fb_instance_name

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

```
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]
     [config_connection_list]
     'END_DEVICE_TYPE'

resource_instance::=
     'RESOURCE' resource_instance_name ':' resource_type_name
     [fb_instance_list]
     [config_connection_list]
     '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]
     config_connection_list
     'END_RESOURCE_TYPE'

segment_type_specification::= 'SEGMENT_TYPE' segment_type_name
     {parameter_declaration}
     'END_SEGMENT_TYPE'

parameter_declaration:= parameter_name ':' var_spec_init ';'

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

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

fb_resource_reference::= resource_hierarchy ['.' fb_instance_name]
     <When the optional element ['.' fb_instance_name] is not given, the
     instance name of the FB in the resource is the same as its instance name
     in the corresponding fb_instance_reference of the mapping.>

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
```

B.4 Common elements

Where syntactic productions are not given for non-terminal symbols in Annex B, the syntactic productions and corresponding semantics given in Annex B of IEC 61131-3:2003 shall apply.

B.5 Supporting productions for management commands

The syntax defined in this clause 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

referenced_parameter::=
     [(resource_instance_name | fb_instance_name)'.'] parameter
     <resource_instance_name refers to a resource located in the same device
     as the MANAGER block defined in 6.3.2>
     <fb_instance_name refers to an FB contained in the same device or
     resource as the <MANAGER> block>
     <if no resource or FB instance name is given, the parameter refers to a
     parameter of the device or resource containing the MANAGER block>

parameter_reference::=
     [(resource_instance_name | fb_instance_name)'.'] parameter_name
     <see above for semantics>

all_data_types::= 'ALL_DATA_TYPES'

all_fb_types::= 'ALL_FB_TYPES'

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

B.6 Tagged data types

The syntax defined below shall be used for the assignment of tags as defined in ISO/IEC 8824-1 to derived data types defined as specified in Annex B and 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.

SYNTAX:

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

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

B.7 Adapter interface types

See 5.5 for the semantics associated with the following syntax.

SYNTAX:

```
adapter_type_declaration::=
     'ADAPTER' adapter_type_name
          fb_interface_list
          [fb_service_declaration]
     'END_ADAPTER'

adapter_type_name::= identifier
```

Annex C
(informative)

Object models

C.1 Model notation

Annex C 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 standard.

The notation used in Annex C 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 ESS overview

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.

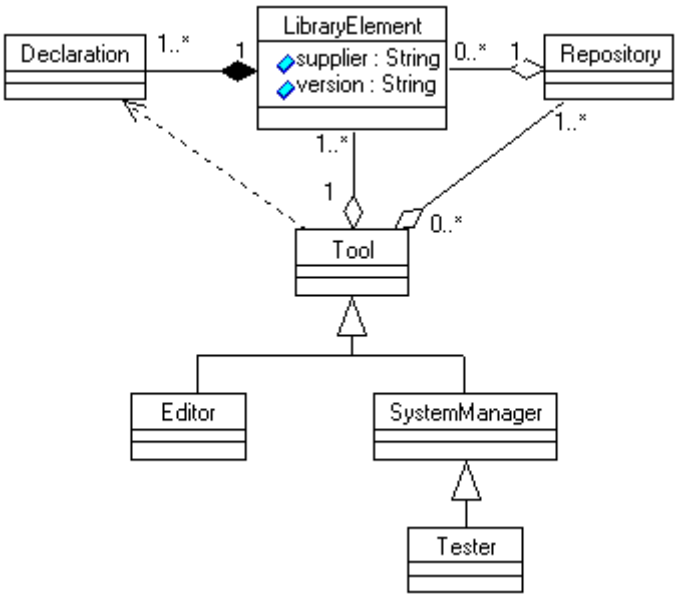


Figure C.1 – ESS overview

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.2.2) 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.

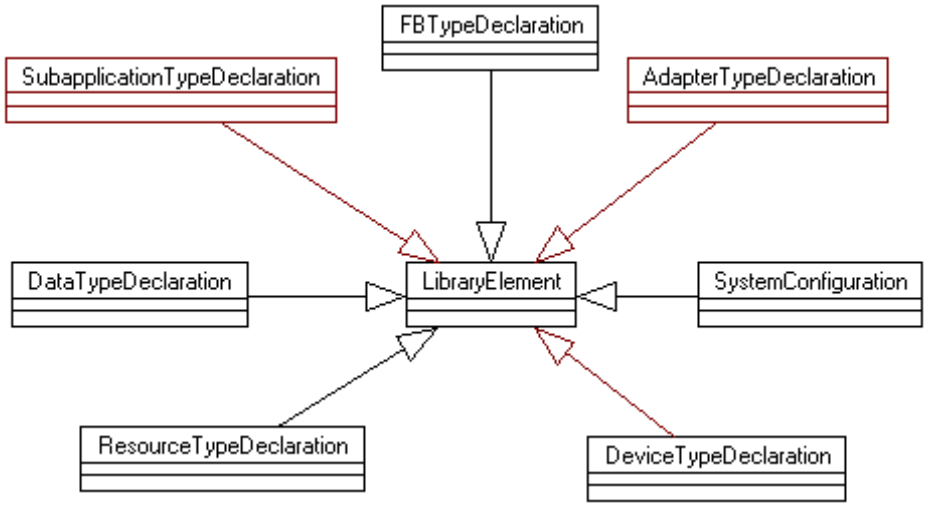


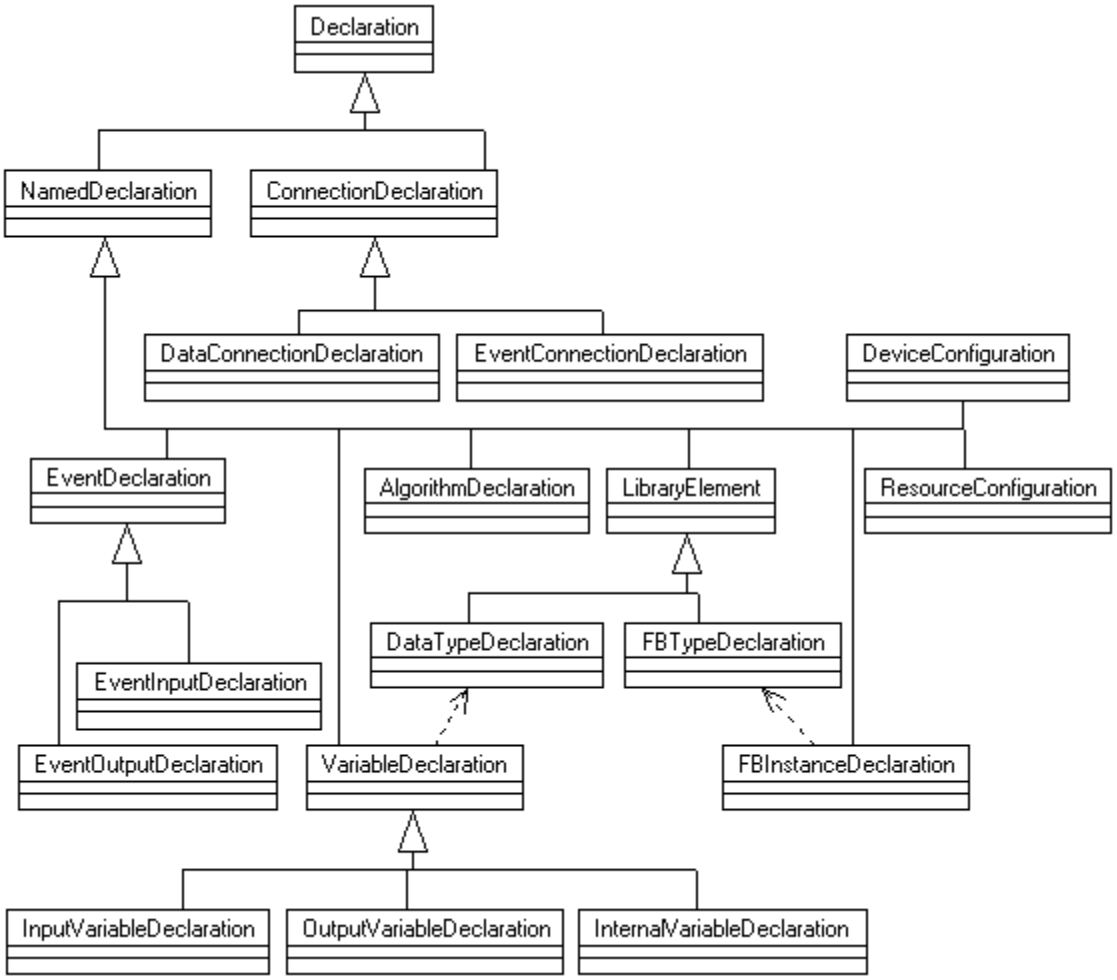
Figure C.2 – Library elements

Table C.2 – Syntactic productions for library elements

Class	Syntactic production
DataTypeDeclaration	type_declaration
FBTypeDeclaration	fb_type_declaration
AdapterTypeDeclaration	adapter_type_declaration
SubapplicationTypeDeclaration	subapplication_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.



NOTE To avoid clutter, classes related to *adapter types, instances* and *connections* are not shown in this Figure; however, they are listed in Table C.3 for reference.

Figure C.3 – Declarations

Table C.3 – Syntactic productions for declarations

Class	Syntactic production
AdapterConnectionDeclaration	adapter_conn
AdapterTypeDeclaration	adapter_type_declaration
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
PlugDeclaration	Part of plug_list
ResourceConfiguration	resource_instance
SocketDeclaration	Part of socket_list

C.2.4 Function block network declarations

Figure C.4 shows the relationships among the elements of *function block network declarations*. See C.2.2 for definitions of the aggregated classes in this diagram.

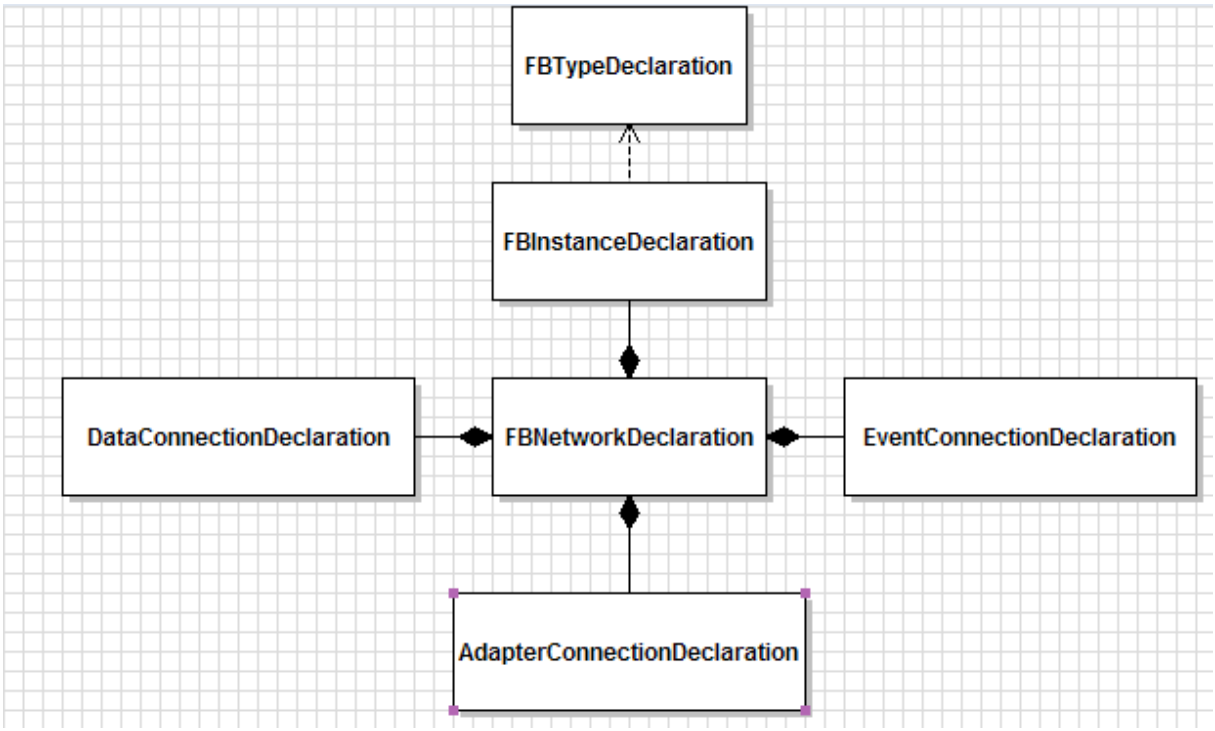


Figure C.4 – Function block network declarations

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 **CompositeFBTypeDeclaration** which contain no event `WITH` data associations.

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

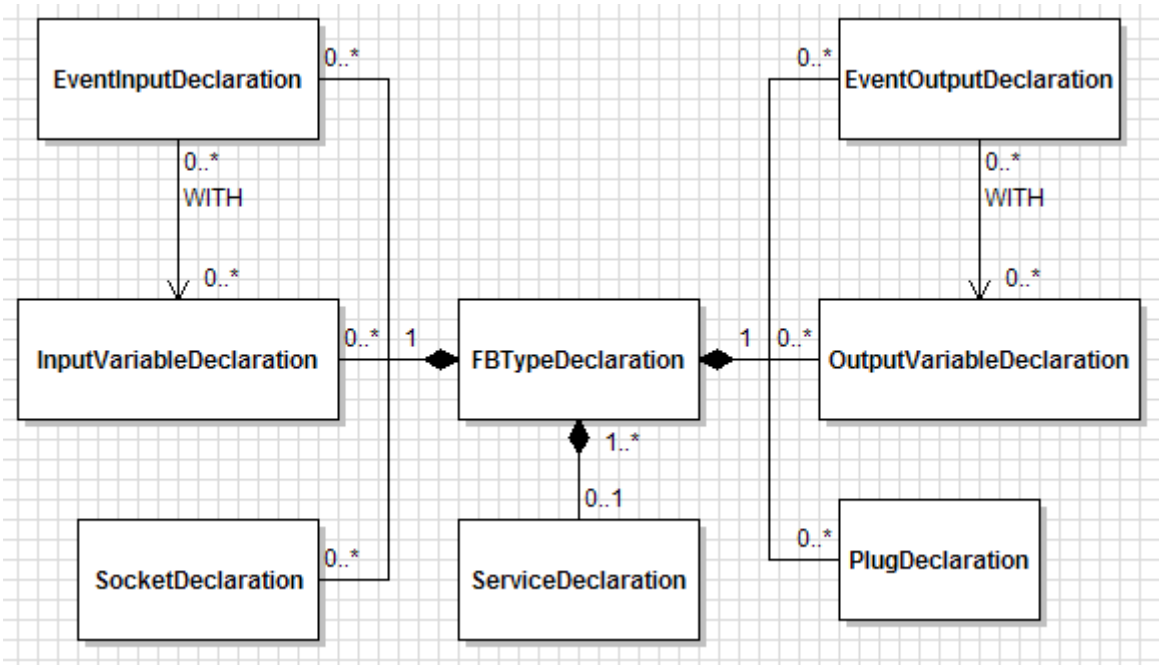


Figure C.5a – Composition

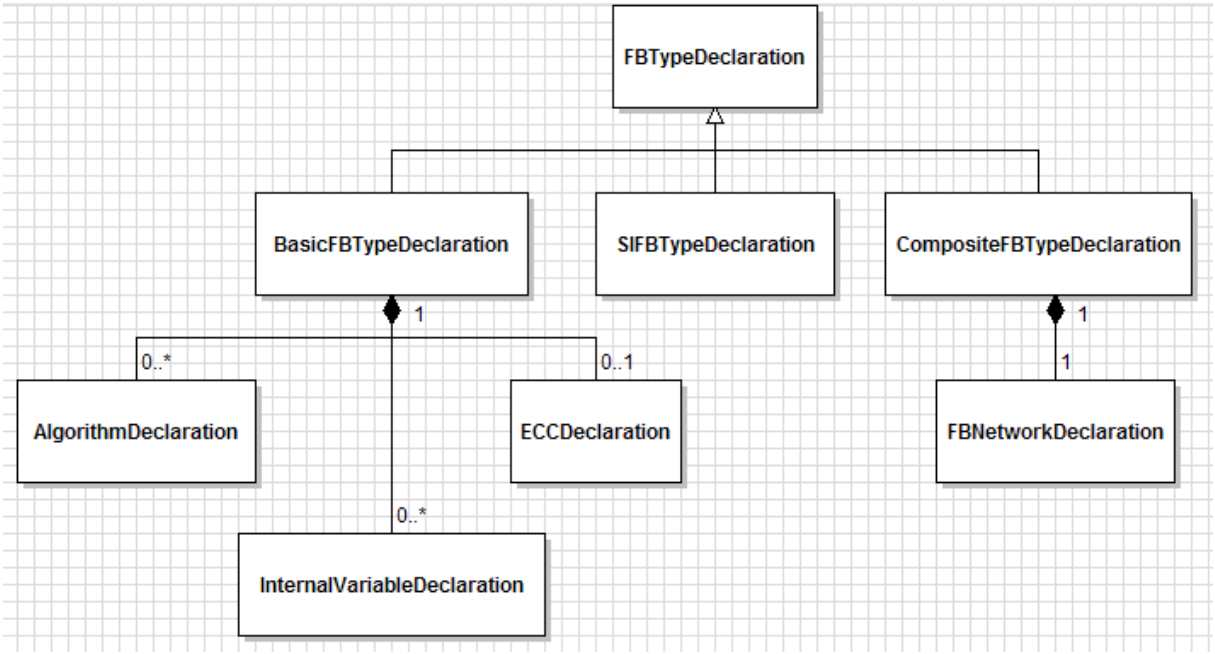


Figure C.5b – Class hierarchy

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.

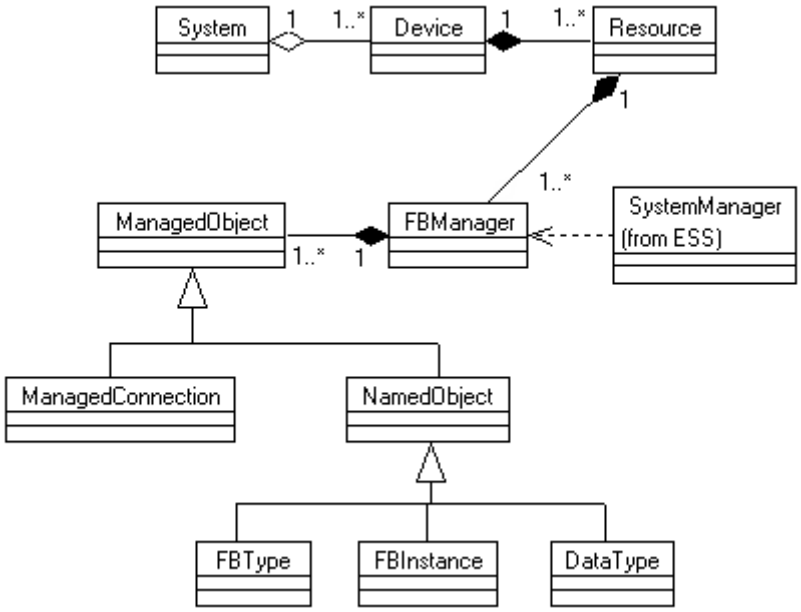


Figure C.6 – IPMCS overview

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*.

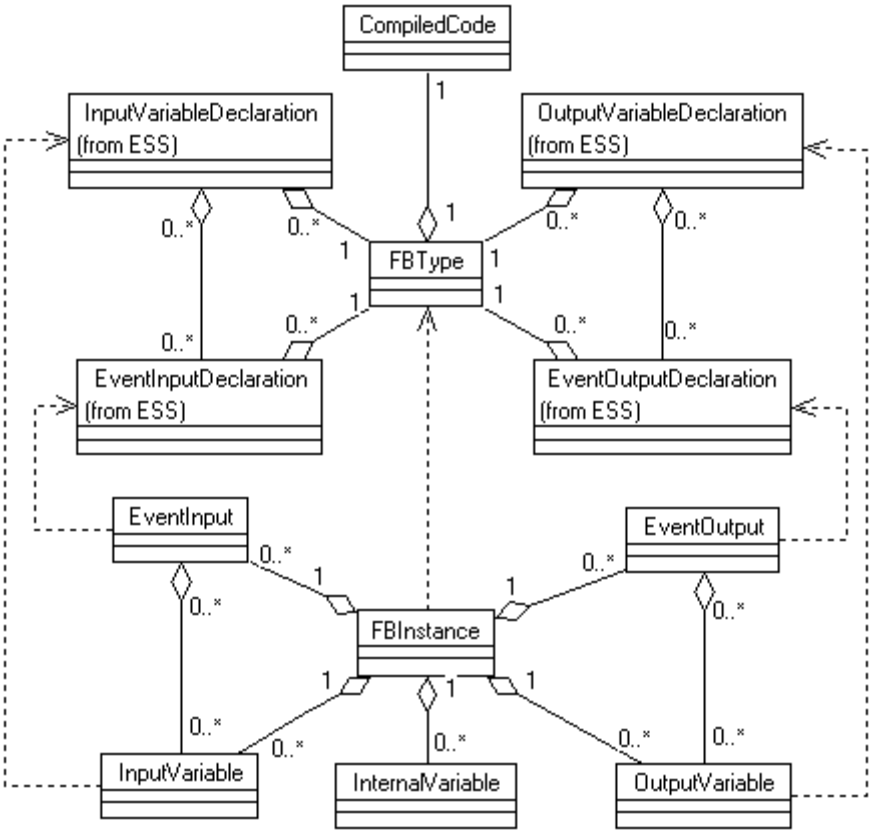


Figure C.7 – Function block types and instances

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 5.2.1. 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

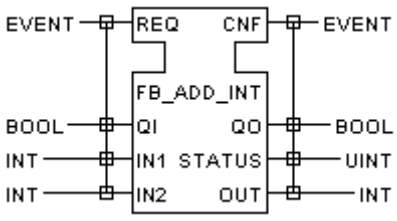
As illustrated in Figure D.1, IEC 61131-3 functions and function blocks can be converted to “simple” function blocks according to the following rules:

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

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). The prefix `F_` instead of `FB_` may optionally be used for simple function block types that are the result of conversions of IEC 61131-3 *functions*.

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:

e) Initialization is as specified in 2.4.2 of IEC 61131-3:2003 for *variables*, and as specified in 2.6 of IEC 61131-3:2003 for Sequential Function Chart (SFC) elements.

f) The occurrence of an INIT+ service primitive is equivalent to "cold restart" initialization as defined in the above mentioned subclauses of IEC 61131-3:2003, followed by an INITO+ service primitive with a STATUS value of zero (0).

g) 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).

h) 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.

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

j) 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
12	Invalid subscript value
13	Array size error

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_, e.g., 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 shall 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 standard shall comply with the requirements of the subclauses 1.5.1, 2.1, 2.2, 2.3 and 2.4 of IEC 61131-3:2003, and the associated elements of Annex B of IEC 61131-3:2003 for the syntax and semantics of textual representation of common elements, with the exceptions and extensions noted in Clause D.5.

Where syntactic productions are not given for non-terminal symbols in Annex B, the corresponding syntactic productions given in Annex B of IEC 61131-3:2003 shall apply.

D.5 Exceptions

Implementations of this standard shall **not** utilize the *directly represented variable* notation defined in 2.4.1.1 of IEC 61131-3:2003 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.

D.6 Interoperation with programmable controllers

D.6.1 Overview

A programmable controller may act as a *server,* as defined in IEC 61131-5, to a *device* as defined in this standard, acting as a *client* as defined in IEC 61131-5. These services are provided using the means defined in IEC 61131-5, and are accessed from the IEC 61499 device using instances of the *function block types* specified in Annex D. These function block types are modeled as *communication function block types* as defined in this standard.

The IEC 61499 client device may exist on a communication network along with the programmable controller acting as a server, or may be an implementer-specific subsystem within the “main processing unit” of the programmable controller, as illustrated in Figure 4 of IEC 61131-5:2000. In either case, the interaction between the IEC 61499 client device and the main processing unit is modelled as occurring over one or more *communication connections* as defined in IEC 61499-1, utilizing instances of the function block types defined in Annex D.

D.6.2 Service conventions

Except for the extensions defined in Annex D, the conventions for naming of input and output variables and events, and for describing the *services* (as defined in this standard) provided by instances of the function block types described in Annex D, are as defined in IEC 61499-1 for the descriptions of *service interface function block types* and *communication function block types*.

For the purposes of Annex D, the `PARAMS` input of type `ANY` defined in this standard is replaced by an `ID` input of type `WSTRING`. The contents of this string specify an **implementation-dependent** representation of the path to the *variable* of interest in the server.

EXAMPLE 1 In the case where the IEC 61499 client device is in logical proximity to the IEC 61131 server, it may be sufficient to simply name the IEC 61131-3 *access path* to the desired variable in the `ID` input, for instance `"CELL_1.CHARLIE"` in the example shown in Figure 19a of IEC 61131-3:2003.

EXAMPLE 2 In the case where the IEC 61499 client device is remotely connected to the IEC 61131-3 server via a communication network, it may be possible to use the `ID` input to encapsulate a Universal Resource Identifier (URI) to specify the desired access path, for instance, `"http://192.168.0.1:61131/CELL_1.CHARLIE"`.

NOTE Where supported by an implementation, the `ID` input may specify an access path to a status variable, such as the pre-defined access paths `P_PCSTATUS` and `P_PCSTATE` specified in IEC 61131-5.

Where used, the contents of the `TYPE` input of a function block type defined in Annex D specify the name of the *data type* of the data (`SD` or `RD`) being transferred. This may be the name of an elementary data type such as `"BOOL"` or a derived data type such as `"ANALOG_16_INPUT_DATA"`.

Where used, the contents of the TASK input of a function block type defined in Annex D specify an **implementation-dependent** representation of the path to the *task* of interest in the server.

EXAMPLE 3 In the case where an IEC 61499 client device is in logical proximity to an IEC 61131-3 server configured as shown in Figure 19a of IEC 61131-3:2003, a path to the task named SLOW_1 in resource STATION_1 could be represented as "CELL_1.STATION_1.SLOW_1".

Values of the STATUS output of the function block types defined in D.6.3 are as given in Table 24 of IEC 61131-5:2000.

D.6.3 Function block types

D.6.3.1 READ

An instance of the READ function block type shown graphically in Figure D.2 and textually in Table D.2 can be used by an IEC 61499 client device to read program or status variable values from an IEC 61131-3 server.

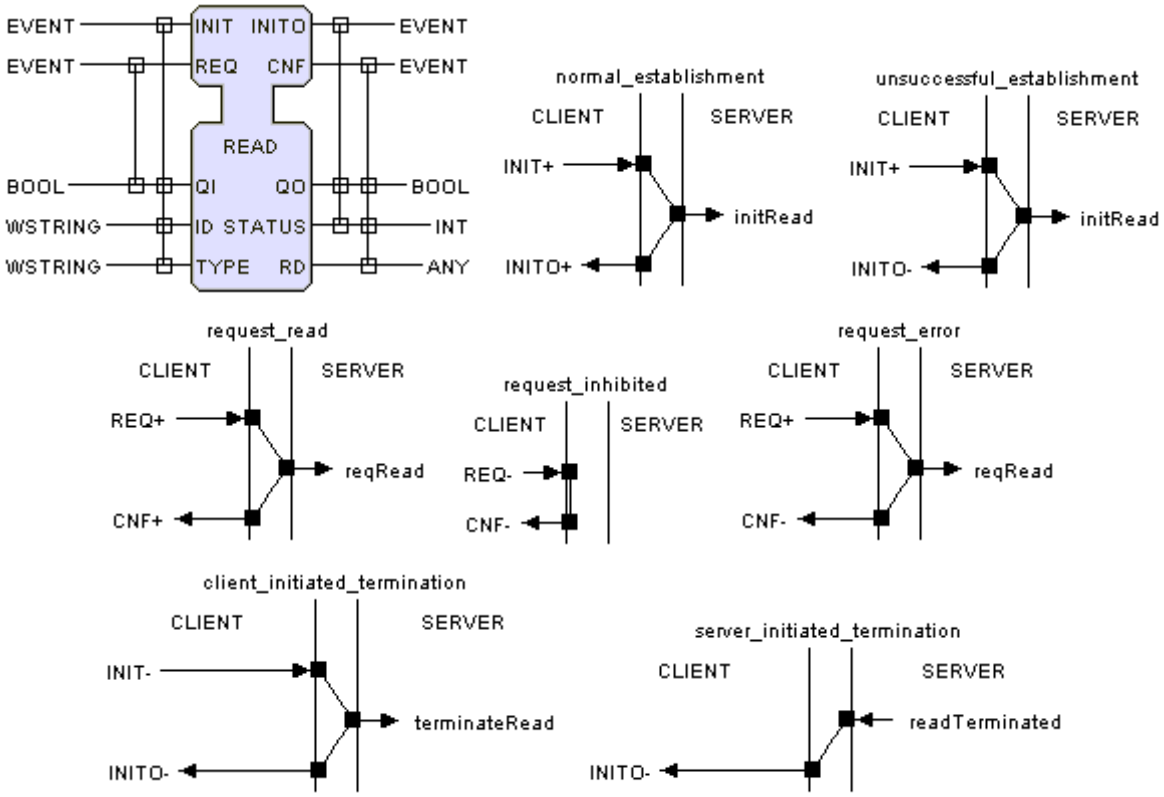


Figure D.2 – Function block type READ

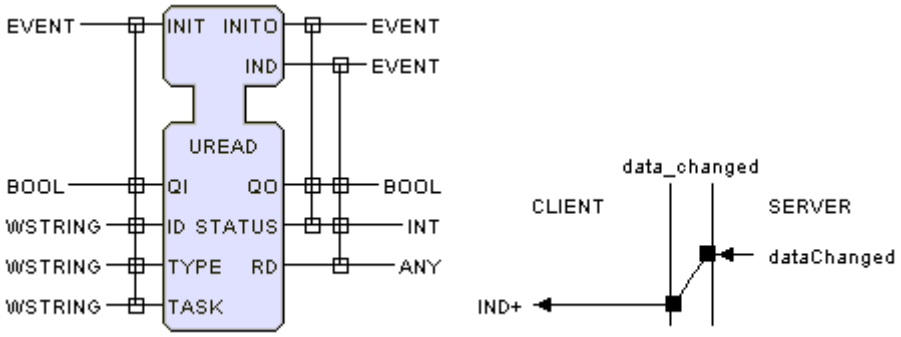
Table D.2 – Source code of function block type READ

```
FUNCTION_BLOCK READ (* Read server status or program variable *)
EVENT_INPUT
  INIT WITH QI,ID,TYPE; (* Initialize/Terminate Service *)
  REQ WITH QI; (* Service Request *)
END_EVENT

EVENT_OUTPUT
  INITO WITH QO,STATUS; (* Initialize/Terminate Confirm *)
  CNF WITH QO,STATUS,RD; (* Confirmation of Requested Service *)
END_EVENT

VAR_INPUT
  QI: BOOL; (* Event Input Qualifier *)
  ID: WSTRING; (* Path to variable to be read *)
  TYPE: WSTRING; (* Data type of RD variable *)
END_VAR

VAR_OUTPUT
  QO: BOOL; (* 1=Normal operation, 0=Abnormal operation *)
  STATUS: INT;
  RD: ANY; (* Variable data from IEC 61131 device *)
END_VAR

SERVICE CLIENT/SERVER
SEQUENCE normal_establishment
  CLIENT.INIT+(ID,TYPE) -> SERVER.initRead(ID,TYPE) -> CLIENT.INITO+();
END_SEQUENCE

SEQUENCE unsuccessful_establishment
  CLIENT.INIT+(ID,TYPE) -> SERVER.initRead(ID,TYPE) -> CLIENT.INITO-(STATUS);
END_SEQUENCE

SEQUENCE request_read
  CLIENT.REQ+() -> SERVER.reqRead(ID) -> CLIENT.CNF+(RD);
END_SEQUENCE

SEQUENCE request_inhibited
  CLIENT.REQ-() -> CLIENT.CNF-(STATUS);
END_SEQUENCE

SEQUENCE request_error
  CLIENT.REQ+() -> SERVER.reqRead(ID) -> CLIENT.CNF-(STATUS);
END_SEQUENCE

SEQUENCE client_initiated_termination
  CLIENT.INIT-() -> SERVER.terminateRead(ID) -> CLIENT.INITO-(STATUS);
END_SEQUENCE

SEQUENCE server_initiated_termination
  SERVER.readTerminated(ID,STATUS) -> CLIENT.INITO-(STATUS);
END_SEQUENCE
END_SERVICE
END_FUNCTION_BLOCK
```

D.6.3.2 UREAD

An instance of the UREAD function block type shown graphically in Figure D.3 and textually in Table D.3 can be used by an IEC 61499 client device to request asynchronous notification of a change in value of a program or status variable from an IEC 61131-3 server. Notification is received via the block's IND event output upon completion of the execution of the specified task when a change in the value of the specified variable (with respect to its value upon initiation of task execution) is detected.

An instance of this function block type can also be used to receive notification of the completion of each execution of the specified task by leaving unspecified the ID and TYPE inputs of the block.



NOTE The graphical representation of other service sequences listed in Table D.3 is similar to Figure D.2.

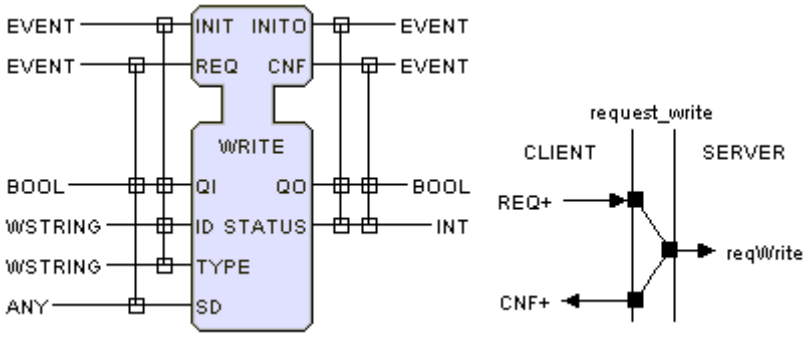
Figure D.3 – Function block type UREAD

Table D.3 – Source code of function block type UREAD

```
FUNCTION_BLOCK UREAD (* Unsolicited read of IEC 61131 program or status variable *)

EVENT_INPUT
  INIT WITH QI,ID,TASK,TYPE; (* Initialize/Terminate Service *)
END_EVENT

EVENT_OUTPUT
  INITO WITH QO,STATUS; (* Initialize/Terminate Confirm *)
  IND WITH QO,STATUS,RD; (* Indication of changed RD value *)
END_EVENT

VAR_INPUT
  QI: BOOL; (* Event Input Qualifier *)
  ID: WSTRING; (* Path to variable to be read *)
  TYPE: WSTRING; (* Data type of RD variable *)
  TASK: WSTRING; (* Path to IEC 61131 TASK triggering read on changed value *)
END_VAR

VAR_OUTPUT
  QO: BOOL; (* 1=Normal operation, 0=Abnormal operation *)
  STATUS: INT;
  RD: ANY; (* Input data from resource *)
END_VAR

SERVICE CLIENT/SERVER
SEQUENCE normal_establishment
  CLIENT.INIT+(ID,TYPE,TASK) -> SERVER.initURead(ID,TYPE,TASK) -> CLIENT.INITO+();
END_SEQUENCE

SEQUENCE unsuccessful_establishment
  CLIENT.INIT+(ID,TYPE,TASK) -> SERVER.initURead(ID,TYPE,TASK)
    -> CLIENT.INITO-(STATUS);
END_SEQUENCE

SEQUENCE data_changed
  SERVER.dataChanged() -> CLIENT.IND+(RD);
END_SEQUENCE

SEQUENCE client_initiated_termination
  CLIENT.INIT-() -> SERVER.terminateURead() -> CLIENT.INITO-(STATUS);
END_SEQUENCE

SEQUENCE server_initiated_termination
  SERVER.UReadTerminated(ID,STATUS) -> CLIENT.INITO-(STATUS);
END_SEQUENCE
END_SERVICE
END_FUNCTION_BLOCK
```

D.6.3.3 WRITE

An instance of the `WRITE` function block type shown graphically in Figure D.4 and textually in Table D.4 can be used by an IEC 61499 client device to write variable data values to an IEC 61131-3 server.



NOTE The graphical representation of other service sequences listed in Table D.4 is similar to Figure D.2.

Figure D.4 – Function block type WRITE

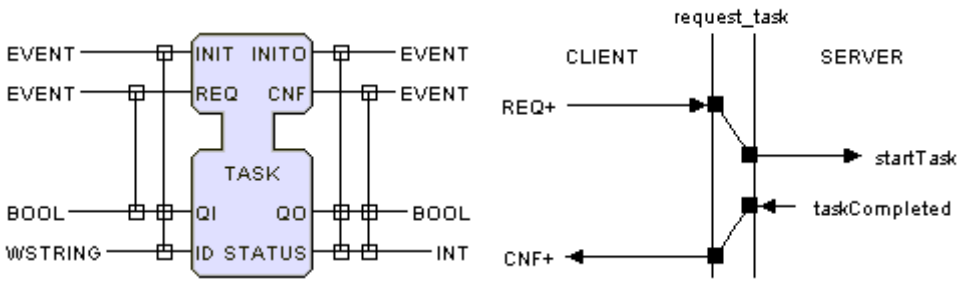
Table D.4 – Source code of function block type WRITE

```
FUNCTION_BLOCK WRITE (* Write a variable value to an IEC 61131 server *)
EVENT_INPUT
  INIT WITH QI,ID,TYPE; (* Initialize/Terminate Service *)
  REQ WITH QI,SD; (* Service Request *)
END_EVENT

EVENT_OUTPUT
  INITO WITH QO,STATUS; (* Initialize/Terminate Confirm *)
  CNF WITH QO,STATUS; (* Confirmation of Requested Service *)
END_EVENT

VAR_INPUT
  QI: BOOL; (* Event Input Qualifier *)
  ID: WSTRING; (* Path to variable to be written *)
  TYPE: WSTRING; (* Data type of SD variable *)
  SD: ANY; (* Variable value to write *)
END_VAR

VAR_OUTPUT
  QO: BOOL; (* 1=Normal operation, 0=Abnormal operation *)
  STATUS: INT;
END_VAR

SERVICE CLIENT/SERVER
SEQUENCE normal_establishment
  CLIENT.INIT+(ID,TYPE) -> SERVER.initWrite(ID,TYPE) -> CLIENT.INITO+();
END_SEQUENCE

SEQUENCE unsuccessful_establishment
  CLIENT.INIT+(ID,TYPE) -> SERVER.initWrite(ID,TYPE) -> CLIENT.INITO-(STATUS);
END_SEQUENCE

SEQUENCE request_write
  CLIENT.REQ+(ID,SD) -> SERVER.reqWrite(ID,SD) -> CLIENT.CNF+();
END_SEQUENCE

SEQUENCE request_inhibited
  CLIENT.REQ-(ID,SD) -> CLIENT.CNF-(STATUS);
END_SEQUENCE

SEQUENCE request_error
  CLIENT.REQ+(ID,SD) -> SERVER.reqWrite(ID,SD) -> CLIENT.CNF-(STATUS);
END_SEQUENCE

SEQUENCE client_initiated_termination
  CLIENT.INIT-() -> SERVER.terminateWrite(ID) -> CLIENT.INITO-(STATUS);
END_SEQUENCE

SEQUENCE server_initiated_termination
  SERVER.writeTerminated(ID,STATUS) -> CLIENT.INITO-(STATUS);
END_SEQUENCE
END_SERVICE
END_FUNCTION_BLOCK
```

D.6.3.4 TASK

An instance of the `TASK` function block type shown graphically in Figure D.5 and textually in Table D.5 can be used by an IEC 61499 client device to request the execution of a task on an IEC 61131-3 server.

When an implementation supports this feature, no value is configured for either the `SINGLE` or `INTERVAL` input of the corresponding TASK block as defined in Table 50 of IEC 61131-3:2003; rather, execution of the corresponding task is triggered as shown in the `request_task` service sequence shown in Figure D.5.



NOTE The graphical representation of other service sequences listed in Table D.5 is similar to Figure D.2.

Figure D.5 – Function block type TASK

Table D.5 – Source code of function block type TASK

```
FUNCTION_BLOCK TASK (* Trigger IEC 61131 task *)
EVENT_INPUT
  INIT WITH QI,ID; (* Initialize/Terminate Service *)
  REQ WITH QI; (* Service Request *)
END_EVENT

EVENT_OUTPUT
  INITO WITH QO,STATUS; (* Initialize/Terminate Confirm *)
  CNF WITH QO,STATUS; (* Confirmation of Requested Service *)
END_EVENT

VAR_INPUT
  QI: BOOL; (* Event Input Qualifier *)
  ID: WSTRING; (* Path to task to be triggered *)
END_VAR

VAR_OUTPUT
  QO: BOOL; (* 1=Normal operation, 0=Abnormal operation *)
  STATUS: INT;
END_VAR

SERVICE CLIENT/SERVER
SEQUENCE normal_establishment
  CLIENT.INIT+(ID) -> SERVER.initTask(ID) -> CLIENT.INITO+();
END_SEQUENCE

SEQUENCE unsuccessful_establishment
  CLIENT.INIT+(ID) -> SERVER.init(ID) -> CLIENT.INITO-(STATUS);
END_SEQUENCE

SEQUENCE request_task
  CLIENT.REQ+(ID) -> SERVER.reqTask(ID) -> CLIENT.CNF+();
END_SEQUENCE

SEQUENCE request_inhibited
  CLIENT.REQ-() -> CLIENT.CNF-(STATUS);
END_SEQUENCE

SEQUENCE request_error
  CLIENT.REQ+(ID) -> SERVER.reqTask(ID) -> CLIENT.CNF-(STATUS);
END_SEQUENCE

SEQUENCE client_initiated_termination
  CLIENT.INIT-() -> SERVER.terminateTask(ID) -> CLIENT.INITO-(STATUS);
END_SEQUENCE

SEQUENCE server_initiated_termination
  SERVER.taskTerminated(ID,STATUS) -> CLIENT.INITO-(STATUS);
END_SEQUENCE
END_SERVICE
END_FUNCTION_BLOCK
```

D.6.4 Compliance

The specifications given in Annex D may be referenced in compliance profiles according to the rules given in IEC 61499-4.

When a programmable controller system compliant with IEC 61131-3 supports interoperability with one or more of the IEC 61499 function block types defined in Annex D, it should include in its list of supported features a reference to the supported features taken from Table D.6, and should include specifications of the values for implementation specific features and parameters as defined in 8.1 and 8.2 of IEC 61131-5:2000, respectively.

Table D.6 – IEC 61499 interoperability features

No.	Description
1	READ function block type
2	UREAD function block type
3	WRITE function block type
4	TASK function block type

Annex E
(informative)

Information exchange

E.1 Use of application layer facilities

Subclause 7.1.3.2 of ISO/IEC 7498-1:1994 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.* The communication function blocks defined in Clause E.2 may use these facilities, when provided by appropriate application-entities, in the following ways.

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

b) The following facilities may be used during service initialization as represented by the INIT and INITO events, using elements of the PARAMS data structure as necessary:

- identification of the intended communications partners;
- determination of the acceptable quality of service;
- agreement on responsibility for error recovery;
- agreement on security aspects;
- identification of abstract syntax.

c) Facilities for *selection of mode of dialog* may be used by the specific function block types, e.g., by a SUBSCRIBER to ensure that it is interacting properly with a PUBLISHER.

Many of the facilities listed above may not be provided by application-entities of industrial-process measurement and control systems (IPMCSs). In this case, the communication function blocks shall implement equivalent facilities to provide the required services.

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, transfer syntaxes for both information transfer and application management are defined in Clause E.3.

NOTE 1 See ISO/IEC 7498-1 for definitions of terms used in this annex, but not defined in this standard.

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

NOTE 3 The contents of Annex E could be considered normative in that compliance profiles as defined in IEC 61499-4, other standards and specifications can specify a context within which some or all of its provisions are employed.

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 Clause E.2.

E.2.2 Function blocks for unidirectional transactions

Figures E.1 through 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 standard.

NOTE 5 Data transfer might be required in order to determine whether `RD_1,...,RD_m` meet the constraints expressed in Note 3.

NOTE 6 The transfer syntaxes defined in Clause E.3 can be used to make the determination described in Note 5.

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

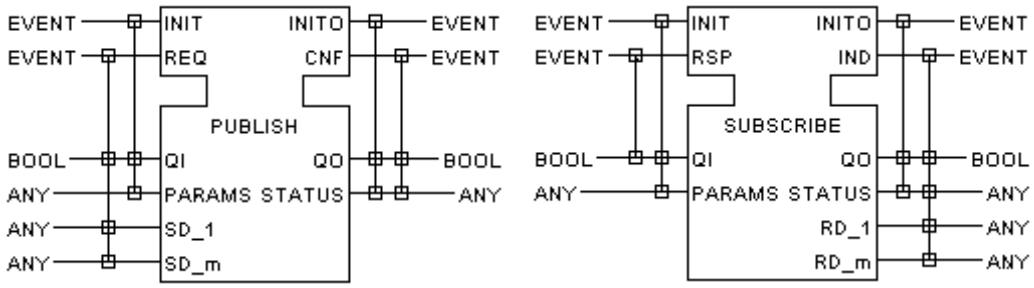


Figure E.1 – Type specifications for unidirectional transactions

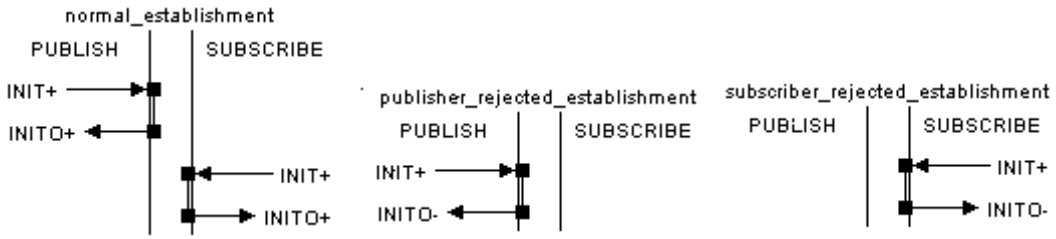


Figure E.2 – Connection establishment for unidirectional transactions

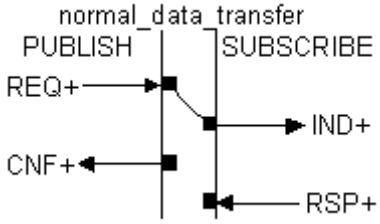


Figure E.3 – Normal unidirectional data transfer

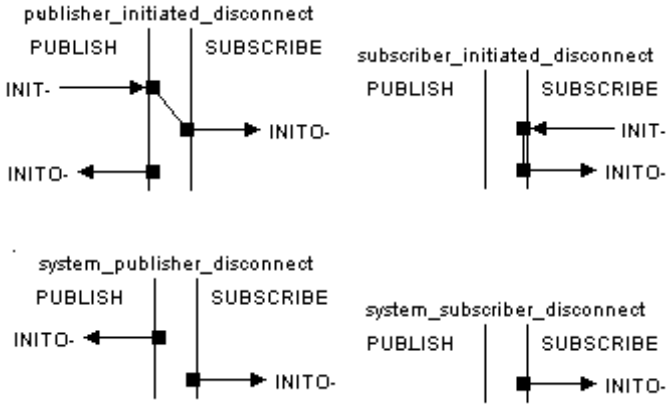


Figure E.4 – Connection release in unidirectional data transfer

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 `RD_1,...,RD_m` and `RD_1,...,RD_n` meet the constraints expressed in Notes 3 and 4.

NOTE 6 The transfer syntaxes defined in Clause E.3 may be used to make the determination described in Note 5.

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

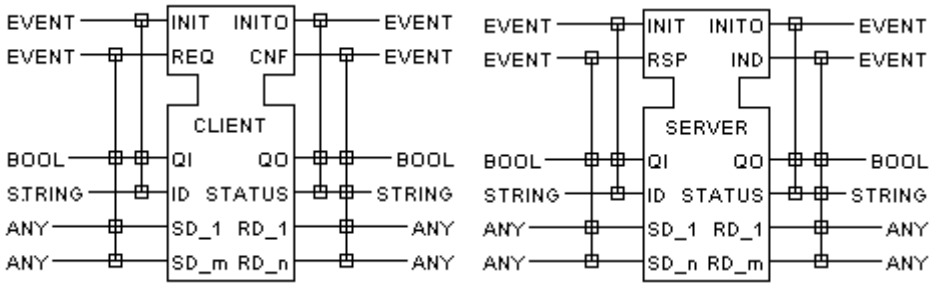


Figure E.5 – Type specifications for bidirectional transactions

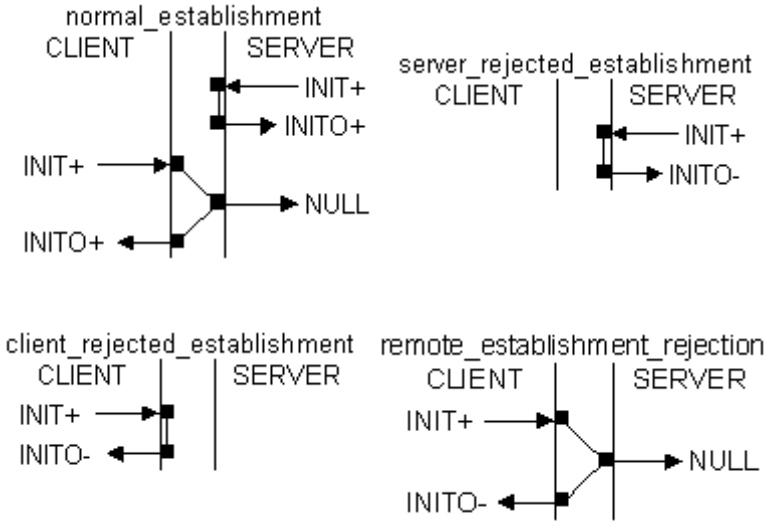


Figure E.6 – Connection establishment for bidirectional transaction

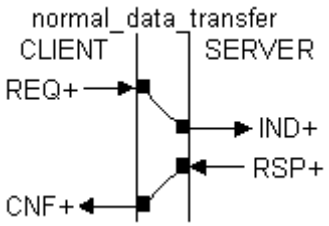


Figure E.7 – Bidirectional data transfer

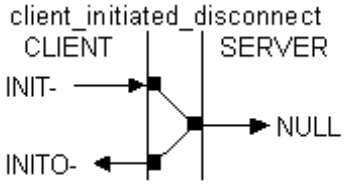


Figure E.8a – Client initiated

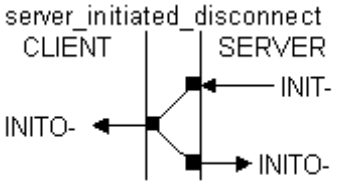


Figure E.8b – Server initiated

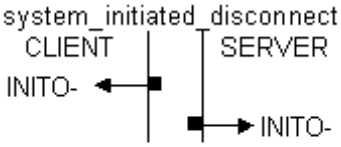


Figure E.8c – System initiated

Figure E.8 – Connection release in bidirectional data transfer

E.3 Transfer syntaxes

E.3.1 Background

A transfer syntax is defined in terms of an *abstract syntax* describing the types of data to be transferred, and a set of *encoding rules* for encoded representation of instances of the data types so defined. Subclause E.3.2 utilizes Abstract Syntax Notation One (ASN.1), as defined in ISO/IEC 8824-1, to define the IEC61499-FBDATA syntax for data transfer.

Two sets of encoding rules are given in Annex E:

a) Subclause E.3.3.1 defines BASIC encoding rules, utilizing the rules defined in ISO/IEC 8825-1.

b) Subclause E.3.3.2 utilizes the special characteristics of the data types in the IEC61499-FBDATA syntax to obtain a set of COMPACT encoding rules according to the following principles:

- Where the number of "contents octets" is fixed, "length octets" are not used in the encoding.
- Special encodings are used to minimize the number of octets and encoding/decoding effort required for fixed length types.
- "Identifier octets" are not used for individual elements of STRUCT and ARRAY data types, since the type of each element is fixed in the corresponding *type declaration*.

E.3.2 IEC61499-FBDATA abstract syntax

The transfer syntax obtained by applying the COMPACT encoding rules in E.3.3.2 to the abstract syntax in E.3.2 is recommended for:

- transferring values from the `SD` inputs of a *communication function block* to the `RD` outputs of the communication function block(s) at the opposite end of a *communication connection*;
- determining whether the constraints on corresponding number and type of variables between `SD` inputs and `RD` outputs are met as noted in Figures E.1 and E.5.

The use of the abstract syntax defined in E.3.2 for the transfer of data expressed as *literals* and values of *variables* is subject to the following semantic RULES:

a) Where the name of a data type in this module (for example, `BOOL`) corresponds to the name of a data type defined in IEC 61131-3, the type definition given is intended for the transfer of data of the corresponding IEC 61131-3 data type.

b) The values of "VisibleString" for the data types `DATE` and `TIME_OF_DAY` is restricted to the textual syntax for these data types as defined in IEC 61131-3.

c) The notation `[typeID]` implies that the tag of the data consists of the value of the `ASN.1 tag` of the corresponding derived data type, established as specified in Annex A of IEC 61499-2:2005 or by other means beyond the scope of this standard.

d) The value of an `EnumeratedData` item consists of the cardinal position (beginning at zero) of the corresponding identifier in the sequence of identifiers defined for the corresponding enumerated data type, established as specified in IEC 61131-3.

e) The specific type of a `SubrangeData` item is as for its particular *subrange data type,* declared as specified in IEC 61131-3.

f) The type of the elements of an `ARRAY` data item is established as specified for *array data types* in IEC 61131-3.

g) The types of the elements of a `STRUCT` data item are established as specified for *structured data types* in IEC 61131-3.

ASN.1 MODULE

```
IEC61499-FBDATA DEFINITIONS::=

BEGIN

EXPORTS FBDataSequence, FBData, ElementaryData, BOOL, FixedLengthInteger,
     FixedLengthReal, TIME, AnyDate, AnyString, FixedLengthBitString,
     SignedInteger, UnsignedInteger, REAL, LREAL, DATE, TIME_OF_DAY,
     DATE_AND_TIME, STRING, WSTRING, BYTE, WORD, DWORD, LWORD,
     DirectlyDerivedData, EnumeratedData, SubrangeData, ARRAY, STRUCT;

FBDataSequence::= [APPLICATION 23] IMPLICIT SEQUENCE OF FBData

FBData::= CHOICE{ElementaryData, DerivedData}
```

```
ElementaryData::= CHOICE{
     BOOL,
     FixedLengthInteger,
     FixedLengthReal,
     TIME,
     AnyDate,
     AnyString,
     FixedLengthBitString}

FixedLengthInteger::= CHOICE{SignedInteger, UnsignedInteger}

SignedInteger::= CHOICE{SINT, INT, DINT, LINT}

UnsignedInteger::= CHOICE{USINT, UINT, UDINT, ULINT}

FixedLengthReal::= CHOICE{REAL, LREAL}

AnyDate::= CHOICE{DATE, TIME_OF_DAY, DATE_AND_TIME}

AnyString::= CHOICE{STRING, WSTRING}

FixedLengthBitString::= CHOICE{BYTE, WORD, DWORD, LWORD}

BOOL::= CHOICE{BOOL0, BOOL1}

BOOL0::= [APPLICATION 0] IMPLICIT NULL

BOOL1::= [APPLICATION 1] IMPLICIT NULL

SINT::= [APPLICATION 2] IMPLICIT INTEGER(-128..127)

INT::= [APPLICATION 3] IMPLICIT INTEGER(-32768..32767)

DINT::= [APPLICATION 4] IMPLICIT INTEGER(-2147483648..2147483647)

LINT::= [APPLICATION 5]
     IMPLICIT INTEGER(-9223372036854775808..9223372036854775807)

USINT::= [APPLICATION 6] IMPLICIT INTEGER(0..255)

UINT::= [APPLICATION 7] IMPLICIT INTEGER(0..65535)

UDINT::= [APPLICATION 8] IMPLICIT INTEGER(0..4294967295)

ULINT::= [APPLICATION 9] IMPLICIT INTEGER(0..18446744073709551615)

REAL::= [APPLICATION 10] IMPLICIT OCTET STRING (SIZE(4))

LREAL::= [APPLICATION 11] IMPLICIT OCTET STRING (SIZE(8))

TIME::= [APPLICATION 12] IMPLICIT LINT -- Duration in 1µs units

DATE::= [APPLICATION 13] IMPLICIT ULINT -- See Table E.1.

TIME_OF_DAY::= [APPLICATION 14] IMPLICIT ULINT -- See Table E.1.

DATE_AND_TIME::= [APPLICATION 15] IMPLICIT ULINT -- See Table E.1.

STRING::= [APPLICATION 16] IMPLICIT OCTET STRING -- 1 octet/char

BYTE::= [APPLICATION 17] IMPLICIT BIT STRING (SIZE(8))

WORD::= [APPLICATION 18] IMPLICIT BIT STRING (SIZE(16))

DWORD::= [APPLICATION 19] IMPLICIT BIT STRING (SIZE(32))

LWORD::= [APPLICATION 20] IMPLICIT BIT STRING (SIZE(64))

WSTRING::= [APPLICATION 21] IMPLICIT OCTET STRING -- 2 octets/char

DerivedData::= CHOICE{
     DirectlyDerivedData,
     EnumeratedData,
     SubrangeData,
     ARRAY,
     STRUCT}
```

```
DirectlyDerivedData::= [typeID] IMPLICIT ElementaryData

EnumeratedData::= [typeID] IMPLICIT UINT

SubrangeData::= [typeID] IMPLICIT FixedLengthInteger

ARRAY::= CHOICE {ArrayVariable, TypedArray}

ArrayVariable::= [APPLICATION 22] IMPLICIT FBDataSequence -- same type

TypedArray::= [typeID] IMPLICIT FBDataSequence - same type

STRUCT::= [typeID] IMPLICIT SEQUENCE -- different types

END
```

E.3.3 Encoding rules

E.3.3.1 BASIC encoding

This encoding shall be the result of applying the basic encoding rules of ISO/IEC 8825-1 to variables of the types defined in E.3.2.

E.3.3.2 COMPACT encoding

This encoding shall be the result of modifying the rules for BASIC encoding given in E.3.3.1 as follows.

a) "Length octets" shall not be included in the encoding of values of the data types shown in Table E.1.

b) The length (in octets) and encoding of the "contents octets" described in ISO/IEC 8825-1 shall be as defined in Table E.1 for values of the data types shown there.

c) Encoding of variables of `TIME`, `DirectlyDerivedData`, `EnumeratedData`, or `SubrangeData` types shall follow the same encoding rules as the base type.

d) "Type octets" shall not be included in the encoding of individual elements of `STRUCT` types, except for the encoding of elements of type `BOOL`, which shall be encoded according to rule (1) of Table E.1.

e) The encoding of values of `STRING` and `WSTRING` types shall be primitive.

f) The encoding of `ARRAY` elements shall be *constructed* in the sense of ISO/IEC 8825-1, with the following provisions for COMPACT encoding:

   1) The "length" subfield of the `ARRAY` element shall be encoded as a value of the `UINT` type without identifier or length octets, i.e., as a 16-bit unsigned integer;

      NOTE 1 This would appear to restrict the maximum number of elements of an `ARRAY` to 65535. However, the actual length may be further restricted by the maximum number of octets that can be transferred by the underlying transport protocol.

      EXAMPLE For UDP messages with a maximum number of is 65508 octets, the maximum transmittable length of an `ARRAY` of `BYTE` elements would be (maximum octets - tag octets - length octets - element type octets)/(element length) = (65508-1-2-1)/1 = 65504 elements.

   2) COMPACT encoding shall be used for the first element of the "values" field;

   3) Subsequent elements, if any, shall be encoded using the COMPACT syntax without an "identifier" subfield, except for elements of type `BOOL`, which shall be encoded according to rule (1) of Table E.1;

   4) If the specified length of the received `ARRAY` is less than the locally allocated space, the remaining elements of the local array are unaffected; if the length of the received `ARRAY` is greater than the locally allocated space, the remaining received elements are ignored.

NOTE 2 Since `ARRAY` is a subclass of `FBData`, a multidimensional `ARRAY` can be encoded recursively as an `ARRAY` whose elements are `ARRAY` elements.

Table E.1 – COMPACT encoding of fixed length data types

Data type	Contents octets	
	Length	Encoding rule
BOOL	0	(1)
SINT	1	(2)
INT	2	(2)
DINT	4	(2)
LINT	8	(2)
USINT	1	(3)
UINT	2	(3)
UDINT	4	(3)
ULINT	8	(3)
REAL	4	(4)
LREAL	8	(4)
DATE	8	(5)
TIME	8	(7)
TIME_OF_DAY	12	(5)
DATE_AND_TIME	20	(5)
BYTE	1	(6)
WORD	2	(6)
DWORD	4	(6)
LWORD	8	(6)

ENCODING RULES FOR TABLE E.1

(1) Values of this data type shall be encoded as a single identifier octet containing the tag encoding for the `BOOL0` or `BOOL1` class, as defined in E.3.2, corresponding to values of `FALSE (0)` or `TRUE (1)`, respectively.

(2) Values of these `SignedInteger` data types shall be encoded in the same manner as an `UnsignedInteger` of the same length as the `SignedInteger` type with a value of $N - N_{min}$, where N is the value of the `SignedInteger` variable to be encoded and N_{min} is the lower end point of the value range of the `SignedInteger` subtype as defined in E.3.2.

(3) Values of these `UnsignedInteger` data types shall be encoded by numbering the bits in the contents octets, starting with bit 1 of the last octet as bit zero and ending the numbering with bit 8 of the first octet. Each bit is assigned a value of 2^N, where N is its position in the above numbering sequence. The value of the unsigned integer is obtained by summing the numerical values assigned to each bit for those bits which are set to one.

(4) Values of these data types shall be encoded as 32-bit single format and 64-bit double format numbers, respectively, as defined in ISO/IEC/IEEE 60559, where the "lsb" defined in ISO/IEC/IEEE 60559 corresponds to "bit zero" as defined in Rule (3).

(5) Values of these types shall be encoded as for type `ULINT`, representing the number of milliseconds since midnight for `TIME_OF_DAY`, the number of milliseconds since `1970-01-01-00:00:00.000` for `DATE_AND_TIME`, or the number of milliseconds from `1970-01-01-00:00:00.000` to `YYYY-MM-DD-00:00:00.000` for `DATE`, where `YYYY-MM_DD` is the current date.

(6) Encoding of values of these `FixedLengthBitString` data types shall be primitive, and shall be obtained by placing the bits in the bitstring, commencing with the first bit and proceeding to the trailing bit, in bits 8 to 1 of the first contents octet, followed in turn by bits 8 to 1 of each of the subsequent octets, where the notation "first bit" and "trailing bit" is specified in ISO/IEC 8824-1.

(7) Encoding of values of this data type shall be the same as for values of type `LINT`, representing a time interval in units of 1 μs.

Annex F
(normative)

Textual specifications

Annex F provides textual specifications, in the syntax defined in Annex B, for all function block and adapter types illustrated in this standard. The contents of Annex F are normative to the extent defined in the description of each such function block type or adapter type in this standard.

NOTE The specifications are listed alphabetically by type name.

```
(*************************************************************************)
FUNCTION_BLOCK E_CTU (* Event-Driven Up Counter *)
EVENT_INPUT
  CU WITH PV; (* Count Up *)
  R; (* Reset *)
END_EVENT
EVENT_OUTPUT
  CUO WITH Q,CV; (* Count Up Output Event *)
  RO WITH Q,CV; (* Reset Output Event *)
END_EVENT
VAR_INPUT
  PV: UINT; (* Preset Value *)
END_VAR
VAR_OUTPUT
  Q: BOOL; (* CV>=PV *)
  CV: UINT;
END_VAR
EC_STATES
  START;
  CU: CU -> CUO;
  R: R -> RO;
END_STATES
EC_TRANSITIONS
  START TO CU:= CU [CV<65535];
  CU TO START:= 1;
  START TO R:= R;
  R TO START:= 1;
END_TRANSITIONS
ALGORITHM CU IN ST: (* Count Up *)
CV:= CV + 1;
Q:= (CV >= PV);
END_ALGORITHM
ALGORITHM R IN ST: (* Reset *)
CV:= 0;
Q:= FALSE;
END_ALGORITHM
END_FUNCTION_BLOCK
(*************************************************************************)
FUNCTION_BLOCK E_CYCLE (* Periodic (cyclic) Generation of an Event *)
EVENT_INPUT
  START WITH DT;
  STOP;
END_EVENT
EVENT_OUTPUT
 EO;   (* Periodic event at period DT, starting at DT after GO *)
END_EVENT
VAR_INPUT
  DT: TIME; (* Period between events *)
END_VAR
FBS
 DLY: E_DELAY;
END_FBS
EVENT_CONNECTIONS
 START TO DLY.START;
```

```
 STOP TO DLY.STOP;
 DLY.EO TO DLY.START;
 DLY.EO TO EO;
END_CONNECTIONS
DATA_CONNECTIONS
 DT TO DLY.DT;
END_CONNECTIONS
END_FUNCTION_BLOCK
(*=================================================================*)
FUNCTION_BLOCK E_D_FF (* Event-driven Data(D)Latch *)
EVENT_INPUT
  CLK WITH D; (* Data Clock *)
END_EVENT
EVENT_OUTPUT
  EO WITH Q; (* Output Event when Q output changes *)
END_EVENT
VAR_INPUT
  D: BOOL; (* Data Input *)
END_VAR
VAR_OUTPUT
  Q: BOOL; (* Latched Data *)
END_VAR
EC_STATES
  Q0; (* Q is FALSE initially *)
  RESET: LATCH -> EO; (* Reset Q and issue EO *)
  SET: LATCH -> EO; (* Latch and issue EO *)
END_STATES
EC_TRANSITIONS
  Q0 TO SET:= CLK [D];
  SET TO RESET:= CLK [NOT D];
  RESET TO SET:= CLK [D];
END_TRANSITIONS
ALGORITHM LATCH IN ST:
Q:=D;
END_ALGORITHM
END_FUNCTION_BLOCK
(*=================================================================*)
FUNCTION_BLOCK E_DELAY
 (* Delayed propagation of an event - Cancellable *)
EVENT_INPUT
 START WITH DT; (* Begin Delay *)
 STOP;       (* Cancel Delay *)
END_EVENT
EVENT_OUTPUT
 EO;      (* Delayed Event *)
END_EVENT
VAR_INPUT
 DT: TIME; (* Delay Time *)
END_VAR
SERVICE E_DELAY/RESOURCE
SEQUENCE event_delay
 E_DELAY.START(DT) ->E_DELAY.EO();
END_SEQUENCE
SEQUENCE delay_canceled
 E_DELAY.START(DT);
 E_DELAY.STOP();
END_SEQUENCE
SEQUENCE no_multiple_delay
 E_DELAY.START(DT);
 E_DELAY.START(DT);
 ->E_DELAY.EO();
END_SEQUENCE
END_SERVICE
END_FUNCTION_BLOCK
(*=================================================================*)
FUNCTION_BLOCK E_DEMUX (* Event demultiplexer *)
EVENT_INPUT
 EI WITH K;  (* Event to demultiplex *)
END_EVENT
```

```
EVENT_OUTPUT
 EO0;
 EO1;
 EO2;
 EO3;      (* Number of outputs is implementation dependent *)
END_EVENT
VAR_INPUT
 K: UINT;  (* Event index, maximum is implementation dependent *)
END_VAR
EC_STATES
 START; (* Initial State *)
 TRIGGERED; (* Intermediate state after EI arrives *)
 EO0: -> EO0;
 EO1: -> EO1;
 EO2: -> EO2;
 EO3: -> EO3;
END_STATES
EC_TRANSITIONS
 START TO TRIGGERED:= EI;
 TRIGGERED TO EO0:= [K=0];
 TRIGGERED TO EO1:= [K=1];
 TRIGGERED TO EO2:= [K=2];
 TRIGGERED TO EO3:= [K=3];
 TRIGGERED TO START:= [K>3];
 EO0 TO START:= 1;
 EO1 TO START:= 1;
 EO2 TO START:= 1;
 EO3 TO START:= 1;
END_TRANSITIONS
END_FUNCTION_BLOCK
================================================================
FUNCTION_BLOCK E_F_TRIG (* Boolean falling edge detection *)
EVENT_INPUT
 EI WITH QI;   (* Event Input *)
END_EVENT
EVENT_OUTPUT
 EO;  (* Event Output *)
END_EVENT
VAR_INPUT
 QI: BOOL;  (* Boolean input for falling edge detection *)
END_VAR
FBS
 D: E_D_FF;
 SW: E_SWITCH;
END_FBS
EVENT_CONNECTIONS
  EI TO D.CLK;
  D.EO TO SW.EI;
  SW.EO0 TO EO;
END_CONNECTIONS
DATA_CONNECTIONS
  QI TO D.D;
  D.Q TO SW.G;
END_CONNECTIONS
END_FUNCTION_BLOCK
================================================================
FUNCTION_BLOCK E_MERGE (* Merge (OR) of multiple events *)
EVENT_INPUT
  EI1; (* First input event *)
  EI2; (* Second input event *)
  END_EVENT
EVENT_OUTPUT EO; (* Output Event *)
END_EVENT
EC_STATES
  START;  (* Initial State *)
  EO:    (* Issue EO Event *)
    ->EO;
END_STATES
EC_TRANSITIONS
  START TO EO:= EI1;
```

```
  START TO EO:= EI2;
  EO TO START:= 1;
END_TRANSITIONS
END_FUNCTION_BLOCK
(*=================================================================*)
FUNCTION_BLOCK E_N_TABLE (* Generation of a finite train of separate events,
table driven *)
EVENT_INPUT
  START WITH DT, N;
  STOP;
END_EVENT
EVENT_OUTPUT
 EO0;   (* N events at periods DT, starting at DT[0] after START *)
 EO1;
 EO2;
 EO3;   (* Extensible *)
END_EVENT
VAR_INPUT
  DT: TIME[3]; (* Periods between events *)
  N: UINT;  (* Number of events to generate (=3 in this example) *)
END_VAR
SERVICE E_N_TABLE/RESOURCE
SEQUENCE typical_operation
  E_N_TABLE.START(DT,N) -> E_N_TABLE.EO0() -> E_N_TABLE.EO1() ->
E_N_TABLE.EO2() -> E_N_TABLE.EO3();
END_SEQUENCE
END_SERVICE
END_FUNCTION_BLOCK
(*=================================================================*)
FUNCTION_BLOCK E_PERMIT (* Permissive propagation of an event *)
EVENT_INPUT EI WITH PERMIT; (* Event input *)
END_EVENT
EVENT_OUTPUT EO; (* Event output *)
END_EVENT
VAR_INPUT PERMIT: BOOL; END_VAR
EC_STATES
  START;  (* Initial State *)
  EO:    (* Issue EO Event *)
    ->EO;
END_STATES
EC_TRANSITIONS
  START TO EO:= EI [PERMIT];
  EO TO START:= 1;
END_TRANSITIONS
END_FUNCTION_BLOCK
(*=================================================================*)
FUNCTION_BLOCK E_R_TRIG (* Boolean rising edge detection *)
EVENT_INPUT
 EI WITH QI;   (* Event Input *)
END_EVENT
EVENT_OUTPUT
 EO;  (* Event Output *)
END_EVENT
VAR_INPUT
 QI: BOOL;  (* Boolean input for rising edge detection *)
END_VAR
FBS
 D: E_D_FF;
 SW: E_SWITCH;
END_FBS
EVENT_CONNECTIONS
  EI TO D.CLK;
  D.EO TO SW.EI;
  SW.EO1 TO EO;
END_CONNECTIONS
DATA_CONNECTIONS
  QI TO D.D;
  D.Q TO SW.G;
END_CONNECTIONS
```

```
END_FUNCTION_BLOCK
(*********************************************************************)
FUNCTION_BLOCK E_REND (* Rendezvous of two events *)
EVENT_INPUT
  EI1; (* First Event Input *)
  EI2; (* Second Event Input *)
  R;  (* Reset Event *)
END_EVENT
EVENT_OUTPUT
 EO;  (* Rendezvous Output Event *)
END_EVENT
EC_STATES
  START;  (* Initial State *)
  EI1;   (* EI1 has arrived, wait for EI2 or R *)
  EO:    (* Issue rendezvous event *)
    ->EO;
  EI2;   (* EI2 has arrived, wait for EI1 or R *)
END_STATES
EC_TRANSITIONS
  START TO EI1:= EI1;
  EI1 TO START:= R;
  START TO EI2:= EI2;
  EI2 TO START:= R;
  EI1 TO EO:= EI2;
  EI2 TO EO:= EI1;
  EO TO START:= 1;
END_TRANSITIONS
END_FUNCTION_BLOCK
(*********************************************************************)
FUNCTION_BLOCK E_RESTART (* Generation of Restart Events *)
EVENT_OUTPUT
 COLD;     (* Cold Restart *)
 WARM;     (* Warm Restart *)
END_EVENT
SERVICE RESOURCE/E_RESTART
SEQUENCE cold_restart ->E_RESTART.COLD(); END_SEQUENCE
SEQUENCE warm_restart ->E_RESTART.WARM(); END_SEQUENCE
END_SERVICE
END_FUNCTION_BLOCK
(*********************************************************************)
FUNCTION_BLOCK E_RS (* Event-driven bistable *)
EVENT_INPUT
  S; (* Set Event *)
  R; (* Reset Event *)
END_EVENT
EVENT_OUTPUT
  EO WITH Q; (* Output Event *)
END_EVENT
VAR_OUTPUT
  Q: BOOL; (* Current Output State *)
END_VAR
EC_STATES
  Q0; (* Q is FALSE initially *)
  RESET: RESET -> EO; (* Reset Q and issue EO *)
  SET: SET -> EO; (* Set Q and issue EO *)
END_STATES
EC_TRANSITIONS
  Q0 TO SET:= S;
  SET TO RESET:= R;
  RESET TO SET:= S;
END_TRANSITIONS
ALGORITHM SET IN ST: (* Set Q *)
Q:=TRUE;
END_ALGORITHM
ALGORITHM RESET IN ST: (* Reset Q *)
Q:=FALSE;
END_ALGORITHM
END_FUNCTION_BLOCK
(*********************************************************************)
FUNCTION_BLOCK E_SELECT (* Selection between two events *)
```

```
EVENT_INPUT
  EI0 WITH G; (* Input event, selected when G=0 *)
  EI1 WITH G; (* Input event, selected when G=1 *)
END_EVENT
EVENT_OUTPUT EO; (* Output Event *)
END_EVENT
VAR_INPUT G: BOOL; (* Select EI0 when G=0, EI1 when G=1 *)
END_VAR
EC_STATES
  START;   (* Initial State *)
  EO: -> EO; (* Issue Output Event *)
END_STATES
EC_TRANSITIONS
  START TO EO:= EI0 [NOT G];
  START TO EO:= EI1 [G];
  EO TO START:= 1;
END_TRANSITIONS
END_FUNCTION_BLOCK
================================================================
FUNCTION_BLOCK E_SPLIT (* Split an event *)
EVENT_INPUT
  EI; (* Input event *)
END_EVENT
EVENT_OUTPUT
 EO1;  (* First output event *)
 EO2;  (* Second output event, etc. *)
END_EVENT
EC_STATES
  START; (* Initial State *)
  EO: (* Extensible *)
    ->EO1, (* Output first event *)
    ->EO2; (* Output second event, etc. *)
END_STATES
EC_TRANSITIONS
  START TO EO:= EI;
  EO TO START:= 1;
END_TRANSITIONS
END_FUNCTION_BLOCK
================================================================
FUNCTION_BLOCK E_SWITCH (* Switch (demultiplex) an event *)
EVENT_INPUT EI WITH G; (* Event Input *)
END_EVENT
EVENT_OUTPUT
  EO0; (* Output, switched from EI when G=0 *)
  EO1; (* Output, switched from EI when G=1 *)
END_EVENT
VAR_INPUT G: BOOL; (* Switch EI to EI0 when G=0, to EI1 when G=1 *)
END_VAR
EC_STATES
  START; (* Initial State *)
  G0:  (* Issue EO0 when EI arrives with G=0 *)
    ->EO0;
  G1: (* Issue EO1 when EI arrives with G=1 *)
    ->EO1;
END_STATES
EC_TRANSITIONS
  START TO G0:= EI [NOT G];
  G0 TO START:= 1;
  START TO G1:= EI [G];
  G1 TO START:= 1;
END_TRANSITIONS
END_FUNCTION_BLOCK
================================================================
FUNCTION_BLOCK E_TABLE (* Generation of a finite train of events, table
driven *)
EVENT_INPUT
 START WITH DT, N;
 STOP; (* Cancel *)
END_EVENT
```

```
EVENT_OUTPUT
 EO WITH CV; (* N events at periods DT, starting at DT[0] after START *)
END_EVENT
VAR_INPUT
  DT: TIME[4]; (* Periods between events *)
  N: UINT; (* Number of events to generate *)
END_VAR
VAR_OUTPUT
  CV: UINT; (* Current event index, 0..N-1 *)
END_VAR
FBS
  CTRL: E_TABLE_CTRL;
  DLY: E_DELAY;
END_FBS
EVENT_CONNECTIONS
  START TO CTRL.INIT;
  CTRL.CLKO TO DLY.START;
  DLY.EO TO EO;
  DLY.EO TO CTRL.CLK;
  STOP TO DLY.STOP;
END_CONNECTIONS
DATA_CONNECTIONS
  DT TO CTRL.DT;
  N TO CTRL.N;
  CTRL.DTO TO DLY.DT;
  CTRL.CV TO CV;
END_CONNECTIONS
END_FUNCTION_BLOCK
(*=================================================================*)
FUNCTION_BLOCK E_TABLE_CTRL (* Control for E_TABLE *)
EVENT_INPUT
 INIT WITH DT, N;
 CLK;
END_EVENT
EVENT_OUTPUT
 CLKO WITH DTO, CV;
END_EVENT
VAR_INPUT
 DT: TIME[4]; (* Array length is implementation dependent *)
 N: UINT;   (* Actual number of time steps *)
END_VAR
VAR_OUTPUT
 DTO: TIME;  (* Current delay interval *)
 CV: UINT;   (* Current event index, 0..N-1 *)
END_VAR
EC_STATES
 START;
 INIT0: INIT;
 INIT1: -> CLKO;
 STEP: STEP -> CLKO;
END_STATES
EC_TRANSITIONS
 START TO INIT0:= INIT;
 INIT0 TO INIT1:= [N>0];
 INIT0 TO START:= [N=0]; (* Don't run if N=0 *)
 INIT1 TO START:= 1;
 START TO STEP:= CLK [CV < MIN(3,N-1)];
 STEP TO START:= 1;
END_TRANSITIONS
ALGORITHM STEP IN ST:
  CV:= CV+1;
  DTO:= DT[CV];
END_ALGORITHM
ALGORITHM INIT IN ST:
  CV:= 0;
  DTO:= DT[0];
END_ALGORITHM
END_FUNCTION_BLOCK
(*=================================================================*)
FUNCTION_BLOCK E_TRAIN (* Generation of a finite train of events *)
```

```
EVENT_INPUT
 START WITH DT, N;
 STOP;
END_EVENT
EVENT_OUTPUT
 EO WITH CV; (* N events at period DT, starting at DT after START *)
END_EVENT
VAR_INPUT
  DT: TIME; (* Period between events *)
  N: UINT; (* Number of events to generate *)
END_VAR
VAR_OUTPUT
 CV: UINT; (* EO index (0..N-1) *)
END_VAR
FBS
 CTR: E_CTU;
 GATE: E_SWITCH;
 DLY: E_DELAY;
END_FBS
EVENT_CONNECTIONS
 START TO CTR.R;
 STOP TO DLY.STOP;
 DLY.EO TO EO;
 DLY.EO TO CTR.CU;
 CTR.CUO TO GATE.EI;
 CTR.RO TO GATE.EI;
 GATE.EO0 TO DLY.START;
END_CONNECTIONS
DATA_CONNECTIONS
 DT TO DLY.DT;
 N TO CTR.PV;
 CTR.Q TO GATE.G;
 CTR.CV TO CV;
END_CONNECTIONS
END_FUNCTION_BLOCK
=================================================================
FUNCTION_BLOCK FB_ADD_INT (* INT Addition *)
EVENT_INPUT
 REQ WITH QI, IN1, IN2;
END_EVENT
EVENT_OUTPUT
 CNF WITH QO, STATUS, OUT;
END_EVENT
VAR_INPUT
 QI: BOOL;    (* Event Qualifier *)
 IN1: INT;    (* Augend *)
 IN2: INT;    (* Addend *)
END_VAR
VAR_OUTPUT
 QO: BOOL;     (* Output Qualifier *)
 STATUS: UINT;  (* Operation Status *)
 OUT: INT;    (* Sum *)
END_VAR
VAR
 RESULT: DINT;
END_VAR
EC_STATES
 START;
 REQ: REQ -> CNF;
END_STATES
EC_TRANSITIONS
 START TO REQ:= REQ;
 REQ TO START:= 1;
END_TRANSITIONS
ALGORITHM REQ IN ST:
  QO:= QI;
  IF QI THEN
   STATUS:= 0;
   RESULT:= INT_TO_DINT(IN1) + INT_TO_DINT(IN2);
```

```
    IF (RESULT > 32767) OR (RESULT < -32768) THEN
     QO = FALSE;
     STATUS = 3;
     IF (RESULT > 32767) THEN OUT:= 32767;
     ELSE OUT:= -32768;
     END_IF;
    ELSE OUT:= RESULT;
    END_IF;
   ELSE STATUS = 1;
   END_IF;
END_ALGORITHM
END_FUNCTION_BLOCK
================================================================================
FUNCTION_BLOCK INTEGRAL_REAL
EVENT_INPUT
 INIT: INIT_EVENT WITH CYCLE;
 EX WITH HOLD, XIN;
END_EVENT
EVENT_OUTPUT
 INITO: INIT_EVENT WITH XOUT;
 EXO WITH XOUT;
END_EVENT
VAR_INPUT
 HOLD: BOOL;  (* 0 = Run, 1 = Hold *)
 XIN: REAL;  (* Integrand *)
 CYCLE: TIME; (* Sampling period *)
END_VAR
VAR_OUTPUT
 XOUT: REAL;  (* Integrated output *)
END_VAR
VAR DT: REAL; END_VAR
EC_STATES
 START;         (* EC Initial state *)
 INIT:INIT -> INITO; (* EC State with Algorithm and EC Action *)
 MAIN: MAIN -> EXO;
END_STATES
EC_TRANSITIONS
 START TO INIT:= INIT; (* An EC Transition *)
 START TO MAIN:= EX;
 INIT TO START:= 1;
 MAIN TO START:= 1;
END_TRANSITIONS
ALGORITHM INIT IN ST:
 XOUT:= 0.0;
 DT:= TIME_TO_REAL(CYCLE);
END_ALGORITHM
ALGORITHM MAIN IN ST:
 IF NOT HOLD THEN
  XOUT:= XOUT + XIN * DT;
 END_IF;
END_ALGORITHM
END_FUNCTION_BLOCK
================================================================================
ADAPTER LD_UNLD (* LOAD/UNLOAD Adapter Interface *)
EVENT_INPUT
 UNLD;   (* UNLOAD Request *)
END_EVENT
EVENT_OUTPUT
 LD WITH WO,WKPC; (* LOAD Request *)
 CNF WITH WO,WKPC; (* UNLD Confirm *)
END_EVENT
VAR_OUTPUT
 WO: BOOL;  (* Workpiece present *)
 WKPC: COLOR; (* Workpiece Color *)
END_VAR
SERVICE PLUG/SOCKET
SEQUENCE normal_operation
 PLUG.LD(WO,WKPC) -> SOCKET.LD(WO,WKPC);
 SOCKET.UNLD() -> PLUG.UNLD();
 PLUG.CNF() -> SOCKET.CNF();
```

```
END_SEQUENCE
END_SERVICE
END_ADAPTER
================================================================
FUNCTION_BLOCK MANAGER (* Management Service Interface *)
EVENT_INPUT
  INIT WITH QI, PARAMS; (* Service Initialization *)
  REQ WITH QI, CMD, OBJECT; (* Service Request *)
END_EVENT
EVENT_OUTPUT
  INITO WITH QO, STATUS; (* Initialization Confirm *)
  CNF WITH QO, STATUS, RESULT; (* Service Confirmation *)
END_EVENT
VAR_INPUT
  QI: BOOL; (* Event Input Qualifier *)
  PARAMS: WSTRING; (* Service Parameters *)
  CMD: UINT; (* Enumerated Command *)
  OBJECT: BYTE[512]; (* Command Object *)
END_VAR
VAR_OUTPUT
  QO: BOOL; (* Event Output Qualifier *)
  STATUS: UINT; (* Service Status *)
  RESULT: BYTE[512]; (* Result Object *)
END_VAR
SERVICE MANAGER/resource
SEQUENCE normal_establishment
  MANAGER.INIT+(PARAMS) -> resource.initManagement() -> MANAGER.INITO+();
END_SEQUENCE
SEQUENCE unsuccessful_establishment
  MANAGER.INIT+(PARAMS) -> resource.initManagement(PARAMS) -> MANAGER.INITO-
(STATUS);
END_SEQUENCE
SEQUENCE normal_command_sequence
  MANAGER.REQ+(CMD,OBJECT) -> resource.performCommand(CMD,OBJECT) ->
MANAGER.CNF+(STATUS,RESULT);
END_SEQUENCE
SEQUENCE command_error
  MANAGER.REQ+(CMD,OBJECT) -> resource.performCommand(CMD,OBJECT) ->
MANAGER.IND-(STATUS);
END_SEQUENCE
SEQUENCE application_initiated_termination
  MANAGER.INIT-() -> resource.terminateService() -> MANAGER.INITO-(STATUS);
END_SEQUENCE
SEQUENCE resource_initiated_termination
  resource.serviceTerminated(STATUS) -> MANAGER.INITO-(STATUS);
END_SEQUENCE
END_SERVICE
END_FUNCTION_BLOCK
================================================================
FUNCTION_BLOCK PI_REAL
EVENT_INPUT
  INIT WITH KP, KI, CYCLE;
  EX WITH HOLD, PV, SP, KP, KI, CYCLE;
END_EVENT
EVENT_OUTPUT
  INITO WITH XOUT;
  EXO WITH XOUT;
END_EVENT
VAR_INPUT
  HOLD: BOOL; (* Hold when TRUE *)
  PV: REAL; (* Process variable *)
  SP: REAL; (* Set point *)
  KP: REAL; (* Proportionality constant *)
  KI: REAL; (* Integral constant,1/s *)
  CYCLE: TIME; (* Sampling period *)
END_VAR
VAR_OUTPUT
  XOUT: REAL;
END_VAR
```

```
FBS
  CALC: PID_CALC;
  INTEGRAL_TERM: INTEGRAL_REAL;
END_FBS
EVENT_CONNECTIONS
  INIT TO CALC.INIT;
  EX TO CALC.PRE;
  CALC.POSTO TO EXO;
  INTEGRAL_TERM.INITO TO INITO;
  CALC.INITO TO INTEGRAL_TERM.INIT;
  CALC.PREO TO INTEGRAL_TERM.EX;
  INTEGRAL_TERM.EXO TO CALC.POST;
END_CONNECTIONS
DATA_CONNECTIONS
  HOLD TO INTEGRAL_TERM.HOLD;
  PV TO CALC.PV;
  SP TO CALC.SP;
  KP TO CALC.KP;
  KI TO CALC.KI;
  CYCLE TO INTEGRAL_TERM.CYCLE;
  CALC.XOUT TO XOUT;
  CALC.ETERM TO INTEGRAL_TERM.XIN;
  INTEGRAL_TERM.XOUT TO CALC.ITERM;
  0 TO CALC.TD;
  0 TO CALC.DTERM;
END_CONNECTIONS
END_FUNCTION_BLOCK
(*********************************************************************)
SUBAPPLICATION PI_REAL_APPL (* A Subapplication *)
EVENT_INPUT
 INIT;
 EX;
END_EVENT
EVENT_OUTPUT
 INITO;
 EXO;
END_EVENT
VAR_INPUT
 HOLD: BOOL; (* Hold when TRUE *)
 PV: REAL; (* Process variable *)
 SP: REAL; (* Set point *)
 KP: REAL; (* Proportional gain *)
 KI: REAL; (* Integral gain = Sample period/Reset time *)
 X0: REAL; (* Initial integrator output *)
END_VAR
VAR_OUTPUT XOUT: REAL; END_VAR
FBS
 ETERM: FB_SUB_REAL;
 INTEGRATOR: ACCUM_REAL;
 CALC: PI_CALC;
END_FBS
EVENT_CONNECTIONS
  INIT TO INTEGRATOR.INIT;
  INTEGRATOR.INITO TO INITO;
  EX TO ETERM.REQ;
  ETERM.CNF TO INTEGRATOR.EX;
  INTEGRATOR.EXO TO CALC.EX;
  CALC.EXO TO EXO;
END_CONNECTIONS
DATA_CONNECTIONS
  X0 TO INTEGRATOR.X0;
  HOLD TO INTEGRATOR.HOLD;
  PV TO ETERM.IN1;
  SP TO ETERM.IN2;
  KP TO CALC.KP;
  KI TO CALC.KI;
  ETERM.OUT TO INTEGRATOR.XIN;
  ETERM.OUT TO CALC.ETERM;
  INTEGRATOR.XOUT TO CALC.ITERM;
  CALC.XOUT TO XOUT;
```

```
  1 TO ETERM.QI;
END_CONNECTIONS
END_SUBAPPLICATION
(*=================================================================*)
FUNCTION_BLOCK REQUESTER
          (* Service Requester Interface *)
EVENT_INPUT
 INIT WITH QI, PARAMS;    (* Service Initialization *)
 REQ WITH QI, SD_1, SD_m;  (* Service Request *)
END_EVENT
EVENT_OUTPUT
 INITO WITH QO, STATUS;      (* Initialization Confirm *)
 CNF WITH QO, STATUS, RD_1, RD_n; (* Service Confirmation *)
END_EVENT
VAR_INPUT
 QI: BOOL;   (* Event Input Qualifier *)
 PARAMS: ANY; (* Service Parameters *)
 SD_1: ANY;  (* Data to transfer, extensible *)
 SD_m: ANY;  (* Last data item to transfer *)
END_VAR
VAR_OUTPUT
 QO: BOOL;   (* Event Output Qualifier *)
 STATUS: ANY; (* Service Status *)
 RD_1: ANY;  (* Received data, extensible *)
 RD_n: ANY;  (* Last received data item *)
END_VAR
SERVICE REQUESTER/RESOURCE
SEQUENCE normal_establishment
 REQUESTER.INIT+(PARAMS) -> REQUESTER.INITO+();
END_SEQUENCE
SEQUENCE unsuccessful_establishment
 REQUESTER.INIT+(PARAMS) -> REQUESTER.INITO-(STATUS);
END_SEQUENCE
SEQUENCE normal_data_transfer
 REQUESTER.REQ+(SD_1,...,SD_m) -> REQUESTER.CNF+(RD_1,...,RD_n);
END_SEQUENCE
SEQUENCE data_transfer_error
 REQUESTER.REQ+(SD_1,...,SD_m) -> REQUESTER.CNF-(STATUS);
END_SEQUENCE
SEQUENCE application_initiated_termination
 REQUESTER.INIT-() -> REQUESTER.INITO-(STATUS);
END_SEQUENCE
SEQUENCE resource_initiated_termination
 -> REQUESTER.INITO-(STATUS);
END_SEQUENCE
END_SERVICE
END_FUNCTION_BLOCK
(*=================================================================*)
FUNCTION_BLOCK XBAR_MVCA (* XBAR_MVC + Adapters *)
EVENT_INPUT
 INIT WITH VF,VR,DTL,DT,BKGD,LEN,DIA,DIR; (* Initialize *)
END_EVENT
EVENT_OUTPUT
 INITO;
END_EVENT
VAR_INPUT
 VF: INT:= 20;     (* ADVANCE speed in +%/s *)
 VR: INT:= -40;     (* RETRACT speed in -%/s *)
 DTL: TIME:= t#750ms; (* LOAD Delay *)
 DT: TIME:= t#250ms; (* Simulation Interval *)
 BKGD: COLOR:= COLOR#blue; (* Transfer Bar Color *)
 LEN: UINT:= 5;  (* Bar Length in Diameters *)
 DIA: UINT:= 20; (* Workpiece diameter *)
 DIR: UINT;      (* Orientation: 0=L/R, 1=T/B, 2=R/L, 3=B/T *)
END_VAR
SOCKETS
 LDU_SKT: LD_UNLD;
END_SOCKETS
PLUGS
```

```
 LDU_PLG: LD_UNLD;
END_PLUGS
FBS
 MVC: XBAR_MVC;
END_FBS
EVENT_CONNECTIONS
  INIT TO MVC.INIT;
  MVC.INITO TO INITO;
  MVC.LOADED TO LDU_SKT.UNLD;
  LDU_SKT.LD TO MVC.LOAD;
  MVC.ADVANCED TO LDU_PLG.LD;
  LDU_PLG.UNLD TO MVC.UNLOAD;
  MVC.UNLOADED TO LDU_PLG.CNF;
END_CONNECTIONS
DATA_CONNECTIONS
  LDU_SKT.WO TO MVC.WI;
  LDU_SKT.WKPC TO MVC.LDCOL;
  MVC.WO TO LDU_PLG.WO;
  MVC.WKPC TO LDU_PLG.WKPC;
  VF TO MVC.VF;
  VR TO MVC.VR;
  DTL TO MVC.DTL;
  DT TO MVC.DT;
  BKGD TO MVC.BKGD;
  LEN TO MVC.LEN;
  DIA TO MVC.DIA;
  DIR TO MVC.DIR;
END_CONNECTIONS
END_FUNCTION_BLOCK
=====================================================================
```

Annex G
(informative)

Attributes

G.1 General principles

Attributes may be associated with *data types, variables, applications*, and *types* and *instances* of *function blocks, devices, resources*, and their component elements. Attributes have values that may be modified and accessed at various points in the life cycle of the function block type or instance.

In addition to the descriptions of function block *algorithms*, supplementary information is necessary to support the use of a function block during the course of its software life cycle. This information may be provided by attaching *attributes* to the component elements of function block *types* or *instances*.

Attributes can be applied to elements such as *data types, variables*, and *parameters* that are used in the specification of function block types or instances. Graphical language elements may require additional attributes for holding information such as position, color, size, etc.

Attributes can also be applied directly to function block types and instances, for instance to hold the version of a function block type specification.

Certain attributes may be used throughout the life cycle of a function block. For instance, an attribute related to a function block type specification may be accessed when the function block type is selected from a library, when an instance of the function block type is queried, etc.

Other attributes may only exist at certain points in the life cycle. For instance, text defining the purpose of a particular function block instance might be applied only when the function block is instantiated, and might be modified during the life of the function block instance.

Certain function block attributes may be installed in associated *resources* and be accessible during the lifetime of the distributed *application*. Such attributes are typically used to support access to function block parameter values by external devices, e.g., to restrict the values of parameters that may be set using a hand-held configurator to predefined safe limits.

G.2 Attribute definitions

An attribute definition provides the information specified in Table G.1. Each attribute has a name and a data type of its associated value. An attribute may have a default value that will be used until a value is given at some point in the software life cycle. In the example given in G.1, the `DESCRIPTION` attribute has an initial value of '' (the empty string) that may be overwritten with a more meaningful description when a function block instance is configured or even during its active use.

Attributes themselves may require additional information to that shown in Table G.1. Such information is designated as *sub-attributes*.

Table G.1 – Elements of attribute definitions

Element	Example	
Name	DESCRIPTION	
Data type	WSTRING(30)	
Default value	""	
Associated element	**Function block types**	**Function block instances**
Usage	Configuration	Run-time

G.3 Examples

NOTE The following examples are for the purpose of illustrating the use of attributes and are not to be considered as normative definitions of standard attributes.

An example of a *data type attribute* is:

- Max_System_Value - This attribute defines the maximum supported value of a numeric data type. It is applied to the generic data type ANY_NUM, so that all numeric types such as INT and REAL will inherit this attribute. Note that each specific data type will have its own value for this attribute, and that standard values for this attribute for some data types are given in Table E.1.

Examples of attributes that apply to *variables* are:

- Diagnostic_Access – This determines whether the value of a variable is accessible by a run-time diagnostic system.
- Write_Access – This defines the access level required to change the value of a variable, e.g., 'Operator', 'System', 'Diagnostics'.
- Units – The dimensional units that apply to a variable, e.g., 'l', 'm/s', 'cm'.
- Usage – A multi-line textual description of the usage of the associated variable.

Examples of function block type attributes are:

- Usage_Class – This describes the general usage of the function block, e.g., 'Input', 'Output', 'Control'.
- Version – This describes the version number of the function block type definition, e.g., '1.2'.
- Help – A multi-line textual description that may be accessed at various points in the life cycle.

Attributes which are relevant to the scheduling of *algorithms* for *execution* include:

- ExecutionTime – This attribute, of type TIME, specifies the worst-case time for execution of a particular *algorithm* of a specified *function block type* in a particular *resource type*.
- Priority – This attribute is associated with a particular *event connection* within a *resource*, and may be inherited from the *resource type*. This attribute may be used by a resource which supports pre-emptive *multitasking* to determine the priority of *execution* of an *algorithm* invoked by an *EC action* associated with an *EC state* which is activated by an event with the specified priority.

G.4 Attribute sources

Attributes may come from the following main sources:

- **Implicit** attributes such as function block *type names*, *instance names*, *variable names* and their *data types*, are defined as part of the normal *declaration* process for the function block.
- **Standard** attributes are those which are required as part of a standard, such function block type versions, maximum range of parameters, parameter descriptions, etc.
- **Product-specific** attributes are those which a system vendor has provided, such as function block type product codes, hardware addresses of function block instances, etc.
- **Application-specific** attributes are those which a system developer specifies to support the use of a particular data type or function block in an application, such as an additional function block instance identifier to fit a customer's desired style, a fail-safe default value for output parameters, an alternative parameter description in a national language, etc.

G.5 Attribute inheritance

Function block elements will inherit attributes from more primitive elements. For instance, a *variable* within a *function block type declaration* will inherit attributes of its associated *data type*, and a function block *instance* will inherit attributes of the associated function block *type*.

Data types will inherit attributes down the generic type hierarchy defined in IEC 61131-3. For example, attributes applied to `ANY_REAL` will also apply to `LREAL` and `REAL`.

G.6 Declaration syntax

The assignment of an attribute value to a declared element is similar to assigning a value to an *instance* of an *attribute type* in which the instance has the same name as the type.

The declaration of an attribute *type* uses the same syntax as the declaration of a *data type* as defined in IEC 61131-3, with the exception that the delimiting keywords are `ATTRIBUTE...END_ATTRIBUTE` instead of `TYPE...END_TYPE`. For instance, the declaration of the attribute type `DESCRIPTION` in Table G.1 would be:

```
ATTRIBUTE DESCRIPTION: WSTRING(30); END_ATTRIBUTE
```

The assignment of a value to an attribute *instance* uses the same syntax as that for assigning an initial value to a *variable* as described in IEC 61131-3, with the following extensions:

a) the name of the attribute instance is the same as the name of the corresponding attribute type;

b) no data type is specified for the attribute instance;

c) the value assignment is enclosed in the **pragma** construct defined in IEC 61131-3;

d) multiple attribute value assignments, separated by semicolons, may be included in the pragma construct;

e) the pragma construct shall be located in such a manner that the declaration to which it applies can be determined unambiguously.

An example of the application of these rules is:

```
FUNCTION_BLOCK PID
     {DESCRIPTION:= "Proportional + Integral + Derivative Control;
      AUTHOR:= "JHC"; VERSION:= "19990103/JHC"}
     INPUT_EVENT
      INIT WITH QI, PARAMS; {DESCRIPTION:= "Initialization Request"}
     ...etc.
```

Bibliography

IEC 60050-351:2006, *International Electrotechnical Vocabulary – Part 351: Control technology*

IEC 61131-5:2000, *Programmable controllers – Part 5: Communications*

IEC 61499 (all parts), *Function blocks*

IEC 61499-2:2012, *Function blocks – Part 2: Software tools requirements*

IEC 61499-4, *Function blocks – Part 4: Rules for compliance profiles*

ISO/IEC 7498-4, *Information processing systems – Open systems interconnection – Basic reference model – Part 4: Management framework*

ISO/IEC 8825-1:2008, *Information technology – ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)*

ISO/IEC 10040:1998, *Information technology – Open Systems Interconnection – Systems management overview*

ISO/IEC/IEEE 60559, *Information technology – Microprocessor systems – Floating-point arithmetic*

ISO 2382 (all parts), *Information technology – Vocabulary*

———————

SOMMAIRE

COMMISSION ÉLECTROTECHNIQUE INTERNATIONALE

BLOCS FONCTIONNELS –

Partie 1: Architecture

AVANT-PROPOS

1) La Commission Electrotechnique Internationale (CEI) est une organisation mondiale de normalisation composée de l'ensemble des comités électrotechniques nationaux (Comités nationaux de la CEI). La CEI a pour objet de favoriser la coopération internationale pour toutes les questions de normalisation dans les domaines de l'électricité et de l'électronique. A cet effet, la CEI – entre autres activités – publie des Normes internationales, des Spécifications techniques, des Rapports techniques, des Spécifications accessibles au public (PAS) et des Guides (ci-après dénommés "Publication(s) de la CEI"). Leur élaboration est confiée à des comités d'études, aux travaux desquels tout Comité national intéressé par le sujet traité peut participer. Les organisations internationales, gouvernementales et non gouvernementales, en liaison avec la CEI, participent également aux travaux. La CEI collabore étroitement avec l'Organisation Internationale de Normalisation (ISO), selon des conditions fixées par accord entre les deux organisations.
2) Les décisions ou accords officiels de la CEI concernant les questions techniques représentent, dans la mesure du possible, un accord international sur les sujets étudiés, étant donné que les Comités nationaux de la CEI intéressés sont représentés dans chaque comité d'études.
3) Les Publications de la CEI se présentent sous la forme de recommandations internationales et sont agréées comme telles par les Comités nationaux de la CEI. Tous les efforts raisonnables sont entrepris afin que la CEI s'assure de l'exactitude du contenu technique de ses publications; la CEI ne peut pas être tenue responsable de l'éventuelle mauvaise utilisation ou interprétation qui en est faite par un quelconque utilisateur final.
4) Dans le but d'encourager l'uniformité internationale, les Comités nationaux de la CEI s'engagent, dans toute la mesure possible, à appliquer de façon transparente les Publications de la CEI dans leurs publications nationales et régionales. Toutes divergences entre toutes Publications de la CEI et toutes publications nationales ou régionales correspondantes doivent être indiquées en termes clairs dans ces dernières.
5) La CEI elle-même ne fournit aucune attestation de conformité. Des organismes de certification indépendants fournissent des services d'évaluation de conformité et, dans certains secteurs, accèdent aux marques de conformité de la CEI. La CEI n'est responsable d'aucun des services effectués par les organismes de certification indépendants.
6) Tous les utilisateurs doivent s'assurer qu'ils sont en possession de la dernière édition de cette publication.
7) Aucune responsabilité ne doit être imputée à la CEI, à ses administrateurs, employés, auxiliaires ou mandataires, y compris ses experts particuliers et les membres de ses comités d'études et des Comités nationaux de la CEI, pour tout préjudice causé en cas de dommages corporels et matériels, ou de tout autre dommage de quelque nature que ce soit, directe ou indirecte, ou pour supporter les coûts (y compris les frais de justice) et les dépenses découlant de la publication ou de l'utilisation de cette Publication de la CEI ou de toute autre Publication de la CEI, ou au crédit qui lui est accordé.
8) L'attention est attirée sur les références normatives citées dans cette publication. L'utilisation de publications référencées est obligatoire pour une application correcte de la présente publication.
9) L'attention est attirée sur le fait que certains des éléments de la présente Publication de la CEI peuvent faire l'objet de droits de brevet. La CEI ne saurait être tenue pour responsable de ne pas avoir identifié de tels droits de brevets et de ne pas avoir signalé leur existence.

La Norme internationale CEI 61499-1 a été établie par le sous-comité 65B: Equipements de mesure et de contrôle-commande, du comité d'études 65 de la CEI: Mesure, commande et automation dans les processus industriels.

Cette deuxième édition annule et remplace la première édition parue en 2005. Elle constitue une révision technique.

Cette édition inclut les modifications techniques majeures suivantes par rapport à l'édition précédente:

- Le terme *contrôle d'exécution* dans les blocs fonctionnels de base (5.2) a été clarifié et étendu:
  - Les parties dynamiques et statiques de la condition de transition EC sont clairement délimitées par l'utilisation de la syntaxe `ec_transition_event[guard_condition]` du langage de modélisation unifié (UML) (5.2.1.3, B.2.1).

  - La terminologie "franchissement d'une transition EC" est utilisée de préférence à "libération" (3.10) afin d'éviter toute erreur d'interprétation qui laisserait supposer que l'ensemble de la condition de transition correspond à une variable booléenne pouvant être "libérée."
  - L'exploitation du diagramme d'états ECC au 5.2.2.2 a été clarifiée et rendue plus rigoureuse.
  - Les sorties d'événements et les sorties de données des instances d'adaptateur (prises mâles et prises femelles) peuvent être utilisées dans les conditions de transition EC, et les entrées d'événements des instances d'adaptateur peuvent être utilisées en tant que sorties d'actions EC.
- Les *variables temporaires* (3.97) peuvent être déclarées (B.2.1) et utilisées dans les algorithmes des blocs fonctionnels de base.
- Les *séquences de service* (6.1.3) peuvent à présent être définies pour les types bloc fonctionnel de base et composé et les types adaptateur, ainsi que les types interface de service.
- La syntaxe pour le *mapping* des instances FB entre les applications et les ressources a été simplifiée (Article B.3).
- La syntaxe relative à la définition des *types de segment* (7.2.3) pour les segments de réseau des configurations de système a été ajoutée (Article B.3).
- Les types de blocs fonctionnels pour l'interfonctionnement avec des dispositifs de commande programmables sont définis (Article D.6).
- Les commandes de gestion READ/WRITE (Tableau 8) s'appliquent désormais uniquement aux *paramètres*.

Le texte de cette partie de la CEI 61499 est issu des documents suivants:

FDIS	Rapport de vote
65B/845/FDIS	65B/855/RVD

Le rapport de vote indiqué dans le tableau ci-dessus donne toute information sur le vote ayant abouti à l'approbation de cette norme (à la fin du vote).

Cette publication a été rédigée selon les Directives ISO/CEI, Partie 2.

Une liste de toutes les parties de la série CEI 61499, présentées sous le titre général *Blocs fonctionnels*, peut être consultée sur le site web de la CEI.

Les termes utilisés dans la présente Norme internationale, définis à l'Article 3, sont présentés en *italiques*.

Le comité a décidé que le contenu de cette publication ne sera pas modifié avant la date de stabilité indiquée sur le site web de la CEI sous "http://webstore.iec.ch" dans les données relatives à la publication recherchée. A cette date, la publication sera

- reconduite,
- supprimée,
- remplacée par une édition révisée, ou
- amendée.

IMPORTANT – Le logo *"colour inside"* qui se trouve sur la page de couverture de cette publication indique qu'elle contient des couleurs qui sont considérées comme utiles à une bonne compréhension de son contenu. Les utilisateurs devraient, par conséquent, imprimer cette publication en utilisant une imprimante couleur.

INTRODUCTION

La CEI 61499 comprend les parties suivantes, sous le titre général *Blocs fonctionnels*:

- La Partie 1 (le présent document) contient:
  - des exigences générales, y compris le domaine d'application, les références normatives, des définitions et des modèles de référence;
  - des règles pour la déclaration des *types de blocs fonctionnels*, et des règles pour le comportement d'*instances* des types ainsi déclarés;
  - des règles pour l'utilisation des blocs fonctionnels dans la *configuration* de *systèmes* de mesure et commande dans les processus industriels (les IPMCS);
  - des règles pour l'utilisation des blocs fonctionnels pour satisfaire aux exigences de communication des systèmes IPMCS distribués;
  - des règles pour l'utilisation de blocs fonctionnels dans la gestion *d'applications, de ressources* et d'*équipements* dans les IPMCS distribués.
- La Partie 2 définit les exigences relatives aux *outils logiciels* pour prendre en charge les tâches d'études des systèmes suivantes:
  - la spécification des *types de blocs fonctionnels*;
  - la spécification fonctionnelle des *types de ressource* et des *types d'équipement*;
  - la spécification, l'analyse et la validation des IPMCS distribués;
  - la *configuration*, la *mise en œuvre*, l'exploitation et la maintenance de systèmes IPMCS distribués;
  - l'échange d'*informations* entre des *outils logiciels*.
- La Partie 3 (informations tutorielles) a été retirée en raison des nombreux supports tutoriels et éducationnels disponibles concernant la CEI 61499. Cependant, il n'est pas exclu qu'une deuxième édition mise à jour de la Partie 3 puisse être développée dans le futur.
- La Partie 4 définit des règles pour le développement des *profils de conformité* qui spécifient les caractéristiques de la CEI 61499-1 et de la CEI 61499-2 devant être mises en œuvre afin de promouvoir les qualités suivantes des systèmes, équipements et outils logiciels basés sur la CEI 61499:
  - interopérabilité des équipements issus de différents fournisseurs;
  - portabilité des logiciels entre les outils logiciels de plusieurs fournisseurs; et
  - aptitude à configurer des équipements provenant de plusieurs vendeurs par des outils logiciels de différents fournisseurs.

BLOCS FONCTIONNELS –

Partie 1: Architecture

1 Domaine d'application

La présente partie de la CEI 61499 définit une architecture générique et présente des lignes directrices pour l'utilisation de *blocs fonctionnels* dans des systèmes de mesure et de commande dans les processus industriels distribués (IPMCS). Cette architecture est présentée en termes de *modèles* de référence pouvant être mis en œuvre, de syntaxe textuelle et de représentations graphiques. Ces modèles, représentations et syntaxe **peuvent être utilisés pour**:

- la spécification et la normalisation des *types de blocs fonctionnels*;
- la spécification fonctionnelle et la normalisation d'éléments de système;
- la spécification indépendante de toute mise en œuvre, l'analyse et la validation des systèmes IPMCS distribués;
- la *configuration*, la *mise en œuvre*, l'exploitation et la maintenance de systèmes IPMCS distribués;
- l'échange d'*informations* parmi des *outils logiciels* pour l'accomplissement des *fonctions* ci-dessus.

La présente partie de la CEI 61499 ne limite ni ne spécifie les capacités fonctionnelles des IPMCS ou de leurs éléments de système, à l'exception de la manière dont ces capacités sont représentées en utilisant les éléments définis dans la présente norme. La CEI 61499-4 traite de la mesure dans laquelle les éléments définis dans la présente norme sont parfois limités par les capacités fonctionnelles des systèmes, sous-systèmes et équipements conformes.

Un des buts de la présente norme est de fournir des modèles de référence pour l'utilisation de blocs fonctionnels dans d'autres normes traitant de la prise en charge du cycle de vie du système, y compris la planification, la conception, la mise en œuvre, la validation, l'exploitation et la maintenance du système. Les modèles donnés dans la présente norme sont censés être génériques, indépendants vis-à-vis de tout domaine et extensibles à la définition et à l'utilisation de blocs fonctionnels dans d'autres normes ou pour des applications particulières ou des domaines d'application particuliers. L'intention est de faire en sorte que les spécifications écrites selon les règles données dans la présente norme soient concises, réalisables, complètes, non ambiguës et cohérentes.

NOTE 1 Les dispositions de la présente norme seule ne suffisent pas à assurer l'interopérabilité entre les équipements de différents vendeurs. Des normes conformes à la présente partie de la CEI 61499 sont susceptibles de spécifier des dispositions supplémentaires pour assurer ladite interopérabilité.

NOTE 2 Des normes conformes à la présente partie de la CEI 61499 sont susceptibles de spécifier des dispositions supplémentaires pour activer l'accomplissement des *fonctions* de gestion de *système*, d'*équipement*, de *ressource* et d'*application*.

2 Références normatives

Les documents suivants sont cités en référence de manière normative, en intégralité ou en partie, dans le présent document et sont indispensables pour son application. Pour les références datées, seule l'édition citée s'applique. Pour les références non datées, la dernière édition du document de référence s'applique (y compris les éventuels amendements).

CEI 61131-1, *Automates programmables – Partie 1: Informations générales*

CEI 61131-3:2003, *Programmable controllers – Part 3: Programming languages* (disponible en anglais seulement)

ISO/CEI 7498-1:1994, *Technologies de l'information – Interconnexion de systèmes ouverts (OSI) – Modèle de référence de base: le modèle de base*

ISO/CEI 8824-1:2008, *Technologies de l'information – Notation de syntaxe abstraite numéro un (ASN.1): Spécification de la notation de base*

ISO/CEI 10646:2003, *Technologies de l'information – Jeu universel de caractères codés (JUC)*

3 Termes and définitions

Pour les besoins du présent document, les termes et définitions suivants s'appliquent.

NOTE Les termes définis dans l'Article 3 sont en *italiques* lorsqu'ils apparaissent dans les définitions et Notes à l'article d'autres termes ainsi que dans l'entier du document.

3.1
utilisateur
instance de bloc fonctionnel qui fournit un *support adaptateur* d'un *type d'adaptateur d'interface* défini

3.2
connexion d'adaptateur
connexion d'une *fiche d'adaptation* à un *support adaptateur* du même *type d'adaptateur d'interface*, qui transporte les flots de *données* et d'*événements* définis par le *type d'adaptateur d'interface*

3.3
type adaptateur d'interface
type qui consiste en la définition d'un jeu d'*événements d'entrées,* d'*événements de sorties,* de *données d'entrées* et de *données* de *sorties* dont des *instances* sont des *fiches d'adaptation* et des *supports adaptateurs*

3.4
algorithme
jeu fini de règles bien définies pour la solution d'un problème en un nombre fini d'*opérations*

3.5
application
unité fonctionnelle logicielle qui est spécifique à la solution d'un problème pour la mesure et la commande dans les processus industriels

Note 1 à l'article: Une application peut être distribuée parmi des *ressources* et est apte à communiquer avec d'autres applications.

3.6
attribut
propriété ou caractéristique d'une *entité*, par exemple, l'identificateur de version d'une spécification du *type de bloc fonctionnel*

3.7
type de bloc fonctionnel de base
type de bloc fonctionnel qui ne peut pas être décomposé en d'autres blocs fonctionnels et qui utilise un *graphe de contrôle d'exécution (ECC)* pour contrôler l'*exécution* de ses *algorithmes*

3.8
transaction bidirectionnelle
transaction dans laquelle une demande et éventuellement des *données* sont acheminées d'un *demandeur* vers un *répondeur* et dans laquelle une réponse et éventuellement des données sont acheminées du répondeur vers le demandeur

3.9
caractère
membre d'un jeu d'éléments qui est utilisé pour la représentation, l'organisation ou le contrôle de *données*

3.10
franchissement
libération
<d'une transition EC> *opération* par laquelle le contrôle passe d'un *état EC* précédent d'une *transition EC* à son état EC suivant

Note 1 à l'article: Cette opération consiste à désactiver l'état EC précédent, puis à activer l'état EC suivant.

3.11
connexion de communication
connexion qui utilise la fonction de mapping de la communication d'une ou plusieurs *ressources* pour l'acheminement des *informations*

3.12
bloc fonctionnel de communication
bloc fonctionnel interface de service qui représente l'*interface* entre une *application* et la fonction de mapping d'une *ressource*

3.13
type de bloc fonctionnel de communication
type de bloc fonctionnel dont les *instances* sont des *blocs fonctionnels de communication*

3.14
bloc fonctionnel composant
instance de bloc fonctionnel qui est utilisée dans la spécification d'un *algorithme* d'un *type de bloc fonctionnel composé*

Note 1 à l'article: Un bloc fonctionnel composant peut être du *type basic* (de base), *composite* (composé) ou *service interface* (interface service).

3.15
sous-application constitutive
instance de sous-application qui est utilisée dans la spécification d'un *type de sous-application*

3.16
type de bloc fonctionnel composé
type de bloc fonctionnel dont les *algorithmes* et le contrôle de leur *exécution* sont entièrement exprimés en termes de *blocs fonctionnels composants, événements,* et *variables* interconnectés

3.17
concomitant
relatif à des *algorithmes* qui sont *exécutés* pendant une période de temps commune au cours de laquelle ils peuvent devoir partager en alternance des *ressources* communes

3.18
configuration (d'un *système* ou d'un *équipement*)
action consistant à sélectionner des *unités fonctionnelles*, affecter leurs emplacements et définir leurs interconnexions

3.19
paramètre de configuration
paramètre relatif à la *configuration* d'un *système,* d'un *équipement* ou d'une *ressource*

3.20
primitive confirm
primitive de service qui représente une interaction dans laquelle une *ressource* indique l'achèvement d'un *algorithme invoqué* précédemment par une interaction via une *primitive "request"*

3.21
connexion
association établie entre des *unités fonctionnelles* pour acheminer de l'*information*

3.22
région critique
opération ou séquence d'opérations qui est *exécutée* sous le contrôle exclusif d'un objet verrou qui est associé aux *données* sur lesquelles les opérations sont accomplies

3.23
donnée
représentation réinterprétable d'*information* d'une manière formalisée adaptée à la communication, à l'interprétation ou au traitement

3.24
connexion de données
association entre deux *blocs fonctionnels* pour l'acheminement de *données*

3.25
entrée de données
interface d'un *bloc fonctionnel* qui reçoit des *données* d'une *connexion de données*

3.26
sortie de données
interface d'un *bloc fonctionnel* qui fournit des *données* à une *connexion de données*

3.27
type de données
jeu de valeurs accompagné d'un jeu d'*opérations* autorisées

3.28
déclaration
mécanisme pour établir la définition d'une *entité*

Note 1 à l'article: Une déclaration est susceptible d'impliquer le rattachement d'un *identificateur* à l'entité et de lui affecter des *attributs* tels que *types de données* et *algorithmes*.

3.29
équipement
entité physique indépendante capable d'accomplir une ou plusieurs *fonctions* spécifiées dans un contexte particulier et délimitée par ses *interfaces*

Note 1 à l'article: Un *système d'équipement de commande programmable* comme défini dans la CEI 61131-1 est un *équipement*.

3.30
application de gestion d'équipement
application dont la fonction principale est la gestion de plusieurs *ressources* au sein d'un *équipement*

3.31
entité
objet particulier, telle qu'une personne, un lieu, un *processus*, un objet, un concept, une association ou un *événement*

3.32
événement
occurrence instantanée qui est significative pour la programmation de l'*exécution* d'un *algorithme*

Note 1 à l'article: L'exécution d'un algorithme peut utiliser des *variables* associées à un événement.

3.33
connexion d'événements
association parmi des *blocs fonctionnels* pour l'acheminement d'*événements*

3.34
entrée d'événements
interface d'un *bloc fonctionnel* qui peut recevoir des *événements* d'une *connexion d'événements*

3.35
sortie d'événements
interface d'un *bloc fonctionnel* qui peut émettre des *événements* vers une *connexion d'événements*

3.36
exception
événement qui entraîne la suspension d'une *exécution* normale

3.37
exécution
processus consistant à accomplir une séquence d'*opérations* spécifiée par un *algorithme*

Note 1 à l'article: La séquence d'opérations à exécuter peut varier d'une *invocation* d'une *instance de bloc fonctionnel* à l'autre, selon les règles spécifiées par l'*algorithme* du bloc fonctionnel et des valeurs courantes de *variables* dans la structure de données du bloc fonctionnel.

3.38
action de contrôle d'exécution
action EC
élément associé à un *état de contrôle d'exécution*, qui identifie un *algorithme* devant être *exécuté* et/ou un *événement* devant être émis

Note 1 à l'article: La temporisation de l'exécution d'algorithme et l'émission d'un événement sont traitées en 5.2.2.

3.39
graphe de contrôle d'exécution
graphe ECC
représentation graphique ou textuelle des relations causales entre des *événements* aux *entrées d'événements* et aux *sorties d'événements* d'un *bloc fonctionnel* et l'*exécution* des *algorithmes* du bloc fonctionnel, en utilisant des *états de contrôle d'exécution,* des *transitions de contrôle d'exécution* et des *actions de contrôle d'exécution*

3.40
état initial de contrôle d'exécution
état initial EC
état de contrôle d'exécution qui est actif à la suite de l'initialisation d'un *graphe de contrôle d'exécution*

3.41
état de contrôle d'exécution
état EC
situation dans laquelle le comportement d'un *bloc fonctionnel de base* par rapport à ses *variables* est déterminé par les *algorithmes* associés à un jeu spécial d'*actions de contrôle d'exécution*

3.42
transition de contrôle d'exécution
transition EC
moyen par lequel le contrôle passe d'un *état de contrôle d'exécution* précédent à un *état de contrôle d'exécution* suivant

3.43
défaut
condition anormale pouvant être à l'origine d'une diminution ou d'une perte de capacité d'une *unité fonctionnelle* pour accomplir une *fonction* requise

3.44
fonction
but spécifique d'une *entité* ou son action caractéristique

3.45
bloc fonctionnel
instance de bloc fonctionnel
unité fonctionnelle logicielle consistant en une copie individuelle nommée d'une structure de données sur laquelle des *opérations* associées peuvent être accomplies comme spécifié par un *type de bloc fonctionnel* correspondant

Note 1 à l'article: Les opérations typiques d'un bloc fonctionnel comprennent la modification des valeurs des données dans sa structure de données associée.

Note 2 à l'article: L'*instance de bloc fonctionnel* et son *type de bloc fonctionnel* correspondant défini dans la CEI 61131-3 sont des éléments de langage de programmation avec un jeu différent de caractéristiques.

3.46
réseau de blocs fonctionnels
réseau dont les nœuds sont des *blocs fonctionnels* ou des *sous-applications* et leurs *paramètres* et dont les branches sont des *connexions de données* et des *connexions d'événements*

Note 1 à l'article: Il s'agit d'une généralisation du *diagramme de blocs fonctionnels* défini dans la CEI 61131-3.

3.47
type de bloc fonctionnel
type dont les *instances* sont des *blocs fonctionnels*

Note 1 à l'article: Les types de bloc fonctionnel comprennent des types de bloc fonctionnel de base, des types de bloc fonctionnel composé et des types de bloc fonctionnel interface de service

3.48
unité fonctionnelle
entité d'équipement *matériel* et/ou *logiciel* capable d'accomplir une action spécifiée

3.49
matériel
équipement physique, par opposition aux programmes, procédures, règles et documentation associée

3.50
identificateur
un ou plusieurs *caractères* utilisés pour nommer une *entité*

3.51
mise en œuvre
phase de développement dans laquelle le *matériel* et le *logiciel* d'un *système* deviennent opérationnels

3.52
primitive indication
primitive de service qui représente une interaction dans laquelle une *ressource* soit
a) indique qu'elle a, de sa propre initiative, *invoqué* un certain *algorithme*;
b) indique qu'un *algorithme* a été invoqué par une *application* homologue

3.53
information
signification qui est attribuée actuellement à une *donnée* au moyen de conventions appliquées à la donnée en question

3.54
variable d'entrée
variable dont la valeur est fournie par une *entrée de données* et qui peut être utilisée dans une ou plusieurs *opérations* d'un *bloc fonctionnel*

Note 1 à l'article: Un *paramètre d'entrée* d'un *bloc fonctionnel*, tel que défini dans la CEI 61131-3, est une *variable d'entrée*.

3.55
instance
unité fonctionnelle consistant en une *entité* nommée individuelle avec les *attributs* d'un *type* défini

3.56
nom d'instance
identificateur associé à une *instance* et la désignant

3.57
instanciation
création d'une *instance* d'un *type* spécifié

3.58
interface
frontière partagée entre deux *unités fonctionnelles*, définies par les caractéristiques fonctionnelles, caractéristiques de signal ou d'autres caractéristiques appropriées selon le cas

3.59
opération interne
<d'un *bloc fonctionnel*> *opération* associée à un *algorithme* d'un bloc fonctionnel, avec son contrôle d'*exécution* ou avec les capacités fonctionnelles de la *ressource* associée

3.60
variable interne
variable dont la valeur est utilisée ou modifiée par une ou plusieurs *opérations* d'un *bloc fonctionnel*, mais n'est pas fournie par une *entrée de données* ou à une *sortie de données*

3.61
invocation
processus de lancement de l'*exécution* de la séquence d'*opérations* spécifiées dans un *algorithme*

3.62
liaison
élément de conception décrivant la *connexion* entre un *équipement* et un *segment* du *réseau*

3.63
libellé
unité lexicale qui représente directement une valeur

3.64
bloc fonctionnel de gestion
bloc fonctionnel dont la *fonction* principale est la gestion d'*applications* dans les limites d'une *ressource*

3.65
ressource de gestion
ressource dont la *fonction* principale est la gestion d'autres *ressources*

3.66
mapping
ensemble de caractéristiques ou d'*attributs* ayant une correspondance définie avec les membres d'un autre ensemble

3.67
message
série ordonnée de *caractères* destinés à acheminer de l'*information*

3.68
puits de messages
partie intégrante d'un *système* de communication dans laquelle des *messages* sont considérés comme reçus

3.69
source de messages
partie intégrante d'un *système* de communication de laquelle des *messages* sont considérés provenir

3.70
modèle
représentation mathématique ou physique d'un système ou d'un processus

3.71
multitâche
mode de fonctionnement qui prévoit l'*exécution concomitante* de deux *algorithmes* ou plus

3.72
réseau
agencement de nœuds et de branches d'interconnexion

3.73
opération
action bien définie qui, lorsqu'elle est appliquée à n'importe quelle combinaison admissible d'*entités* connues, produit une nouvelle *entité*

3.74
variable de sortie
variable dont la valeur est établie par une ou plusieurs *opérations* d'un *bloc fonctionnel* et est fournie à une *donnée de sortie*

Note 1 à l'article: Un *paramètre de sortie* d'un *bloc fonctionnel*, comme défini dans la CEI 61131-3, est une *variable de sortie.*

3.75
paramètre
variable à laquelle est donnée une valeur constante pour une *application* spécifiée et qui peut représenter l'application

3.76
prise mâle
fiche d'adaptation
instance d'un *type d'adaptateur d'interface* qui fournit un point de départ d'une *connexion d'adaptation* à partir d'un bloc fonctionnel *fournisseur*

3.77
fournisseur
instance de bloc fonctionnel qui fournit une *fiche d'adaptation* d'un *type d'adaptateur d'interface* défini

3.78
primitive request
primitive de service qui représente une interaction dans laquelle une *application* invoque un certain *algorithme* fourni par un *service*

3.79
demandeur
unité fonctionnelle qui lance une *transaction* par le biais d'une *primitive request*

3.80
ressource
unité fonctionnelle qui a un contrôle indépendant de son fonctionnement et qui fournit divers *services* à des *applications,* y compris la programmation et l'*exécution* d'*algorithmes*

Note 1 à l'article: Le terme RESOURCE défini dans la CEI 61131-3:2003, 1.3.66 est un élément de langage de programmation correspondant à la *ressource* définie ci-dessus.

Note 2 à l'article: Un *équipement* contient une ou plusieurs ressources.

3.81
application de gestion de ressource
application dont la fonction principale est la gestion d'une seule *ressource*

3.82
répondeur
unité fonctionnelle qui conclut une *transaction* par le biais d'une *primitive response*

3.83
primitive response
primitive de service qui représente une interaction dans laquelle une *application* indique qu'elle a exécuté un *algorithme* précédemment *invoqué* par une interaction représentée par une *primitive indication*

3.84
échantillonner, verb
capter et retenir la valeur instantanée d'une *variable* pour une utilisation ultérieure

3.85
fonction de programmation
fonction qui sélectionne des *algorithmes* ou des *opérations* pour l'*exécution* et lance et arrête cette exécution

3.86
segment
partition physique d'un *réseau de communication*

3.87
service
capacité fonctionnelle d'une *ressource* qui peut être modélisée par une séquence de *primitives de service*

3.88
bloc fonctionnel interface de service
bloc fonctionnel qui fournit un ou plusieurs *services* à une *application*, en se basant sur un *mapping* des *primitives de service* avec les *événements d'entrée,* les *événements de sortie,* les *données d'entrée* et les *données de sortie* du bloc fonctionnel

3.89
primitive de service
représentation abstraite et indépendante vis-à-vis de toute mise en œuvre, d'une interaction entre une *application* et une *ressource*

3.90
diagramme de séquence d'un service
diagramme représentant une séquence de *primitives de service*

3.91
prise femelle
support adaptateur
instance d'un *type adaptateur d'interface* qui fournit un point d'extrémité pour une *connexion d'adaptation* à un bloc fonctionnel d'*utilisateur*

3.92
logiciel
création intellectuelle comprenant les programmes, procédures, règles, *configurations* et toute documentation associée relatifs à l'exploitation d'un *système*

3.93
outil logiciel
logiciel qui est utilisé pour la production, l'inspection ou l'analyse d'un autre logiciel

3.94
instance de sous-application
instance d'un *type sous-application* à l'intérieur d'une *application* ou à l'intérieur d'un type sous-application

Note 1 à l'article: Une instance de sous-application peut être distribuée parmi des *ressources*, c'est-à-dire que ses blocs fonctionnels composants ou le contenu de ses sous-applications constitutives peu(ven)t être affecté(s) à des ressources différentes.

3.95
type de sous-application
unité fonctionnelle dont le corps est constitué de *blocs fonctionnels composants* ou de *sous-applications constitutives*

Note 1 à l'article: Un type sous-application permet la création de sous-structures d'*applications* sous la forme d'une hiérarchie autosemblable.

3.96
système
ensemble d'éléments reliés entre eux, considérés dans un contexte défini comme un tout et séparés de leur environnement

Note 1 à l'article: De tels éléments peuvent être tant des objets matériels que des concepts et les résultats de ceux-ci (par exemple: formes d'organisation, méthodes mathématiques, langages de programmation)

Note 2 à l'article: Le système est considéré comme séparé de l'environnement et des autres systèmes extérieurs par une surface imaginaire à travers laquelle peuvent passer les liaisons entre le système considéré et l'environnement.

3.97
variable temporaire
variable dont la valeur est initialisée, utilisée et éventuellement modifiée durant l'*exécution* d'un *algorithme*; qui n'est pas visible à l'extérieur du corps de l'algorithme, et dont la valeur ne persiste pas d'une exécution de l'algorithme à l'autre

3.98
transaction
unité de service dans laquelle une demande et éventuellement des *données* sont acheminées d'un *demandeur* vers un *répondeur* et dans laquelle une réponse et éventuellement des données peuvent également être acheminées en retour du répondeur vers le demandeur

3.99
type
élément *logiciel* qui spécifie les *attributs* communs partagés par toutes les *instances* du type

3.100
nom de type
identificateur associé à un *type* et le désignant

3.101
transaction unidirectionnelle
transaction dans laquelle une demande et éventuellement des *données* sont acheminées d'un *demandeur* vers un *répondeur* et dans laquelle une réponse n'est pas acheminée en retour du répondeur vers le demandeur

3.102
variable
entité logicielle qui peut prendre différentes valeurs, une à la fois

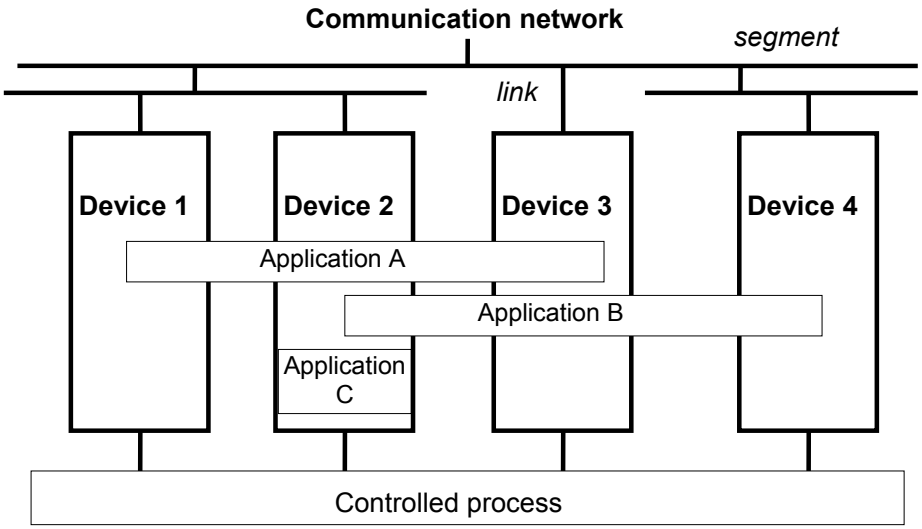
Note 1 à l'article: Les valeurs d'une variable sont habituellement limitées à un certain *type de données*.

Note 2 à l'article: Les variables peuvent être classées en *variables d'entrée, variables de sortie*, *variables internes* et *variables temporaires*.

4 Modèles de référence

4.1 Modèle pour un système

Pour les besoins de la CEI 61499, un *système* de mesure et de commande dans les processus industriels (IPMCS) est modélisé (voir Figure 1) comme étant un ensemble d'*équipements* connectés entre eux et communiquant les uns avec les autres au moyen d'un *réseau* de communication constitué de *segments* et de *liaisons*. Les équipements sont connectés à des segments de réseau par des *liaisons*.



NOTE Le processus commandé n'est pas partie intégrante du système de mesure et de commande.

Légende

Anglais	Français
Communication network	Réseau de communication
segment	segment
link	liaison
Device	Équipement
Application A	Application A
Application B	Application B
Appl. C	Application C
Controlled process	Processus commandé

Figure 1 – Modèle de système

Une *fonction* accomplie par l'IPMCS est modélisée comme une *application* qui peut résider dans un seul équipement, telle que l'application C à la Figure 1, ou peut être distribuée parmi plusieurs équipements, telle que les applications A et B de la Figure 1. Par exemple, une application peut être constituée d'une ou plusieurs boucles de commande dans lesquelles l'échantillonnage des entrées est accompli dans un équipement, le traitement de commande est accompli dans un autre équipement et la conversion des sorties dans un troisième.

4.2 Modèle pour un équipement

Comme illustré à la Figure 2, un *équipement* doit contenir au moins une *interface,* à savoir, une interface de processus ou une interface de communication et peut contenir zéro *ressource* ou plus.

NOTE 1 Un équipement est considéré être une *instance* d'un *type* d'équipements correspondant comme spécifié en 7.2.2.

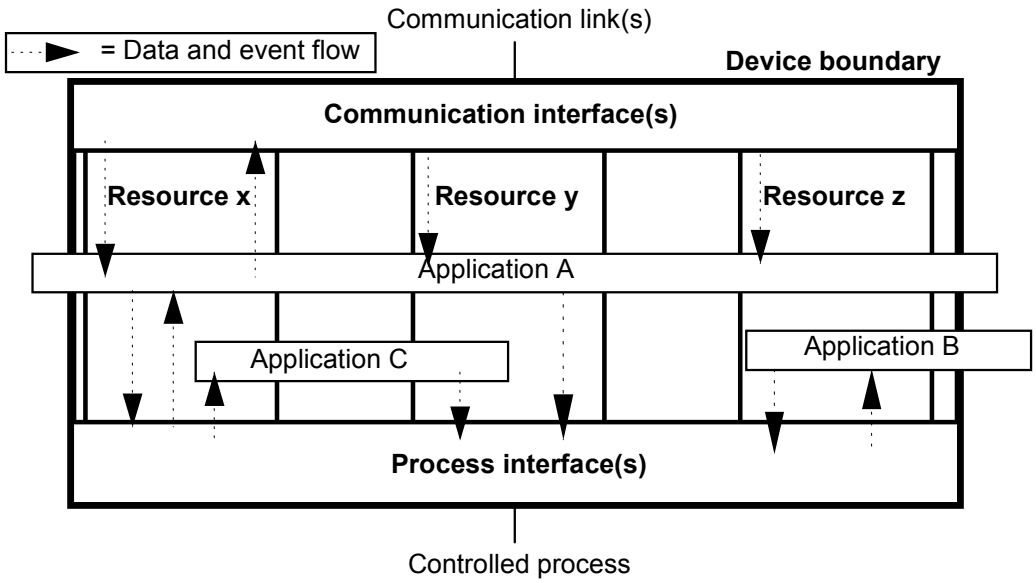
NOTE 2 Un équipement qui ne contient aucune ressource est considéré être fonctionnellement l'équivalent d'une *ressource* telle que définie en 4.3.

Une "interface de processus" assure un *mapping* entre le processus physique (mesures analogiques, E/S discrètes, etc.) et les ressources. Les informations échangées avec le processus physique sont présentées à la ressource sous forme de *données* et/ou d'*événements*.

Les *interfaces* de communication assurent un mapping entre des ressources et les informations échangées par l'intermédiaire d'un *réseau*. Les services fournis par les interfaces de communication peuvent comprendre:

- la présentation d'informations de communication à la ressource sous forme de *données* et/ou d'*événements*;
- des services complémentaires pour prendre en charge la programmation, la *configuration*, le diagnostic, etc.

Les *liaisons* de communication peuvent être associées soit directement à un *équipement*, soit à une instance de type *ressource* spécifique (ressource de communication) à laquelle une partie de l'application distribuée peut ou peut ne pas être mappée, en fonction du type de ressource.



NOTE Cette figure illustre une structure interne possible de l'Équipement 2 de la Figure 1.

Légende

Anglais	Français
Communication link(s)	Liaison(s) de communication
Resource x	Ressource x
Controlled process	Processus commandé
Resource z Resource y	Ressource z Ressource y
Application B Application C	Application B Application C
Application A	Application A
Device boundary	Limite de l'équipement
Communication interface(s)	Interface(s) de communication
Process interface(s)	Interface(s) du processus
= Data & event flow	Flot de données et d'événements

Figure 2 – Modèle d'un équipement

4.3 Modèle pour une ressource

Pour les besoins de la CEI 61499, une *ressource* est considérée être une *unité fonctionnelle* dotée d'une commande indépendante pour son fonctionnement et qui est contenue dans un *équipement*. Elle peut être créée, configurée, paramétrée, démarrée, supprimée, etc., sans avoir d'influence sur d'autres ressources.

NOTE 1 Une ressource est considérée être une *instance* du *type* de ressources correspondant comme spécifié en 7.2.1.

NOTE 2 Bien qu'une ressource ait une commande indépendante pour son fonctionnement, ses états opérationnels pourraient nécessiter d'être coordonnés avec ceux d'autres ressources à des fins d'installation, d'essai, etc.

Les *fonctions* d'une ressource sont d'accepter des *données* et/ou des *événements* provenant des *interfaces* de processus et/ou de communication, de traiter les données et/ou événements et de retourner les données et/ou événements aux interfaces du processus et/ou de communication, comme spécifié par les *applications* qui utilisent la ressource.

NOTE 3 Outre la prise en charge des fonctions énumérées ci-dessus, des types spécifiques de ressources pourraient représenter une capacité de mise en œuvre des fonctions d'interface telles que les interfaces de processus ou des services de communications de couche inférieure sur des liaisons de communication. En fonction du type de ressource en question, ces services sont susceptibles ou non d'être les seuls qu'elles soient capables de fournir.

NOTE 4 La prise en considération d'autres aspects possibles de ressources ne relève pas du domaine d'application de la présente norme.

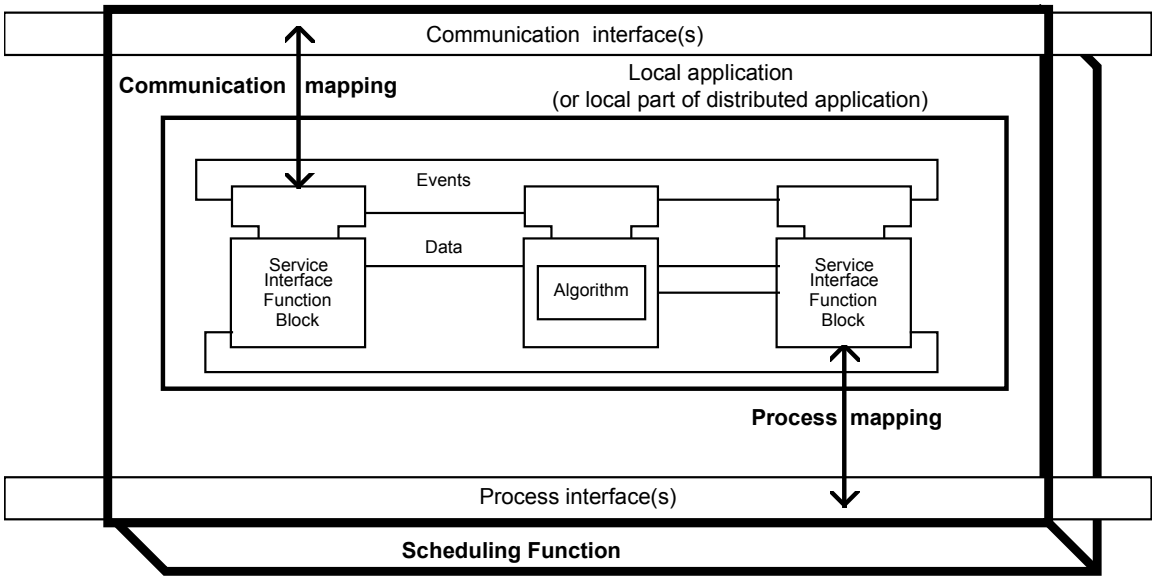
Comme illustré à la Figure 3, une ressource est modélisée par ce qui suit:

- Une ou plusieurs "applications locales" (ou parties locales d'applications distribuées). Les *variables* et *événements* manipulés dans la présente partie sont des *variables d'entrée* et de *sortie* et des événements aux *entrées d'événements* et *sorties d'événements* des *blocs fonctionnels* qui accomplissent les *opérations* nécessaires à l'application.
- Une partie "mapping du processus" qui consiste à définir un *mapping* des *données* et des *événements* entre les *applications* et une ou plusieurs *interfaces* du processus. Comme montré à la Figure 3, ce mapping peut être modélisé par des *blocs fonctionnels interface de service* spécialisés dans ce but.
- Une partie "mapping de la communication" qui consiste à définir un *mapping* des *données* et des *événements* entre des *applications* et des *interfaces de communication*. Comme montré à la Figure 3, ce mapping peut être modélisée par des *blocs fonctionnels interface de service* spécialisés dans ce but.
- Une *fonction* de programmation qui effectue l'exécution des blocs fonctionnels dans les applications et le transfert de données entre ceux-ci, conformément aux exigences relatives à la temporisation et à la séquence déterminées par:

  a) l'occurrence des événements;

  b) les interconnexions des blocs fonctionnels; et

  c) les informations de programmation telles que les périodes et les priorités.



NOTE 1 Cette figure est uniquement illustrative. Ni la représentation graphique ni l'emplacement des blocs fonctionnels ne sont normatifs.

NOTE 2 Des interfaces de communication et de processus peuvent 'être partagées entre les ressources.

Légende

Anglais	Français
Communication interface(s)	Interface(s) de communication
Communication mapping	Mapping de la communication
Local application (or local part of distributed application)	Application locale (ou partie locale d'une application distribuée)
Events	Événements
Data	Données
Service Interface Function Block	Bloc fonctionnel interface de service
Algorithm	Algorithme
Process mapping	Mapping du processus
Process interface(s)	Interface(s) du processus
Scheduling Function	Fonction de programmation

Figure 3 – Modèle d'une ressource

4.4 Modèle pour une application

Pour les besoins du présent document, une *application* consiste en un *réseau de blocs fonctionnels,* dont les nœuds sont des *blocs fonctionnels* ou des *sous-applications* et leurs *paramètres* et dont les branches sont des *connexions de données* et des *connexions d'événements*.

Les sous-applications sont des *instances* de *type sous-application*, qui, comme les applications, sont constituées de *réseaux de blocs fonctionnels*. Les noms d'application et les *noms d'instances* de sous-application et de blocs fonctionnels peuvent donc être utilisés pour créer une hiérarchie d'*identificateurs* qui peuvent identifier de façon univoque chaque *instance* de *bloc fonctionnel* dans un *système*.

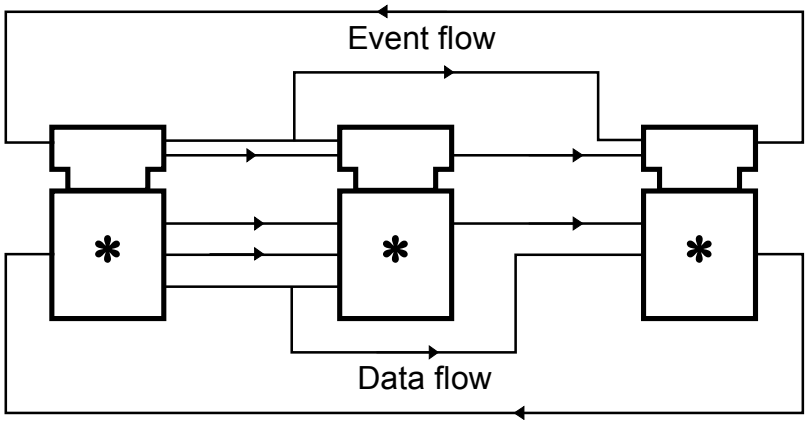
Une application peut être distribuée parmi plusieurs *ressources* dans des *équipements* identiques ou différents. Une *ressource* utilise les relations causales spécifiées par l'application pour déterminer les réponses appropriées à des *événements* qui peuvent se

produire à partir des interfaces de communication et du processus ou à partir d'autres fonctions de la ressource. Ces réponses peuvent comprendre:

- la programmation et l'*exécution* d'*algorithmes;*
- la modification de *variables;*
- la génération d'événements supplémentaires;
- les interactions avec des interfaces de communication et du processus.

Dans le contexte du présent document, les applications sont définies par des *réseaux de blocs fonctionnels* spécifiant un flot d'événements et de données parmi les *instances* de *blocs fonctionnels* ou de *sous-applications*, comme illustré à la Figure 4. Le flot d'événements détermine la programmation et l'*exécution*, par la ressource associée, des *opérations* spécifiées par le ou les *algorithmes* de chaque bloc fonctionnel, selon les règles données en 5.2.2.

Les normes, composants et systèmes conformes à la présente norme peuvent utiliser des moyens de substitution pour programmer l'exécution. Ces moyens de substitution doivent être spécifiés exactement à l'aide des éléments définis dans la présente norme.



NOTE 1 "*" représente des instances de bloc fonctionnel ou de sous-application.

NOTE 2 Cette figure est uniquement illustrative. La représentation graphique n'est pas normative.

Légende

Anglais	Français
Event flow	Flot d'événements
Data flow	Flot de données

Figure 4 – Modèle d'application

4.5 Modèle de bloc fonctionnel

4.5.1 Caractéristiques des instances de bloc fonctionnel

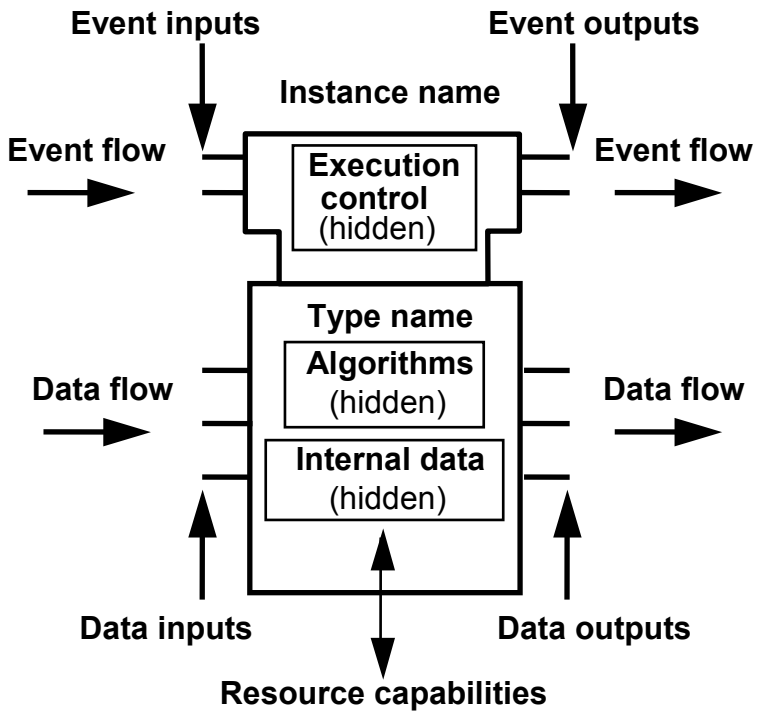
Un *bloc fonctionnel (une instance de bloc fonctionnel)* est une *unité fonctionnelle* de *logiciel* comprenant une copie nommée qui lui est propre de la structure de données spécifiée par un *type de bloc fonctionnel*, qui demeure d'une *invocation* du bloc fonctionnel à la suivante. Les caractéristiques des instances de bloc fonctionnel sont décrites en 4.5.1, tandis que les spécifications des types de bloc fonctionnel sont décrites en 4.5.2.

Une *instance de bloc fonctionnel* présente les éléments caractéristiques suivants, tels qu'illustrés à la Figure 5:

- son *nom de type* et son *nom* d'*instance*;

- un ensemble d'*entrées d'événements,* chacune de celles-ci pouvant recevoir des *événements* provenant d'une *connexion d'événements* qui peuvent altérer l'exécution d'un ou plusieurs *algorithmes*;
- un ensemble de *sorties d'événements,* chacune de celles-ci pouvant émettre des *événements* vers une *connexion d'événements* en fonction de l'exécution d'*algorithmes* ou d'autre capacité fonctionnelle de la *ressource* dans laquelle le bloc fonctionnel est situé;
- un ensemble d'*entrées de données,* qui peuvent être *mises en correspondance* avec les *variables d'entrée* correspondantes;
- un ensemble *de sorties de données,* qui peuvent être *mises en correspondance* avec les *variables de sortie* correspondantes;
- des *données* internes, qui peuvent être *mises en correspondance* avec un ensemble de *variables internes*;
- des caractéristiques fonctionnelles qui sont déterminées en combinant des données internes et/ou des informations d'état internes à un ensemble d'*algorithmes* et/ou de caractéristiques fonctionnelles de la *ressource* associée. Ces caractéristiques fonctionnelles sont définies dans la spécification du *type* de bloc fonctionnel.

NOTE Les informations d'état internes peuvent être représentées par des *variables internes* ou par une représentation interne d'un diagramme d'états de contrôle d'exécution.



NOTE Cette figure est uniquement illustrative. La représentation graphique n'est pas normative.

Légende

Anglais	Français
Event inputs	Entrées d'événements
Event outputs	Sorties d'événements
Instance name	Nom d'instance
Event flow	Flot d'événements
Execution Control (hidden)	Contrôle d'exécution (caché)
Data flow	Flot de données

Anglais	Français
Type name	Nom du type
Algorithms (hidden)	Algorithmes (cachés)
Internal data (hidden)	Données internes (cachées)
Data inputs	Entrées de données
Data outputs	Sorties de données
Resource capabilities	Capacités des ressources
(Scheduling, communication mapping, process mapping)	(Programmation, mapping de la communication, mapping du processus)

Figure 5 – Caractéristiques des blocs fonctionnels

Les algorithmes contenus dans un bloc fonctionnel sont, en principe, invisibles de l'extérieur du bloc fonctionnel, avec les exceptions décrites, de façon formelle ou informelle, par le fournisseur du bloc fonctionnel. En outre, le bloc fonctionnel peut contenir des *variables* internes et/ou des informations d'état internes, qui persistent entre les invocations des algorithmes du bloc fonctionnel, mais qui ne sont pas accessibles avec les connexions des flots de données depuis l'extérieur du bloc fonctionnel.

L'accès aux variables et informations d'état internes des instances du bloc fonctionnel peut être fourni par des capacités fonctionnelles supplémentaires de la ressource associée.

Les moyens pour spécifier les relations causales parmi les entrées d'événements, les sorties d'événements et l'exécution des algorithmes sont définis aux Articles 5 et 6.

4.5.2 Spécifications des types de bloc fonctionnel

Un *type de bloc fonctionnel* est l'élément *logiciel* qui spécifie les caractéristiques de toutes les *instances* du type, y compris:

- son *nom de type*.
- le nombre, les noms, les noms de type et l'ordre des *entrées d'événements* et des *sorties d'événements.*
- le nombre, les noms, le *type de données* et l'ordre des *variables* d'entrée, de sortie et internes.

Les mécanismes pour la *déclaration* de ces caractéristiques sont définis en 5.2.1.

En outre, la spécification des types de bloc fonctionnel définit la fonctionnalité des *instances* du type. Cette fonctionnalité peut être exprimée comme suit:

- Pour les *types de bloc fonctionnel de base*, des mécanismes de déclaration sont donnés en 5.2.1.3 pour la spécification des *algorithmes*, qui opèrent sur les valeurs de *variables d'entrée,* de *variables de sortie* et des *variables internes* pour produire de nouvelles valeurs des *variables* de *sortie* et des *variables internes*. Les associations entre l'*invocation* des algorithmes et l'occurrence d'*événements* aux entrées et sorties d'événements sont exprimées au moyen d'un *graphe de contrôle d'exécution* (ECC), en utilisant les mécanismes de déclaration définis en 5.2.1.4.
- La fonctionnalité d'une *instance* d'un *type de bloc fonctionnel composé* ou d'un *type de sous-application* est déclarée, en utilisant respectivement les mécanismes définis en 5.3.1 et en 5.4.1, en termes de *connexions de données* et de *connexions d'événements* parmi ses *blocs fonctionnels composants* ou sous-applications et les entrées et sorties d'événements et de données du bloc fonctionnel composé ou de la sous-application.
- La fonctionnalité d'une instance d'un *type de bloc fonctionnel interface de service* est décrite par un *mapping* de *primitives de service* avec des *entrées d'événements, sorties*

d'événements, entrées de données et *sortie de données*, en utilisant les mécanismes de déclaration définis en 6.1.

- D'autres moyens tels qu'un texte en langage naturel peuvent être utilisés pour décrire la fonctionnalité du type de bloc fonctionnel; cependant, la spécification d'un tel moyen ne relève pas du domaine d'application de la présente norme.

4.5.3 Modèle d'exécution pour les blocs fonctionnels de base

Comme montré à la Figure 6, l'*exécution* des *algorithmes* pour les *blocs fonctionnels de base* est invoquée par la partie **contrôle d'exécution** d'une *instance de bloc fonctionnel* en réponse à des événements au niveau des *entrées d'événements*. Cette *invocation* prend la forme d'une demande faite à la **fonction de programmation** de la *ressource* associée de programmer l'exécution des *opérations* de l'algorithme. À la fin de l'exécution d'un algorithme, le contrôle d'exécution génère zéro événement ou plus au niveau des *sorties d'événements* selon le cas.

Les *événements aux entrées d'événements* sont fournis par une connexion aux *sorties d'événements* d'autres instances de blocs fonctionnels ou de la même instance de bloc fonctionnel. Les événements sur ces sorties d'événements peuvent être générés par le contrôle d'exécution tel que décrit ci-dessus ou par le "mapping de la communication", le "mapping du processus", la "programmation" ou autre capacité fonctionnelle de la *ressource.*

NOTE 1 Le contrôle d'exécution dans des blocs fonctionnels composés est réalisé par le biais d'un flot d'événements au sein du corps du bloc fonctionnel.

La Figure 6 montre l'ordre des événements et l'exécution de l'algorithme dans le cas de l'association d'une seule entrée d'événements, d'un seul algorithme et d'une seule sortie d'événements. Les temps pertinents dans ce diagramme sont définis comme suit:

t_1: les valeurs des variables d'entrée pertinentes (c'est-à-dire: celles qui sont associées à l'entrée d'événements par le qualificateur WITH défini en 5.2.1.2) sont rendues disponibles;

t_2: l'événement sur l'entrée d'événements se produit;

t_3: la fonction de contrôle d'exécution notifie à la fonction de programmation de ressource de programmer un algorithme pour l'exécution;

t_4: l'exécution de l'algorithme commence;

t_5: l'algorithme parachève l'établissement des valeurs pour les variables de sortie associées à la sortie d'événements par le qualificateur WITH défini en 5.2.1.2;

t_6: la fonction de programmation de ressource reçoit notification que l'exécution d'algorithme est terminée;

t_7: la fonction de programmation invoque la fonction de contrôle d'exécution;

t_8: la fonction de contrôle d'exécution signale un événement sur la sortie d'événements.

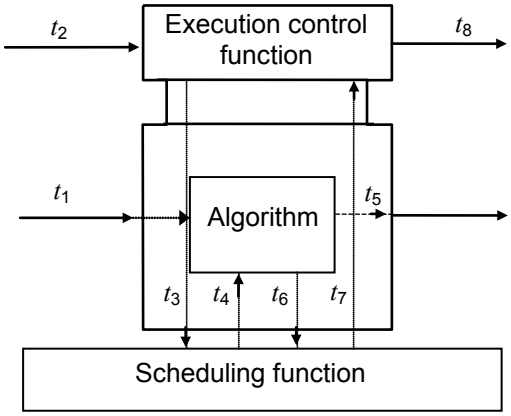
Comme montré à la Figure 7, les retards significatifs de temporisation qui présentent dans ce cas un intérêt dans la conception de l'application sont:

T_{setup} = $t_2 - t_1$

T_{start} = $t_4 - t_2$ (temps depuis l'apparition de l'événement sur l'entrée d'événements jusqu'au début de l'exécution d'algorithme)

T_{alg} = $t_6 - t_4$ (temps d'exécution d'algorithme)

T_{finish} = $t_8 - t_6$ (temps depuis la fin de l'exécution d'algorithme jusqu'à l'apparition de l'événement sur la sortie d'événements)

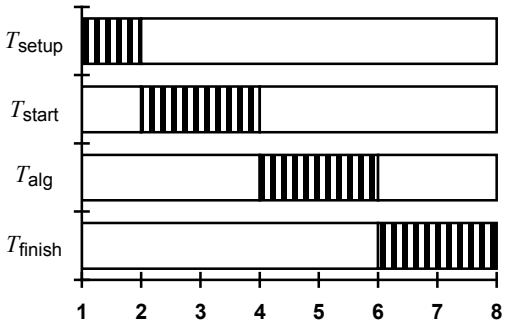


Légende

Anglais	Français
Execution control function	Fonction de contrôle d'exécution
Algorithm	Algorithme
Scheduling function	Fonction de programmation

NOTE Cette figure est uniquement illustrative. La représentation graphique n'est pas normative.

Figure 6 – Modèle d'exécution



NOTE les étiquettes d'axe 1, 2,... dans la figure ci-dessus correspondent aux temps t_1, t_2,... de la Figure 6.

Figure 7 – Temporisation de l'exécution

Les exigences spécifiques pour la présentation graphique des types de de bloc fonctionnel sont définies en 5.2.1.1.

NOTE 2 En fonction du problème à résoudre, des exigences diverses pourraient exister pour la synchronisation des valeurs des *variables d'entrée* avec l'*exécution* des *algorithmes afin d'assurer la prédictibilité des résultats de l'exécution de l'algorithme*. De telles exigences pourraient comprendre, par exemple:

- l'assurance que les valeurs des variables utilisées par un algorithme restent stables pendant l'exécution de l'algorithme;

- l'assurance que les valeurs des variables utilisées par un algorithme correspondent aux données présentes lors de la survenue de l'événement sur l'entrée d'événements qui a entrainé la programmation de l'algorithme en vue d'une exécution;
- l'assurance que les valeurs des variables utilisées par tous les algorithmes programmés en vue de l'exécution dans un bloc fonctionnel correspondent aux données présentes lors de survenue de l'événement sur l'entrée d'événements qui a entrainé la programmation du premier de ces algorithmes en vue d'une exécution.

NOTE 3 Des *ressources* pouraient avoir besoin de programmer l'*exécution* d'*algorithmes* de manière *multitâche*. La spécification des attributs pour faciliter une telle programmation est décrite dans l'Annexe G.

4.6 Modèle de distribution

Comme illustré à la Figure 8a), une *application* ou une *sous-application* peut être distribuée en allouant ses *instances de bloc fonctionnel* à différentes *ressources* dans un ou plusieurs *équipement*. Les détails internes d'un bloc fonctionnel étant cachés à tout application ou sous-application utilisant celui-ci, un bloc fonctionnel doit former une unité atomique de distribution. Autrement dit, tous les éléments contenus dans une instance de bloc fonctionnel donnée doivent être contenus dans la même ressource.

Les relations fonctionnelles entre les blocs fonctionnels d'une application ou d'une sous-application ne doivent pas être altérées par sa distribution. Cependant, contrairement à une application ou sous-application confinée à une seule ressource, la temporisation et la fiabilité des fonctions de communications auront une incidence sur la temporisation et la fiabilité d'une application ou sous-application distribuée.

Les articles suivants s'appliquent lorsque des applications ou sous-applications sont distribuées entre plusieurs ressources:

- l'Article 6 définit les exigences pour les services de communication pour prendre en charge la distribution d'applications ou de sous-applications entre plusieurs équipements;
- l'Article 7 définit les exigences pour le cas dans lequel plusieurs applications ou sous-applications sont distribuées entre plusieurs ressources et équipements.

4.7 Modèle de gestion

Les Figures 8b et 8c donnent une représentation schématique de la gestion des *ressources* et des *équipements*. La Figure 8b illustre un cas dans lequel une *ressource de gestion* fournit des moyens partagés pour la gestion d'autres *ressources* au sein d'unéquipement, tandis que la Figure 8c illustre la distribution de services de gestion parmi des ressources au sein d'un équipement. Des *applications* de gestion peuvent être modélisées à l'aide des *blocs fonctionnels interface de service* et des *blocs fonctionnels de communication* **dépendants de la mise en œuvre**.

NOTE 1 Le 6.3 définit des *types de blocs fonctionnels interface de service* pour la gestion d'*applications*, et la CEI 61499-2 donne des exemples de leur utilisation.

NOTE 2 Les *applications de gestion* pourraient contenir des *instances de blocs fonctionnels interface de service* représentant des instances d'*équipement* ou de *ressource* pour les besoins d'interrogation ou de modification des *paramètres* de l'équipement ou de la ressource.

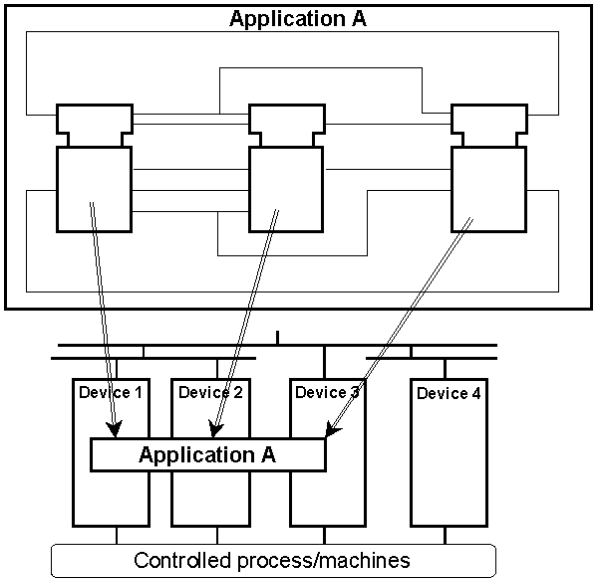


Figure 8a – Modèle de distribution

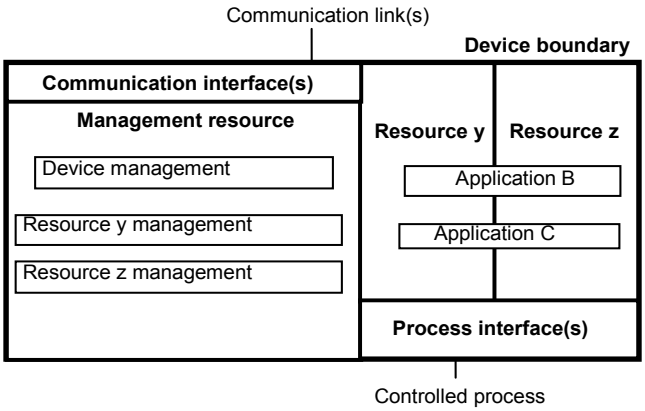


Figure 8b – Modèle de gestion partagé

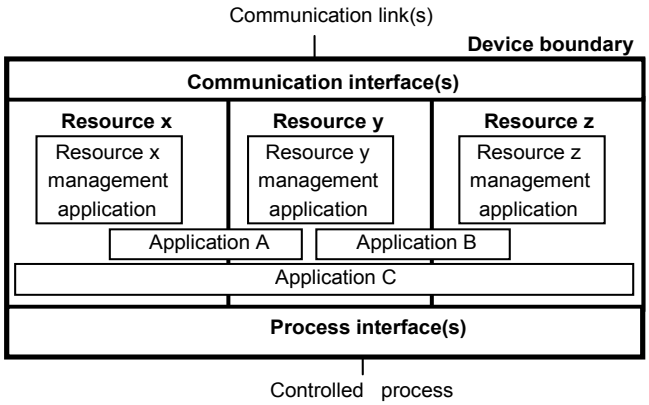


Figure 8c – Modèle de gestion distribué

Légende

Anglais	Français
Application A	Application A
Device 1, 2, 3, 4	Équipement 1, 2, 3, 4
Controlled process/machines	Diagrammes/processus commandé
a) Distribution model	a) Modèle de distribution
Communication link(s)	Liaison(s) de communication
Device boundary	Limite de l'équipement
Communication interface(s)	Interface(s) de communication
Management resource	Gestion des ressources
Resource z Resource y	Ressource z Ressource y
Device management	Gestion de l'équipement
Application B	Application B
Resource y management	Gestion de la ressource y
Application C	Application C
Resource z management	Gestion de la ressource z
Process interface(s)	Interface(s) du processus
Controlled process	Processus commandé
b) Shared management model	b) Modèle de gestion partagée
Communication link(s)	Liaison(s) de communication
Device boundary	Limite de l'équipement
Communication interface(s)	Interface(s) de communication
Resource x	Ressource x
Resource y	Ressource y
Resource z	Ressource z
Process interface(s)	Interface(s) du processus
Resource x management application	Gestion de l'application ressource x
Resource y management application	Gestion de l'application ressource y
Resource z management application	Gestion de l'application ressource z
Application A	Application A
Application B	Application B
Process interface(s)	Interface(s) du processus
Controlled process	Processus commandé
c) Distributed management model	c) Modèle de gestion distribuée

Figure 8 – Modèles de distribution et de gestion

4.8 Modèles d'état opérationnel

Tout *système* donné doit être conçu, mis en service, exploité et maintenu. Cela est modélisé par le concept du "cycle de vie" du système. Réciproquement, un système est composé de plusieurs *unités fonctionnelles* (telles que des *équipements,* des *ressources* et des *applications*), chacune de celles-ci ayant son propre cycle de vie.

Différentes actions peuvent devoir être accomplies pour prendre en charge des *unités fonctionnelles* à chaque étape du cycle de vie. Afin de caractériser quelle action peut être réalisée et de maintenir l'intégrité des unités fonctionnelles, il convient de définir les états opérationnels; par exemple: OPERATIONAL, CONFIGURABLE, LOADED, STOPPED, etc.

Chaque état opérationnel d'une unité fonctionnelle spécifie quelles actions sont autorisées, ainsi que le comportement prévu.

Un système peut être organisé de telle manière que certaines unités fonctionnelles puissent posséder ou acquérir le droit de modifier les états opérationnels d'autres unités fonctionnelles.

Des exemples d'utilisation d'états opérationnels sont:

- une unité fonctionnelle dans un état RUNNING, c'est-à-dire en exécution, peut ne pas être capable de recevoir une action de téléchargement;
- une unité fonctionnelle distribuée peut avoir besoin de maintenir un état opérationnel cohérent parmi tous ses éléments composants et développer une stratégie pour propager à travers eux les changements d'état opérationnel.

Des états opérationnels spécifiques pour des *instances de bloc fonctionnel* gérées sont définis en 6.3.2.

5 Spécification des types de bloc fonctionnel, de sous-application et d'adaptateurs d'interface

5.1 Vue d'ensemble

Comme illustré à la Figure 9, l'Article 5 définit les moyens pour la spécification du type de trois catégories de blocs:

- Le paragraphe 5.2 définit le moyen de spécifier et déterminer le comportement d'instances des *blocs fonctionnels de base*, comme illustré à la Figure 9 a). Dans ce type de bloc fonctionnel, le contrôle d'exécution est spécifié par un *graphe de contrôle d'exécution (ECC)* tandis que les *algorithmes* devant être exécutés sont déclarés comme spécifié dans les Normes conformes telles que définies dans la CEI 61499-4.
- Le paragraphe 5.3 définit le moyen de spécifier les *types de bloc fonctionnel composé*, comme illustré à la Figure 9b. Dans ce type de bloc fonctionnel, les algorithmes et leur contrôle d'exécution sont spécifiés par le biais de connexions d'événements et de données en un ou plusieurs *réseaux de blocs fonctionnels.*
- Le paragraphe 5.4 définit le moyen de spécifier les *types de sous-application*, comme illustré à la Figure 9c. Dans ce type de bloc, les algorithmes et leur contrôle d'exécution sont spécifiés comme dans le cas des types de bloc fonctionnel composé, mais avec la propriété spécifique que les *blocs fonctionnels composants* des sous-applications peuvent être distribués entre plusieurs *ressources.* Les sous-applications peuvent être imbriquées et, de ce fait, le corps d'une sous-application peut aussi contenir des *sous-applications constitutives.*

D'autres moyens peuvent être utilisés pour décrire le comportement d'instances d'un type de bloc fonctionnel. La spécification d'un tel moyen ne relève pas du domaine d'application de la présente norme; par conséquent, l'exigence est la suivante: en cas d'utilisation d'un tel moyen, un *mapping* non ambigü doit être donné entre leurs termes et les concepts et les termes et concepts correspondants de la présente norme.

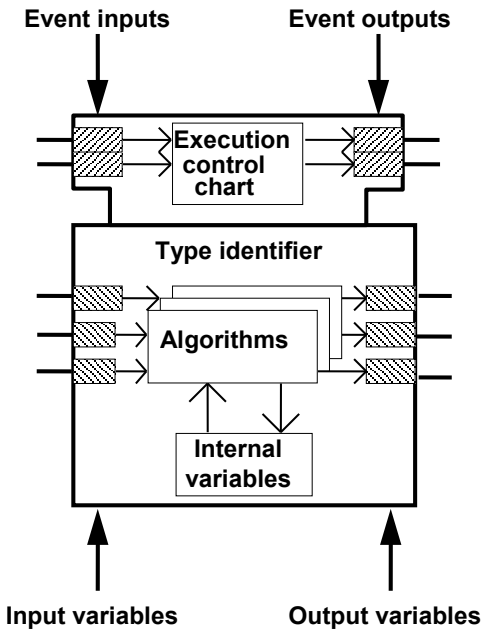


Figure 9a – Bloc fonctionnel de base (5.2)

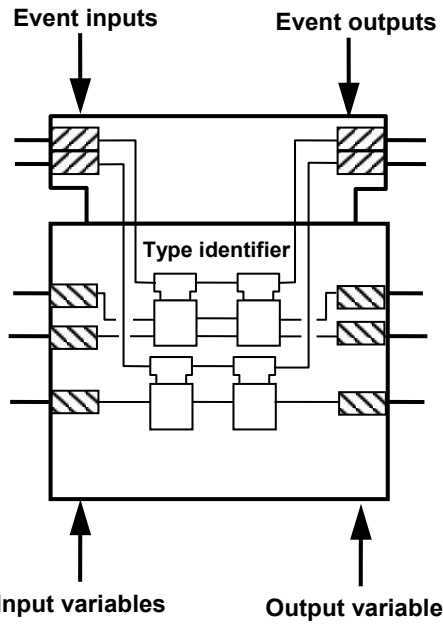


Figure 9b – Bloc fonctionnel composé (5.3)

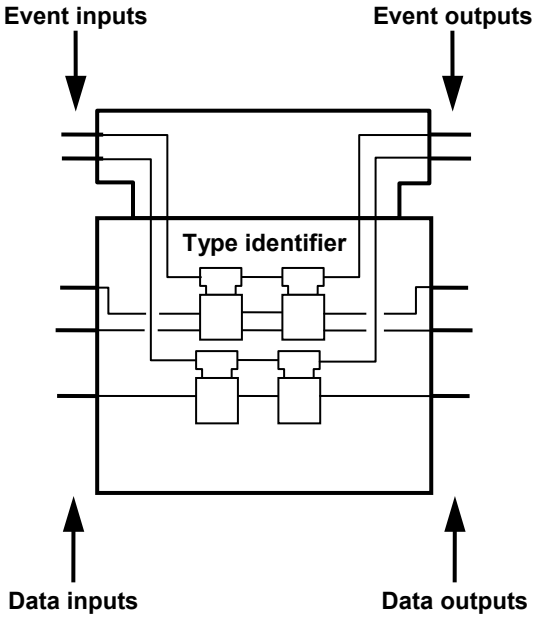


Figure 9c – Sous-application (5.4)

Légende

Anglais	Français
Event Inputs	Entrées d'événements
Event Outputs	Sorties d'événements
Execution Control Chart	Graphe de contrôle d'exécution
Type identifier	Identificateur du type
Algorithms	Algorithme
Internal variables	Variables internes
Input variables	Variables d'entrée
Output variables	Variables de sortie
Data inputs	Entrées des données
Data outputs	Sorties des données

NOTE Cette figure est uniquement illustrative. La représentation graphique n'est pas normative.

Figure 9 – Types de bloc fonctionnel et de sous-application

5.2 Blocs fonctionnels de base

5.2.1 Déclaration du type

5.2.1.1 Généralités

Un *bloc fonctionnel de base* utilise un *graphe de contrôle d'exécution (ECC)* pour contrôler l'*exécution* de ses *algorithmes*.

Comme illustré à la Figure 10, un *type* de *bloc fonctionnel de base* peut être déclaré textuellement conformément à la syntaxe spécifiée en B.2 ou graphiquement conformément aux règles suivantes:

a) le *nom de type* du bloc fonctionnel est montré au centre en haut de la partie inférieure du bloc;

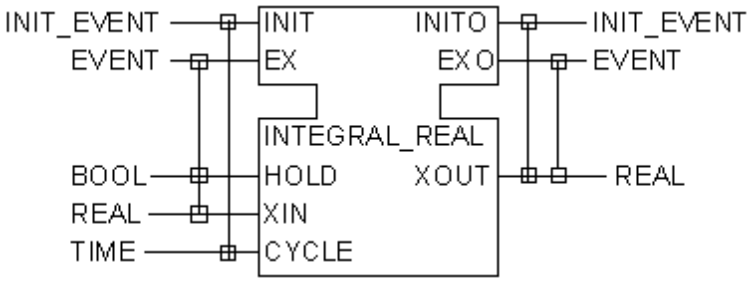
b) les noms et *déclarations de type* des *variables d'entrée* et des *supports adaptateurs* sont montrés sur le bord gauche de la partie inférieure du bloc;

c) les noms et *déclarations de type* des *variables de sortie* et des *fiches d'adaptation* sont montrés sur le bord droit de la partie inférieure du bloc;

d) l'*interface* du type du bloc fonctionnel avec les *événements* est déclarée dans la partie supérieure comme spécifié en 5.2.1.2;

e) les *algorithmes* associés au type du bloc fonctionnel sont déclarés comme spécifié en 5.2.1.3;

f) le contrôle de l'*exécution* des algorithmes associés est déclaré comme spécifié en 5.2.1.4.



NOTE 1 Voir Annexe F pour la déclaration textuelle de cet exemple.

NOTE 2 Cet exemple est uniquement illustratif. Les détails de la spécification ne sont pas normatifs.

Figure 10 – Déclaration du type de bloc fonctionnel de base

5.2.1.2 Déclaration de l'interface événement

Comme montré à la Figure 10, l'*interface événement* d'un *type de bloc fonctionnel de base* peut être déclarée textuellement conformément à la syntaxe donnée à l'Article B.2 ou graphiquement conformément aux règles suivantes:

a) Les *interfaces événements* sont placées dans une zone distincte en haut du bloc.

b) Les noms des *entrées d'événements* sont montrés à gauche de la partie supérieure du bloc.

c) Les noms des *sorties d'événements* sont montrés à droite de la partie supérieure du bloc.

d) Les *types* d'événement sont montrés à l'extérieur du bloc adjacent aux entrées ou sorties d'événements qui leur sont associées.

NOTE 1 Si aucun type d'événement n'est donné pour une entrée ou sortie d'événements, elle est considérée être du type par défaut EVENT.

NOTE 2 Une sortie d'événements de type EVENT peut être reliée à une entrée d'événements de n'importe quel type, et une entrée d'événements de type EVENT peut recevoir un événement de tout type.

NOTE 3 Une sortie d'événements de n'importe quel type autre que EVENT peut seulement être reliée à une entrée d'événements du même type ou du type EVENT.

NOTE 4 Un *type* d'événement est déclaré implicitement par son utilisation dans une déclaration d'événements.

Comme illustré à la Figure 10 et dans l'Annexe F, le qualificateur `WITH` ou un équivalent graphique doit être utilisé pour spécifier respectivement une association entre des *variables d'entrée* ou des *variables de sortie* et un *événement* sur l'*entrée d'événements* ou la *sortie d'événements* associée.

Chaque *variable d'entrée* et chaque *variable de sortie* apparaissent dans zéro, une ou plusieurs clauses `WITH` ou équivalents graphiques.

NOTE 5 Ces informations peuventêtre utilisées pour déterminer les *services* de communication requis pour *configurer* une *application* distribuée telle que décrite à l'Article 7.

NOTE 6 Une variable d'entrée qui n'apparaît dans aucune clause `WITH` n'est pas en mesure d'être reliée à une variable de sortie d'un autre bloc fonctionnel. Soit les valeurs de telles variables restent à leurs valeurs initiales déclarées, soit elles sont établies par des commandes de gestion telles que WRITE, comme décrit en 6.3.2.

NOTE 7 Une variable de sortie qui n'apparaît dans aucune clause `WITH` est en mesure d'être reliée à une variable d'entrée d'un autre bloc fonctionnel ou d'être "lue" par des commandes de gestion telles que READ, comme décrit en 6.3.2.

NOTE 8 Voir 4.5.3 pour une application du qualificateur WITH au modèle d'exécution d'un bloc fonctionnel de base.

5.2.1.3 Déclaration d'algorithme

Comme montré à l'Annexe F, des *algorithmes* associés à un *type de bloc fonctionnel de base* peuvent être inclus dans la déclaration du type de bloc fonctionnel conformément aux règles pour la déclaration de la spécification du type de bloc fonctionnel donnée dans l'Annexe B. D'autres moyens peuvent aussi être utilisés pour la spécification des identificateurs et des corps des algorithmes; cependant, la spécification d'un tel moyen ne relève pas du domaine d'application de la présente norme.

La déclaration d'un algorithme peut inclure la déclaration de variables temporaires qui:

- sont seulement visibles dans le corps de l'algorithme;
- sont initialisées à chaque invocation de l'algorithme;
- peuvent être utilisées et modifiées pendant l'exécution de l'algorithme; et
- n'ont pas de valeurs qui persistent entre des exécutions de l'algorithme.

5.2.1.4 Déclaration du contrôle d'exécution d'algorithme

L'ordonnancement des invocations d'algorithmes pour les *types de bloc fonctionnel de base* peut être déclaré dans la spécification type du bloc fonctionnel. Si les algorithmes d'un type de bloc fonctionnel de base sont définis comme spécifié en 5.2.1.3 (ou autrement identifiés), l'ordonnancement des invocations d'algorithme pour un tel bloc fonctionnel peut alors être sous la forme d'un *Graphe de contrôle d'exécution (ECC)* consistant en *états EC, transitions EC* et *actions EC.* Ces éléments sont représentés et interprétés comme suit:

a) l'ECC est inclus dans une section *contrôle d'exécution* de la déclaration type du bloc fonctionnel, considérée résider dans la partie supérieure du bloc;

b) l'ECC doit contenir exactement un seul *état initial EC*, représenté graphiquement comme une forme avec un contour double et un *identificateur* associé. L'état initial EC ne doit avoir aucune action EC associée;

c) l'ECC doit contenir un ou plusieurs *états EC*, représentés graphiquement comme des formes avec un contour simple, chacune avec un *identificateur* associé;

d) l'ECC peut utiliser, mais pas modifier, des variables déclarées dans la spécification type du bloc fonctionnel;

e) un *état EC* peut avoir zéro, une ou plusieurs *actions EC* associées. L'association des actions EC avec l'état EC peut être exprimée sous forme graphique ou textuelle;

f) l'*algorithme* (s'il y en a) associé à une action EC et l'*événement* (s'il y en a) devant être émis à la fin de l'algorithme doivent être exprimés sous forme graphique ou textuelle;

g) une *transition EC* est représentée sous forme graphique ou textuelle comme une liaison orientée allant d'un état EC à un autre (ou au même état);

h) chaque transition EC doit avoir une condition de transition associée, contenant une référence à un *événement*, à une *condition de garde*, ou les deux, exprimée dans la syntaxe définie pour le `ec_transition_condition` non-terminal en B.2.1.

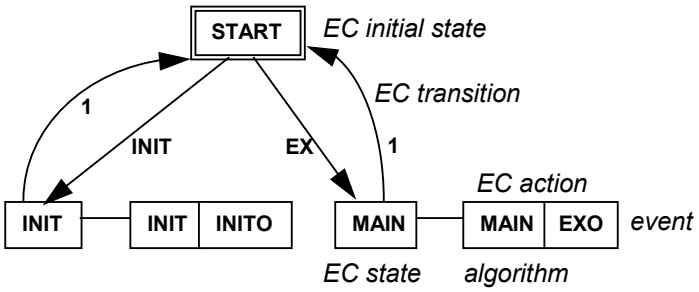
La Figure 11 illustre les éléments d'un ECC. Des déclarations textuelles similaires utilisant la syntaxe de l'Article B.2 sont données dans l'Annexe F.

NOTE 1 La notation `1` (une), illustrée à la Figure 11, est considérée être l'équivalent de `[TRUE]` représentant une condition de transition sans événement associé et avec une condition de garde qui est toujours `TRUE`.

NOTE 2 Dans ce domaine restreint, le même symbole (par exemple: `INIT`) peut être utilisé pour représenter un état EC et un nom d'algorithme, car le référent du symbole peut être déduit facilement à partir de son usage.

NOTE 3 Le texte en *italiques* ne fait pas partie de l'ECC.

NOTE 4 Une relation un à un d'événements avec des algorithmes, comme illustré par cette figure, se rencontre fréquemment, mais n'est pas le seul usage possible. Voir le Tableau A.1 pour des exemples d'autres usages: le bloc E_SPLIT montre une association de deux sorties d'événements avec un seul état, mais pas d'algorithme; E_MERGE montre une association d'un événement de sortie, mais sans algorithme, avec deux entrées d'événements; E_DEMUX montre un algorithme parmi plusieurs associés à un seul événement d'entrée; etc.



Légende

Anglais	Français
algorithm	algorithme
EC initial state	état initial de l'EC
EC state	état EC
EC action	action EC
EC transition	transition EC
event	événement

Figure 11 – Exemple d'ECC

5.2.2 Comportement des instances

5.2.2.1 Initialisation

L'initialisation d'une *instance* de bloc fonctionnel de base par une *ressource* doit être fonctionnellement l'équivalent de la procédure suivante:

a) La valeur de chaque variable d'*entrée*, de *sortie* et *variable interne* doit être initialisée à la valeur initiale correspondante donnée dans la spécification *type* du bloc fonctionnel. Si aucune valeur initiale n'est définie, la valeur de la variable doit être initialisée à la valeur initiale par défaut définie pour le type de données de la variable.

b) Toutes les éventuelles initialisations supplémentaires spécifiques à un algorithme doivent être effectuées; par exemple, toutes les *étapes initiales* des *Graphes séquentiels de fonction (SFC)* de la CEI 61131-3 doivent être activées et toutes les autres *étapes* doivent être désactivées.

c) L'*état initial EC* du *Graphe de contrôle d'exécution (ECC)* du bloc fonctionnel doit être activé, tous les autres *états EC* doivent être désactivés et le diagramme d'états d'opération de l'ECC défini en 5.2.2.2 doit être placé dans son état initial (`s0`).

NOTE Les conditions dans lesquelles une ressource doit accomplir une telle initialisation sont **dépendantes de la mise en œuvre.**

Le *type* de bloc fonctionnel peut aussi spécifier un *algorithme* d'initialisation devant être exécuté à l'apparition d'un événement approprié; par exemple, l'algorithme `INIT` montré à la Figure 11. Une *application* peut spécifier les conditions dans lesquelles cet algorithme doit être exécuté, par exemple en reliant la sortie d'une instance du type `E_RESTART` défini dans l'Annexe A à une entrée d'événements appropriée, par exemple l'entrée `INIT` illustrée à la Figure 10.

5.2.2.2 Invocation d'algorithme

L'*exécution* d'un *algorithme* associé à une *instance de bloc fonctionnel* est *invoquée* par une demande à la **fonction de programmation** de la *ressource* de programmer l'exécution des *opérations* de l'algorithme.

NOTE 1 Les opérations accomplies par un algorithme peuvent varier d'une exécution à l'autre en raison d'états internes modifiés du bloc fonctionnel, même si le bloc fonctionnel peut n'avoir qu'un seul algorithme et une seule entrée d'événements déclenchant son exécution.

L'invocation d'algorithme pour une instance d'un *type de bloc fonctionnel de base* doit être accomplie par l'équivalent fonctionnel de l'opération de son *Graphe de contrôle d'exécution (ECC)*. L'opération de l'ECC doit exposer le comportement défini par le diagramme d'états de la Figure 12 et du Tableau 1.

NOTE 2 Une conséquence de ce modèle est qu'une occurrence d'un événement à une entrée d'événements ne provoquera pas le franchissement d'une transition contenant l'événement, si la transition n'est pas associée à un état actuellement actif, c'est-à-dire, si l'événement n'est pas pertinent dans l'état donné. Cependant, l'*échantillonnage* des variables d'entrée associées à l'événement avec la construction WITH aura lieu dans tous les cas.

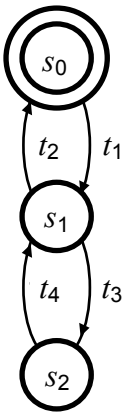


Figure 12 – Diagrammes d'états des opérations de l'ECC

Tableau 1 – États et transitions du diagramme d'états des opérations de l'ECC

Etat		Opérations
s_0		--
s_1		évaluer les transitions[c,e]
s_2		accomplir des actions[d,e]
Transition	**Condition**	**Opérations**
t_1	un événement d'entrée se produit[a]	Échantillonner des entrées[b,e]
t_2	aucune transition n'est franchie	
t_3	une transition est franchie	
t_4	actions achevées	

[a] La ressource doit assurer que pas plus d'un événement d'entrée ne se produit à un instant donné.

[b] Cette opération consiste en l'*échantillonnage* (ou son équivalent fonctionnel) des variables d'entrée associées à l'événement d'entrée courant par une déclaration WITH telle que décrite en 5.2.1.2.

[c] Cette opération consiste à évaluer les conditions de transition au niveau des transitions EC suivant l'état EC actif et à franchir la première transition EC (s'il y en a) pour laquelle une condition de garde (guard_condition) TRUE telle que définie en B.2.1 est rencontrée, conformément aux règles suivantes:

1 "Le franchissement de la transition EC" doit consister à désactiver son état EC précédent et à activer son état EC suivant.

2 L'ordre dans lequel les conditions de transition sont évaluées doit correspondre à l'ordre dans lequel les transitions sont déclarées comme défini en B.2.1 ou, de manière équivalente, dans la syntaxe XML définie dans la CEI 61499-2.

3 La guard_condition d'une condition de transition contenant seulement un `event_input_name` doit avoir la valeur par défaut TRUE.

4 Si l'état s_1 a été engagé via t_1, seules les conditions de transition avec l'événement d'entrée courant via son event_input_name tel que défini en B.2.1, ou les conditions de transition sans associations d'événement, doivent être évaluées.

5 Si l'état s_1 a été engagé via t_4, seules les conditions de transition sans associations d'événement doivent être évaluées.

[d] L'opération consiste, pour chaque *action EC* associée à l'*étape EC* active, à exécuter l'algorithme associé (s'il y en a) et à émettre un événement à la sortie d'événements associée (s'il y en a). L'ordre dans lequel les actions sont accomplies correspond à l'ordre dans lequel elles apparaissent graphiquement de haut en bas ou à l'ordre dans lequel elles sont déclarées en suivant la syntaxe textuelle définie en B.2.1 ou, de manière équivalente, dans la syntaxe XML définie dans la CEI 61499-2.

[e] Toutes les opérations accomplies à partir d'une occurrence de transition t_1 jusqu'à l'occurrence de t_2 doivent être mises en œuvre comme une *région critique* avec un verrou sur l'instance de bloc fonctionnel.

5.2.2.3 Exécution d'algorithme

L'*exécution* d'algorithme dans un bloc fonctionnel de base doit consister en l'exécution d'une séquence finie d'*opérations* déterminées par des règles **dépendant de la mise en œuvre** appropriées au langage dans lequel l'algorithme est écrit, à la *ressource* dans laquelle il s'exécute et au domaine auquel il s'applique. L'exécution d'algorithme se termine après l'exécution de la dernière opération de cette séquence.

Si un algorithme met en œuvre un diagramme d'états, des exécutions répétitives de l'algorithme sont nécessaires pour reconnaître ou accomplir des changements d'états. Normalement, il n'y a pas d'association entre ces changements d'états et l'achèvement de l'algorithme. De telles associations doivent être créées par les moyens de génération de sortie d'événements décrits en 5.2.2.2.

5.3 Blocs fonctionnels composés

5.3.1 Spécification de type

La déclaration des *types de bloc fonctionnel composé* doit suivre les règles données en 5.2.1 avec l'exception que des *entrées d'événements* et des *sorties d'événements* des *blocs fonctionnels composants* peuvent être interconnectées avec les entrées d'événements et les sorties d'événements du bloc fonctionnel composé pour représenter l'ordonnancement et la causalité des invocations de blocs fonctionnels. Les règles suivantes doivent s'appliquer à cet usage:

a) Chaque entrée d'événements du bloc fonctionnel composé est strictement reliée à une seule entrée d'événements d'exactement un seul bloc fonctionnel composant ou à strictement une seule sortie d'événements du bloc fonctionnel composé, avec l'exception que le raccourci graphique pour la division d'événements montrée à la Figure A.1 peut être employé.

b) Chaque entrée d'événements d'un bloc fonctionnel composant est reliée à une sortie d'événements au maximum de strictement un seul autre bloc fonctionnel composant ou à une entrée d'événements au maximum du bloc fonctionnel composé, avec l'exception que le raccourci graphique pour la fusion d'événements montrée à la Figure A.1 peut être employé.

c) Chaque sortie d'événements du bloc fonctionnel composant est reliée à une entrée d'événements au maximum de strictement un seul autre bloc fonctionnel composant ou à une sortie d'événements au maximum du bloc fonctionnel composé, avec l'exception que le raccourci graphique pour la division d'événements montrée à la Figure A.1 peut être employé.

d) Chaque sortie d'événements du bloc fonctionnel composé est reliée à strictement une seule sortie d'événements d'exactement un seul bloc fonctionnel composant ou à partir de strictement une seule entrée d'événements du bloc fonctionnel composé, avec l'exception que le raccourci graphique pour la fusion d'événements montrée à la Figure A.1 peut être employé.

e) L'utilisation du qualificateur `WITH` dans la déclaration d'entrées d'événements des types de bloc fonctionnel composé est exigée. L'utilisation du qualificateur `WITH` peut résulter en l'*échantillonnage* des entrées de données associées comme dans le cas des blocs fonctionnels de base ou d'interface de service, ou des outils logiciels peuvent fournir un moyen d'élimination des échantillonnages redondants dans la phase de mise en œuvre.

f) Les *instances* des *types sous-application* tels que définis en 5.4 ne doivent pas être utilisées dans la spécification d'un type de bloc fonctionnel composé.

Les *entrées de données* et *sorties de données* des *blocs fonctionnels composants* peuvent être interconnectées avec les entrées de données et les sorties de données du bloc fonctionnel composé pour représenter le flot de données au sein du bloc fonctionnel composé. Les règles suivantes doivent s'appliquer à cet usage:

- Chaque entrée de données du bloc fonctionnel composé peut être reliée à zéro, une ou plusieurs entrées de données de zéro, un ou plusieurs blocs fonctionnels composants et/ou à zéro, une ou plusieurs sorties de données du bloc fonctionnel composé.
- Chaque entrée de données d'un bloc fonctionnel composant peut être reliée à une sortie de données au maximum d'exactement un seul autre bloc fonctionnel composant ou à une entrée de données au maximum du bloc fonctionnel composé.
- Chaque sortie de données d'un bloc fonctionnel composant peut être reliée à zéro, une ou plusieurs entrées de données de zéro, un ou plusieurs blocs fonctionnels composants et/ou à zéro, une ou plusieurs sorties de données du bloc fonctionnel composé.
- Chaque sortie de données du bloc fonctionnel composé doit être reliée à strictement une seule sortie de données d'exactement un seul bloc fonctionnel composant ou à partir de strictement une seule entrée de données du bloc fonctionnel composé.

NOTE 1 Si un élément déclaré dans une construction VAR_INPUT...END_VAR ou VAR_OUTPUT...END_VAR est respectivement associé à un événement d'entrée ou de sortie par une construction WITH, cela se traduira par la création respective d'une variable d'entrée ou de sortie associée, comme dans le cas des types de bloc fonctionnel de base. Si un tel élément n'est pas associé à un événement d'entrée ou de sortie, le flot de données associé est transmis directement à destination ou en provenance des blocs fonctionnels composants par l'intermédiaire des connexions décrites ci-dessus.

NOTE 2 Les règles pour l'interconnexion des entrées et sorties d'événements et de variables des *prises mâles* et *prises femelles* dans le corps du bloc fonctionnel composé sont les mêmes que pour l'interconnexion des entrées et sorties des *blocs fonctionnels composants*. Voir 5.5 pour des exigences supplémentaires concernant l'*adaptateur d'interface*.

La Figure 13 illustre l'application de ces règles à l'exemple du bloc fonctionnel PI_REAL. La Figure 13a montre la représentation graphique des interfaces externes, tandis que 13(b) montre la construction graphique de son corps. La Figure 14 montre les interfaces et le contrôle d'exécution pour le type du bloc fonctionnel PID_CALC utilisé dans le corps de l'exemple PI_REAL.

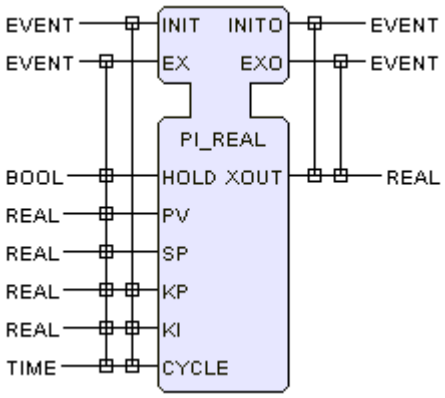


Figure 13a – Interface externe

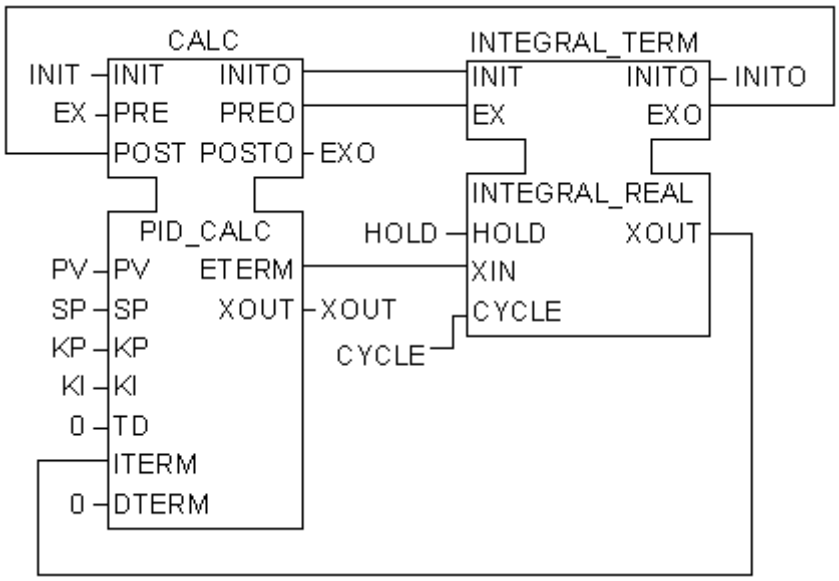


Figure 13b – Corps graphique

NOTE 1 Une déclaration textuelle complète de ce type de bloc fonctionnel est donnée dans l'Annexe F.

NOTE 2 Cet exemple est uniquement illustratif. Les détails de la spécification ne sont pas normatifs.

Figure 13 – Exemple de bloc fonctionnel composé PI_REAL

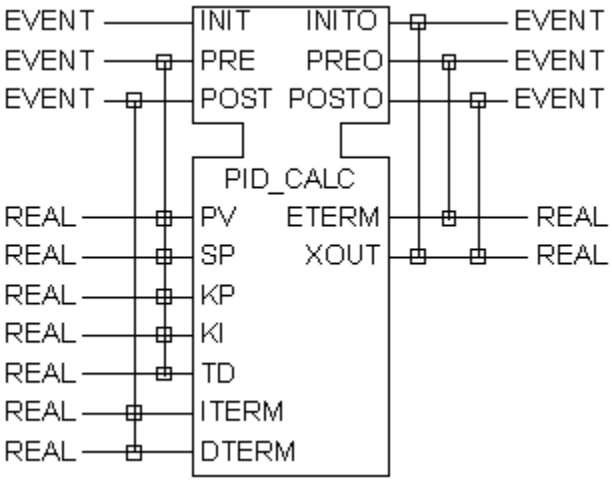


Figure 14a – Interface externe

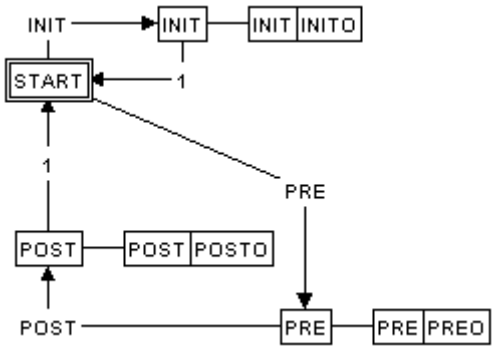


Figure 14b – Contrôle d'exécution

NOTE Cet exemple est uniquement illustratif. Les détails de la spécification ne sont pas normatifs.

Figure 14 – Exemple de bloc fonctionnel de base `PID_CALC`

5.3.2 Comportement d'instances

L'*invocation* et l'*exécution* des *blocs fonctionnels composants* dans des blocs fonctionnels composés doivent être accomplies comme suit:

a) Si une *entrée d'événements* du bloc fonctionnel composé est reliée à une *sortie d'événements* du bloc, l'apparition d'un *événement* sur l'entrée d'événements doit entraîner la génération d'un événement à la sortie d'événements associée.

b) Si une entrée d'événements du bloc fonctionnel composé est reliée à une entrée d'événements d'un bloc fonctionnel composant, l'apparition d'un événement sur l'entrée d'événements du bloc fonctionnel composé doit entraîner la programmation d'une invocation de la fonction de contrôle d'exécution du bloc fonctionnel composant, avec une apparition d'un événement sur l'entrée d'événements associée du bloc fonctionnel composant.

c) Si une sortie d'événements d'un bloc fonctionnel composant est reliée à une entrée d'événements d'un second bloc fonctionnel composant, l'apparition d'un événement sur la sortie d'événements du premier bloc doit entraîner la programmation d'une invocation de la fonction de contrôle d'exécution du second bloc, avec une apparition d'un événement sur l'entrée d'événements associée du second bloc.

d) Si une sortie d'événements d'un bloc fonctionnel composant est reliée à une sortie d'événements du bloc fonctionnel composé, l'apparition d'un événement sur la sortie d'événements du bloc composant doit entraîner la génération d'un événement sur la sortie d'événements associée du bloc fonctionnel composé.

L'initialisation d'instances de blocs fonctionnels composés doit équivaloir à l'initialisation de leurs blocs fonctionnels composants conformément aux dispositions en 5.2.2.1.

5.4 Sous-applications

5.4.1 Spécification de type

La déclaration des *types de sous-application* est semblable à la déclaration des *types de bloc fonctionnel composé* tels que définis en 5.3.1, à l'exception du fait que les mots-clés de délimitation doivent être `SUBAPPLICATION..END_SUBAPPLICATION`. Les règles suivantes doivent s'appliquer à cet usage:

a) Le qualificateur `WITH` n'est pas utilisé dans la déclaration des entrées d'événements et des sorties d'événements des *types sous-application*.

b) Chaque entrée d'événements de la sous-application doit être reliée strictement à une seule entrée d'événements d'exactement un seul bloc fonctionnel composant ou une seule sous-application constitutive ou bien à strictement une seule sortie d'événements de la sous-application.

c) Chaque entrée d'événements d'un bloc fonctionnel composant ou d'une sous-application constitutive est reliée à une sortie d'événements au maximum de strictement un seul autre bloc fonctionnel composant ou une seule autre sous-application constitutive ou bien au maximum à une entrée d'événements de la sous-application.

d) Chaque sortie d'événements d'un bloc fonctionnel composant ou d'une sous-application constitutive est reliée à une entrée d'événements au maximum de strictement un seul autre bloc fonctionnel composant ou une seule autre sous-application constitutive ou bien au maximum à une sortie d'événements de la sous-application.

e) Chaque sortie d'événements de la sous-application est reliée à partir de strictement une seule sortie d'événements d'exactement un seul bloc fonctionnel composant ou une seule sous-application constitutive ou bien à partir de strictement une seule entrée d'événements de la sous-application.

NOTE 1 Les blocs fonctionnels composants peuvent'inclure des instances des blocs de traitement d'événements définis dans l'Annexe A, par exemple pour "diviser" des événements en utilisant des instances du bloc `E_SPLIT`, pour "fusionner" des événements en utilisant le bloc `E_MERGE`, ou pour les deux cas, en utilisant le raccourci graphique équivalent.

Les *entrées de données* et *sorties de données* des *blocs fonctionnels composants* ou des *sous-applications constitutives* peuvent être interconnectées avec les entrées de données et les sorties de données de la sous-application pour représenter le flot de données au sein de la sous-application. Les règles suivantes doivent s'appliquer à cet usage:

- Chaque entrée de données de la sous-application peut être reliée à zéro, une ou plusieurs entrées de données de zéro, un ou plusieurs blocs fonctionnels composants ou de zéro, une ou plusieurs sous-applications constitutives et/ou à zéro, une ou plusieurs sorties de données de la sous-application.
- Chaque entrée de données d'un bloc fonctionnel composant ou d'une sous-application constitutive peut être reliée à une sortie de données au maximum d'exactement un seul autre bloc fonctionnel composant ou une seule autre sous-application constitutive ou bien à une entrée de données au maximum de la sous-application.
- Chaque sortie de données d'un bloc fonctionnel composant ou d'une sous-application constitutive peut être reliée à zéro, une ou plusieurs entrées de données de zéro, un ou plusieurs blocs fonctionnels composants ou de zéro, une ou plusieurs sous-applications constitutives et/ou à zéro, une ou plusieurs sorties de données de la sous-application.
- Chaque sortie de données de la sous-application doit être reliée à partir d'exactement une seule sortie de données d'exactement un seul bloc fonctionnel composant ou une seule

sous-application constitutive ou bien à partir d'exactement une seule entrée de données de la sous-application.

NOTE 2 Bien que les constructions `VAR_INPUT...END_VAR` et `VAR_OUTPUT...END_VAR` soient utilisées pour la déclaration des entrées et sorties de données des types de sous-application, cela ne se traduit pas par la création de variables d'entrée et de sortie; le flot de données est au contraire transmis aux blocs fonctionnels composants ou aux sous-applications constitutives par les connexions décrites ci-dessus.

NOTE 3 Les règles pour l'interconnexion des entrées et sorties d'événements et de variables des *prises mâles* et *prises femelles* dans le corps de la sous-application sont les mêmes que pour l'interconnexion des entrées et sorties des *blocs fonctionnels composants*. Voir 5.5 pour des exigences supplémentaires concernant l'*adaptateur d'interface*.

EXEMPLE La Figure 15 illustre l'application de ces règles à l'exemple de sous-application `PI_REAL_APPL`. La Figure 15 a) montre la représentation graphique de ses interfaces externes, tandis que la Figure 15 b) montre la construction graphique de son corps. Le corps de la sous-application `PI_REAL_APPL` utilise le type de bloc fonctionnel PID_CALC issu de l'exemple de bloc fonctionnel composé en 5.3.1, qui est montré à la Figure 14.

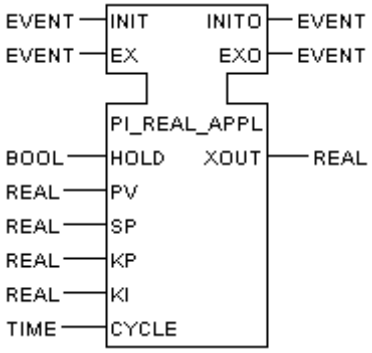


Figure 15a – Interface externe

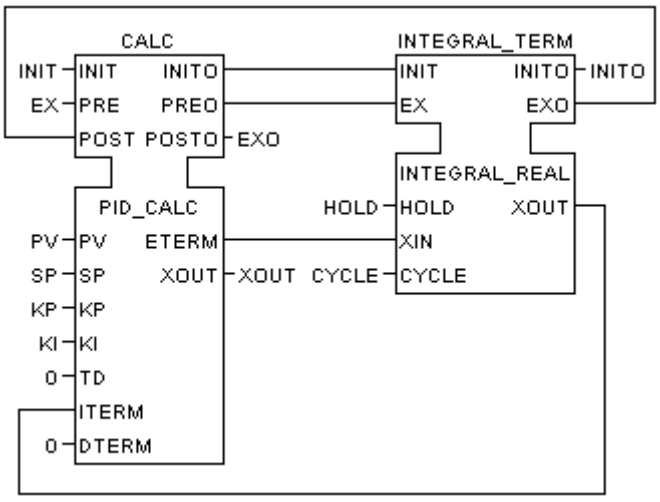


Figure 15b – Corps graphique

NOTE 1 Une déclaration textuelle complète de cette sous-application est donnée dans l'Annexe F.

NOTE 2 Cet exemple est uniquement illustratif. Les détails de la spécification ne sont pas normatifs.

Figure 15 – Exemple de sous-application `PI_REAL_APPL`

5.4.2 Comportement d'instances

L'*invocation* des *opérations* des *blocs fonctionnels composants* ou des *sous-applications constitutives* au sein des *sous-applications* doit être accomplie comme suit:

a) Si une *entrée d'événements* de la sous-application est reliée à une *sortie d'événements* du bloc, l'apparition d'un *événement* sur l'entrée d'événements doit entraîner la génération d'un événement sur la sortie d'événements associée.

b) Si une entrée d'événements de la sous-application est reliée à une entrée d'événements d'un bloc fonctionnel composant ou d'une sous-application constitutive, l'apparition d'un événement sur l'entrée d'événements de la sous-application doit entraîner la programmation d'une invocation de la fonction de contrôle d'exécution du bloc fonctionnel composant ou de la sous-application constitutive, avec une apparition d'un événement sur l'entrée d'événements associée du bloc fonctionnel composant ou de la sous-application constitutive.

c) Si une sortie d'événements d'un élément bloc fonctionnel composant ou d'une sous-application constitutive est reliée à une entrée d'événements d'un second élément bloc fonctionnel composant ou d'une seconde sous-application constitutive, l'apparition d'un événement sur la sortie d'événements du premier bloc doit entraîner la programmation d'une invocation de la fonction de contrôle d'exécution du second bloc, avec une apparition d'un événement sur l'entrée d'événements associée du second bloc.

d) Si une sortie d'événements d'un bloc fonctionnel composant ou d'une sous-application constitutive est reliée à une sortie d'événements de la sous-application, l'apparition d'un événement sur la sortie d'événements du bloc fonctionnel composant doit entraîner la génération d'un événement sur la sortie d'événements associée de la sous-application.

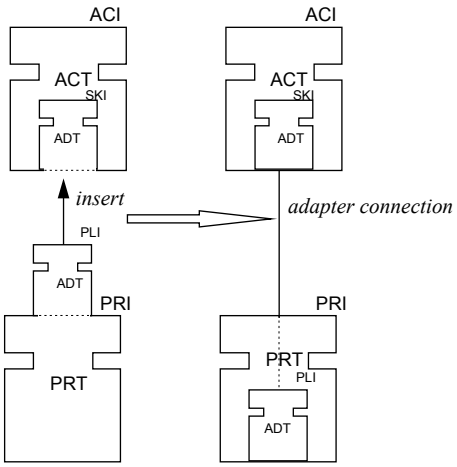
Étant donné que les sous-applications ne créent pas des variables de façon explicite, aucune procédure d'initialisation spécifique n'est applicable aux instances de sous-application.

5.5 Adaptateur d'interface

5.5.1 Principes généraux

Des *adaptateurs d'interface* peuvent être utilisés pour donner une représentation compacte d'un ensemble spécifié de flots d'événements et de données. Comme illustré à la Figure 16, un *type adaptateur d'interface* fournit un moyen de définir un sous-ensemble (la *fiche d'adaptation)* des *entrées* et des *sorties* d'un bloc fonctionnel du *fournisseur* qui peut être inséré en un sous-ensemble conjugué de *sorties* et d'*entrées* (le *support adaptateur)* d'un bloc fonctionnel *utilisateur*. Ainsi, l'adaptateur interface représente les chemins d'événements et de données par lesquels le fournisseur fournit un *service* vers l'utilisateur, ou vice versa, en fonction des profils d'interactions fournisseur/utilisateur, qui peuvent être représentées par des séquences de *primitives de service* telles que décrites en 6.1.3.

NOTE Un *type de bloc fonctionnel* donné pourrait fonctionner comme un *fournisseur,* un *utilisateur*, ou les deux, ou ni l'un ni l'autre, et de contenir plus d'une instance de *prise mâle* ou de *prise femelle* d'un ou plusieurs *types d'adaptateur d'interface.*



Légende

Anglais	Français
insert	insertion
adapter connection	connexion de l'adaptateur

Légende

PRT Provider type (*type de fournisseur*),

PRI Provider instance (*instance de fournisseur*),

ACT Acceptor type (*type d'utilisateur*),

ACI Acceptor instance (*instance de l'utilisateur*),

ADT Adapter type (*type d'adaptateur*),

PLI Plug instance (*instance de prise mâle*),

SKI Socket instance (*instance de prise femelle*)

NOTE Cette figure est uniquement illustrative. La représentation graphique n'est pas normative.

Figure 16 – Adaptateur d'interface – Modèle conceptuel

5.5.2 Spécification de type

Une *déclaration de type adaptateur d'interface* doit définir seulement le nom du *type d'interface* et ses interfaces d'*événements* et de *données* contenues. Ceux-ci sont définis graphiquement ou textuellement de la même manière que le *nom de type, les interfaces d'événement* et les *interfaces de données* d'un *type de bloc fonctionnel de base* tels que définis en 5.2.1.1 et 5.2.1.2, à l'exception du fait que les mots-clés pour le début et la fin de la déclaration textuelle de type doivent être `ADAPTER...END_ADAPTER`. La syntaxe textuelle pour la déclaration des adaptateurs d'interface est donnée à l'Article B.7.

EXEMPLE L'adaptateur d'interface illustré à la Figure 17 représente le fonctionnement du transfert d'un équipement de transfert "amont" représenté par un *fournisseur* de la *fiche* d'adaptation vers un équipement "aval" représenté par un *utilisateur* avec un *support* adaptateur correspondant. Comme illustré à la Figure 17(b), le fonctionnement typique de cette interaction consiste en la séquence suivante:

a) Un événement dans l'équipement amont (par exemple: l'arrivée d'une pièce à travailler à la position de déchargement) entraîne qu'un événement LD, typiquement interprété comme une commande de "chargement", est émis vers l'équipement aval. Associée à cet événement, une valeur de capteur WO indique si, oui ou non, une pièce à travailler est effectivement présente pour le transfert, plus une certaine propriété mesurée ou un certain ensemble de propriétés de la pièce à travailler, en l'occurrence sa couleur.

b) Un événement ultérieur dans l'équipement aval (par exemple: l'achèvement du montage de la charge) entraîne qu'un événement UNLD, typiquement interprété comme une commande de libération de la pièce à travailler, est envoyé vers l'équipement amont.

c) Ensuite, un événement CNF, typiquement interprété comme une confirmation de la libération de la pièce à travailler, est transmis de l'équipement amont vers l'équipement aval afin de parachever l'opération. À ce stade, la sortie WO est typiquement FALSE et la valeur de la sortie WKPC n'a pas de signification.

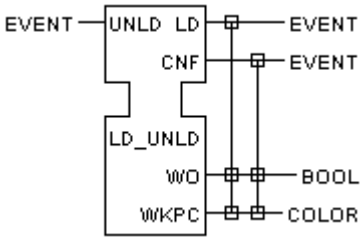


Figure 17a – Interface

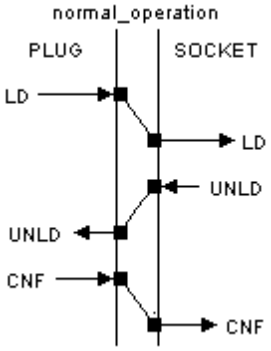


Figure 17b – Séquence de service

NOTE 1 Une déclaration textuelle complète de ce type d'adaptateur est donnée dans l'Annexe F.

NOTE 2 Cet exemple est uniquement illustratif. Les détails de la spécification ne sont pas normatifs.

NOTE 3 Voir 6.1.2 pour une explication des séquences de service.

Figure 17 – Déclaration du type d'adaptateur – Exemple graphique

5.5.3 Usage

L'usage des *types* et des *instances adaptateur d'interface* doit être conforme aux règles suivantes:

a) Les instances adaptateur d'interface devant être utilisées comme *prises mâles* dans des instances d'un *type de bloc fonctionnel* doivent être déclarées dans sa *déclaration de type* dans un bloc `PLUGS...END_PLUGS`, déclarant le *nom d'instance* et le *type adaptateur d'interface* de chaque prise mâle. Dans la représentation graphique des *types de bloc fonctionnel* et des *instances*, les fiches sont montrées comme *variables de sortie* avec une indication textuelle ou graphique spécialisée pour signifier qu'elles ne sont pas des variables de sortie ordinaires.

b) Les instances adaptateur d'interface devant être utilisées comme *prises femelles* dans des instances d'un *type bloc fonctionnel* doivent être déclarées dans sa *déclaration de type* dans un bloc `SOCKETS...END_SOCKETS`, déclarant le *nom d'instance* et le *type d'interface d'adaptateur* de chaque prise femelle. Dans la représentation graphique des *types bloc fonctionnel* et des *instances*, les prises femelles sont montrées comme *variables d'entrée* avec une indication textuelle ou graphique spécialisée pour signifier qu'elles ne sont pas des variables d'entrée ordinaires.

c) Les *entrées* et les *sorties* d'une *prise mâle* doivent être utilisées au sein de sa *déclaration de type bloc fonctionnel* de la même manière que les entrées et les sorties du bloc fonctionnel.

d) Les *entrées* et les *sorties* d'une *prise femelle* doivent respectivement être utilisées au sein de sa *déclaration de type bloc fonctionnel* de la même manière que les *sorties* et les *entrées* du bloc fonctionnel.

e) L'insertion de *prises mâles* dans des *prises femelles* doit être spécifiée dans un bloc `ADAPTER_CONNECTIONS ... END_CONNECTIONS` dans la *déclaration* de *l'application, de la sous-application, du type de ressource, de l'instance de ressource* ou du *type bloc fonctionnel composé* contenant les instances respectives de *fournisseur* et d'*utilisateur*.

f) Dans le corps d'un *type bloc fonctionnel composé* ou d'une *sous-application*, une *prise femelle* est représentée comme un *bloc fonctionnel* avec les mêmes entrées et sorties que le *type adaptateur d'interface* correspondant. De manière similaire, dans ce cas une *prise mâle* est représentée comme un bloc fonctionnel avec les entrées et les sorties du type d'interface d'adaptateur correspondant inversées.

g) L'insertion des prises mâles dans des prises femelles doit être soumise aux contraintes suivantes:

1) une prise mâle peut seulement être insérée dans une prise femelle du même *type adaptateur d'interface*;
2) une prise mâle peut seulement être insérée dans zéro ou une seule prise femelle à la fois;
3) une prise femelle peut seulement accepter zéro ou une seule prise mâle à la fois;
4) une prise mâle peut seulement être insérée dans une fiche femelle si toutes les deux sont dans le même *bloc fonctionnel composé*, dans la même *ressource*, dans la même *application* ou dans la même *sous-application*.

Une connexion allant d'une prise mâle à une prise femelle peut être montrée dans une *application* ou *sous-application* même si les instances de bloc fonctionnel correspondantes peuvent être *mises en correspondance* avec des *ressources* distinctes. Dans ce cas, des moyens appropriés, tels que des blocs fonctionnels interfaces de services de communication décrits en 6.2, doivent être utilisés pour mettre en œuvre le transfert correspondant d'événements et de données entre les ressources.

Les *blocs fonctionnels de gestion* tels que décrits en 6.3 peuvent fournir des moyens pour la création, la suppression et l'interrogation dynamiques des connexions d'adaptateur.

EXEMPLE 1 Une instance du type XBAR_MVCA illustré à la Figure 18 agit tant comme fournisseur d'une interface de prise mâle (LDU_PLG) que comme un utilisateur avec une interface de prise femelle (LDU_SKT). Ce faisant, elle sert à abstraire et encapsuler les interactions d'une instance du type XBAR_MVC avec des unités fonctionnelles "amont" et "aval".

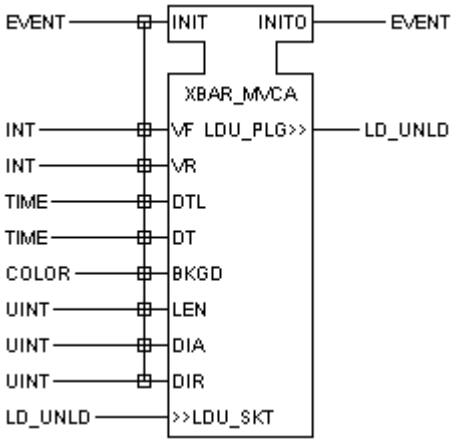


Figure 18a – Interface

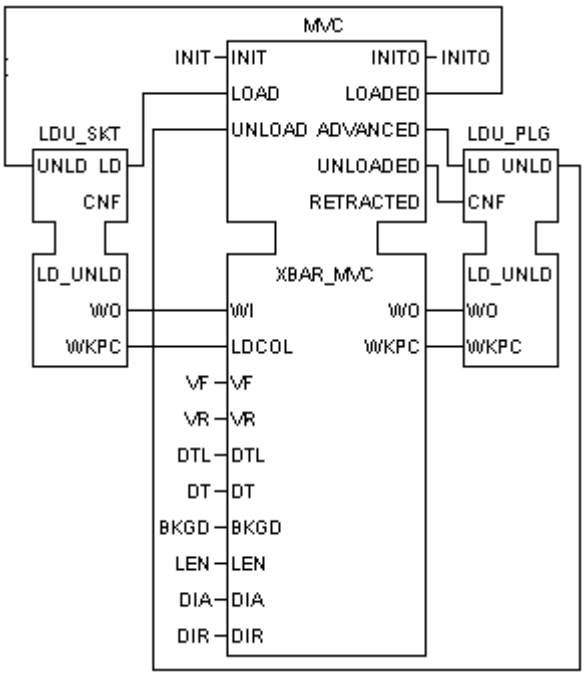


Figure 18b – Corps

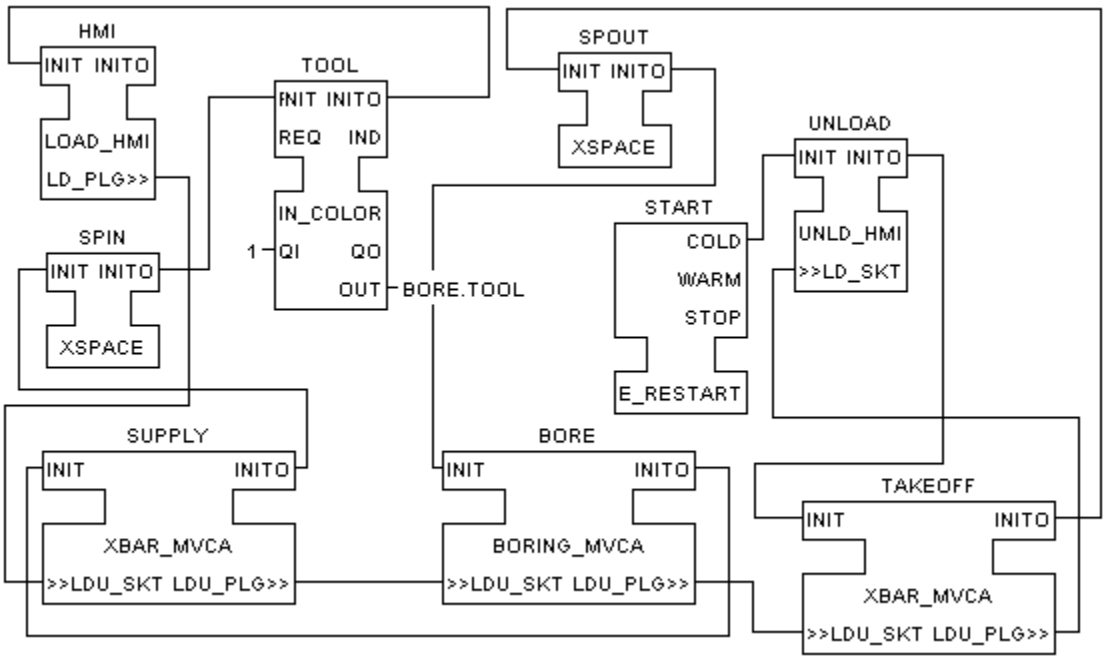
NOTE 1 Une déclaration textuelle complète de cet exemple est donnée dans l'Annexe F.

NOTE 2 Cet exemple est uniquement illustratif. Les détails de la spécification ne sont pas normatifs.

NOTE 3 Bien que cet exemple présente seulement un type composé, les *types* de bloc fonctionnel *fournisseur* et *utilisateur* sont susceptibles d'être soit *de base*, soit *composés*.

Figure 18 – Illustration des déclarations des types de bloc fonctionnel fournisseur et utilisateur

EXEMPLE 2
La Figure 19 illustre une configuration de ressource contenant deux instances du type XBAR_MVCA illustré à la Figure 18. L'instance SUPPLY agit comme un utilisateur ("unité aval") pour le bloc HMI et comme un fournisseur ("unité amont") pour le bloc BORE, alors que l'instance TAKEOFF remplit respectivement des rôles correspondants pour les blocs BORE et UNLOAD.



NOTE 1 Cet exemple est uniquement illustratif. Les détails de la spécification ne sont pas normatifs.

NOTE 2 Par souci de clarté, les connexions de *paramètre* sont omises dans ce diagramme.

NOTE 3 Les déclarations de types pour blocs autres que le type XBAR_MVCA ne sont pas données dans l'Annexe F.

Figure 19 – Illustration des connexions d'adaptateur

5.6 Traitement des exceptions et des défauts

Des moyens supplémentaires pour la prévention, la reconnaissance et le traitement des *exceptions* et des *défauts* peuvent être fournis par des *ressources*. De telles capacités peuvent être modélisées sous la forme de *blocs fonctionnels interface de service*. La définition de types spécifiques de bloc fonctionnel pour la prévention, la reconnaissance et le traitement des exceptions et des défauts ne relève pas du domaine d'application de la présente norme. Cependant, les sorties `INIT`, `CNF` et `IND` des blocs fonctionnels interface de service, et les valeurs associées de `STATUS`, peuvent être utilisées pour indiquer l'occurrence et le type des exceptions et des défauts, comme noté en 6.1.3.

6 Blocs fonctionnels interface de service

6.1 Principes généraux

6.1.1 Généralités

Un *bloc fonctionnel interface de service* fournit un ou plusieurs *services* à une application, en se basant sur un *mapping* de *primitives de service* avec les *entrées d'événements, sorties d'événements, entrées de données* et *sorties de données* du bloc fonctionnel.

Les interfaces externes des *types de bloc fonctionnel interface de service* ont la même apparence générale que les *types de bloc fonctionnel de base*. Cependant, certaines entrées

et sorties de type bloc fonctionnel interface de service ont une sémantique spécialisée et le comportement d'*instances* de ces types est défini par une notation graphique spécialisée pour les séquences de *primitives de service.*

NOTE La spécification des opérations internes des blocs fonctionnels interface de service ne relève pas du domaine d'application de la présente norme.

6.1.2 Spécification de type

La *déclaration* des *types bloc fonctionnel interface de service* peut utiliser les *entrées d'événements, sorties d'événements, entrées de données* et *sorties de données* normalisées qui sont énumérées dans le Tableau 2, selon ce qui est approprié au service particulier fourni. Lorsque celles-ci sont utilisées, leur sémantique doit être telle que définie en 6.1.2. Le nom du *type* de bloc fonctionnel doit indiquer le service fourni.

EXEMPLE Les Figures 20(a) et (b) montrent des exemples de blocs fonctionnels interface de service dans lesquels l'interaction principale est respectivement déclenchée par l'application et par la ressource.

NOTE 1 Les services peuvent fournir des interactions déclenchées tant par la ressource que par l'application dans le même bloc fonctionnel interface de service.

NOTE 2 Les types interface de service peuvent également utiliser des entrées et des sorties, y compris les *prises mâles* et les *prises femelles*, avec des noms différents de ceux donnés ici; dans un tel cas, leur usage doit être défini en termes de séquences appropriées de primitives de service.

Tableau 2 – Entrées et sorties normalisées pour les blocs fonctionnels interface de service (1 de 2)

Entrées d'événements
INIT Cette entrée d'événements doit être *mappée avec* une *primitive* "*request*" qui demande une initialisation du service fourni par l'instance de bloc fonctionnel, par exemple, l'initialisation locale d'une *connexion de la communication* ou un module d'interface de processus.
REQ Cette entrée d'événements doit être mappée avec une *primitive* "*request*" du service fourni par l'instance bloc fonctionnel.
RSP Cette entrée d'événements doit être mappée avec une *primitive* "*response*" du service fourni par l'instance bloc fonctionnel.
Sorties d'événements
INITO Cette sortie d'événements doit être mappée avec une *primitive* "*confirm*" qui indique la fin d'une procédure d'initialisation du service.
CNF Cette sortie d'événements doit être mappée avec une *primitive* "*confirm*" du service fourni par l'instance bloc fonctionnel.
IND Cette sortie d'événements doit être mappée avec une *primitive* "*indication*" du service fourni par l'instance bloc fonctionnel.

Tableau 2 (2 de 2)

Entrées de données
QI: **BOOL** Cette entrée représente un qualificateur sur les *primitives de service* mises en correspondance avec les *entrées d'événements*. Par exemple, si cette entrée est TRUE à l'apparition d'un événement INIT, l'initialisation du service est demandée; si elle est FALSE, l'arrêt du service est demandé.
PARAMS: **ANY** Cette entrée contient un ou plusieurs *paramètres* associés au service, typiquement comme éléments d'une *instance* d'un *type de données structuré*. Lorsque cette entrée est présente, la spécification du *type* de *bloc fonctionnel* doit définir son *type de données* et sa/ses valeur(s) initiale(s) par défaut. Une spécification du type bloc fonctionnel d'interface service peut remplacer cette entrée par une ou plusieurs entrées de paramètres de service.
SD_1, ..., SD_m: ANY Ces entrées contiennent les données associées aux *primitives* "*request*" et "*response*". La spécification du *type* de *bloc fonctionnel* doit définir les *types de données* et les valeurs par défaut de ces entrées et doit définir leurs associations avec des entrées d'événements dans un diagramme de séquences d'événement tel que déclaré en 6.1.3. La spécification du type de bloc fonctionnel peut définir d'autres noms pour ces entrées.
Sorties de données
QO: BOOL Cette variable représente un qualificateur sur les *primitives de service* mises en correspondance avec les *sorties d'événements*. Par exemple, une valeur TRUE de cette sortie à l'apparition d'un événement INITO indique une initialisation réussie du service; une valeur FALSE indique une initialisation infructueuse.
STATUS: ANY Cette sortie doit être d'un *type de données* approprié pour exprimer le statut du service à l'apparition d'une sortie d'événements. Une spécification de service peut indiquer que la valeur de cette sortie n'est pas pertinente pour certaines situations, par exemple, pour INITO+, IND+ et CNF+ comme décrit en 6.1.3.
RD_1, ..., RD_n: ANY Ces sorties contiennent les données associées aux primitives "*confirm*" et "*indication*". La *spécification du type* du bloc fonctionnel doit définir les *types de données* et les valeurs initiales de ces sorties et doit définir leurs associations avec des sorties d'événements dans un diagramme de séquences d'événement tel que décrit en 6.1.3. La spécification du type de bloc fonctionnel peut définir d'autres noms pour ces sorties.

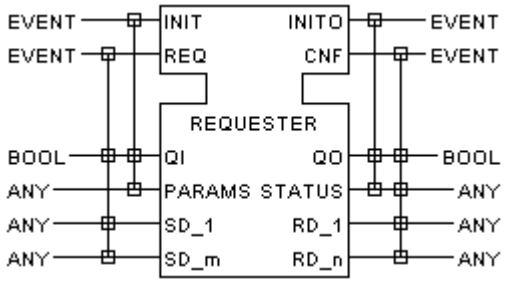


Figure 20a – Interactions déclenchées par une application

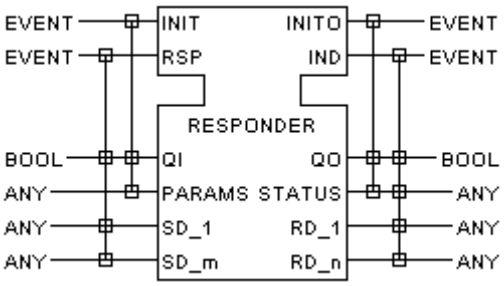


Figure 20b – Interactions déclenchées par une ressource

NOTE 1 `REQUESTER` et `RESPONDER` représentent les services particuliers fournis par des instances des types de bloc fonctionnel.

NOTE 2 Les *types de données* des entrées `SD_1,...,SD_n` et des sorties `RD_1,...RD_m` seront typiquement fixés comme un certain type de données non générique, par exemple `INT` ou `WORD`, dans des mises en œuvre concrètes des types de bloc fonctionnel génériques illustrés ici.

NOTE 3 Voir l'Annexe F pour une déclaration textuelle complète du type de bloc fonctionnel `REQUESTER`.

Figure 20 – Exemples de blocs fonctionnels interface de service

6.1.3 Comportement des instances

Le comportement des *instances* des *blocs fonctionnels interface de service* doit être défini dans la spécification correspondante du *type* de *bloc fonctionnel*, qui peut utiliser des *diagrammes de séquences de service* assujettis aux règles suivantes:

a) La sémantique suivante doit s'appliquer:
   1) Le temps augmente dans le sens descendant.
   2) Les événements qui sont liés séquentiellement sont reliés ensemble au travers ou au sein des ressources.
   3) S'il n'y a pas de relation spécifique entre des événements, en ce qu'il est impossible de prévoir lequel se produira le premier, mais que tous les deux doivent se produire dans un délai fini, un tilde (~) ou une notation textuelle similaire est utilisé(e).

b) Dans le cas où le service est représenté par un seul bloc fonctionnel interface de service, le diagramme doit être partagé par une seule ligne verticale en deux champs tels qu'illustrés en Figure 21:
   1) Dans le cas où le service est fourni principalement par une interaction déclenchée par une application, l'*application* doit être dans le champ de gauche et la *ressource* dans le champ de droite (voir Figure 21 a)).
   2) Dans le cas où le service est fourni principalement par une interaction déclenchée par une ressource, la *ressource* doit être dans le champ de gauche et l'*application* dans le champ de droite (voir Figure 21 b)).

c) Dans le cas où le service est représenté par deux ou plusieurs blocs fonctionnels interface de service, la notation illustrée en E.2.2 et en E.2.3 peut être utilisée.

d) Les *primitives de service* doivent être indiquées par des flèches horizontales. Le nom de l'*événement* représentant la primitive de service doit être écrit au voisinage de la flèche et des moyens doivent être fournis pour déterminer les noms des *variables* d'entrée et/ou de sortie représentant les *données* associées à la primitive.

e) Lorsqu'une entrée `QI` est présente dans la définition du type du bloc fonctionnel, le suffixe "+" doit être utilisé conjointement à un nom d'*entrée d'événements* pour indiquer que la

valeur de l'entrée QI est TRUE à l'apparition de l'événement associé, et le suffixe "-" doit être utilisé pour indiquer qu'elle est FALSE.

f) Lorsqu'une sortie QO est présente dans la définition du type du bloc fonctionnel, le suffixe "+" doit être utilisé conjointement à un nom de *sortie d'événements* pour indiquer que la valeur de la sortie QO est TRUE à l'apparition de l'événement associé, et le suffixe "-" doit être utilisé pour indiquer qu'elle est FALSE.

g) La sémantique normalisée des événements affirmés (+) et niés (-) doit être telle que spécifiée dans le Tableau 3.

La Figure 21 illustre des séquences normales de lancement du service, de transfert de données et d'arrêt du service. Des spécifications du *type de bloc fonctionnel interface de service* peuvent utiliser des diagrammes similaires pour spécifier toutes les séquences pertinentes des primitives de service et leurs données associées dans des conditions normales et anormales.

NOTE Des diagrammes de séquences sont également susceptibles d'être utilisés pour documenter les comportements observables de l'extérieur des types du bloc fonctionnel de base et composé.

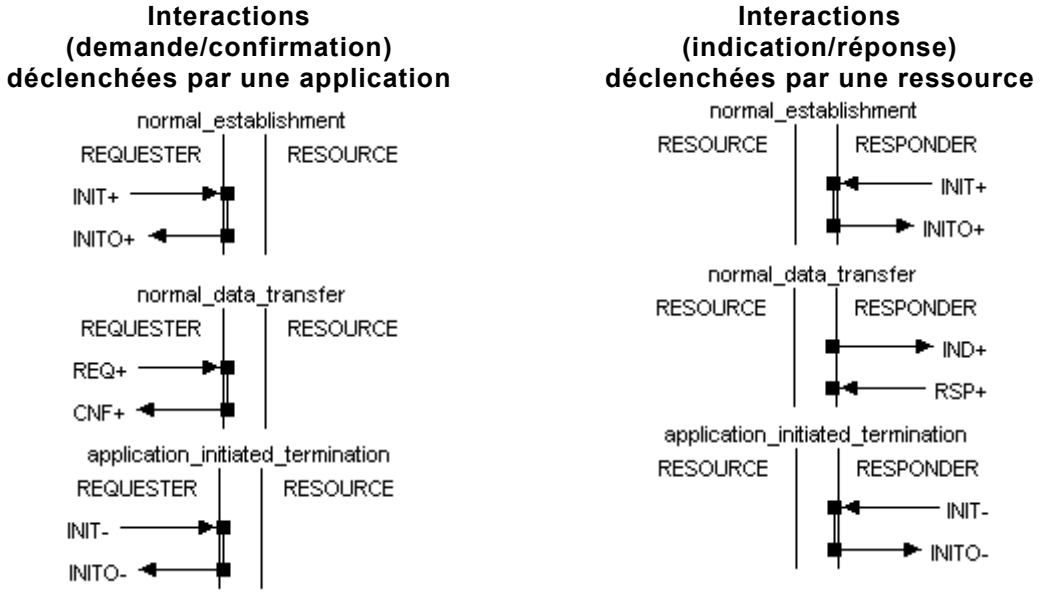


Figure 21 – Exemples de diagrammes des séquences de service

Tableau 3 – Sémantique des primitives de service

Primitive	Sémantique
INIT+	Demande d'établissement d'un service
INIT-	Demande d'arrêt d'un service
INITO+	Indication d'établissement d'un service normal
INITO-	Rejet de demande d'établissement d'un service ou indication d'arrêt d'un service
REQ+	Demande normale pour un service
REQ-	Demande désactivée pour un service
CNF+	Confirmation normale d'un service
CNF-	Indication d'état anormal d'un service
IND+	Indication d'arrivée normale d'un service
IND-	Indication d'état anormal d'un service
RSP+	Réponse normale par une application
RSP-	Réponse anormale par une application

6.2 Blocs fonctionnels de communication

6.2.1 Spécification de type

Les blocs fonctionnels de communication fournissent des *interfaces* entre des *applications* et les fonctions de "mapping de la communication" des *ressources* telles que définies en 4.3; aussi sont-ils des *blocs fonctionnels interface de service* tels que décrits en 6.1.

Comme d'autres blocs fonctionnels interface de service, un bloc fonctionnel de communication peut être de type *basic* (c'est-à-dire: de base) ou *composite* (c'est-à-dire: composé), tant que son opération peut être représentée par un *mapping* de *primitives de service* avec les *entrées d'événements, les sorties d'événements, les entrées de données* et les *sorties de données* du bloc fonctionnel.

Le paragraphe 6.2.1 donne des règles pour la *déclaration* des *types bloc fonctionnel de communication.* Le paragraphe 6.2.2 donne des règles pour le comportement des *instances* de ces types de bloc fonctionnel. L'Article E.2 définit des types génériques de bloc fonctionnel de communication pour les transactions *unidirectionnelles* et *bidirectionnelles* et donne des règles de personnalisation dépendant de la mise en œuvre de ces types.

La *déclaration* des *types de bloc fonctionnel communication* doit utiliser le moyen défini en 6.1 pour la déclaration des *types de bloc fonctionnel interface de service*, avec la sémantique spécialisée montrée dans le Tableau 4 pour les variables d'*entrée* et de *sortie*.

Tableau 4 – Sémantique des variables pour les blocs fonctionnels de communication

Variable	Sémantique
PARAMS	Cette entrée fournit des *paramètres* pour la *connexion de communication* associée à l'*instance de bloc fonctionnel de communication*. Elle doit inclure un moyen d'identifier le protocole de communication et la connexion de communication et peut inclure d'autres paramètres de connexion de communication tels que les contraintes de temporisation, etc.
SD_1,..., SD_m	Ces entrées représentent des *données* devant être transférées sur la *connexion de communication* spécifiée par l'entrée PARAMS à l'apparition d'une *primitive* REQ+ ou RSP+, selon ce qui est approprié. [a]
STATUS	Cette sortie représente le statut de la *connexion de communication*, par exemple: - Fin normale du lancement, de l'arrêt, ou de transfert de données - Causes de lancement anormal, d'arrêt anormal ou de transfert anormal de données
RD_1,..., RD_n	Ces sorties représentent des *données* reçues sur la *connexion de communication* spécifiée par l'entrée PARAMS à l'apparition d'une primitive IND+ ou CNF+ selon ce qui est approprié. [a]
NOTE Les déclarations de type de bloc fonctionnel de communication peuvent définir des contraintes entre les sorties RD_1,...,RD_n et les entrées SD_1,..., SD_m d'instances correspondantes de bloc fonctionnel. Par exemple, le nombre et les types des sorties RD pourraient'être contraints pour concorder avec le nombre et les types des entrées SD correspondantes.	
[a] *Les déclarations de type de bloc fonctionnel de communication* définissent le nombre et le type des entrées SD_1,..., SD_m et des sorties RD_1,...,RD_n et peuvent leur attribuer d'autres noms.	

6.2.2 Comportement des instances

Comme illustré à l'Article E.2, le comportement des *instances* des *types de bloc fonctionnel de communication* doit être défini dans la *déclaration* correspondante de type bloc fonctionnel de communication, en utilisant le moyen spécifié pour les *blocs fonctionnels interface de service* en 6.1 avec la sémantique spécialisée des primitives de service donnée dans le Tableau 5. Une telle spécification doit inclure des séquences de *primitives de service* pour:

- l'établissement et la libération normaux et anormaux des *connexions de communication;*
- le transfert normal et anormal de données.

Tableau 5 – Sémantique des primitives de service pour les blocs fonctionnels de communication

Primitive	Sémantique
INIT+	Demande pour établissement d'une connexion de communication
INIT-	Demande pour libération d'une connexion de communication
INITO+	Indication d'établissement d'une connexion de communication
INITO-	Rejet d'une demande d'établissement de connexion de communication ou indication d'une libération d'une connexion de communication
REQ+	Demande normale pour transfert de données
REQ-	Demande désactivée pour transfert de données
CNF+	Confirmation normale de transfert de données
CNF-	Indication de transfert anormal de données
IND+	Indication d'arrivée normale de données
IND-	Indication d'arrivée anormale de données
RSP+	Réponse normale par l'application à l'arrivée de données
RSP-	Réponse anormale par l'application à l'arrivée de données

6.3 Blocs fonctionnels de gestion

6.3.1 Exigences

L'extension des exigences fonctionnelles pour la "gestion d'application" en 8.3.2 de l'ISO/CEI 7498-1:1994 au modèle d'application distribuée de la présente norme indique qu'il convient que les *services* pour la gestion des ressources et d'applications dans les systèmes IPMCS puissent accomplir les *fonctions* suivantes*:*

a) Dans une *ressource,* créer, initialiser, démarrer, arrêter, supprimer, vérifier l'existence et les *attributs* des changements de disponibilité et de statut des éléments ci-après et en donner notification:

   1) types de données

   2) types et instances de bloc fonctionnel

   3) connexions entre des instances de bloc fonctionnel

b) Dans un *équipement,* créer, initialiser, démarrer, arrêter, supprimer, vérifier l'existence et les *attributs* des changements de disponibilité et de statut des *ressources* et en donner notification.

NOTE 1 Les dispositions de la présente norme ne visent pas à satisfaire aux exigences pour la *gestion de système* traitées dans l'ISO/CEI 7498-4 et l'ISO/CEI 10040, avec l'exception que de telles exigences sont traitées par les fonctions énumérées ci-dessus.

NOTE 2 La présente norme traite seulement de l'élément a) ci-dessus, c'est-à-dire la gestion des ressources des *applications*. Un cadre de travail pour la gestion des équipements est décrit dans la CEI 61499-2.

NOTE 3 Les associations entre les *ressources*, les *applications* et les *instances de bloc fonctionnel* sont définies dans les *configurations de système* telles que décrites en 7.3.

NOTE 4 Le démarrage et l'arrêt d'une *application* distribuée sont accomplis par un *outil logiciel* approprié.

6.3.2 Spécification de type

La Figure 22 illustre la forme générale des *types de bloc fonctionnel de gestion* dont les *instances* satisfont aux exigences de gestion d'application définies ci-dessus.

NOTE 1 Dans des mises en œuvre particulières, le nom de type (`MANAGER` dans cet exemple) pourrait représenter le type de la ressource gérée.

NOTE 2 Pour ces types de bloc fonctionnel, les entrées spécifiques `CMD` et `OBJECT` et la sortie spécifique `RESULT` remplacent les entrées génériques `SD_1` et `SD_2` et la sortie générique `RD_1` décrites en 6.1.

NOTE 3 Les entrées `INIT` et `PARAMS` et la sortie `INITO` pourraientêtre présentes ou non dans une mise en œuvre particulière.

NOTE 4 Lorsqu'elles sont présentes, le type et les valeurs de l'entrée `PARAMS` sont des paramètres **dépendants de la mise en œuvre** du type de ressource.

NOTE 5 Une spécification textuelle complète de ce type de bloc fonctionnel, y compris toutes les séquences de service, est donnée dans l'Annexe F.

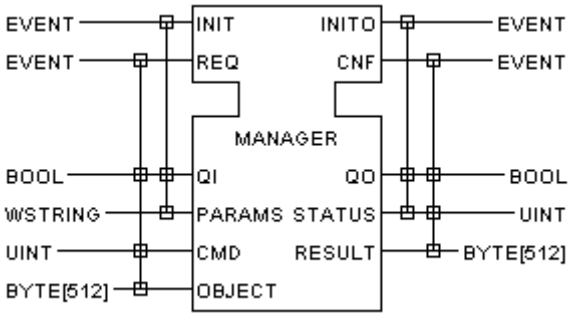
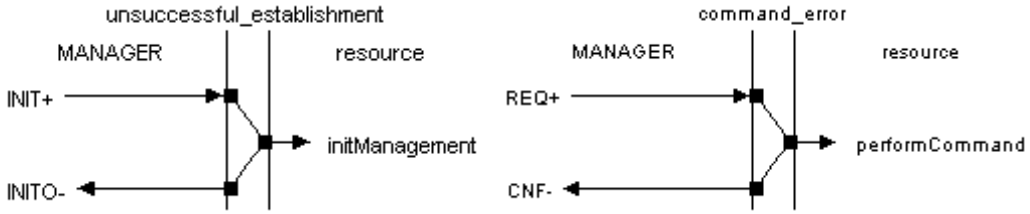


Figure 22 – Type générique d'un bloc fonctionnel de gestion

Le comportement des instances et la sémantique d'entrée/sortie des types de bloc fonctionnel de gestion doivent suivre les règles données en 6.1 pour les *types de bloc fonctionnel interface de service* avec des interactions lancées par une application, avec les comportements complémentaires montrés à la Figure 23 pour un lancement et des demandes de service infructueuses.



NOTE Une spécification textuelle complète de ce type de bloc fonctionnel, y compris toutes les séquences de service, est donnée dans l'Annexe F.

Figure 23 – Séquences des primitives de service pour un service infructueux

L'*opération* de gestion devant être *exécutée* doit être exprimée par la valeur de l'entrée `CMD` d'un bloc fonctionnel de gestion conformément à la sémantique définie dans le Tableau 6.

Tableau 6 – Valeurs et sémantique de l'entrée `CMD`

Valeur	Commande	Sémantique
0	`CREATE`	Créer un objet spécifié
1	`DELETE`	Supprimer un objet spécifié
2	`START`	Démarrer un objet spécifié
3	`STOP`	Arrêter un objet spécifié
4	`READ`	Lire des données paramètre
5	`WRITE`	Écrire des données paramètre
6	`KILL`	Rendre inexécutable un objet spécifié
7	`QUERY`	Demander des informations relatives à un objet spécifié
8	`RESET`	Réinitialiser un objet spécifié

Les valeurs et la sémantique correspondante de la sortie `STATUS` d'un bloc fonctionnel de gestion doivent être telles que décrites dans le Tableau 7 pour exprimer le résultat de l'exécution de la commande spécifiée.

Tableau 7 – Valeurs et sémantique de la sortie STATUS

Valeur	Statut	Sémantique
0	RDY	Absence d'erreurs
1	BAD_PARAMS	Valeur non valide de l'entrée PARAMS
2	LOCAL_TERMINATION	Arrêt déclenché par l'application
3	SYSTEM_TERMINATION	Arrêt déclenché par le système
4	NOT_READY	Le gestionnaire n'est pas capable de traiter la commande.
5	UNSUPPORTED_CMD	La commande demandée n'est pas prise en charge.
6	UNSUPPORTED_TYPE	Le type d'objet demandé n'est pas pris en charge.
7	NO_SUCH_OBJECT	L'objet référencé n'existe pas.
8	INVALID_OBJECT	Syntaxe non valide pour la spécification d'objet
9	INVALID_OPERATION	L'opération commandée n'est pas valide pour l'objet spécifié.
10	INVALID_STATE	L'opération commandée n'est pas valide pour l'état actuel de l'objet.
11	OVERFLOW	Transaction précédente toujours en cours

Les longueurs effectives de l'entrée OBJECT et de la sortie RESULT des instances de bloc fonctionnel de gestion sont **dépendantes de la mise en œuvre**.

L'entrée OBJECT doit spécifier l'objet devant subir l'opération conformément à l'entrée CMD et la sortie RESULT doit contenir une description de l'objet résultant de l'opération si elle a réussi. Le contenu de ces chaînes doit inclure des codages **dépendants de la mise en œuvre** des objets définis comme des symboles non terminaux dans l'Annexe B et référencés dans le Tableau 8.

NOTE 6 La longueur maximale admissible de l'entrée OBJECT et de la sortie RESULT est un **paramètre dépendant de la mise en œuvre**; la valeur de 512 donnée dans la Figure 22 est illustrative.

Tableau 8 – Syntaxe de commande

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
DELETE	data_type_name	data_type_name
	fb_type_name	fb_type_name
	fb_instance_reference	fb_instance_reference
	connection_definition	connection_definition
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

CMD	OBJECT	RESULT
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
READ	parameter_reference	parameter
WRITE	referenced_parameter	parameter_reference
RESET	fb_instance_reference	fb_status
NOTE Voir Tableau 6 pour les valeurs entières de l'entrée CMD correspondant aux commandes énumérées ci-dessus.		

Cela doit être considéré comme une **erreur**, donnant lieu à un code STATUS égal à INVALID_OBJECT, si une commande CREATE tente de créer

- un *bloc fonctionnel* dont le *nom d'instance* duplique celui d'un bloc fonctionnel existant au sein de la même *ressource*,
- un doublon de *connexion*, ou
- plusieurs connexions sur une même *entrée de données*.

La seule exception à la règle ci-dessus consiste en ce qu'une commande CREATE peut remplacer une connexion d'un *parameter* (paramètre) à une *entrée de données* avec une nouvelle connexion de paramètre.

Cela doit être considéré comme une **erreur**, donnant lieu à un code de STATUS égal à UNSUPPORTED_TYPE, si une commande CREATE tente de créer une instance de bloc fonctionnel ou un paramètre d'un *type* qui n'est pas connu du bloc fonctionnel de gestion.

Cela doit être considéré comme une **erreur**, donnant lieu à un code STATUS égal à INVALID_OPERATION, si une commande DELETE tente de supprimer un *type bloc fonctionnel*, une instance de bloc fonctionnel, un *type de données* ou une connexion qui est défini(e) dans la *spécification type* de la *ressource* gérée.

La sémantique des commandes START et STOP doit être comme suit:

- les commandes START et STOP d'une *instance de bloc fonctionnel* doivent être telles que définies en 6.3.2;
- les commandes START et STOP d'une *application* doivent respectivement équivaloir à START et STOP de toutes les *instances du bloc fonctionnel* dans l'application contenue dans la *ressource* gérée;
- la commande STOP d'une *instance de bloc fonctionnel de gestion* doit être l'équivalent de STOP de toutes les *instances de bloc fonctionnel* au sein de la *ressource* gérée;
- la commande START d'une *instance de bloc fonctionnel de gestion* doit être l'équivalent de START de toutes les *instances de bloc fonctionnel* au sein de la *ressource* gérée. Si la ressource gérée a été arrêtée précédemment, cela doit être suivi de l'émission d'un événement sur la sortie appropriée de chaque instance du type de bloc fonctionnel E_RESTART défini dans l'Annexe A. Ces événements doivent se produire aux sorties WARM des blocs E_RESTART si la ressource a été arrêtée en raison d'une précédente commande STOP, ou bien sur les sorties COLD.

La sémantique spécialisée pour la commande QUERY doit être comme suit:

- lorsque l'entrée OBJECT spécifie une *entrée d'événements, sortie d'événements* ou *sortie de données,* la sortie RESULT doit contenir zéro, un ou plusieurs points d'extrémité opposés;
- lorsque l'entrée OBJECT spécifie une *entrée de données,* la sortie RESULT doit énumérer zéro ou un seul point d'extrémité opposé;
- lorsque l'entrée OBJECT spécifie le nom d'une *application,* la sortie RESULT doit énumérer les noms de tous les blocs fonctionnels dans l'application contenus au sein de la *ressource* gérée.

6.3.3 Comportement des blocs fonctionnels gérés

Les blocs fonctionnels qui sont sous le contrôle d'un *bloc fonctionnel de gestion* doivent manifester des comportements opérationnels équivalant à celui montré dans le diagramme de transitions d'états de la Figure 24, en respectant les règles suivantes:

a) Les conditions de transitions en majuscules dans la Figure 24 renvoient à une valeur de l'entrée CMD, telle que spécifiée dans le Tableau 6, du bloc fonctionnel de gestion à l'apparition d'une primitive de service REQ+.

b) La séquence de primitives command_error pour le type de bloc fonctionnel MANAGER doit apparaître, avec la valeur indiquée de la sortie STATUS telle que définie dans le Tableau 7, dans les conditions suivantes:

   1) UNSUPPORTED_CMD: Aucun état n'existe dans la Figure 24 avec une condition de transition pour la valeur CMD spécifiée;

   2) INVALID_STATE: L'état actif courant n'a pas de condition de transition pour la valeur de CMD spécifiée;

   3) UNSUPPORTED_TYPE: La valeur de CMD est CREATE et l'instance de bloc fonctionnel n'existe pas, mais le type de bloc fonctionnel n'est pas connu de l'instance MANAGER, c'est-à-dire que la condition de garde **type_defined** est FALSE;

   4) INVALID_OPERATION: La valeur de CMD est DELETE et l'instance de bloc fonctionnel est dans l'état STOPPED ou KILLED, mais l'instance de bloc fonctionnel est *déclarée* dans la spécification de *type* de l'*équipement ou des ressources*, c'est-à-dire que la condition de garde **is_deletable** est FALSE.

c) La séquence normal_command_sequence de primitives montrées pour le type de bloc fonctionnel MANAGER doit suivre une primitive de service CMD+ dans toutes les autres conditions, avec une valeur de RDY pour la sortie STATUS telle que définie dans le Tableau 7 et avec une valeur correspondante pour la sortie RESULT telle que définie dans le Tableau 8.

d) La sémantique des actions montrées à la Figure 24 doit être telle que montrée dans le Tableau 9 pour les *blocs fonctionnels de base* et *interface de service* gérés.

e) Les actions décrites dans la règle précédente s'appliquent de manière récursive à tous les *blocs fonctionnels composants* des *blocs fonctionnels composés* gérés.

NOTE 1 Les comportements des blocs fonctionnels qui ne sont pas sous le contrôle des blocs fonctionnels de gestion ne relèvent pas du domaine d'application de la présente norme.

NOTE 2 La spécification du comportement des blocs fonctionnels gérés dans des conditions de coupure et de rétablissement de puissance électrique ne relève pas du domaine d'application de la présente norme. Un tel comportement est susceptible d'être spécifié par le fabricant d'un équipement conforme, par exemple par référence à une norme appropriée.

NOTE 3 Des *applications* peuvent'utiliser des *instances* du bloc E_RESTART décrit dans l'Annexe A pour générer des événements susceptibles d'être utilisés pour déclencher des algorithmes appropriés en cas de coupure et de rétablissement de puissance électrique.

NOTE 4 Comme décrit en 5.4.2, le contrôle d'exécution dans des *sous-applications* est entièrement cédé aux mécanismes de contrôle d'exécution de leurs blocs fonctionnels composants et des sous-applications constitutives.

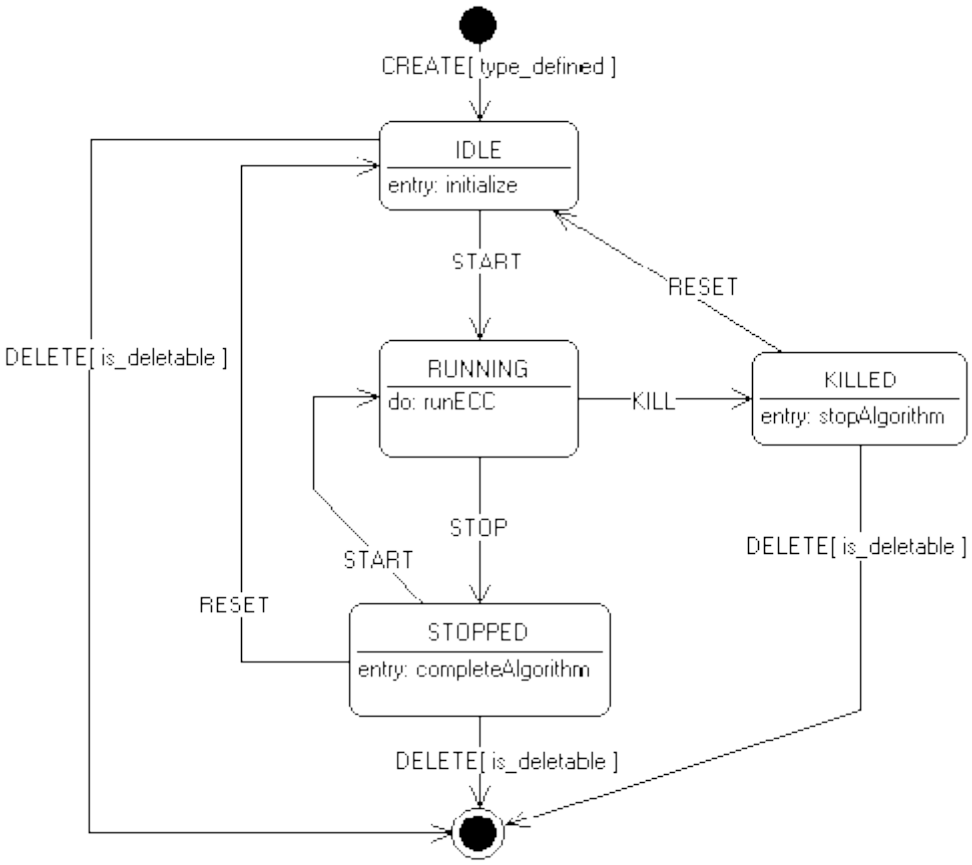


Figure 24 – Diagramme d'états opérationnels d'un bloc fonctionnel géré

Tableau 9 – Sémantique des actions de la Figure 24

Action	Blocs fonctionnels de base	Bloc fonctionnel interface de service
initialize	Initialiser toutes les variables comme défini en 5.2.2.1.	
	Accomplir d'autres opérations d'initialisation comme défini en 5.2.2.1.	Placer le service dans l'état adéquat pour répondre correctement à une primitive INIT+.
runECC	Activer le diagramme d'états d'opération de l'ECC défini en 5.2.2.2	Activer l'invocation des primitives de service par des événements aux entrées d'événements, et création d'événements aux sorties d'événements.
completeAlgorithm	Permettre l'algorithme actuellement actif (s'il y en a) sans autre génération d'événements de sortie.	Laisser se terminer la primitive de service actuellement active.
stopAlgorithm	Mettre immédiatement fin aux opérations de l'algorithme actuellement actif (s'il y en a).	Mettre immédiatement fin à toutes les opérations du service.

7 Configuration d'unités fonctionnelles et de systèmes

7.1 Principes de configuration

L'Article 7 contient des règles pour la *configuration* des *systèmes* de mesure et commande dans les processus industriels (les IPMCS) conformément au modèle suivant:

a) un IPMCS est constitué d'*équipements* connectés entre eux;

b) un *équipement* est une *instance* d'un *type* de l'équipement correspondant;

c) les capacités fonctionnelles d'un *type d'équipement* sont décrites en termes des *ressources* qui lui sont associées;

d) une *ressource* est une *instance* d'un *type de ressource* correspondant;

e) les capacités fonctionnelles d'un *type de ressource* sont décrites en termes des *types de blocs fonctionnels* qui peuvent être *instanciés* et des *instances de blocs fonctionnels* qui existent, dans toutes les *instances* du *type* de ressource.

On considère donc que la *configuration* d'un IPMCS consiste en la *configuration* de ses *équipements* et *applications,* associés, y compris l'allocation d'*instances de bloc fonctionnel* dans chaque *application* aux *ressources* associées aux *équipements*. L'Article 7 définit les ensembles de règles suivantes pour venir à l'appui de ce processus:

- les règles pour la spécification fonctionnelle des *types* de *ressources* et des *équipements* sont définies en 7.2;
- les règles pour la *configuration* d'un IPMCS en termes des *équipements* et d'*applications* associés sont définies en 7.3.

7.2 Spécification fonctionnelle des types de ressources, d'équipements et de segments

7.2.1 Spécification fonctionnelle des types de ressources

La spécification fonctionnelle d'un *type de ressource* comprend:

- le *nom du type* de la ressource;
- le *nom d'instance,* le *type de données* et l'initialisation de chacun des *paramètres* de la ressource;
- une déclaration des *types de données* et des *types de bloc fonctionnel* que chaque *instance* du *type de ressource* est capable d'*instancier*;
- les noms d'instance, les types et les valeurs initiales de toutes les instances de bloc fonctionnel qui sont toujours présentes dans chaque instance du type de la ressource;
- toutes les *connexions de données, connexions d'adaptateurs* et *connexions d'événements* qui sont toujours présentes dans chaque instance du type de la ressource.

NOTE 1 Des informations complémentaires peuvent'être fournies avec les spécifications types des ressources, y compris:

– les nombres maximum de *connexions de données,* de *connexions d'adaptateurs* et de *connexions d'événements* qui peuvent exister dans une instance type de la ressource;

– le temps (identifié comme étant "T_{alg}" à la Figure 7) requis pour l'*exécution* de chaque *algorithme* des blocs fonctionnels d'un type spécifié dans une instance de la ressource;

– le nombre maximal d'instances de type de blocs fonctionnels spécifiés qui sont susceptibles d'exister dans chaque instance de la ressource;

– des compromis entre les instances de blocs fonctionnels, par exemple: savoir si deux instances du type bloc fonctionnel "A" peuvent être échangées contre une seule instance de type "B", etc.

NOTE 2 Les spécifications fonctionnelles des *interfaces* de communication et des processus d'une ressource, y compris le genre et le degré de conformité à des normes applicables, ne relèvent pas du domaine d'application de la présente norme, sauf lorsque ces interfaces sont représentées par des *blocs fonctionnels interface de service*.

7.2.2 Spécification fonctionnelle des types d'équipements

La spécification fonctionnelle d'un *type d'équipement* comprend:

a) le *nom de type* de l'équipement;

b) le *nom d'instance,* le *type de données* et l'initialisation de chacun des *paramètres* de l'équipement;

c) le nom d'instance, le nom du type et l'initialisation de chaque *instance de bloc fonctionnel* qui est toujours présente dans chaque *instance* du type d'équipement;

d) toutes *connexions de données, connexions d'adaptateurs* et *connexions d'événements* qui sont toujours présentes dans chaque instance du type d'équipement.

e) déclarations des *instances des ressources* qui sont présentes dans chaque instance du type d'équipement. Chaque déclaration doit contenir:

   1) le nom d'instance et le nom du type de la ressource;

   2) le nom d'instance, le nom du type et l'initialisation de chaque *instance de bloc fonctionnel* qui est toujours présente dans l'instance de ressource dans chaque instance du type d'équipement;

   3) toutes les *connexions de données, connexions d'adaptateurs* et *connexions d'événements* qui sont toujours présentes dans l'instance de ressource dans chaque instance du type d'équipement.

NOTE 1 Les points (2) et (3) ci-dessus sont considérés comme un ajout aux éléments correspondants déclarés dans la spécification du type de ressources telle que définie en 7.2.1.

NOTE 2 Les spécifications fonctionnelles des *interfaces* de communication et de processus d'un équipement, y compris le genre et le degré de conformité à des normes applicables, ne relèvent pas du domaine d'application de la présente norme, sauf lorsque ces interfaces sont représentées par des *blocs fonctionnels interface de service*.

NOTE 3 Un type d'équipement est susceptible de contenir un réseau de blocs fonctionnels seulement si l'on considère qu'il est constitué d'une seule ressource (non déclarée); dans ce cas, le type d'équipement ne contient aucune déclaration d'instances de ressource.

7.2.3 Spécification fonctionnelle des types de segments

La spécification fonctionnelle d'un *type de segment* comprend:

- le *nom du type* de segment;
- le *nom d'instance,* le *type de données* et l'initialisation de chacun des *paramètres* du segment.

7.3 Exigences relatives à la configuration

7.3.1 Configuration des systèmes

La configuration d'un *système* comprend:

- le *nom* du système;
- la spécification de chaque *application* dans le système, telle que définie en 7.3.2;
- la configuration de chaque *équipement* et de ses *ressources* associées, telle que définie en 7.3.3;
- la configuration de chaque *segment de réseau* et de ses *liaisons* associées à des équipements ou à des ressources, telle que spécifiée en 7.3.4.

7.3.2 Spécification d'applications

La spécification d'une *application* consiste en:

- son nom sous la forme d'un *identificateur*;
- le *nom d'instance,* le *nom du type,* les *connexions de données,* les *connexions d'événements* et les *connexions d'adaptateur* de chaque *bloc fonctionnel* et *sous-application* dans l'application.

Cela doit être considéré comme une **erreur** si le nom d'une application n'est pas unique au sein de la portée du *système*.

7.3.3 Configuration des équipements et des ressources

La configuration d'un *équipement* comprend:

- le *nom d'instance* et le *nom du type* d'équipement;

- les valeurs spécifiques à la configuration pour les *paramètres* de léquipement;
- les *types de ressources* pris en charge par l'*instance* de l'équipement en plus de ceux qui sont spécifiés pour le *type* d'équipement;
- le *nom d'instance* et le *nom du type* de chaque *instance de bloc fonctionnel* qui est présente dans l'instance de l'équipement en plus de ceux qui sont définis pour le type d'équipement;
- toutes les *connexions de données, connexions d'adaptateurs* et *connexions d'événements* qui sont présentes dans l'instance de l'équipement en plus de celles qui sont définies pour le type d'équipement;
- les *types de ressources* pris en charge par l'*instance* de l'équipement en plus de ceux qui sont spécifiés pour le *type* d'équipement;
- la configuration de chacune des *ressources* dans l'équipement. Celles-ci sont constituées de toutes les instances de ressources définies dans la spécification des *types* d'équipements plus toutes les éventuelles ressources associées à l'*instance* de l'équipement spécifique.

NOTE Une instance d'équipement est susceptible de contenir un réseau de blocs fonctionnels seulement si l'on considère qu'il est constitué d'une seule ressource (non déclarée); dans ce cas, la déclaration de l'instance de l'équipement ne contient aucune déclaration d'instances de ressource.

Cela doit être considéré comme une **erreur** si le nom d'instance dans chaque équipement n'est pas unique au sein de la portée du système.

La configuration d'une *ressource* comprend:

a) son *nom d'instance* et son *nom de type*;

b) les *types de données* et les *types de bloc fonctionnel* pris en charge par l'*instance* ressource*;*

c) le *nom d'instance*, le *nom de type* et l'initialisation de chaque instance de bloc fonctionnel qui est présente dans l'instance ressource;

d) toutes *connexions de données, connexions d'événements* et *connexions d'adaptateur* qui sont présentes dans l'instance ressource.

La configuration des ressources est assujettie aux règles suivantes:

– Les points b), c), et d) ci-dessus sont considérés comme un ajout aux éléments correspondants déclarés dans les spécifications des types d'équipements et de ressources telles que respectivement définies en 7.2.2 et 7.2.1.

– Les points c) et d) comprennent des *instances de bloc fonctionnel*, des *connexions de données, connexions d'adaptateurs* et des *connexions d'événements* issues des parties d'*applications* allouées à la ressource.

– Les points c) et d) comprennent des *blocs fonctionnels de communication, des connexions de données,* des *connexions d'événements* et des *connexions d'adaptateurs*, selon les besoins, pour établir et maintenir les flots de données et d'événements pour les *applications* associées.

– Les éléments dans le point c) peuvent inclure le *mapping* d'instances de bloc fonctionnel dans l'application à des instances de bloc fonctionnel existant dans la ressource en raison de la définition de types décrite en 7.2.1.

– Cela doit être considéré comme une **erreur** si le nom d'instance d'une ressource n'est pas unique dans la portée de l'équipement la contenant ou si une instance de bloc fonctionnel quelconque dans une application n'est pas allouée à exactement une seule ressource.

Un moyen automatisé peut être fourni pour satisfaire aux exigences ci-dessus. Les fournisseurs de tels moyens doivent donner des règles non ambiguës permettant de déterminer leur opération ou bien ils doivent fournir un moyen permettant d'examiner et de modifier les résultats de l'application de ces moyens.

7.3.4 Configuration des segments et des liaisons réseau

La configuration d'un *segment de réseau* comprend:

- le *nom d'instance* et le *nom du type* du segment;
- des valeurs spécifiques à une configuration pour les paramètres du segment du réseau.

Cela doit être considéré comme une **erreur** si le *nom d'instance* de chaque segment de réseau n'est pas unique dans la portée du *système* ou si les valeurs déclarées des paramètres de segment sont incompatibles avec la déclaration (éventuelle) du *type de segment* défini en 7.2.3.

La configuration d'une *liaison* comprend:

– le nom d'un *équipement* ou le nom hiérarchique d'une "*ressource* de communication" à l'intérieur d'un équipement et le nom du segment du réseau auquel l'équipement ou la ressource est relié(e);

– des valeurs spécifiques à une configuration pour les *paramètres* de la liaison.

Annexe A
(normative)

Blocs fonctionnels d'événements

Des *instances* des *types* de bloc fonctionnel montrées dans le Tableau A.1 peuvent être utilisées pour la génération et le traitement d'*événements* dans des *blocs fonctionnels composés*; dans des *sous-applications*; pour la définition des types de *ressource* et d'*équipement*; et dans la *configuration* d'*applications*, des *ressources* et des *équipements*.

Les types de bloc fonctionnel montrés dans l'Annexe A qui utilisent des *graphiques de contrôle d'exécution* sont des *types de bloc fonctionnel de base*. Lorsque des déclarations textuelles d'*algorithmes* sont données pour ces types de bloc fonctionnel, le langage utilisé est le langage Structured Text (ST) défini dans la CEI 61131-3.

Des mises en œuvre de référence pour certains des types de bloc fonctionnel dans l'Annexe A sont données comme des définitions de *type de bloc fonctionnel composé*. Ces mises en œuvre sont normatives seulement dans le sens où les comportements fonctionnels des mises en œuvre conformes doivent être équivalents à ceux de la mise en œuvre de référence, avec les considérations suivantes qui s'appliquent aux paramètres de temporisation définis en 4.5.3:

- Les paramètres T_{setup}, T_{start} et T_{finish} sont considérés être égaux à zéro (0) pour tous les *blocs fonctionnels composants* dans la mise en œuvre de référence.
- Le paramètre T_{alg} est considéré égal au paramètre `DT` pour toutes les instances du type `E_DELAY` utilisées comme *blocs fonctionnels composants* dans la mise en œuvre de référence, tandis qu'il est considéré égal à zéro (0) pour tous les autres blocs fonctionnels composants dans la mise en œuvre de référence.

Tous les autres types de bloc fonctionnel donnés dans l'Annexe A sont des *blocs fonctionnels de type interface de service*.

NOTE L'Annexe F donne des spécifications textuelles complètes de tous les types de bloc fonctionnel montrés dans le Tableau A.1.

Tableau A.1 – Blocs fonctionnels d'événements (1 de 7)

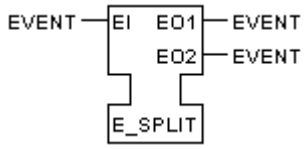
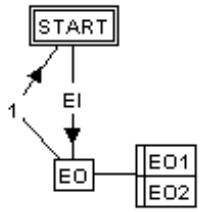
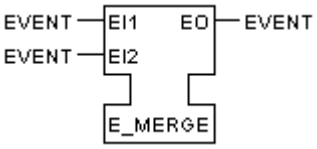
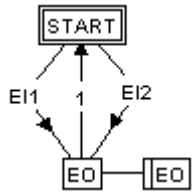
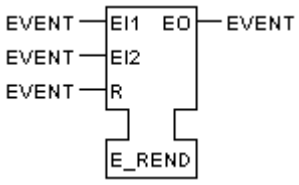
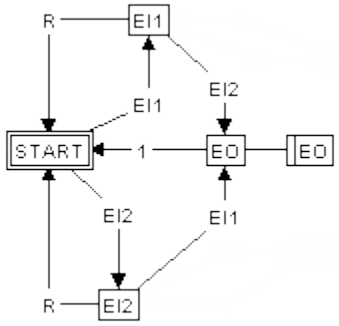
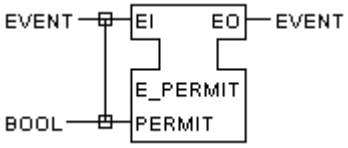
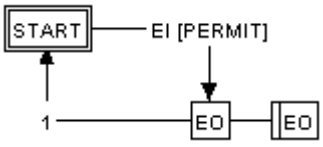
<table>
<tr><th>No.</th><th colspan="2">Description</th></tr>
<tr><th colspan="2">Interface</th><th>Séquences ECC/Algorithmes/Service</th></tr>
<tr><td>1</td><td colspan="2">Diviser un événement</td></tr>
<tr><td colspan="2"> 
</td><td> 
</td></tr>
<tr><td colspan="3">L'apparition d'un événement en EI entraîne l'apparition d'événements en EO1, EO2,…,Eon (n=2 dans l'exemple ci-dessus).</td></tr>
<tr><td>2</td><td colspan="2">Fusion (OR) de plusieurs événements</td></tr>
<tr><td colspan="2"> 
</td><td> 
</td></tr>
<tr><td colspan="3">L'apparition d'un événement en l'une quelconque des entrées EI1, EI2,…,EIn entraîne l'apparition d'un événement en EO (n=2 dans l'exemple ci-dessus).</td></tr>
<tr><td>3</td><td colspan="2">Rendez-vous de deux événements</td></tr>
<tr><td colspan="2"> 
</td><td> 
</td></tr>
<tr><td>4</td><td colspan="2">Propagation permissive d'un événement</td></tr>
<tr><td colspan="2"> 
</td><td> 
</td></tr>
</table>

Tableau A.1 (2 de 7)

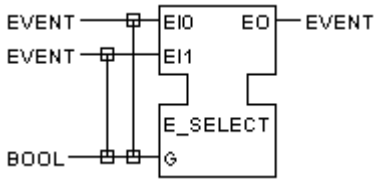
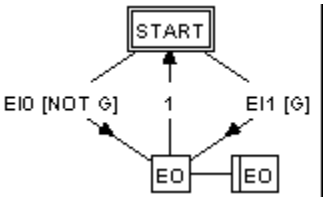
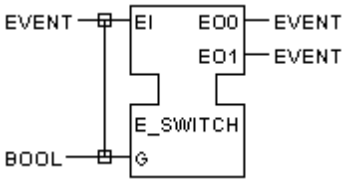
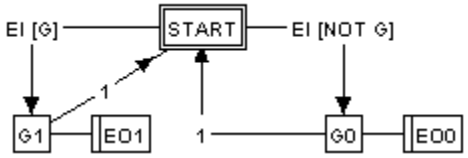
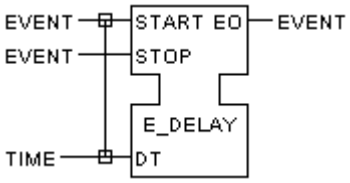
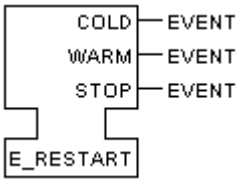
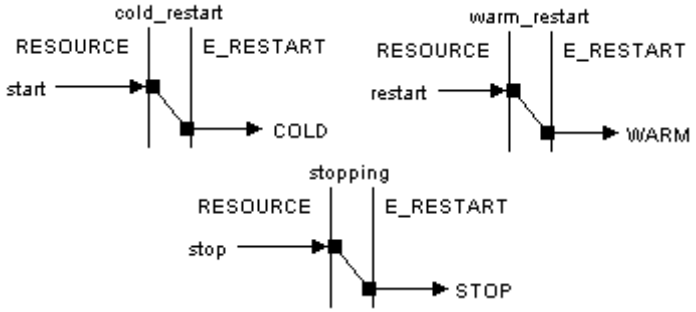
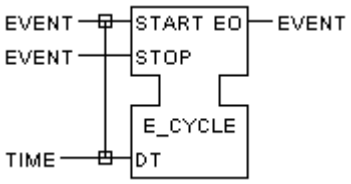
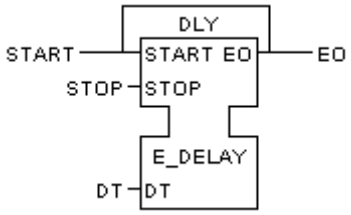
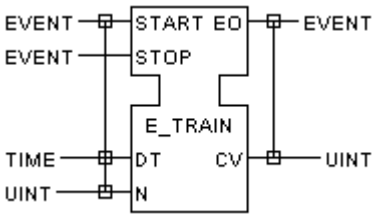
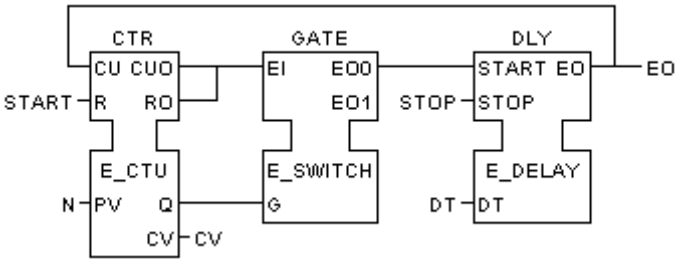
<table>
<tr><th>No.</th><th colspan="2">Description</th></tr>
<tr><th colspan="2">Interface</th><th>Séquences ECC/Algorithmes/Service</th></tr>
<tr><td>5</td><td colspan="2">Sélection entre deux événements</td></tr>
<tr><td colspan="2"> 
</td><td> 
</td></tr>
<tr><td>6</td><td colspan="2">Commutation (démultiplexage) d’un événement</td></tr>
<tr><td colspan="2"> 
</td><td> 
</td></tr>
<tr><td>7</td><td colspan="2">Propagation différée d’un événement</td></tr>
<tr><td colspan="2"> 
</td><td>Un événement en EO est généré à un intervalle de temps DT après l’apparition d’un événement sur l’entrée START. Le retard pour l’événement est annulé par l’apparition d’un événement sur l’entrée STOP. Si plusieurs événements apparaissent sur l’entrée START avant l’apparition d’un événement en EO, seul un événement simple apparaît en EO, avec un temps DT après l’apparition du premier événement sur l’entrée START. Aucun retard d’événement ne sera déclenché si un événement apparaît sur l’entrée START avec une valeur de DT qui n’est pas supérieure à t#0s.</td></tr>
<tr><td>8</td><td colspan="2">Génération d’événements de redémarrage</td></tr>
<tr><td colspan="2"> 
</td><td> 
</td></tr>
<tr><td colspan="3">a) Un événement est émis sur la sortie COLD à la suite d’un «redémarrage à froid» de la ressource associée.
b) Un événement est émis sur la sortie WARM à la suite d’un «redémarrage à chaud» de la ressource associée.
c) Un événement est émis sur la sortie STOP (si possible) avant «l’arrêt» de la ressource associée.
NOTE 1 Voir la CEI 61131-1 pour un débat relatif au «redémarrage à froid» et «redémarrage à chaud».</td></tr>
</table>

Tableau A.1 (3 de 7)

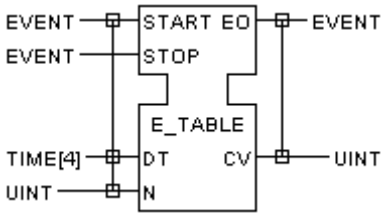
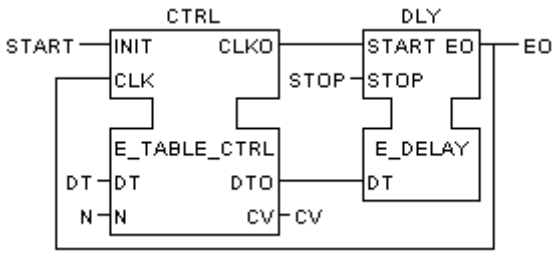
<table>
<tr><th>No.</th><th colspan="2">Description</th></tr>
<tr><th colspan="2">Interface</th><th>Séquences ECC/Algorithmes/Service</th></tr>
<tr><td>9</td><td colspan="2">Génération périodique (cyclique) d'un événement</td></tr>
<tr><td colspan="2"> 

Un événement apparaît en EO avec un intervalle DT après l'apparition d'un événement en START, et avec des intervalles DT par la suite jusqu'à l'apparition d'un événement en STOP.</td><td> 
</td></tr>
<tr><td>10</td><td colspan="2">Génération d'un train fini d'événements</td></tr>
<tr><td colspan="2"> 
</td><td> 

NOTE 2 Voir l'entrée 18 du tableau pour une définition du type E_CTU.</td></tr>
<tr><td colspan="3">Un événement apparaît en EO avec un intervalle DT après l'apparition d'un événement en START et à des intervalles DT par la suite jusqu'à ce que N apparitions aient été générées ou qu'un événement apparaisse sur l'entrée STOP.

NOTE 3 Le compte CV est réinitialisé chaque fois qu'un événement apparaît sur l'interface START, mais le retard ne redémarre pas à moins qu'il n'ait déjà été arrêté. Ce comportement maintient l'intervalle entre EO lors du redémarrage du compte.</td></tr>
</table>

Tableau A.1 (4 de 7)

<table>
<tr><th>No.</th><th colspan="2">Description</th></tr>
<tr><th colspan="2">Interface</th><th>Séquences ECC/Algorithmes/Service</th></tr>
<tr><td>11</td><td colspan="2">Génération d'un train d'événements (guidée par table)</td></tr>
<tr><td colspan="2"> 
</td><td> 
</td></tr>
<tr><td colspan="3">Un événement apparaît en EO avec un intervalle DT[0] après l'apparition d'un événement en START. Un deuxième événement apparaît avec un intervalle DT[1] après le premier, etc., jusqu'à ce que N apparitions aient été générées ou qu'un événement apparaisse sur l'entrée STOP. Le compte d'événements courant est maintenu sur la sortie CV.

NOTE 4 Dans cet exemple de mise en œuvre, N <= 4.

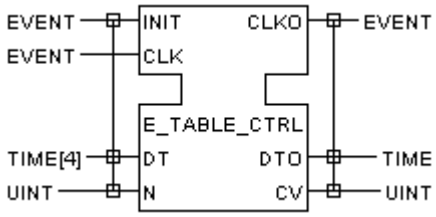
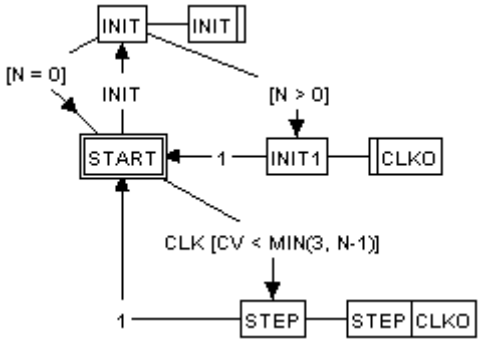
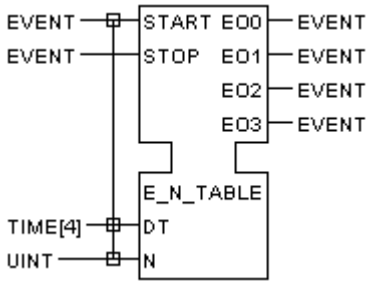
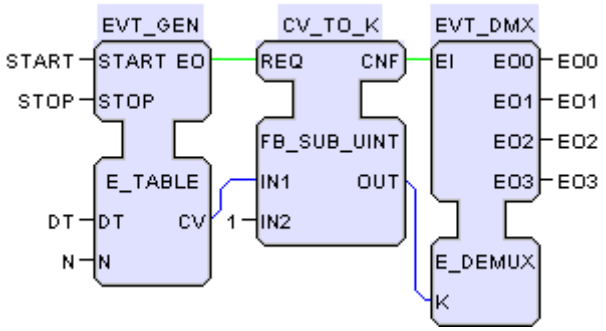
NOTE 5 La mise en œuvre utilisant le type de bloc fonctionnel E_TABLE_CTRL illustré ci-dessous n'est pas une exigence normative. Une fonctionnalité équivalente peut être mise en œuvre par divers moyens.</td></tr>
<tr><td colspan="2"> 
</td><td> 
</td></tr>
<tr><td colspan="2">ALGORITHM INIT IN ST:
  CV:= 0;
  DTO:= DT[0];
END_ALGORITHM</td><td>ALGORITHM STEP IN ST:
  CV:= CV+1;
  DTO:= DT[CV];
END_ALGORITHM</td></tr>
</table>

Tableau A.1 (5 de 7)

<table>
<tr><th>No.</th><th colspan="2">Description</th></tr>
<tr><th colspan="2">Interface</th><th>Séquences ECC/Algorithmes/Service</th></tr>
<tr><td>12</td><td colspan="2">Génération d'un train d'événements distincts (guidée par table)</td></tr>
<tr><td colspan="2"> 
</td><td> 
</td></tr>
<tr><td colspan="3">Un événement apparaît en EO0 avec un intervalle DT[0] après l'apparition d'un événement en START. Un événement apparaît en EO1 avec un intervalle DT[1] après l'apparition de l'événement en EO0, etc., jusqu'à ce que N apparitions aient été générées ou qu'un événement apparaisse sur l'entrée STOP.

NOTE 6 Dans cet exemple de mise en œuvre, N <= 4.

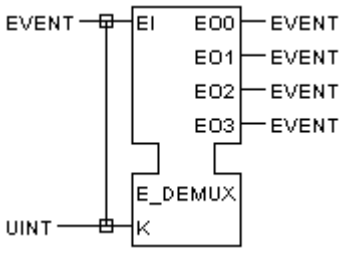
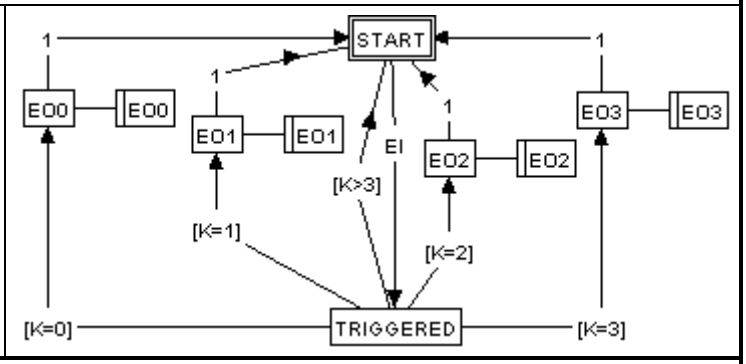
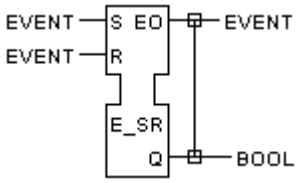
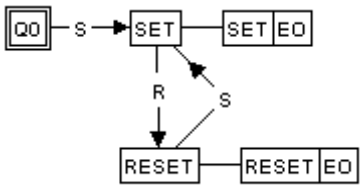
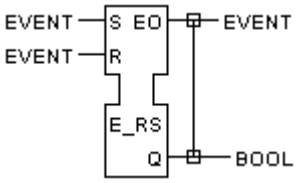
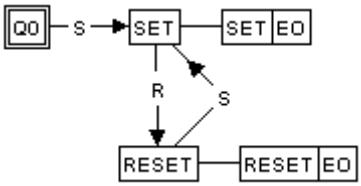
NOTE 7 La mise en œuvre utilisant le type de bloc fonctionnel E_DEMUX illustré ci-dessous n'est pas une exigence normative. Une fonctionnalité équivalente peut être mise en œuvre par divers moyens.</td></tr>
<tr><td colspan="2"> 
</td><td> 
</td></tr>
<tr><td>13</td><td colspan="2">Bistable piloté par des événements</td></tr>
<tr><td colspan="2"> 
</td><td> 
</td></tr>
<tr><td colspan="3">La sortie Q est mise à 1 (TRUE) à la suite de l'apparition d'un événement sur l'entrée S, et elle est réinitialisée à 0 (FALSE) à la suite de l'apparition d'un événement sur l'entrée R. Un événement est émis sur la sortie EO lorsque la valeur de Q change.</td></tr>
<tr><td colspan="2"><pre>ALGORITHM SET IN ST: (* Set Q *)
  Q:= TRUE;
END_ALGORITHM</pre></td><td><pre>ALGORITHM RESET IN ST: (* Reset Q *)
  Q:= FALSE;
END_ALGORITHM</pre></td></tr>
</table>

Tableau A.1 (6 de 7)

<table>
<tr><th>No.</th><th colspan="2">Description</th></tr>
<tr><th colspan="2">Interface</th><th>Séquences ECC/Algorithmes/Service</th></tr>
<tr><td>14</td><td colspan="2">Bistable piloté par des événements</td></tr>
<tr><td colspan="2"> 
</td><td> 
</td></tr>
<tr><td colspan="3">La sortie Q est mise à 1 (TRUE) à la suite de l'apparition d'un événement sur l'entrée S, et elle est réinitialisée à 0 (FALSE) à la suite de l'apparition d'un événement sur l'entrée R. Un événement est émis sur la sortie EO lorsque la valeur de Q change.

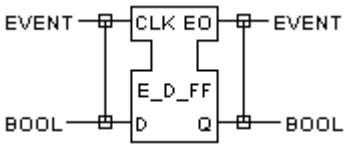
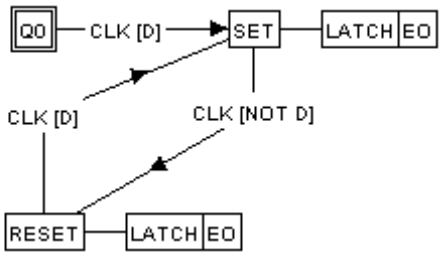
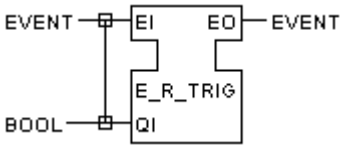
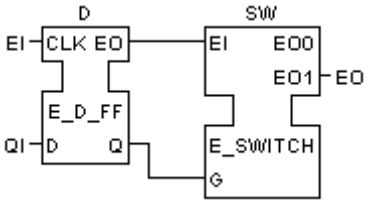
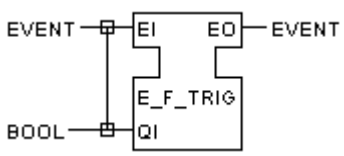
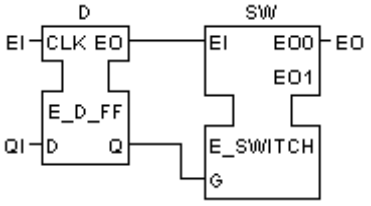
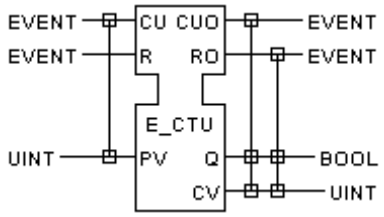
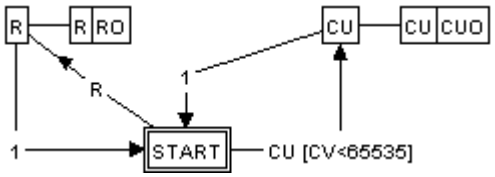
NOTE La mise en œuvre de ce type de bloc fonctionnel est identique à E_SR. E_SR et E_RS sont tous les deux mis en œuvre pour la cohérence avec les types SR et RS de la CEI 61131-3, bien qu'il n'y ait aucune «dominance» d'événements comme il y en aurait pour des entrées R et S commandées par niveau.</td></tr>
<tr><td>15</td><td colspan="2">Bistable D (Verrou de données)</td></tr>
<tr><td colspan="2"> 
</td><td> 
</td></tr>
<tr><td colspan="2"></td><td><pre>ALGORITHM LATCH IN ST:
  Q:= D;
END_ALGORITHM</pre></td></tr>
<tr><td>16</td><td colspan="2">Détection booléenne de front montant</td></tr>
<tr><td colspan="2"> 
</td><td> 
</td></tr>
<tr><td>17</td><td colspan="2">Détection booléenne de front descendant</td></tr>
<tr><td colspan="2"> 
</td><td> 
</td></tr>
</table>

Tableau A.1 (7 de 7)

<table>
<tr><th>No.</th><th colspan="2">Description</th></tr>
<tr><th colspan="2">Interface</th><th>Séquences ECC/Algorithmes/Service</th></tr>
<tr><td>18</td><td colspan="2">Compteur progressif événementiel</td></tr>
<tr><td colspan="2"> 
</td><td> 
</td></tr>
<tr><td colspan="2">ALGORITHM R IN ST: (* Reset *)
  CV:= 0;
  Q:= 0;
END_ALGORITHM</td><td>ALGORITHM CU IN ST: (* Count Up *)
  CV:= CV + 1;
  Q:= (CV >= PV);
END_ALGORITHM</td></tr>
</table>

Des notations graphiques raccourcies peuvent remplacer les blocs E_SPLIT et E_MERGE définis dans le Tableau A.1. Par exemple, la représentation (implicite) raccourcie montrée à la Figure A.1b est l'équivalent de la représentation explicite de la Figure A.1a.

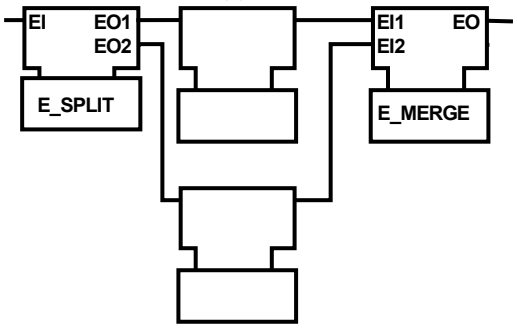


Figure A.1a – Représentation explicite

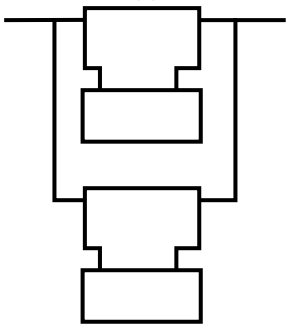


Figure A.1b – Représentation implicite

NOTE Les détails non pertinents ont été supprimés de la figure ci-dessus.

Figure A.1 – Division et fusion d'événements

Annexe B
(normative)

Syntaxe textuelle

B.1 Technique de spécification d'une syntaxe

Les constructions textuelles de l'Annexe B sont spécifiées en termes de *syntaxe*, qui spécifie les combinaisons admissibles de symboles qui peuvent être utilisées pour définir un programme; et d'un jeu de *sémantiques*, qui spécifient les significations des combinaisons de symboles définies par la syntaxe.

Une **syntaxe** est définie par un ensemble de *symboles terminaux* devant être utilisés pour la spécification du programme; un ensemble de *symboles non terminaux* définis en termes de symboles terminaux; et un ensemble de *règles de production* spécifiant ces définitions.

Les **symboles terminaux** pour des spécifications textuelles d'entités définies dans la présente norme sont constitués de combinaisons de caractères parmi l'ensemble de caractères donnés dans le Tableau 2 – Rangée 00 de la table de «Compatibilité Latin de Base et idéogrammes CJK» liée à l'Article 33 défini dans l'ISO/CEI 10646:2003.

Pour les besoins de la présente norme, les symboles textuels terminaux sont constitués d'une chaîne appropriée de caractères placée entre des apostrophes ou des guillemets appariés. Par exemple, un symbole terminal représenté par la chaîne de caractères ABC peut être représenté par `"ABC"` ou `'ABC'`.

Ceci permet la représentation de chaînes contenant soit des apostrophes, soit des guillemets; par exemple, un symbole terminal constitué d'un guillemet lui-même serait représenté par `'"'`

Un symbole terminal spécial utilisé dans la syntaxe est la «chaîne vide», c'est-à-dire, une chaîne ne contenant aucun caractère. Ceci est représenté par le symbole terminal `NIL`.

Les symboles textuels **non terminaux** sont représentés par des chaînes en lettres minuscules, des nombres et le caractère de soulignement (_), commençant par une lettre minuscule. Par exemple, les chaînes `nonterm1` et `non_term_2` sont des symboles non terminaux valides, alors que les chaînes `3nonterm` et `_nonterm4` ne le sont pas.

Les **règles de production** données dans la présente norme forment une *grammaire étendue* dans laquelle chaque règle a la forme

```
non_terminal_symbol::= extended_structure
```

Cette règle peut être lue comme:

«Un `non_terminal_symbol` peut consister en une `extended_structure`.»

Les structures étendues peuvent être construites selon les règles suivantes:

a) La chaîne vide, `NIL`, est une structure étendue.

b) Un symbole terminal est une structure étendue.

c) Un symbole non terminal est une structure étendue.

d) Si `S` est une structure étendue, alors les expressions suivantes sont aussi des structures étendues:

- `(S)`, signifiant `S` lui-même.
- `{S}`, *fermeture*, signifiant zéro, une ou plusieurs concaténations de `S`.
- `[S]`, *option*, signifiant zéro ou une occurrence de `S`.

e) Si `S1` et `S2` sont des structures étendues, alors les expressions suivantes sont des structures étendues:

- `S1 | S2`, *union*, signifiant un choix de `S1` ou `S2`.
- `S1 S2`, *concaténation*, signifiant `S1` suivi de `S2`.

f) La concaténation *précède* l'union, c'est-à-dire que `S1 | S2 S3` équivaut à `S1 | (S2 S3)`, et `S1 S2 | S3` équivaut à `(S1 S2) | S3`.

Les **sémantiques** sont définies dans la présente norme par du texte approprié en langage naturel, accompagnant les règles de production, qui référencent les descriptions données dans les articles appropriés. Les options normalisées disponibles pour l'utilisateur et le vendeur sont spécifiées dans ces sémantiques.

Dans certains cas, il est plus commode d'intégrer les informations sémantiques dans une structure étendue. Dans de tels cas, ces informations sont délimitées par des parenthèses en chevron appariées, par exemple `<informations sémantiques>`.

B.2 Spécification des types de bloc fonctionnel et de sous-application

B.2.1 Spécification des types de bloc fonctionnel

La syntaxe définie en B.2.1 peut être utilisée pour la spécification textuelle des *types de bloc fonctionnel* selon les règles données dans les Articles 5 et 6 de la présente norme.

SYNTAXE:

```
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]
     '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::= event_conn_source 'TO' event_conn_destination ';'


event_conn_source::= ([plug_name '.'] event_input_name)
     | ((fb_instance_name | socket_name) '.' event_output_name)

event_conn_destination::= ([plug_name '.'] event_output_name)
     | ((fb_instance_name | socket_name) '.' event_input_name)
```

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

data_conn::= data_conn_source 'TO' data_conn_destination ';'


data_conn_source::= ([plug_name '.'] input_variable_name)
     | ((fb_instance_name | socket_name) '.' output_variable_name)

data_conn_destination::= ([plug_name '.'] output_variable_name)
     | ((fb_instance_name | socket_name) '.' input_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 | ('->' ec_action_output)
     | (algorithm_name '->' ec_action_output)


ec_action_output:= ([plug_name '.'] event_output_name)
     | (socket_name '.' event_input_name)

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

ec_transition_condition::= '1'
     | ec_transition_event | '[' guard_condition ']'
     | ec_transition_event '[' guard_condition ']'


ec_transition_event::= ([plug_name '.'] event_input_name)
     | (socket_name '.' event_output_name)

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

ec_expression_operand::=
     ([(plug_name | socket_name) '.'] input_variable_name)
     | ([(plug_name | socket_name) '.'] output_variable_name)
     | internal_variable_name
     | constant
```

```
fb_algorithm_declaration::=
     'ALGORITHM' algorithm_name 'IN' language_type ':'
     [temp_var_decls]
     algorithm_body
     'END_ALGORITHM'

temp_var_decls::= <as defined in IEC 61131-3>

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 Spécification des types de sous-application

La syntaxe définie dans ce paragraphe peut être utilisée pour la spécification textuelle des *types* de *sous-application* selon les règles données en 5.4.1.

La production donnée en B.2.1 s'applique aussi au présent paragraphe.

SYNTAXE:

```
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::= subapp_event_source 'TO' subapp_event_destination ';'

subapp_event_source:: = ([plug_name '.'] event_input_name)
     | ((fb_subapp_name | socket_name) '.' event_output_name

subapp_event_destination::= ([plug_name '.'] event_output_name)
     | ((fb_subapp_name | socket_name) '.' event_input_name)

fb_subapp_name::= fb_instance_name | subapp_instance_name

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

subapp_data_conn::= subapp_data_source 'TO' subapp_data_destination ';'

subapp_data_source::= ([plug_name '.'] input_variable_name)
     | ((fb_subapp_name | socket_name) '.' output_variable_name)

subapp_data_destination::= ([plug_name '.'] output_variable_name)
     | ((fb_subapp_name | socket_name) '.' input_variable_name)

subapp_type_name::= identifier

subapp_instance_name::= identifier
```

B.3 Eléments de configuration

La syntaxe définie dans cet article peut être utilisée pour la spécification textuelle des *types de ressource, types d'équipement, types de segment, applications,* et *configurations de système* selon les règles données à l'Article 7.

Les productions données à l'Article B.2 s'appliquent aussi à cet article.

SYNTAXE:

```
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]
     [config_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]
     '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 | input_variable_name)
     'TO'
     (fb_instance_name | resource_instance_name)'.' Input_variable_name ';'
     <resource_instance_name only applies to connections within device_type or
     device_configuration declarations>

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 ';'

fb_instance_reference::= [app_hierarchy_name] fb_instance_name

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

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]
     [config_connection_list]
     'END_DEVICE_TYPE'

resource_instance::=
     'RESOURCE' resource_instance_name ':' resource_type_name
     [fb_instance_list]
     [config_connection_list]
     '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]
     config_connection_list
     'END_RESOURCE_TYPE'

segment_type_specification::= 'SEGMENT_TYPE' segment_type_name
     {parameter_declaration}
     'END_SEGMENT_TYPE'

parameter_declaration:= parameter_name ':' var_spec_init ';'

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

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

fb_resource_reference::= resource_hierarchy ['.' Fb_instance_name]
     <Lorsque l'élément facultatif ['.' Fb_instance_name] n'est pas donné, le
     nom d'instance du FB dans la ressource est le même que son nom d'instance
     dans le fb_instance_reference correspondant du mapping.>

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
```

B.4 Eléments communs

Lorsque des productions syntaxiques ne sont pas données pour les symboles non terminaux dans l'Annexe B, les productions syntaxiques et la sémantique correspondante données dans l'Annexe B de la CEI 61131-3:2003 doivent s'appliquer.

B.5 Représentations prises en charge pour les commandes de gestion

La syntaxe définie dans cet article est référencée dans le Tableau 8.

SYNTAXE:

```
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

referenced_parameter::=
     [(resource_instance_name | fb_instance_name)'.'] parameter
     <resource_instance_name se réfère à une ressource localisée dans le même
     équipement que le bloc MANAGER défini en 6.3.2>
     <fb_instance_name se réfère à un FB contenu dans le même équipement ou la
     même ressource que le bloc <MANAGER> >
     <si aucun nom d'instance de ressource ou de FB n'est donné, le paramètre
     se réfère à un paramètre du équipement ou de la ressource contenant le
     bloc MANAGER>

parameter_reference::=
     [(resource_instance_name | fb_instance_name)'.'] parameter_name
     <see above for semantics>

all_data_types::= 'ALL_DATA_TYPES'

all_fb_types::= 'ALL_FB_TYPES'

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

B.6 Types de données étiquetés

La syntaxe définie ci-dessous doit être utilisée pour l'attribution d'étiquettes telles que définies dans l'ISO/CEI 8824-1 à des types de données dérivés comme spécifié dans l'Annexe B et l'Annexe E. Comme défini dans l'ISO/CEI 8824-1, les étiquettes de classe `APPLICATION` et `PRIVATE` doivent être utilisées, sauf pour les types devant être utilisés seulement dans l'étiquetage spécifique à un contexte.

SYNTAXE:

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

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

B.7 Types d'adaptateurs d'interfaces

Voir 5.5 pour la sémantique associée à la syntaxe suivante.

SYNTAXE:

```
adapter_type_declaration::=
     'ADAPTER' adapter_type_name
          fb_interface_list
          [fb_service_declaration]
     'END_ADAPTER'

adapter_type_name::= identifier
```

Annexe C
(informative)

Modèles d'objets

C.1 Notation du modèle

L'Annexe C donne des modèles d'objets pour certaines des classes qui peuvent être utilisées dans les systèmes Engineering Support Systems (ESS) pour prendre en charge la conception, la mise en œuvre, la mise en service et le fonctionnement des systèmes de mesure et commande dans les processus industriels (les IPMCS) construits selon l'architecture définie dans la présente norme.

La notation utilisée dans l'Annexe C est le langage de modélisation unifié (UML). Des références à une documentation extensive de cette notation peuvent être consultées sur internet à l'URL (localisateur uniforme de ressource «Uniform Resource Locator») **http://www.omg.org/uml/**.

C.2 Modèle des systèmes ESS

C.2.1 Vue d'ensemble du système ESS

La Figure C.1 présente une vue d'ensemble des classes principales dans le système ESS (Engineering Support System «système de support d'ingénierie») pour un système de mesure et commande dans les processus industriels (IPMCS) et leur correspondance avec les classes d'objets dans l'IPMCS. Les descriptions des classes dans la Figure C.1 sont données dans le Tableau C.1.

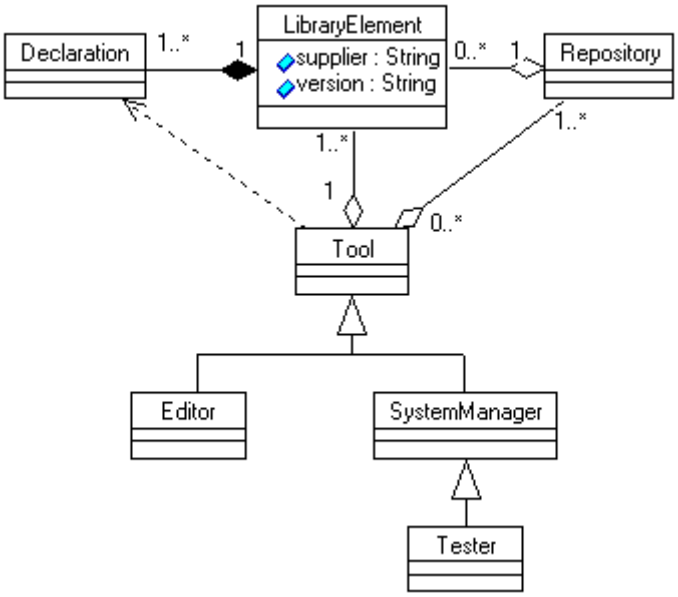


Figure C.1 – Vue d'ensemble du système ESS

Tableau C.1 – Descriptions des classes ESS

Declaration	Il s'agit d'une superclasse abstraite pour les *déclarations*.
Editor	Les instances de cette classe fournissent les fonctions d'édition sur les *déclarations* nécessaires pour prendre en charge le cas d'utilisation EDIT.
LibraryElement	Il s'agit d'une superclasse abstraite qui peut être stockée dans des zones de stockage et qui peut être importée et exportée avec la syntaxe textuelle définie dans l'Annexe B ou avec la syntaxe XML définie dans la CEI 61499-2. De tels objets ont des attributs fournisseur (vendeur, programmeur, etc.) et version (numéro de version, date, etc.) pour aider à la gestion, en plus d'un nom (hérité de **NamedDeclaration –** voir C.2.2) comme attribut-clé.
Repository	Les instances de cette classe assurent le stockage permanent et la récupération d'éléments de la bibliothèque. Elles peuvent aussi fournir des services de contrôle de version.
SystemManager	Les instances de cette classe fournissent les fonctions nécessaires pour prendre en charge les cas d'utilisation INSTALL et OPERATE.
Tester	Cette classe étend les capacités de la classe SystemManager pour prendre en charge les opérations du cas d'utilisation TEST.
Tool	Cette classe modélise les comportements génériques des *outils logiciels* pour le support d'ingénierie des IPMCS.

C.2.2 Eléments bibliothèques

Les sous-classes de **LibraryElement** sont montrées à la Figure C.2. La production syntaxique dans l'Annexe B correspondant à chaque sous-classe est énumérée dans le Tableau C.2.

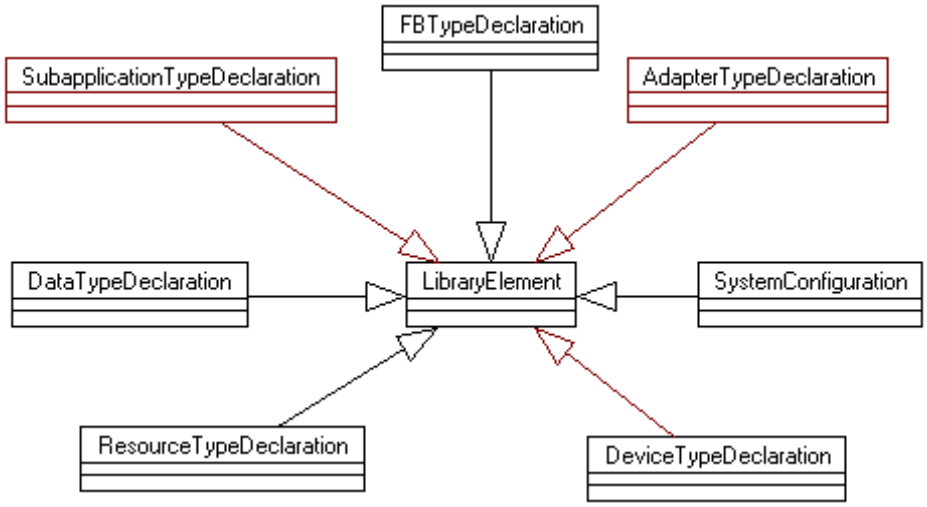


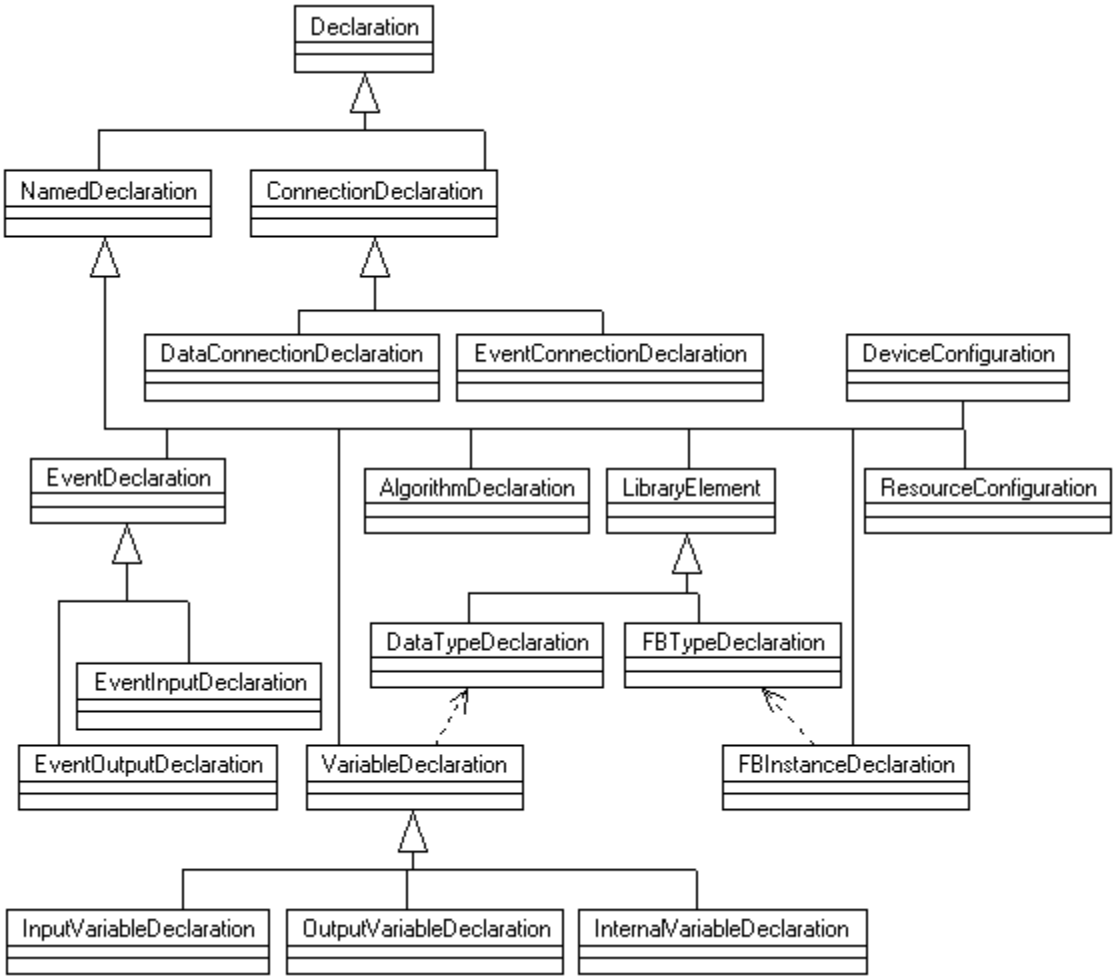
Figure C.2 – Eléments bibliothèques

Tableau C.2 – Productions syntaxiques pour les éléments des bibliothèques

Classe	Production syntaxique
DataTypeDeclaration	type_declaration
FBTypeDeclaration	fb_type_declaration
AdapterTypeDeclaration	adapter_type_declaration
SubapplicationTypeDeclaration	subapplication_type_declaration
ResourceTypeDeclaration	resource_type_specification
DeviceTypeDeclaration	device_type_specification
SystemConfiguration	system_configuration

C.2.3 Déclarations

La Figure C.3 montre la hiérarchie de classe des *déclarations* qui peuvent être manipulées par des *outils logiciels*. Les productions syntaxiques dans l'Annexe B correspondant à chacune de ces sous-classes sont énumérées dans le Tableau C.3.



NOTE Pour éviter la confusion, les classes relatives aux *types, instances* et *connexions* d'adaptateurs ne sont pas montrées dans cette Figure; elles sont toutefois énumérées dans le Tableau C.3 pour référence.

Figure C.3 – Déclarations

Tableau C.3 – Productions syntaxiques pour les déclarations

Classe	Production syntaxique
AdapterConnectionDeclaration	adapter_conn
AdapterTypeDeclaration	adapter_type_declaration
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
PlugDeclaration	Part of plug_list
ResourceConfiguration	resource_instance
SocketDeclaration	Part of socket_list

C.2.4 Déclarations des réseaux de blocs fonctionnels

La Figure C.4 montre les relations entre les éléments de *déclarations des réseaux de blocs fonctionnels*. Voir C.2.2 pour les définitions des classes agrégées dans ce diagramme.

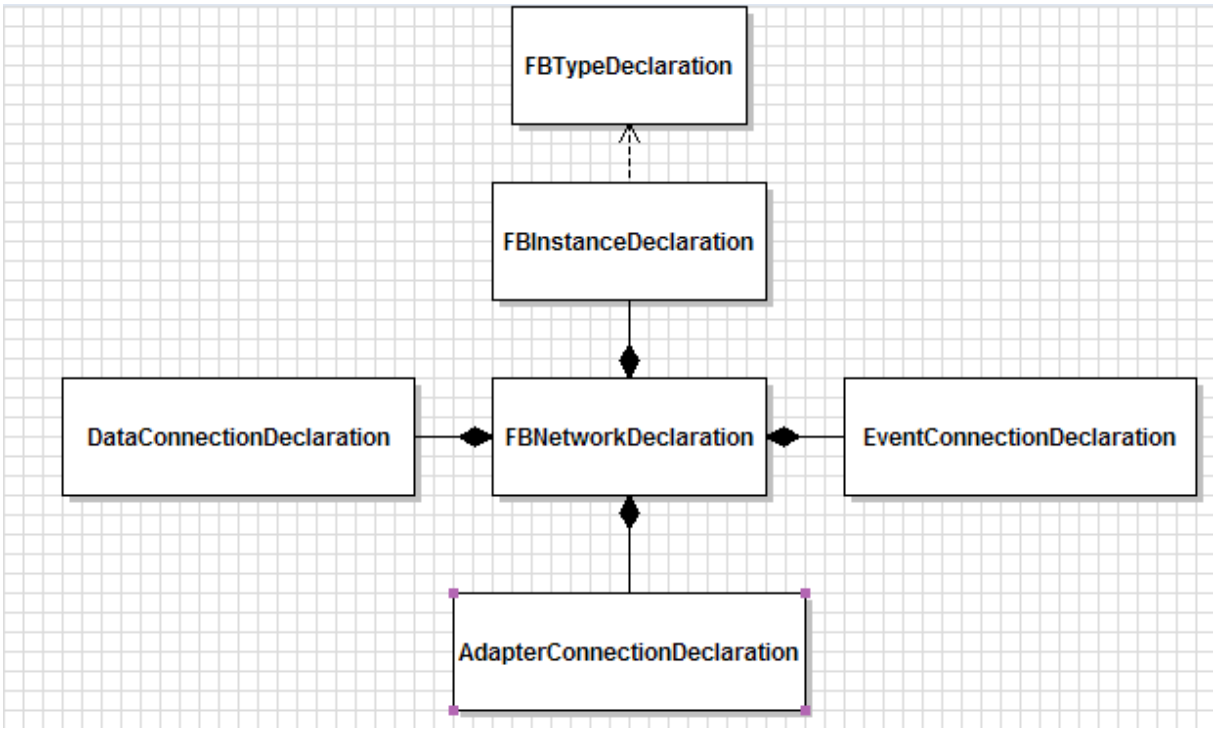


Figure C.4 – Déclarations des réseaux de blocs fonctionnels

C.2.5 Déclarations des types de blocs fonctionnels

La Figure C.5 montre les relations entre les éléments de *déclarations des types de blocs fonctionnels*. Les productions syntaxiques pour les classes **EventInputDeclaration, EventOutputDeclaration, InputVariableDeclaration, OutputVariableDeclaration, InternalVariableDeclaration,** et les classes constitutives de **FBNetworkDeclaration** sont données dans le Tableau C.3. Les productions syntaxiques `fb_ecc_declaration` et `fb_service_declaration` de l'Article B.2 correspondent respectivement aux classes **ECCDeclaration** et **ServiceDeclaration**.

NOTE 1 Les déclarations des *sous-applications* sont représentées par des instances de la classe **CompositeFBTypeDeclaration** qui ne contiennent aucune association de données événement `WITH`.

NOTE 2 **NamedDeclaration** est la superclasse abstraite de déclarations qui ont des noms, par exemple, *noms de type* ou *noms d'instance*.

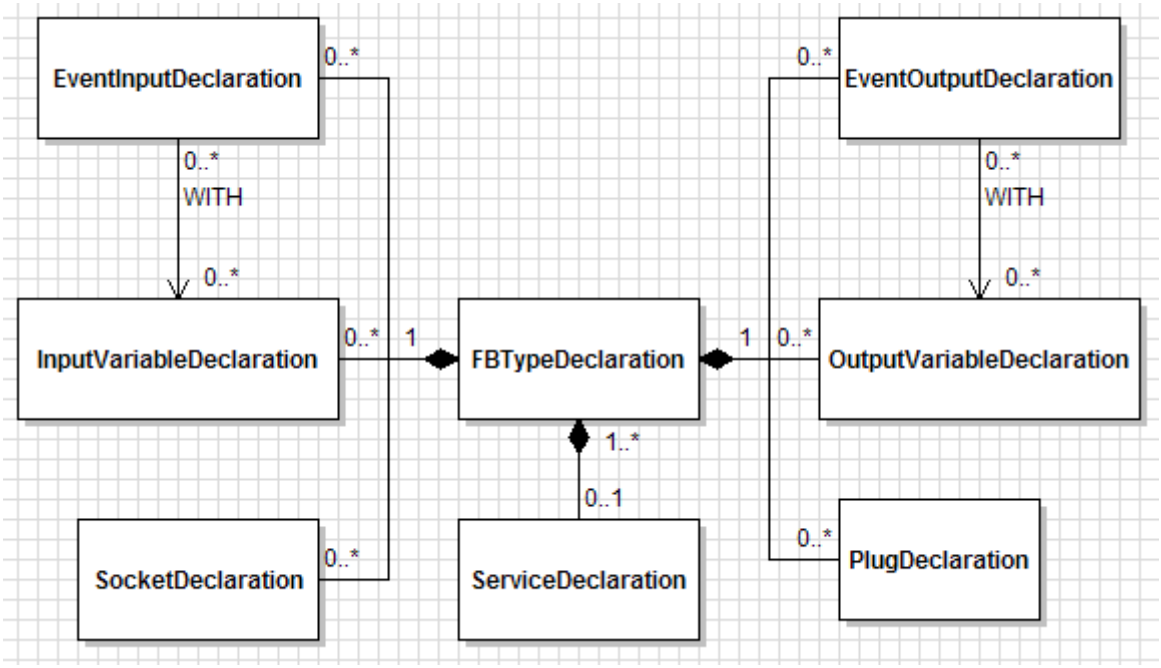


Figure C.5a – Composition

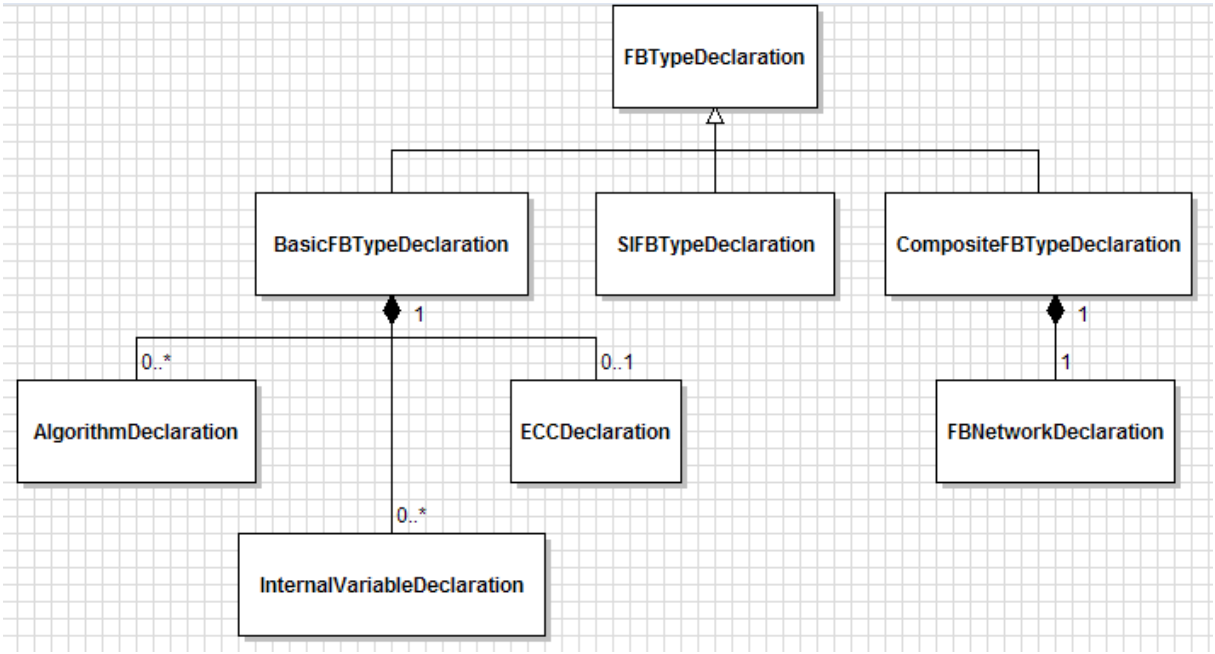


Figure C.5b – Hiérarchie de classes

Figure C.5 – Déclarations des types de blocs fonctionnels

C.3 Modèles IPMCS

La Figure C.6 présente une vue d'ensemble des classes principales dans le système de mesure et commande dans les processus industriels (IPMCS). Les descriptions des classes dans la Figure C.6 et leurs objets correspondants dans le système de support d'ingénierie (ESS) sont donnés dans le Tableau C.4.

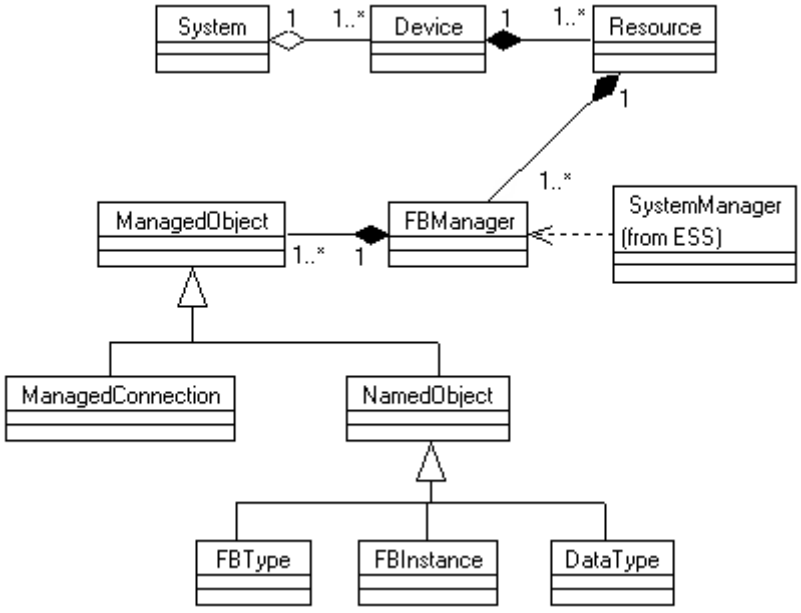


Figure C.6 – Vue d'ensemble du système IPMCS

Tableau C.4 – Classes des IPMCS

Classe des IPMCS	Description	Classe ESS correspondante
DataType	Une instance de cette classe est un *type de données*.	**DataTypeDeclaration**
Device	Une instance de cette classe représente un *équipement*.	**DeviceConfiguration**
FBInstance	Une instance de cette classe est une *instance de bloc fonctionnel*.	**FBInstanceDeclaration**
FBManager	Une instance de cette classe fournit les services de gestion définis à l'Article 6.	**SystemManager**
FBType	Une instance de cette classe est un *type de bloc fonctionnel*.	**FBTypeDeclaration**
ManagedConnection	Des instances de cette classe peuvent être accessibles à une instance de la classe **FBManager** utilisant la combinaison de la source et de la destination comme clé unique.	**ConnectionDeclaration**
ManagedObject	Il s'agit de la superclasse abstraite d'objets qui sont gérés par une instance de la classe **FBManager**. De tels objets peuvent avoir des attributs fournisseur (vendeur, programmateur, etc.) et version (numéro de version, date, etc.) pour aider à la gestion.	None
NamedObject	Il s'agit de la superclasse abstraite d'objets qui peuvent être accessibles par le nom par une instance de la classe **FBManager**.	**NamedDeclaration**
Resource	Une instance de cette classe représente une *ressource*.	**ResourceConfiguration**
System	Une instance de cette classe représente un *Système* de mesure et commande dans les processus industriels (IPMCS).	**SystemConfiguration**

La Figure C.7 montre les relations entre les éléments d'une *instance de bloc fonctionnel* et son *type de bloc fonctionnel* associé.

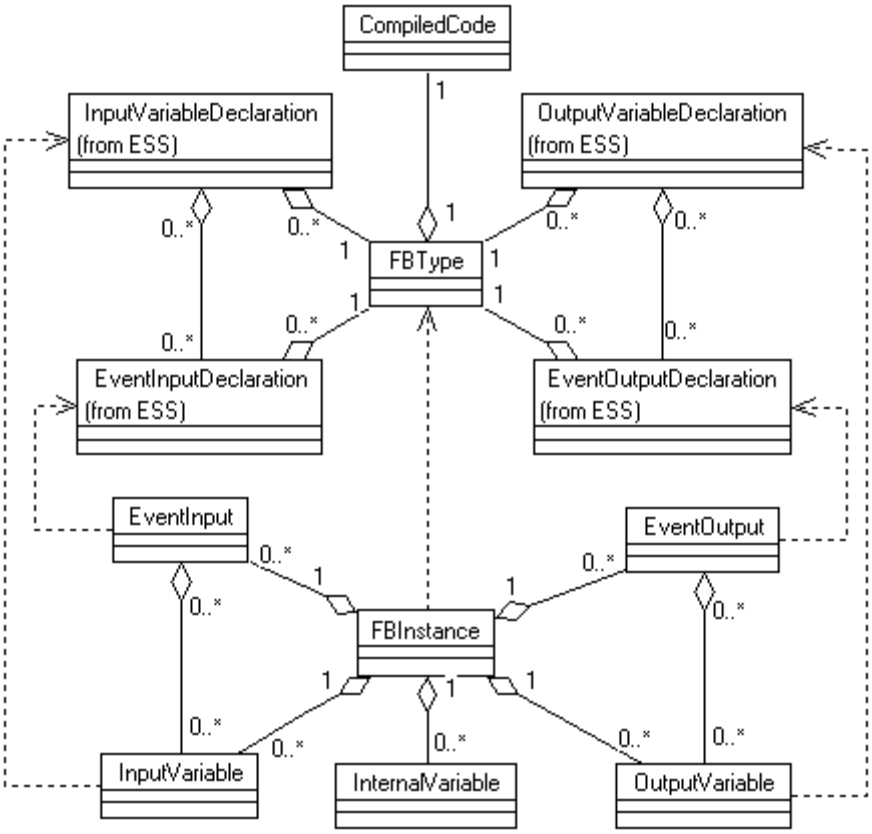


Figure C.7 – Types et instances de bloc fonctionnel

Annexe D
(informative)

Relation à la CEI 61131-3

D.1 Généralités

Les *fonctions* et les *blocs fonctionnels* tels que définis dans la CEI 61131-3 peuvent être utilisés pour la *déclaration* d'*algorithmes* pour les *types de bloc fonctionnel de base* tels que spécifiés en 5.2.1. L'Article D.2 définit des règles pour la conversion des *fonctions* et des *types de bloc fonctionnel* de la CEI 61131-3 en *bloc fonctionnel de type simples* afin qu'ils puissent être utilisés dans la spécification d'*applications* et de *types de ressource*. L'Article D.3 définit des versions événementielles des fonctions et des blocs fonctionnels de la CEI 61131-3 pour les mêmes utilisations.

D.2 Blocs fonctionnels "simples"

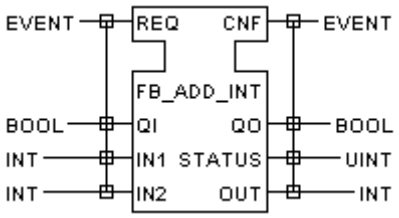
Comme illustré à la Figure D.1, les fonctions et les blocs fonctionnels de la CEI 61131-3 peuvent être convertis en blocs fonctionnels "simples" conformément aux règles suivantes:

a) Les blocs fonctionnels simples sont représentés comme des *blocs fonctionnels interface de service* pour des applications déclenchées par une application telles que montrées à la Figure 21 a).

b) Le *nom de type* du type de bloc fonctionnel simple est le nom du type de fonction ou du type de bloc fonctionnel CEI 61131-3 converti avec le préfixe `FB_` (par exemple, `FB_ADD_INT` à la Figure D.1). Le préfixe `F_` à la place de `FB_` peut être facultativement utilisé pour des types de bloc fonctionnel simple qui sont le résultat des conversions de *fonctions* de la CEI 61131-3.

c) Les variables d'*entrée* et de *sortie* et leurs *types de données* correspondants sont les mêmes que les variables d'entrée et de sortie correspondantes du type de fonction ou du type de bloc fonctionnel de la CEI 61131-3 qui a été converti.

d) L'entrée d'événements `INIT` et la sortie d'événements `INITO` sont utilisées avec des types de bloc fonctionnel simple qui ont été convertis à partir de *types de bloc fonctionnel* de la CEI 61131-3, mais elles ne sont pas utilisées avec des types de bloc fonctionnel simple qui ont été convertis à partir de *fonctions* de la CEI 61131-3.



NOTE Une déclaration textuelle complète de ce type de bloc fonctionnel est donnée dans l'Annexe F.

Figure D.1 – Exemple de type de bloc fonctionnel "simple"

Le comportement des *instances* de *types* bloc fonctionnel simple est conforme aux règles suivantes:

e) L'initialisation est comme spécifié en 2.4.2 de la CEI 61131-3:2003 pour les *variables* et comme spécifié en 2.6 de la CEI 61131-3:2003 pour les éléments graphes séquentiels de fonction (SFC).

f) L'occurrence d'une primitive du service INIT+ équivaut à une initialisation en «redémarrage à froid» telle que définie dans les paragraphes susmentionnés de la CEI 61131-3:2003, suivie d'une primitive de service INITO+ avec une valeur de STATUS de zéro (0).

g) L'occurrence d'une primitive de service INIT- ou REQ- n'a aucun effet, sauf d'induire une primitive de service INITO- ou CNF- avec une valeur de STATUS égale à un (1).

h) L'occurrence d'une primitive de service REQ+ entraîne l'*exécution* de l'*algorithme* spécifié dans le corps du bloc fonctionnel, conformément aux règles données dans la CEI 61131-3 pour le langage dans lequel l'algorithme est programmé.

i) L'exécution réussie de l'algorithme en réponse à une primitive REQ+ donne lieu à une primitive CNF+ avec une valeur de STATUS égale à zéro (0).

j) S'il se produit une erreur au cours de l'exécution de l'algorithme, le résultat est une primitive CNF- avec une valeur de STATUS déterminée conformément au Tableau D.1.

Tableau D.1 – Sémantique des valeurs de STATUS

Valeur	Sémantique
0	Fonctionnement normal
1	Propagation de `INIT-` ou de `REQ-`
2	Erreur de conversion de type
3	Le résultat numérique dépasse la plage pour le type de données
4	Division par zéro
5	Sélecteur (`K`) hors plage pour la fonction `MUX`
6	Position du caractère spécifiée non valide
7	Le résultat dépasse la longueur maximale de la chaîne
8	Simultanément vraies, transactions non classées par ordre de priorité, dans une divergence de sélection
9	Erreur de conflit du contrôle d'action
10	Retour issu d'une fonction sans valeur assignée
11	L'itération ne se termine pas
12	Valeur d'indice non valide
13	Erreur de taille de matrice

D.3 Fonctions et blocs fonctionnels événementiels

Des *fonctions* de la CEI 61131-3 peuvent être converties en blocs fonctionnels pour un usage efficace dans des systèmes événementiels conformément aux règles données à l'Article D.2 avec les modifications suivantes:

a) le *nom de type* du type de bloc fonctionnel événementiel est le même que le nom de la fonction CEI 61131-3 convertie avec le préfixe complémentaire `E_`, par exemple, `E_ADD_INT`;

b) une primitive `CNF+` ou `CNF-` ne suit pas l'exécution de l'algorithme, à moins qu'une telle exécution donne lieu à une valeur modifiée de la sortie de fonction.

NOTE Si le «chaînage en cascade» des sorties `CNF` aux entrées `REQ` est utilisé pour mettre en œuvre une séquence de calculs, la séquence s'arrêtera au premier point où une valeur de sortie ne change pas.

En général, étant donné que les *blocs fonctionnels* de la CEI 61131-3 ont des informations d'états internes, de tels blocs doivent être spécialement convertis pour être utilisables dans des systèmes événementiels. Par exemple, le bloc fonctionnel `E_DELAY` montré dans le Tableau A.1 peut être utilisé par un grand nombre des fonctions de retard fournies par les

blocs fonctionnels temporisateurs de la CEI 61131-3. Un exemple d'une conversion du bloc fonctionnel `CTU` de la norme CEI 61131-3 est donné comme Caractéristique n°18 dans le Tableau A.1.

D.4 Conformité à la CEI 61131-3

Les mises en œuvre de la présente norme doivent se conformer aux exigences des paragraphes 1.5.1, 2.1, 2.2, 2.3 et 2.4 de la CEI 61131-3:2003 et aux éléments associés de l'Annexe B de la CEI 61131-3:2003 pour la syntaxe et la sémantique de représentation textuelle d'éléments communs, avec les exceptions et les extensions notées à l'Article D.5.

Lorsque des productions syntaxiques ne sont pas données pour les symboles non terminaux dans l'Annexe B, les productions syntaxiques correspondantes données dans l'Annexe B de la CEI 61131-3:2003 doivent s'appliquer.

D.5 Exceptions

Les mises en œuvre de la présente norme **ne doivent pas** utiliser la notation de *variable directement représentée* définie en 2.4.1.1 de la CEI 61131-3:2003 et les caractéristiques associées dans d'autres paragraphes. Cependant, un *libellé* de type `STRING` ou `WSTRING`, contenant une chaîne dont la syntaxe et la sémantique correspondent à la notation de variable directement représentée, peut être utilisé comme *paramètre* d'un *bloc fonctionnel interface de service* qui fournit un accès à la variable correspondante.

D.6 Interfonctionnement avec des dispositifs de commande programmables

D.6.1 Vue d'ensemble

Un dispositif de commande programmable peut agir comme un *serveur* tel que défini dans la CEI 61131-5, par rapport à un *équipement* tel que défini dans la présente norme, agissant comme *client* tel que défini dans la CEI 61131-5. Ces services sont fournis en utilisant les moyens définis dans la CEI 61131-5 et sont accessibles à partir d'un équipement CEI 61499 utilisant les instances des *types de bloc fonctionnel* spécifiés dans l'Annexe D. Ces types de blocs fonctionnels sont modélisés comme des *types bloc fonctionnel de communication* tels que définis dans la présente norme.

L'équipement client CEI 61499 peut exister sur un réseau de communication avec le dispositif de commande programmable agissant comme serveur ou peut être un sous-système spécifique au réalisateur au sein du «bloc de traitement principal» du dispositif de commande programmable, tel qu'illustré à la Figure 4 de la CEI 61131-5:2000. Dans un cas comme dans l'autre, l'interaction entre l'équipement client CEI 61499 et le bloc de traitement principal est modélisée comme se produisant sur une ou plusieurs *connexions de communication* telles que définies dans la CEI 61499-1, en utilisant les instances des types de bloc fonctionnel définis dans l'Annexe D.

D.6.2 Conventions de service

À l'exception des extensions définies dans l'Annexe D, les conventions pour nommer des variables et des événements d'entrée et de sortie et pour décrire les *services* (tels que définis dans la présente norme) fournis par des instances des types de bloc fonctionnel décrites dans l'Annexe D, sont telles que définies dans la CEI 61499-1 pour les descriptions des *types bloc fonctionnel interface de service* et des *types bloc fonctionnel de communication*.

Pour les besoins de l'Annexe D, l'entrée `PARAMS` de type `ANY` défini dans la présente norme est remplacée par une entrée `ID` de type `WSTRING`. Le contenu de cette chaîne spécifie une représentation **dépendante de la mise en œuvre** du chemin allant vers la *variable* concernée du serveur.

EXEMPLE 1 Dans le cas où l'équipement client CEI 61499 se situe dans la proximité logique du serveur CEI 61131, il peut suffire de nommer simplement le *chemin d'accès* CEI 61131-3 par la variable souhaitée dans l'entrée ID, par exemple "CELL_1.CHARLIE" dans l'exemple montré à la Figure 19a de la CEI 61131-3:2003.

EXEMPLE 2 Dans le cas où l'équipement client CEI 61499 est relié à distance au serveur 61131-3 par un réseau de communication, il peut être possible d'utiliser l'entrée ID pour encapsuler un identificateur de ressource universel (URI) pour spécifier le chemin d'accès souhaité, par exemple, "http://192.168.0.1:61131/CELL_1.CHARLIE".

NOTE Lorsque cela est pris en charge par une mise en œuvre, l'entrée ID peut spécifier un chemin d'accès vers une variable de statut, tel que les chemins d'accès prédéfinis P_PCSTATUS et P_PCSTATE spécifiés dans la CEI 61131-5.

Lorsque cela est utilisé, le contenu de l'entrée TYPE d'un type de bloc fonctionnel défini dans l'Annexe D spécifie le nom du *type de données* des données (SD ou RD) transférées. Il peut s'agir du nom d'un type de données élémentaire tel que "BOOL" ou un type de données dérivé tel que "ANALOG_ 16_INPUT_DATA".

Lorsque cela est utilisé, le contenu de l'entrée TASK d'un type de bloc fonctionnel défini dans l'Annexe D spécifie une représentation **dépendante de la mise en œuvre** du chemin allant à la *tâche* concernée dans le serveur.

EXEMPLE 3 Dans le cas où l'équipement client CEI 61499 se situe dans la proximité logique d'un serveur CEI 61131-3 configuré comme montré à la Figure 19 a) de la CEI 61131-3:2003, un chemin allant à la tâche nommée SLOW_1 dans la ressource STATION_1 pourrait être représenté comme "CELL_1.STATION_1.SLOW_1".

Les valeurs de la sortie STATUS des types de bloc fonctionnel définis en D.6.3 sont telles que données dans le Tableau 24 de la CEI 61131-5:2000.

D.6.3 Types de bloc fonctionnel

D.6.3.1 READ

Une instance du type de bloc fonctionnel READ montré sous forme graphique à la Figure D.2 et sous forme textuelle dans le Tableau D.2 peut être utilisée par un équipement client CEI 61499 pour lire les valeurs de variables programme ou le statut dans le serveur CEI 61131-3.

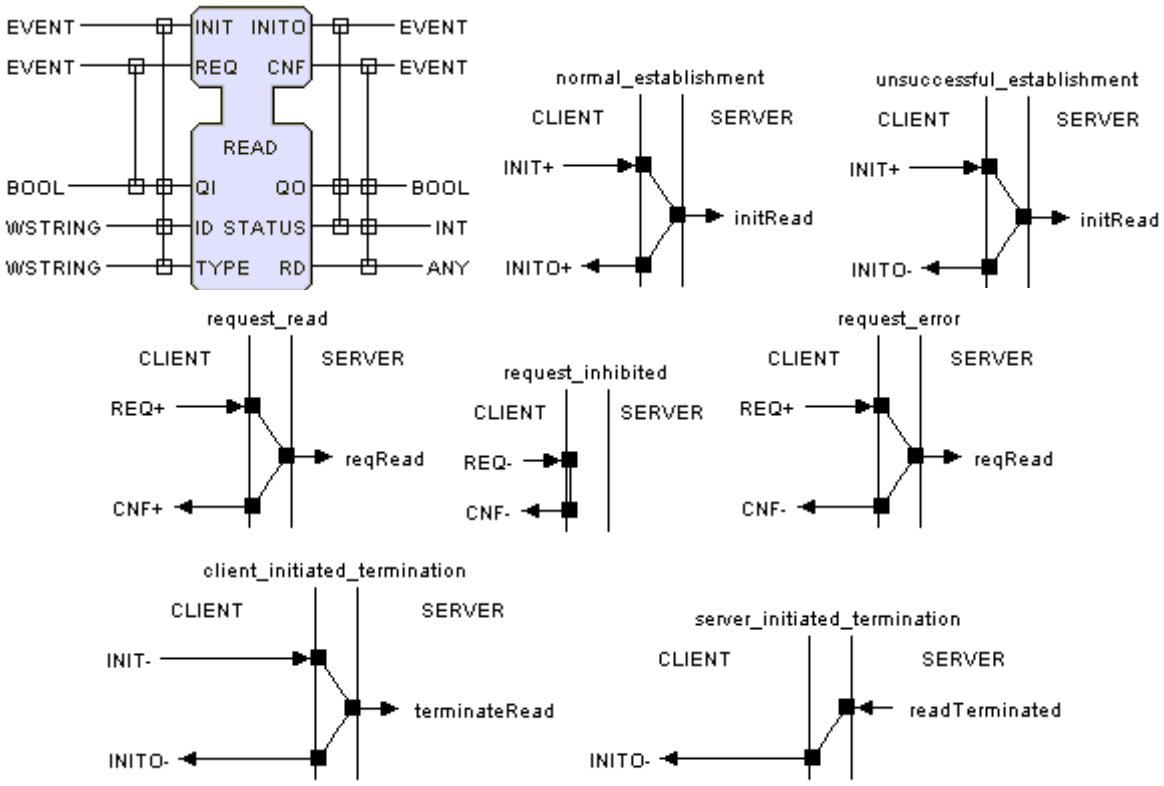


Figure D.2 – Type de bloc fonctionnel READ

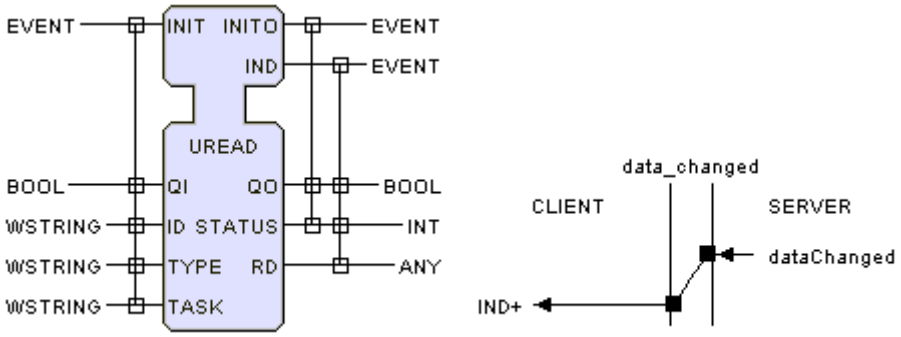
Tableau D.2 – Code source du type de bloc fonctionnel READ

```
FUNCTION_BLOCK READ (* Lire la variable statut ou programme du serveur *)
EVENT_INPUT
  INIT WITH QI,ID,TYPE; (* Initialiser/Arrêter le service *)
  REQ WITH QI; (* Demande de service *)
END_EVENT

EVENT_OUTPUT
  INITO WITH QO,STATUS; (* Initialiser/Arrêter «Confirm» *)
  CNF WITH QO,STATUS,RD; (* Confirmation du service demandée *)
END_EVENT

VAR_INPUT
  QI: BOOL; (* Qualificateur d'entrée d'événements *)
  ID: WSTRING; (* Chemin allant vers la variable devant être lue *)
  TYPE: WSTRING; (* Type de données de variable RD *)
END_VAR

VAR_OUTPUT
  QO: BOOL; (* 1=Fonctionnement normal, 0= Fonctionnement anormal *)
  STATUS: INT;
  RD: ANY; (* Données variables issues de l'équipement CEI 61131 *)
END_VAR

SERVICE CLIENT/SERVER
SEQUENCE normal_establishment
  CLIENT.INIT+(ID,TYPE) -> SERVER.initRead(ID,TYPE) -> CLIENT.INITO+();
END_SEQUENCE

SEQUENCE unsuccessful_establishment
  CLIENT.INIT+(ID,TYPE) -> SERVER.initRead(ID,TYPE) -> CLIENT.INITO-(STATUS);
END_SEQUENCE

SEQUENCE request_read
  CLIENT.REQ+() -> SERVER.reqRead(ID) -> CLIENT.CNF+(RD);
END_SEQUENCE

SEQUENCE request_inhibited
  CLIENT.REQ-() -> CLIENT.CNF-(STATUS);
END_SEQUENCE

SEQUENCE request_error
  CLIENT.REQ+() -> SERVER.reqRead(ID) -> CLIENT.CNF-(STATUS);
END_SEQUENCE

SEQUENCE client_initiated_termination
  CLIENT.INIT-() -> SERVER.terminateRead(ID) -> CLIENT.INITO-(STATUS);
END_SEQUENCE

SEQUENCE server_initiated_termination
  SERVER.readTerminated(ID,STATUS) -> CLIENT.INITO-(STATUS);
END_SEQUENCE
END_SERVICE
END_FUNCTION_BLOCK
```

D.6.3.2 UREAD

Une instance du type de bloc fonctionnel `UREAD` montré sous forme graphique à la Figure D.3 et sous forme textuelle dans le Tableau D.3 peut être utilisée par un équipement client CEI 61499 pour demander une notification asynchrone d'un changement de valeur de variable programme ou de statut auprès d'un serveur CEI 61131-3. La notification est reçue par la sortie d'événements `IND` du bloc à la fin de l'exécution de la tâche spécifiée lorsqu'un changement de la valeur de la variable spécifiée (par rapport à sa valeur à la suite du lancement de l'exécution de la tâche) est détecté.

Une instance de ce type de bloc fonctionnel peut aussi être utilisée pour recevoir une notification de fin de chaque exécution de la tâche spécifiée en laissant non spécifiées les entrées `ID` et `TYPE` du bloc.



NOTE La représentation graphique d'autres séquences de service énumérées dans le Tableau D.3 est similaire à la Figure D.2.

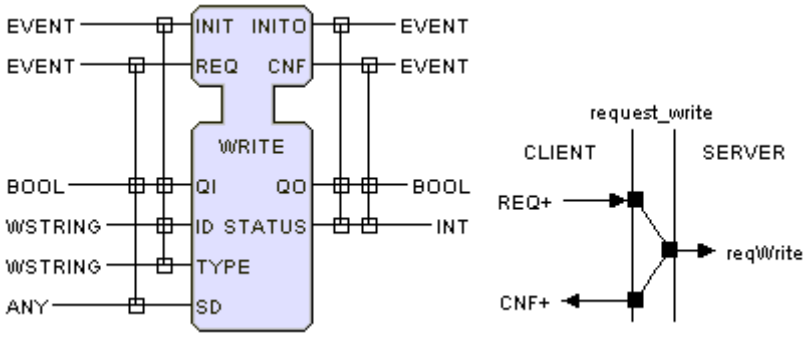
Figure D.3 – Type de bloc fonctionnel UREAD

Tableau D.3 – Code source du type de bloc fonctionnel UREAD

```
FUNCTION_BLOCK UREAD (* Lecture non sollicitée de la variable programme ou statut de
la CEI 61131 *)

EVENT_INPUT
  INIT WITH QI,ID,TASK,TYPE; (* Initialiser/Arrêter le service *)
END_EVENT

EVENT_OUTPUT
  INITO WITH QO,STATUS; (* Initialiser/Arrêter «Confirm» *)
  IND WITH QO,STATUS,RD; (* Indication de la valeur RD modifiée *)
END_EVENT

VAR_INPUT
  QI: BOOL; (* Qualificateur d'entrée d'événements *)
  ID: WSTRING; (* Chemin allant vers la variable devant être lue *)
  TYPE: WSTRING; (* Type de données de variable RD *)
  TASK: WSTRING; (* Chemin vers TASK CEI 61131 déclenchant la lecture sur valeur
modifiée *)
END_VAR

VAR_OUTPUT
  QO: BOOL; (* 1=Fonctionnement normal, 0= Fonctionnement anormal *)
  STATUS: INT;
  RD: ANY; (* Données d'entrée issues de ressource *)
END_VAR

SERVICE CLIENT/SERVER
SEQUENCE normal_establishment
  CLIENT.INIT+(ID,TYPE,TASK) -> SERVER.initURead(ID,TYPE,TASK) -> CLIENT.INITO+();
END_SEQUENCE

SEQUENCE unsuccessful_establishment
  CLIENT.INIT+(ID,TYPE,TASK) -> SERVER.initURead(ID,TYPE,TASK)
    -> CLIENT.INITO-(STATUS);
END_SEQUENCE

SEQUENCE data_changed
  SERVER.dataChanged() -> CLIENT.IND+(RD);
END_SEQUENCE

SEQUENCE client_initiated_termination
  CLIENT.INIT-() -> SERVER.terminateURead() -> CLIENT.INITO-(STATUS);
END_SEQUENCE

SEQUENCE server_initiated_termination
  SERVER.UreadTerminated(ID,STATUS) -> CLIENT.INITO-(STATUS);
END_SEQUENCE
END_SERVICE
END_FUNCTION_BLOCK
```

D.6.3.3 WRITE

Une instance du type de bloc fonctionnel `WRITE` montré sous forme graphique à la Figure D.4 et sous forme textuelle dans le Tableau D.4 peut être utilisée par un équipement client CEI 61499 pour écrire des valeurs de données variables dans un serveur CEI 61131-3.



NOTE La représentation graphique d'autres séquences de service énumérées dans le Tableau D.4 est similaire à la Figure D.2.

Figure D.4 – Type de bloc fonctionnel WRITE

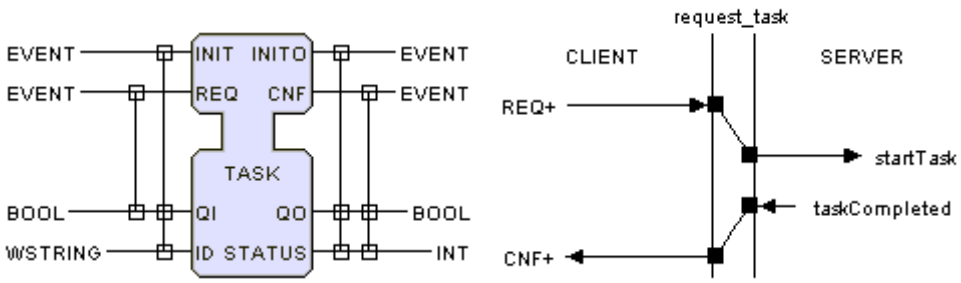
Tableau D.4 – Code source du type de bloc fonctionnel WRITE

```
FUNCTION_BLOCK WRITE (* Écrire une valeur de variable dans un serveur CEI 61131 *)
EVENT_INPUT
  INIT WITH QI,ID,TYPE; (* Initialiser/Arrêter le service *)
  REQ WITH QI,SD; (* Demande de service *)
END_EVENT

EVENT_OUTPUT
  INITO WITH QO,STATUS; (* Initialiser/Arrêter «Confirm» *)
  CNF WITH QO,STATUS; (* Confirmation du service demandée *)
END_EVENT

VAR_INPUT
  QI: BOOL; (* Qualificateur d'entrée d'événements *)
  ID: WSTRING; (* Chemin allant vers la variable devant être lue *)
  TYPE: WSTRING; (* Type de données de variable SD *)
  SD: ANY; (* Valeur de variable à écrire *)
END_VAR

VAR_OUTPUT
  QO: BOOL; (* 1=Fonctionnement normal, 0= Fonctionnement anormal *)
  STATUS: INT;
END_VAR

SERVICE CLIENT/SERVER
SEQUENCE normal_establishment
  CLIENT.INIT+(ID,TYPE) -> SERVER.initWrite(ID,TYPE) -> CLIENT.INITO+();
END_SEQUENCE

SEQUENCE unsuccessful_establishment
  CLIENT.INIT+(ID,TYPE) -> SERVER.initWrite(ID,TYPE) -> CLIENT.INITO-(STATUS);
END_SEQUENCE

SEQUENCE request_write
  CLIENT.REQ+(ID,SD) -> SERVER.reqWrite(ID,SD) -> CLIENT.CNF+();
END_SEQUENCE

SEQUENCE request_inhibited
  CLIENT.REQ-(ID,SD) -> CLIENT.CNF-(STATUS);
END_SEQUENCE

SEQUENCE request_error
  CLIENT.REQ+(ID,SD) -> SERVER.reqWrite(ID,SD) -> CLIENT.CNF-(STATUS);
END_SEQUENCE

SEQUENCE client_initiated_termination
  CLIENT.INIT-() -> SERVER.terminateWrite(ID) -> CLIENT.INITO-(STATUS);
END_SEQUENCE

SEQUENCE server_initiated_termination
  SERVER.writeTerminated(ID,STATUS) -> CLIENT.INITO-(STATUS);
END_SEQUENCE
END_SERVICE
END_FUNCTION_BLOCK
```

D.6.3.4 TASK

Une instance du type de bloc fonctionnel `TASK` montré sous forme graphique à la Figure D.5 et sous forme textuelle dans le Tableau D.5 peut être utilisée par un équipement client CEI 61499 pour demander l'exécution d'une tâche sur un serveur CEI 61131-3.

Lorsqu'une mise en œuvre prend en charge cette caractéristique, aucune valeur n'est configurée pour l'entrée `SINGLE` ou `INTERVAL` du bloc TASK correspondant tel que défini dans le Tableau 50 de la CEI 61131-3:2003; plus exactement, l'exécution de la tâche correspondante est déclenchée comme illustré dans la séquence de service `request_task` montrée à la Figure D.5.



NOTE La représentation graphique d'autres séquences de service énumérées dans le Tableau D.5 est similaire à la Figure D.2.

Figure D.5 – Type de bloc fonctionnel TASK

Tableau D.5 – Code source du type de bloc fonctionnel TASK

```
FUNCTION_BLOCK TASK (* Déclencher une tâche CEI 61131 *)
EVENT_INPUT
  INIT WITH QI,ID; (* Initialiser/Arrêter le service *)
  REQ WITH QI; (* Demande de service *)
END_EVENT

EVENT_OUTPUT
  INITO WITH QO,STATUS; (* Initialiser/Arrêter «Confirm» *)
  CNF WITH QO,STATUS; (* Confirmation du service demandée *)
END_EVENT

VAR_INPUT
  QI: BOOL; (* Qualificateur d'entrée d'événements *)
  ID: WSTRING; (* Chemin allant à la tâche devant être déclenchée *)
END_VAR

VAR_OUTPUT
  QO: BOOL; (* 1=Fonctionnement normal, 0= Fonctionnement anormal *)
  STATUS: INT;
END_VAR

SERVICE CLIENT/SERVER
SEQUENCE normal_establishment
  CLIENT.INIT+(ID) -> SERVER.initTask(ID) -> CLIENT.INITO+();
END_SEQUENCE

SEQUENCE unsuccessful_establishment
  CLIENT.INIT+(ID) -> SERVER.init(ID) -> CLIENT.INITO-(STATUS);
END_SEQUENCE

SEQUENCE request_task
  CLIENT.REQ+(ID) -> SERVER.reqTask(ID) -> CLIENT.CNF+();
END_SEQUENCE

SEQUENCE request_inhibited
  CLIENT.REQ-() -> CLIENT.CNF-(STATUS);
END_SEQUENCE

SEQUENCE request_error
  CLIENT.REQ+(ID) -> SERVER.reqTask(ID) -> CLIENT.CNF-(STATUS);
END_SEQUENCE

SEQUENCE client_initiated_termination
  CLIENT.INIT-() -> SERVER.terminateTask(ID) -> CLIENT.INITO-(STATUS);
END_SEQUENCE

SEQUENCE server_initiated_termination
  SERVER.taskTerminated(ID,STATUS) -> CLIENT.INITO-(STATUS);
END_SEQUENCE
END_SERVICE
END_FUNCTION_BLOCK
```

D.6.4 Conformité

Les spécifications données dans l'Annexe D peuvent être référencées dans les profils de conformité selon les règles données dans la CEI 61499-4.

Lorsqu'un système de dispositif de commande programmable conforme à la CEI 61131-3 prend en charge l'interopérabilité avec un ou plusieurs types de bloc fonctionnel de la CEI 61499 définis dans l'Annexe D, il convient qu'il inclue dans sa liste de caractéristiques prises en charge une référence aux caractéristiques prises en charge issues du Tableau D.6, et il convient qu'il inclue des spécifications des valeurs pour les caractéristiques et paramètres spécifiques à une mise en œuvre tels que définis respectivement en 8.1 et en 8.2 de la CEI 61131-5:2000.

Tableau D.6 – Caractéristiques d'interopérabilité de la CEI 61499

No.	Description
1	Type de bloc fonctionnel READ
2	Type de bloc fonctionnel UREAD
3	Type de bloc fonctionnel WRITE
4	Type de bloc fonctionnel TASK

Annexe E
(informative)

Echange d'informations

E.1 Utilisation des moyens de la couche application

Le paragraphe 7.1.3.2 de l'ISO/CEI 7498-1:1994 identifie un certain nombre de moyens fournis par des *entités d'application* (c'est-à-dire: *entités* de la *couche application*) pour permettre à des *processus d'application* d'échanger des informations. Pour fournir ces moyens, les entités d'application utilisent des *protocoles d'application* et des *services de présentation.* Les blocs fonctionnels de communication définis à l'Article E.2 peuvent utiliser ces moyens, lorsqu'ils sont fournis par des entités d'application appropriées, de la façon suivante:

a) Les blocs fonctionnels de communication utilisent les moyens de *transfert d'informations* fournis par des entités d'application pour assurer la *synchronisation d'applications qui coopèrent*, représentées par les événements `REQ`, `CNF`, `IND` et `RSP` et transférer les données représentées par des entrées `SD` et des sorties `RD`.

b) Les moyens suivants peuvent être utilisés au cours de l'initialisation du service par les événements `INIT` et `INITO`, en utilisant les éléments de la structure de données `PARAMS` selon les besoins:

- identification des partenaires de communications attendus;
- détermination de la qualité acceptable du service;
- accord sur la responsabilité de la reprise sur erreur;
- accord sur des aspects de sécurité;
- identification de la syntaxe abstraite.

c) Les moyens servant à la *sélection du mode de dialogue* peuvent être utilisés par les types spécifiques de bloc fonctionnel, par exemple, par un `SUBSCRIBER` (c'est-à-dire: abonné) pour s'assurer qu'il interagit correctement avec un `PUBLISHER` (c'est-à-dire: éditeur).

Beaucoup des moyens énumérés ci-dessous ne sont pas fournis par des entités d'application des systèmes de mesure et commande dans les processus industriels (les IPMCS). Dans ce cas, les blocs fonctionnels de communication doivent mettre en œuvre des moyens équivalents pour fournir les services requis.

En particulier, les services de présentation ne sont parfois pas fournis par les entités d'application IMPCS. Par conséquent, afin de faciliter la mise en œuvre de ces services par des blocs fonctionnels de communication, les syntaxes de transfert tant pour le transfert d'informations que pour la gestion d'application sont définies à l'Article E.3.

NOTE 1 Voir l'ISO/CEI 7498-1 pour des définitions de termes utilisés dans cette annexe, mais pas définis dans la présente norme.

NOTE 2 Une *ressource* est un «processus d'application» tel que défini dans l'ISO/CEI 7498-1.

NOTE 3 Le contenu de l'Annexe E pourrait être considéré normatif par le fait que les profils de conformité tels que définis dans la CEI 61499-4, dans d'autres normes et spécifications sont susceptibles de spécifier un contexte dans lequel l'ensemble ou une partie de ses dispositions sont employés.

E.2 Types de bloc fonctionnel de communication

E.2.1 Généralités

Ce Paragraphe définit des *types de bloc fonctionnel de communication* génériques pour les transactions *unidirectionnelles* et *bidirectionnelles*. Il convient que les personnalisations **dépendantes de la mise en œuvre** de ces types respectent les règles suivantes:

a) la mise en œuvre doit spécifier les types de données et la sémantique des valeurs des entrées de données et des sorties de données de chaque type de bloc fonctionnel de ce genre;

b) la mise en œuvre doit spécifier le traitement du transfert anormal de données;

c) la mise en œuvre doit spécifier toutes les éventuelles différences entre le comportement d'instances de ces types de bloc fonctionnel et les comportements spécifiés à l'Article E.2.

E.2.2 Blocs fonctionnels pour les transactions unidirectionnelles

Les Figures E.1 à E.4 donnent des déclarations de types et des séquences de primitives de service typiques des blocs fonctionnels qui assurent des *transactions unidirectionnelles* sur une *connexion de communication*. Une telle connexion consiste en une *instance* `PUBLISH` et une ou plusieurs instances du type `SUBSCRIBE`.

NOTE 1 Des spécifications textuelles complètes de ces types de bloc fonctionnel ne sont pas données dans l'Annexe F.

NOTE 2 Les types de données et la sémantique de l'entrée `PARAMS` et de la sortie `STATUS` sont **dépendants de la mise en œuvre**.

NOTE 3 Le nombre (`m`) et les types des données reçues `RD_1,…,RD_m` correspondent au nombre et aux types des données émises `SD_1,…,SD_m`.

NOTE 4 Les moyens par lesquels les connexions de communications sont établies ne relèvent pas du domaine d'application de la présente norme.

NOTE 5 Le transfert de données pourrait être requis afin de déterminer si, oui ou non, `RD_1,…,RD_m` respectent les contraintes exprimées dans la Note 3.

NOTE 6 Les syntaxes de transfert définies à l'Article E.3 sont susceptibles d'être utilisées pour effectuer la détermination décrite dans la Note 5.

NOTE 7 Le traitement du transfert anormal de données est **dépendant de la mise en œuvre.**

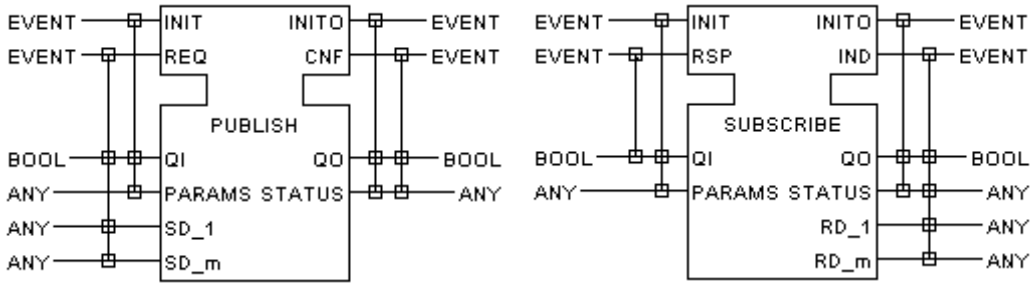


Figure E.1 – Spécifications du type pour les transactions unidirectionnelles

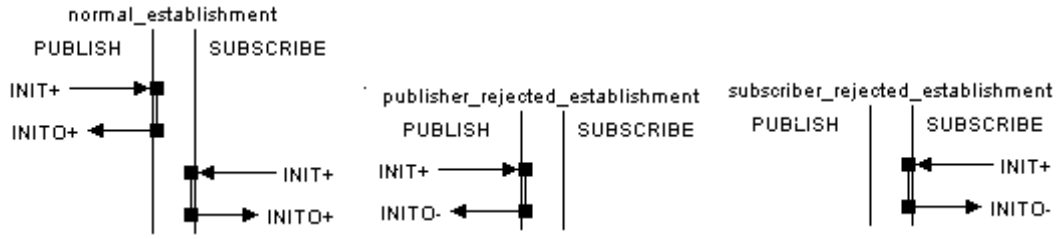


Figure E.2 – Établissement de connexion pour les transactions unidirectionnelles

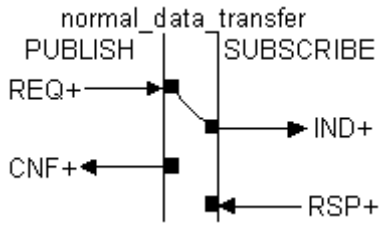


Figure E.3 – Transfert unidirectionnel normal de données

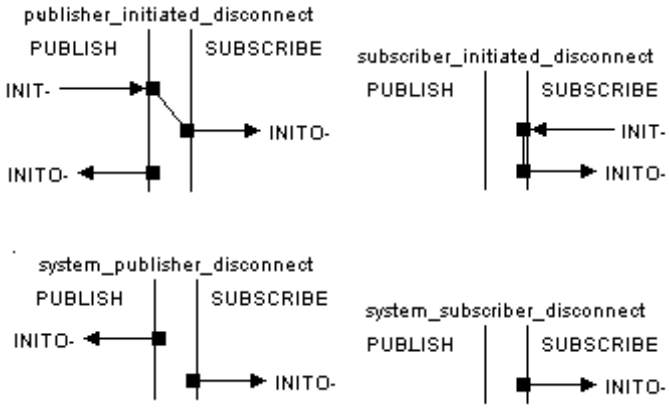


Figure E.4 – Libération de connexion pour le transfert unidirectionnel de données

E.2.3 Blocs fonctionnels pour les transactions bidirectionnelles

Les Figures E.5 à E.8 donnent des déclarations de types et des séquences de primitives de service des blocs fonctionnels qui assurent des *transactions bidirectionnelles* sur une *connexion de communication*. Une telle connexion comprend une instance de type CLIENT et une instance de type SERVER.

NOTE 1 Des spécifications textuelles complètes de ces types de bloc fonctionnel ne sont pas données dans l'Annexe F.

NOTE 2 Les types de données et la sémantique de l'entrée `PARAMS` et de la sortie `STATUS` sont **dépendants de la mise en œuvre**.

NOTE 3 Le nombre (`m`) et les types des données reçues `RD_1,…,RD_m` correspondent au nombre et aux types des données émises `SD_1,…,SD_m`.

NOTE 4 Le nombre (`n`) et les types des données reçues `RD_1,…,RD_n` correspondent au nombre et aux types des données émises `SD_1,…,SD_n`.

NOTE 5 Le transfert de données peut être requis afin de déterminer si, oui ou non, `RD_1,…,RD_m` et `RD_1,…,RD_n` respectent les contraintes exprimées dans les Notes 3 et 4.

NOTE 6 Les syntaxes de transfert définies à l'Article E.3 peuvent être utilisées pour effectuer la détermination décrite dans la Note 5.

NOTE 7 Le traitement du transfert anormal de données est **dépendant de la mise en œuvre.**

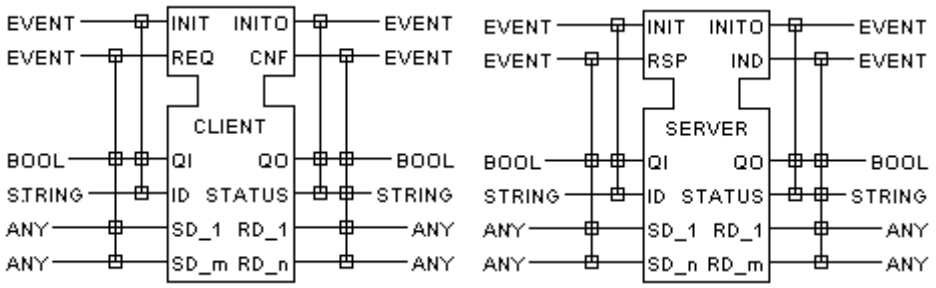


Figure E.5 – Spécifications du type pour les transactions bidirectionnelles

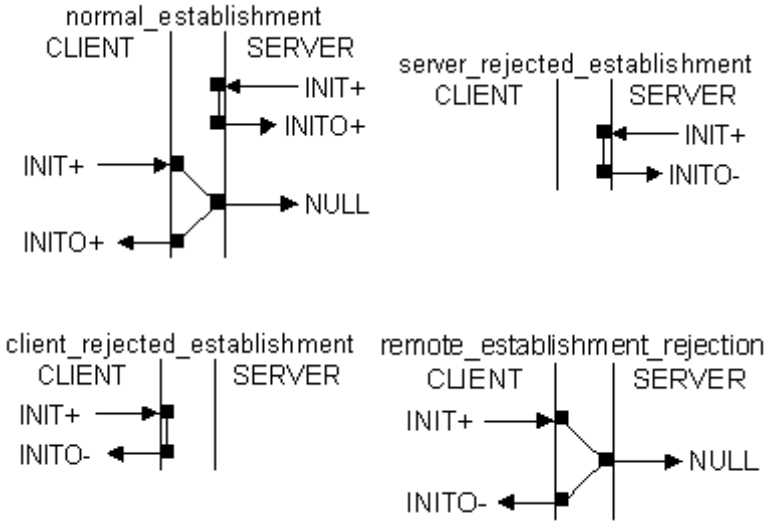


Figure E.6 – Établissement de connexion pour une transaction bidirectionnelle

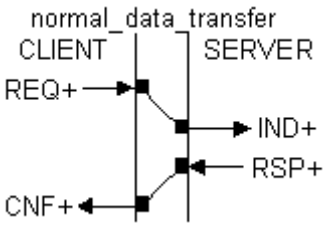


Figure E.7 – Transfert de données bidirectionnel

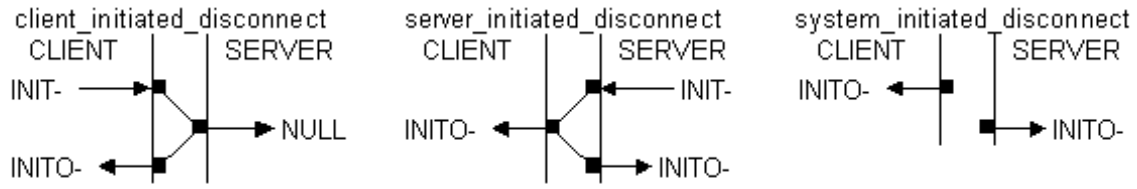


Figure E.8a – Déclenchée par le client

Figure E.8b – Déclenchée par le serveur

Figure E.8c – Déclenchée par le système

Figure E.8 – Libération de connexion dans le transfert bidirectionnel de données

E.3 Syntaxe de transfert

E.3.1 Contexte

Une syntaxe de transfert est définie en termes de *syntaxe abstraite* décrivant les types des données devant être transférées et un jeu de *règles de codage* pour la représentation codée d'instances des types de données ainsi définis.Le paragraphe E.3.2 utilise la Notation de syntaxe abstraite numéro un (ASN.1), comme défini dans l'ISO/CEI 8824-1, pour définir la syntaxe IEC61499-FBDATA pour le transfert de données.

Deux jeux de règles de codage sont donnés dans l'Annexe E:

a) Le paragraphe E.3.3.1 définit des règles de codage BASIC, en utilisant les règles définies dans l'ISO/CEI 8825-1.

b) Le paragraphe E.3.3.2 utilise les caractéristiques spéciales des types de données dans la syntaxe IEC61499-FBDATA pour obtenir un jeu de règles de codage COMPACT selon les principes suivants:

- Lorsque le nombre des «octets de contenu» est fixe, les «octets de longueur» ne sont pas utilisés dans le codage.
- Des codages spéciaux sont utilisés pour réduire au maximum le nombre d'octets et l'effort de codage/décodage requis pour les types de longueur fixe.
- Les «octets d'identificateur» ne sont pas utilisés pour les éléments individuels des types de données STRUCT et ARRAY, car le type de chaque élément est fixé dans la *déclaration de types* correspondante.

E.3.2 Syntaxe abstraite IEC61499-FBDATA

La syntaxe de transfert obtenue en appliquant les règles de codage COMPACT en E.3.3.2 à la syntaxe abstraite en E.3.2 est recommandée pour:

- transférer des valeurs des entrées `SD` d'un *bloc fonctionnel de communication* jusqu'aux sorties `RD` du/des bloc(s) fonctionnel(s) de communication à l'extrémité opposée d'une *connexion de communication*;
- déterminer si, oui ou non, les contraintes sur le nombre et le type correspondants de variables entre les entrées `SD` et les sorties `RD` sont satisfaites comme noté dans les Figures E.1 et E.5.

L'utilisation de la syntaxe abstraite définie en E.3.2 pour le transfert de données exprimées sous forme de *libellés* et de valeurs de *variables* assujettie aux règles sémantiques suivantes (RULES):

a) Lorsque le nom d'un type de données dans ce module (par exemple, `BOOL`) correspond au nom d'un type de données défini dans la CEI 61131-3, la définition de type donnée est destinée au transfert de données du type de données correspondant de la CEI 61131-3.

b) Les valeurs de «VisibleString» pour les types de données `DATE` et `TIME_OF_DAY` sont restreintes à la syntaxe textuelle pour ces types de données tels que définis dans la CEI 61131-3.

c) La notation `[typeID]` implique que l'étiquette des données consiste en la valeur de l'`étiquette ASN.1` du type de données dérivé correspondant, établi comme spécifié dans l'Annexe A de la CEI 61499-2:2005 par un autre moyen ne relevant pas du domaine d'application de la présente norme.

d) La valeur d'un élément `EnumeratedData` consiste en la position cardinale (débutant à zéro) de l'identificateur correspondant dans la séquence d'identificateurs définis pour le type de données «enumerated» (énuméré) correspondant, établi comme spécifié dans la CEI 61131-3.

e) Le type spécifique d'un élément `SubrangeData` est comme pour son *type de données «subrange»* particulier, déclaré comme spécifié dans la CEI 61131-3.

f) Le type des éléments d'un élément de données ARRAY est établi comme spécifié pour les *types de données «array»* dans la CEI 61131-3.

g) Les types des éléments d'un élément de données STRUCT sont établis comme spécifié pour les *types de données structurées* dans la CEI 61131-3.

ASN.1 MODULE

```
IEC61499-FBDATA DEFINITIONS::=

BEGIN

EXPORTS FBDataSequence, FBData, ElementaryData, BOOL, FixedLengthInteger,
     FixedLengthReal, TIME, AnyDate, AnyString, FixedLengthBitString,
     SignedInteger, UnsignedInteger, REAL, LREAL, DATE, TIME_OF_DAY,
     DATE_AND_TIME, STRING, WSTRING, BYTE, WORD, DWORD, LWORD,
     DirectlyDerivedData, EnumeratedData, SubrangeData, ARRAY, STRUCT;

FBDataSequence::= [APPLICATION 23] IMPLICIT SEQUENCE OF FBData

FBData::= CHOICE{ElementaryData, DerivedData}

ElementaryData::= CHOICE{
     BOOL,
     FixedLengthInteger,
     FixedLengthReal,
     TIME,
     AnyDate,
     AnyString,
     FixedLengthBitString}

FixedLengthInteger::= CHOICE{SignedInteger, UnsignedInteger}

SignedInteger::= CHOICE{SINT, INT, DINT, LINT}

UnsignedInteger::= CHOICE{USINT, UINT, UDINT, ULINT}

FixedLengthReal::= CHOICE{REAL, LREAL}

AnyDate::= CHOICE{DATE, TIME_OF_DAY, DATE_AND_TIME}

AnyString::= CHOICE{STRING, WSTRING}

FixedLengthBitString::= CHOICE{BYTE, WORD, DWORD, LWORD}

BOOL::= CHOICE{BOOL0, BOOL1}

BOOL0::= [APPLICATION 0] IMPLICIT NULL

BOOL1::= [APPLICATION 1] IMPLICIT NULL

SINT::= [APPLICATION 2] IMPLICIT INTEGER(-128..127)

INT::= [APPLICATION 3] IMPLICIT INTEGER(-32768..32767)

DINT::= [APPLICATION 4] IMPLICIT INTEGER(-2147483648..2147483647)

LINT::= [APPLICATION 5]
     IMPLICIT INTEGER(-9223372036854775808..9223372036854775807)

USINT::= [APPLICATION 6] IMPLICIT INTEGER(0..255)

UINT::= [APPLICATION 7] IMPLICIT INTEGER(0..65535)

UDINT::= [APPLICATION 8] IMPLICIT INTEGER(0..4294967295)

ULINT::= [APPLICATION 9] IMPLICIT INTEGER(0..18446744073709551615)

REAL::= [APPLICATION 10] IMPLICIT OCTET STRING (SIZE(4))

LREAL::= [APPLICATION 11] IMPLICIT OCTET STRING (SIZE(8))

TIME::= [APPLICATION 12] IMPLICIT LINT – Durée en unités de 1µs

DATE::= [APPLICATION 13] IMPLICIT ULINT – Voir Tableau E.1.
```

```
TIME_OF_DAY::= [APPLICATION 14] IMPLICIT ULINT – Voir Tableau E.1.

DATE_AND_TIME::= [APPLICATION 15] IMPLICIT ULINT – Voir Tableau E.1.

STRING::= [APPLICATION 16] IMPLICIT OCTET STRING – 1 octet par caractère

BYTE::= [APPLICATION 17] IMPLICIT BIT STRING (SIZE(8))

WORD::= [APPLICATION 18] IMPLICIT BIT STRING (SIZE(16))

DWORD::= [APPLICATION 19] IMPLICIT BIT STRING (SIZE(32))

LWORD::= [APPLICATION 20] IMPLICIT BIT STRING (SIZE(64))

WSTRING::= [APPLICATION 21] IMPLICIT OCTET STRING – 2 octets par caractère

DerivedData::= CHOICE{
     DirectlyDerivedData,
     EnumeratedData,
     SubrangeData,
     ARRAY,
     STRUCT}

DirectlyDerivedData::= [typeID] IMPLICIT ElementaryData

EnumeratedData::= [typeID] IMPLICIT UINT

SubrangeData::= [typeID] IMPLICIT FixedLengthInteger

ARRAY::= CHOICE {ArrayVariable, TypedArray}

ArrayVariable::= [APPLICATION 22] IMPLICIT FBDataSequence – même type

TypedArray::= [typeID] IMPLICIT FBDataSequence – même type

STRUCT::= [typeID] IMPLICIT SEQUENCE – types différents

END
```

E.3.3 Règles de codage

E.3.3.1 Codage BASIC

Ce codage doit être le résultat obtenu en appliquant les règles de codage de base selon l'ISO/CEI 8825-1 à des variables des types définis en E.3.2.

E.3.3.2 Codage COMPACT

Ce codage doit être le résultat obtenu en modifiant les règles pour le codage BASIC données en E.3.3.1 comme suit:

a) les «octets de longueur» ne doivent pas être inclus dans le codage des valeurs des types de données montrés au Tableau E.1;

b) la longueur (en octets) et le codage des «octets de contenu» décrits dans l'ISO/CEI 8825-1 doivent être tels que définis dans le Tableau E.1 pour les valeurs des types de données qui y sont montrés;

c) le codage des variables de type `TIME`, `DirectlyDerivedData`, `EnumeratedData` ou `SubrangeData` doit suivre les mêmes règles de codage que le type de base.

d) les «octets de typ» ne doivent pas être inclus dans le codage d'éléments individuels de type `STRUCT`, à l'exception du codage d'éléments de type `BOOL`, qui doivent être codés suivant la règle (1) du Tableau E.1;

e) le codage des valeurs des types `STRING` et `WSTRING` doit être primitif;

f) le codage des éléments `ARRAY` doit être *construit* dans le sens de l'ISO/CEI 8825-1, avec les dispositions suivantes pour le codage COMPACT:

1) Le sous-champ «longueur» de l'élément ARRAY doit être codé comme une valeur du type UINT sans octets d'identificateur ou de longueur, c'est-à-dire comme un nombre entier non signé («unsigned») de 16 bits.

   NOTE 1 Cela semble réduire le nombre maximal d'éléments d'un ARRAY à 65535. Cependant, la longueur réelle peut être réduite encore davantage par le nombre maximal d'octets pouvant être transféré par le protocole de transport sous-jacent.

   EXEMPLE Pour les messages UDP possédant un nombre maximal de 65508 octets, la longueur maximale transmissible d'un ARRAY d'éléments BYTE serait égale à: (nombre maximal d'octets – octets d'étiquette – octets de longueur – octets de type d'élément)/(longueur d'élément) = (65508-1-2-1)/1 = 65504 éléments.

2) Le codage COMPACT doit être utilisé pour le premier élément du champ «valeurs».

3) Les éléments suivants, le cas échéant, doivent être codés en utilisant la syntaxe COMPACT sans sous-champ «identificateur», sauf pour les éléments de type BOOL, qui doivent être codés suivant la règle (1) du Tableau E.1.

4) Si la longueur spécifiée de l'élément ARRAY reçu est inférieure à l'espace alloué localement, les éléments restants de la matrice locale sont inchangés; si la longueur de l'élément ARRAY reçu est supérieure à l'espace alloué localement, les éléments reçus restants sont ignorés.

NOTE 2 Etant donné que ARRAY est une sous-classe de FBData, un élément ARRAY multidimensionnel est susceptible d'être codé de façon récurrente en tant qu'ARRAY dont les éléments sont des éléments ARRAY.

Tableau E.1 – Codage COMPACT des types de données de longueur fixe

Type de données	Octets de contenu	
	Longueur	Règle de codage
BOOL	0	(1)
SINT	1	(2)
INT	2	(2)
DINT	4	(2)
LINT	8	(2)
USINT	1	(3)
UINT	2	(3)
UDINT	4	(3)
ULINT	8	(3)
REAL	4	(4)
LREAL	8	(4)
DATE	8	(5)
TIME	8	(7)
TIME_OF_DAY	12	(5)
DATE_AND_TIME	20	(5)
BYTE	1	(6)
WORD	2	(6)
DWORD	4	(6)
LWORD	8	(6)

RÈGLES DE CODAGE POUR LE TABLEAU E.1

(1) Les valeurs de ce type de données doivent être codées en un seul octet d'identificateur contenant le codage d'étiquette pour la classe `BOOL0` ou `BOOL1`, défini en E.3.2, correspondant respectivement aux valeurs de `FALSE (0)` ou `TRUE (1)`.

(2) Les valeurs de ces types de données `SignedInteger` doivent être codées de la même manière qu'un `UnsignedInteger` de même longueur que le type `SignedInteger` avec une valeur $N - N_{min}$, où N est la valeur de la variable `SignedInteger` à coder et N_{min} est l'extrémité inférieure de la plage de valeurs du sous-type `SignedInteger` tel que défini en E.3.2.

(3) Les valeurs de ces types de données `UnsignedInteger` doivent être codées en numérotant les bits dans les octets de contenu, en commençant par le bit 1 du dernier octet comme bit zéro et en terminant la numérotation par le bit 8 du premier octet. Chaque bit se voit attribuer une valeur de 2^N, où N est sa position dans la séquence de numérotation ci-dessus. La valeur du nombre entier non signé («unsigned») est obtenue en additionnant les valeurs numériques attribuées à chaque bit pour les bits qui sont mis à un.

(4) Les valeurs de ces types de données doivent être codées comme des nombres au format simple 32 bits et au format double 64 bits comme défini dans la CEI 60559, où «lsb» défini dans la CEI 60559 correspond au «bit zéro» tel que défini dans la Règle(3).

(5) Les valeurs de ces types doivent être codées comme dans le cas du type `ULINT`, représentant le nombre de millisecondes écoulées depuis minuit pour `TIME_OF_DAY`, le nombre de millisecondes écoulées depuis le `1970-01-01-00:00:00.000` pour `DATE_AND_TIME`, le nombre de millisecondes écoulées à partir du `1970-01-01-00:00:00.000` jusqu'au `YYYY-MM-DD-00:00:00.000` pour `DATE`, où `YYYY-MM_DD` est la date courante.

(6) Le codage des valeurs de ces types de données `FixedLengthBitString` doit être primitif et doit être obtenu en plaçant les bits dans la chaîne de bits, en commençant par le premier bit et en poursuivant jusqu'au bit de queue, dans les bits 8 à 1 du premier octet de contenu, suivi tour à tour des bits 8 à 1 de chacun des octets suivants, la notation «premier bit» et «bit de queue» étant spécifiée dans l'ISO/CEI 8824-1.

(7) Le codage des valeurs de ce type de données doit être le même que celui des valeurs du type `LINT`, représentant un intervalle de temps en unités de 1 μs.

Annexe F
(normative)

Spécifications textuelles

L'Annexe F fournit des spécifications textuelles, dans la syntaxe définie dans l'Annexe B, pour tous les types des blocs fonctionnels et d'adaptateurs illustrés dans la présente norme. Le contenu de l'Annexe F est normatif dans la mesure définie par la description de chaque type de bloc fonctionnel ou d'adaptateur de ce genre dans la présente norme.

NOTE Les spécifications sont énumérées alphabétiquement par nom de type.

```
(*************************************************************************)
FUNCTION_BLOCK E_CTU (* Compteur progressif événementiel *)
EVENT_INPUT
  CU WITH PV; (* Comptage progressif *)
  R; (* Réinitialisation *)
END_EVENT
EVENT_OUTPUT
  CUO WITH Q,CV; (* Compter progressivement l'événement de sortie *)
  RO WITH Q,CV; (* Réinitialiser l'événement de sortie *)
END_EVENT
VAR_INPUT
  PV: UINT; (* Valeur préétablie *)
END_VAR
VAR_OUTPUT
  Q: BOOL; (* CV>=PV *)
  CV: UINT;
END_VAR
EC_STATES
  START;
  CU: CU -> CUO;
  R: R -> RO;
END_STATES
EC_TRANSITIONS
  START TO CU:= CU [CV<65535];
  CU TO START:= 1;
  START TO R:= R;
  R TO START:= 1;
END_TRANSITIONS
ALGORITHM CU IN ST: (* Comptage progressif *)
CV:= CV + 1;
Q:= (CV >= PV);
END_ALGORITHM
ALGORITHM R IN ST: (* Réinitialisation *)
CV:= 0;
Q:= FALSE;
END_ALGORITHM
END_FUNCTION_BLOCK
(*************************************************************************)
FUNCTION_BLOCK E_CYCLE (* Génération (cyclique) périodique d'un Event *)
EVENT_INPUT
  START WITH DT;
  STOP;
END_EVENT
EVENT_OUTPUT
 EO;   (* Événement périodique à la période DT, commençant à DT après GO *)
END_EVENT
VAR_INPUT
  DT: TIME; (* Période entre événements *)
END_VAR
FBS
 DLY: E_DELAY;
END_FBS
EVENT_CONNECTIONS
 START TO DLY.START;
```

```
  STOP TO DLY.STOP;
  DLY.EO TO DLY.START;
  DLY.EO TO EO;
END_CONNECTIONS
DATA_CONNECTIONS
  DT TO DLY.DT;
END_CONNECTIONS
END_FUNCTION_BLOCK
(*=================================================================*)
FUNCTION_BLOCK E_D_FF (* Verrou de données (D) guidé par des événements *)
EVENT_INPUT
  CLK WITH D; (* Horloge de données *)
END_EVENT
EVENT_OUTPUT
  EO WITH Q; (* Événement de sortie lorsque la sortie Q change *)
END_EVENT
VAR_INPUT
  D: BOOL; (* Entrée de données *)
END_VAR
VAR_OUTPUT
  Q: BOOL; (* Données verrouillées *)
END_VAR
EC_STATES
  Q0; (* Q est initialement FALSE *)
  RESET: LATCH -> EO; (* Réinitialiser Q et émettre EO *)
  SET: LATCH -> EO; (* Verrouiller et émettre EO *)
END_STATES
EC_TRANSITIONS
  Q0 TO SET:= CLK [D];
  SET TO RESET:= CLK [NOT D];
  RESET TO SET:= CLK [D];
END_TRANSITIONS
ALGORITHM LATCH IN ST:
Q:=D;
END_ALGORITHM
END_FUNCTION_BLOCK
(*=================================================================*)
FUNCTION_BLOCK E_DELAY
 (* Propagation retardée d'un événement - Annulable *)
EVENT_INPUT
 START WITH DT; (* Commencer le retard *)
 STOP;       (* Annuler le retard *)
END_EVENT
EVENT_OUTPUT
 EO;      (* Événement retardé *)
END_EVENT
VAR_INPUT
 DT: TIME; (* Temps de retard *)
END_VAR
SERVICE E_DELAY/RESOURCE
SEQUENCE event_delay
 E_DELAY.START(DT) ->E_DELAY.EO();
END_SEQUENCE
SEQUENCE delay_canceled
 E_DELAY.START(DT);
 E_DELAY.STOP();
END_SEQUENCE
SEQUENCE no_multiple_delay
 E_DELAY.START(DT);
 E_DELAY.START(DT);
 ->E_DELAY.EO();
END_SEQUENCE
END_SERVICE
END_FUNCTION_BLOCK
(*=================================================================*)
FUNCTION_BLOCK E_DEMUX (* Démultiplexeur d'événements *)
EVENT_INPUT
 EI WITH K;  (* Événement à démultiplexer *)
END_EVENT
```

```
EVENT_OUTPUT
 EO0;
 EO1;
 EO2;
 EO3;      (* Le nombre de sorties dépend de la mise en œuvre *)
END_EVENT
VAR_INPUT
 K: UINT;  (* Indice d'événement, le maximum dépend de la mise en œuvre *)
END_VAR
EC_STATES
 START; (* État initial *)
 TRIGGERED; (* État intermédiaire après l'arrivée de EI *)
 EO0: -> EO0;
 EO1: -> EO1;
 EO2: -> EO2;
 EO3: -> EO3;
END_STATES
EC_TRANSITIONS
 START TO TRIGGERED:= EI;
 TRIGGERED TO EO0:= [K=0];
 TRIGGERED TO EO1:= [K=1];
 TRIGGERED TO EO2:= [K=2];
 TRIGGERED TO EO3:= [K=3];
 TRIGGERED TO START:= [K>3];
 EO0 TO START:= 1;
 EO1 TO START:= 1;
 EO2 TO START:= 1;
 EO3 TO START:= 1;
END_TRANSITIONS
END_FUNCTION_BLOCK
=================================================================
FUNCTION_BLOCK E_F_TRIG (* Détection booléenne de front descendant *)
EVENT_INPUT
 EI WITH QI;   (* Entrée d'événements *)
END_EVENT
EVENT_OUTPUT
 EO;  (* Sortie d'événements *)
END_EVENT
VAR_INPUT
 QI: BOOL;  (* Entrée booléenne pour détection de front descendant *)
END_VAR
FBS
 D: E_D_FF;
 SW: E_SWITCH;
END_FBS
EVENT_CONNECTIONS
  EI TO D.CLK;
  D.EO TO SW.EI;
  SW.EO0 TO EO;
END_CONNECTIONS
DATA_CONNECTIONS
  QI TO D.D;
  D.Q TO SW.G;
END_CONNECTIONS
END_FUNCTION_BLOCK
=================================================================
FUNCTION_BLOCK E_MERGE (* Fusion (OR) de plusieurs événements *)
EVENT_INPUT
  EI1; (* Premier événement d'entrée *)
  EI2; (* Deuxième événement d'entrée *)
  END_EVENT
EVENT_OUTPUT EO; (* Événement de sortie *)
END_EVENT
EC_STATES
  START;  (* État initial *)
  EO:    (* Émettre l'événement EO *)
    ->EO;
END_STATES
EC_TRANSITIONS
  START TO EO:= EI1;
```

```
  START TO EO:= EI2;
  EO TO START:= 1;
END_TRANSITIONS
END_FUNCTION_BLOCK
(*=================================================================*)
FUNCTION_BLOCK E_N_TABLE (* Génération d'un train fini d'événements
distincts, guidée par table *)
EVENT_INPUT
  START WITH DT, N;
  STOP;
END_EVENT
EVENT_OUTPUT
 EO0;   (* N événements à des périodes DT, en commençant à DT[0] après START
*)
 EO1;
 EO2;
 EO3;   (* Extensible *)
END_EVENT
VAR_INPUT
  DT: TIME[3]; (* Périodes entre événements *)
  N: UINT;  (* Nombre d'événements à générer (=3 dans cet exemple) *)
END_VAR
SERVICE E_N_TABLE/RESOURCE
SEQUENCE typical_operation
  E_N_TABLE.START(DT,N) -> E_N_TABLE.EO0() -> E_N_TABLE.EO1() ->
E_N_TABLE.EO2() -> E_N_TABLE.EO3();
END_SEQUENCE
END_SERVICE
END_FUNCTION_BLOCK
(*=================================================================*)
FUNCTION_BLOCK E_PERMIT (* Propagation permissive d'un événement *)
EVENT_INPUT EI WITH PERMIT; (* Entrée d'événements *)
END_EVENT
EVENT_OUTPUT EO; (* Sortie d'événements *)
END_EVENT
VAR_INPUT PERMIT: BOOL; END_VAR
EC_STATES
  START;  (* État initial *)
  EO:    (* Émettre l'événement EO *)
    ->EO;
END_STATES
EC_TRANSITIONS
  START TO EO:= EI [PERMIT];
  EO TO START:= 1;
END_TRANSITIONS
END_FUNCTION_BLOCK
(*=================================================================*)
FUNCTION_BLOCK E_R_TRIG (* Détection booléenne de front montant *)
EVENT_INPUT
 EI WITH QI;   (* Entrée d'événements *)
END_EVENT
EVENT_OUTPUT
 EO;  (* Sortie d'événements *)
END_EVENT
VAR_INPUT
 QI: BOOL;  (* Entrée booléenne pour détection de front montant *)
END_VAR
FBS
 D: E_D_FF;
 SW: E_SWITCH;
END_FBS
EVENT_CONNECTIONS
  EI TO D.CLK;
  D.EO TO SW.EI;
  SW.EO1 TO EO;
END_CONNECTIONS
DATA_CONNECTIONS
  QI TO D.D;
  D.Q TO SW.G;
```

```
END_CONNECTIONS
END_FUNCTION_BLOCK
(*=================================================================*)
FUNCTION_BLOCK E_REND (* Rendez-vous de deux éléments *)
EVENT_INPUT
  EI1; (* Premier événement d'entrée *)
  EI2; (* Deuxième événement d'entrée *)
  R;  (* Réinitialiser l'événement *)
END_EVENT
EVENT_OUTPUT
 EO;  (* Rendez-vous d'événement de sortie *)
END_EVENT
EC_STATES
  START;  (* État initial *)
  EI1;   (* EI1 est arrivé, attendre EI2 ou R *)
  EO:    (* Émettre l'événement de rendez-vous*)
    ->EO;
  EI2;   (* EI2 est arrivé, attendre EI1 ou R *)
END_STATES
EC_TRANSITIONS
  START TO EI1:= EI1;
  EI1 TO START:= R;
  START TO EI2:= EI2;
  EI2 TO START:= R;
  EI1 TO EO:= EI2;
  EI2 TO EO:= EI1;
  EO TO START:= 1;
END_TRANSITIONS
END_FUNCTION_BLOCK
(*=================================================================*)
FUNCTION_BLOCK E_RESTART (* Génération d'événements de redémarrage *)
EVENT_OUTPUT
 COLD;     (* Redémarrage à froid *)
 WARM;     (* Redémarrage à chaud *)
END_EVENT
SERVICE RESOURCE/E_RESTART
SEQUENCE cold_restart ->E_RESTART.COLD(); END_SEQUENCE
SEQUENCE warm_restart ->E_RESTART.WARM(); END_SEQUENCE
END_SERVICE
END_FUNCTION_BLOCK
(*=================================================================*)
FUNCTION_BLOCK E_RS (* bistable piloté par des événements *)
EVENT_INPUT
  S; (* Établir l'événement *)
  R; (* Réinitialiser l'événement *)
END_EVENT
EVENT_OUTPUT
  EO WITH Q; (* Produire l'événement *)
END_EVENT
VAR_OUTPUT
  Q; BOOL; (* État de sortie courant *)
END_VAR
EC_STATES
  Q0; (* Q est initialement FALSE *)
  RESET; RESET -> EO; (* Réinitialiser Q et émettre EO *)
  SET; SET -> EO; (* Positionner Q et émettre EO *)
END_STATES
EC_TRANSITIONS
  Q0 TO SET:= S;
  SET TO RESET:= R;
  RESET TO SET:= S;
END_TRANSITIONS
ALGORITHM SET IN ST: (* Établir Q*)
Q:=TRUE;
END_ALGORITHM
ALGORITHM RESET IN ST: (* Réinitialiser Q *)
Q:=FALSE;
END_ALGORITHM
END_FUNCTION_BLOCK
(*=================================================================*)
```

```
FUNCTION_BLOCK E_SELECT (* Sélection entre deux événements *)
EVENT_INPUT
  EI0 WITH G; (* Événement d'entrée, sélectionné lorsque G=0 *)
  EI1 WITH G; (* Événement d'entrée, sélectionné lorsque G=1 *)
END_EVENT
EVENT_OUTPUT EO; (* Événement de sortie *)
END_EVENT
VAR_INPUT G: BOOL; (* Séletctionner EI0 lorsque G=0, EI1 lorsque G=1 *)
END_VAR
EC_STATES
  START;   (* État initial *)
  EO: -> EO; (* Émettre l'événement de sortie*)
END_STATES
EC_TRANSITIONS
  START TO EO:= EI0 [NOT G];
  START TO EO:= EI1 [G];
  EO TO START:= 1;
END_TRANSITIONS
END_FUNCTION_BLOCK
=====================================================================
FUNCTION_BLOCK E_SPLIT (* Diviser un événement *)
EVENT_INPUT
  EI; (* Événement d'entrée *)
END_EVENT
EVENT_OUTPUT
 EO1;  (* Premier événement de sortie *)
 EO2;  (* Deuxième événement de sortie, etc. *)
END_EVENT
EC_STATES
  START; (* État initial *)
  EO: (* Extensible *)
    ->EO1, (* Produire un premier événement *)
    ->EO2; (* Produire un deuxième événement, etc. *)
END_STATES
EC_TRANSITIONS
  START TO EO:= EI;
  EO TO START:= 1;
END_TRANSITIONS
END_FUNCTION_BLOCK
=====================================================================
FUNCTION_BLOCK E_SWITCH (* Commuter (démultiplexer) un événement *)
EVENT_INPUT EI WITH G; (* Entrée d'événements *)
END_EVENT
EVENT_OUTPUT
  EO0; (* Sortie, commutée de EI lorsque G=0 *)
  EO1; (* Sortie, commutée de EI lorsque G=1 *)
END_EVENT
VAR_INPUT G: BOOL; (* Commuter EI à EI0 lorsque G=0, à EI1 lorsque G=1 *)
END_VAR
EC_STATES
  START; (* État initial *)
  G0:  (* Émettre EO0 lorsque EI arrive avec G=0 *)
    ->EO0;
  G1: (* Émettre EO1 lorsque EI arrive avec G=1 *)
    ->EO1;
END_STATES
EC_TRANSITIONS
  START TO G0:= EI [NOT G];
  G0 TO START:= 1;
  START TO G1:= EI [G];
  G1 TO START:= 1;
END_TRANSITIONS
END_FUNCTION_BLOCK
=====================================================================
FUNCTION_BLOCK E_TABLE (* Génération d'un train fini d'événements, guidée
par table *)
EVENT_INPUT
 START WITH DT, N;
 STOP; (* Annuler*)
```

```
END_EVENT
EVENT_OUTPUT
 EO WITH CV; (* N événements à des périodes DT, en commençant à DT[0] après
START *)
END_EVENT
VAR_INPUT
  DT: TIME[4]; (* Périodes entre événements *)
  N: UINT; (* Nombre d'événements à générer *)
END_VAR
VAR_OUTPUT
  CV; UINT; (* Indice d'événement courant, 0..N-1 *)
END_VAR
FBS
  CTRL: E_TABLE_CTRL;
  DLY: E_DELAY;
END_FBS
EVENT_CONNECTIONS
  START TO CTRL.INIT;
  CTRL.CLKO TO DLY.START;
  DLY.EO TO EO;
  DLY.EO TO CTRL.CLK;
  STOP TO DLY.STOP;
END_CONNECTIONS
DATA_CONNECTIONS
  DT TO CTRL.DT;
  N TO CTRL.N;
  CTRL.DTO TO DLY.DT;
  CTRL.CV TO CV;
END_CONNECTIONS
END_FUNCTION_BLOCK
================================================================================
FUNCTION_BLOCK E_TABLE_CTRL (* Commande pour E_TABLE *)
EVENT_INPUT
 INIT WITH DT, N;
 CLK;
END_EVENT
EVENT_OUTPUT
 CLKO WITH DTO, CV;
END_EVENT
VAR_INPUT
 DT; TIME[4]; (* La longueur de Array dépend de la mise en œuvre *)
 N; UINT;   (* Nombre réel d'échelons temporels *)
END_VAR
VAR_OUTPUT
 DTO; TIME;  (* Intervalle de retard courant *)
 CV; UINT;   (* Indice d'événement courant, 0..N-1 *)
END_VAR
EC_STATES
 START;
 INIT0: INIT;
 INIT1: -> CLKO;
 STEP: STEP -> CLKO;
END_STATES
EC_TRANSITIONS
 START TO INIT0:= INIT;
 INIT0 TO INIT1:= [N>0];
 INIT0 TO START:= [N=0]; (* Ne pas exécuter si N=0 *)
 INIT1 TO START:= 1;
 START TO STEP:= CLK [CV < MIN(3,N-1)];
 STEP TO START:= 1;
END_TRANSITIONS
ALGORITHM STEP IN ST:
  CV:= CV+1;
  DTO:= DT[CV];
END_ALGORITHM
ALGORITHM INIT IN ST:
  CV:= 0;
  DTO:= DT[0];
END_ALGORITHM
END_FUNCTION_BLOCK
```

```
(*=================================================================*)
FUNCTION_BLOCK E_TRAIN (* Génération d'un train fini d'événements *)
EVENT_INPUT
 START WITH DT, N;
 STOP;
END_EVENT
EVENT_OUTPUT
 EO WITH CV; (* N événements à la période DT, commençant à DT après START *)
END_EVENT
VAR_INPUT
  DT: TIME; (* Période entre événements *)
  N: UINT; (* Nombre d'événements à générer *)
END_VAR
VAR_OUTPUT
 CV: UINT; (* Indice EO (0..N-1) *)
END_VAR
FBS
 CTR: E_CTU;
 GATE: E_SWITCH;
 DLY: E_DELAY;
END_FBS
EVENT_CONNECTIONS
 START TO CTR.R;
 STOP TO DLY.STOP;
 DLY.EO TO EO;
 DLY.EO TO CTR.CU;
 CTR.CUO TO GATE.EI;
 CTR.RO TO GATE.EI;
 GATE.EO0 TO DLY.START;
END_CONNECTIONS
DATA_CONNECTIONS
 DT TO DLY.DT;
 N TO CTR.PV;
 CTR.Q TO GATE.G;
 CTR.CV TO CV;
END_CONNECTIONS
END_FUNCTION_BLOCK
(*=================================================================*)
FUNCTION_BLOCK FB_ADD_INT (* Addition INT *)
EVENT_INPUT
 REQ WITH QI, IN1, IN2;
END_EVENT
EVENT_OUTPUT
 CNF WITH QO, STATUS, OUT;
END_EVENT
VAR_INPUT
 QI; BOOL;    (* Qualificateur d'événement *)
 IN1; INT;    (* Cumulande*)
 IN2; INT;    (* Cumulateur*)
END_VAR
VAR_OUTPUT
 QO; BOOL;     (* Qualificateur de sortie *)
 STATUS; UINT;  (* Statut d'opération *)
 OUT; INT;    (* Somme *)
END_VAR
VAR
 RESULT: DINT;
END_VAR
EC_STATES
 START;
 REQ: REQ -> CNF;
END_STATES
EC_TRANSITIONS
 START TO REQ:= REQ;
 REQ TO START:= 1;
END_TRANSITIONS
ALGORITHM REQ IN ST:
  QO:= QI;
  IF QI THEN
```

```
    STATUS:= 0;
    RESULT:= INT_TO_DINT(IN1) + INT_TO_DINT(IN2);
    IF (RESULT > 32767) OR (RESULT < -32768) THEN
     QO = FALSE;
     STATUS = 3;
     IF (RESULT > 32767) THEN OUT:= 32767;
     ELSE OUT:= -32768;
     END_IF;
    ELSE OUT:= RESULT;
    END_IF;
   ELSE STATUS = 1;
   END_IF;
END_ALGORITHM
END_FUNCTION_BLOCK
================================================================
FUNCTION_BLOCK INTEGRAL_REAL
EVENT_INPUT
 INIT: INIT_EVENT WITH CYCLE;
 EX WITH HOLD, XIN;
END_EVENT
EVENT_OUTPUT
 INITO: INIT_EVENT WITH XOUT;
 EXO WITH XOUT;
END_EVENT
VAR_INPUT
 HOLD; BOOL;  (* 0 = Exécuter, 1 = Bloquer*)
 XIN; REAL;  (* Intégrande *)
 CYCLE; TIME; (* Période d'échantillonnage *)
END_VAR
VAR_OUTPUT
 XOUT; REAL;  (* Sortie intégrée *)
END_VAR
VAR DT; REAL; END_VAR
EC_STATES
 START;         (* État initial EC *)
 INIT:INIT -> INITO; (* État EC avec algorithme et action EC *)
 MAIN: MAIN -> EXO;
END_STATES
EC_TRANSITIONS
 START TO INIT:= INIT; (* Une transition EC *)
 START TO MAIN:= EX;
 INIT TO START:= 1;
 MAIN TO START:= 1;
END_TRANSITIONS
ALGORITHM INIT IN ST:
 XOUT:= 0.0;
 DT:= TIME_TO_REAL(CYCLE);
END_ALGORITHM
ALGORITHM MAIN IN ST:
 IF NOT HOLD THEN
  XOUT:= XOUT + XIN * DT;
 END_IF;
END_ALGORITHM
END_FUNCTION_BLOCK
================================================================
ADAPTER LD_UNLD (* LOAD/UNLOAD Adapter Interface *)
EVENT_INPUT
 UNLD;   (* UNLOAD Request *)
END_EVENT
EVENT_OUTPUT
 LD WITH WO,WKPC; (* LOAD Request *)
 CNF WITH WO,WKPC; (* UNLD Confirm *)
END_EVENT
VAR_OUTPUT
 WO: BOOL;  (* Pièce à travailler présente *)
 WKPC: COLOR; (* Couleur de la pièce à travailler *)
END_VAR
SERVICE PLUG/SOCKET
SEQUENCE normal_operation
 PLUG.LD(WO,WKPC) -> SOCKET.LD(WO,WKPC);
```

```
 SOCKET.UNLD() -> PLUG.UNLD();
 PLUG.CNF() -> SOCKET.CNF();
END_SEQUENCE
END_SERVICE
END_ADAPTER
================================================================
FUNCTION_BLOCK MANAGER (* Management Service Interface *)
EVENT_INPUT
  INIT WITH QI, PARAMS; (* Initialisation de service *)
  REQ WITH QI, CMD, OBJECT; (* Demande de service *)
END_EVENT
EVENT_OUTPUT
  INITO WITH QO, STATUS; (* Confirmation d'initialisation *)
  CNF WITH QO, STATUS, RESULT; (* Confirmation de Service *)
END_EVENT
VAR_INPUT
  QI; BOOL; (* Qualificateur d'entrée d'événements *)
  PARAMS; WSTRING; (* Paramètres de service *)
  CMD; UINT; (* Commande énumérée *)
  OBJECT; BYTE[512]; (* Objet Command *)
END_VAR
VAR_OUTPUT
  QO; BOOL; (* Qualificateur de sortie d'événements*)
  STATUS; UINT; (* Statut de service *)
  RESULT; BYTE[512]; (* Objet Result *)
END_VAR
SERVICE MANAGER/resource
SEQUENCE normal_establishment
  MANAGER.INIT+(PARAMS) -> resource.initManagement() -> MANAGER.INITO+();
END_SEQUENCE
SEQUENCE unsuccessful_establishment
  MANAGER.INIT+(PARAMS) -> resource.initManagement(PARAMS) -> MANAGER.INITO-
(STATUS);
END_SEQUENCE
SEQUENCE normal_command_sequence
  MANAGER.REQ+(CMD,OBJECT) -> resource.performCommand(CMD,OBJECT) ->
MANAGER.CNF+(STATUS,RESULT);
END_SEQUENCE
SEQUENCE command_error
  MANAGER.REQ+(CMD,OBJECT) -> resource.performCommand(CMD,OBJECT) ->
MANAGER.IND-(STATUS);
END_SEQUENCE
SEQUENCE application_initiated_termination
  MANAGER.INIT-() -> resource.terminateService() -> MANAGER.INITO-(STATUS);
END_SEQUENCE
SEQUENCE resource_initiated_termination
  resource.serviceTerminated(STATUS) -> MANAGER.INITO-(STATUS);
END_SEQUENCE
END_SERVICE
END_FUNCTION_BLOCK
================================================================
FUNCTION_BLOCK PI_REAL
EVENT_INPUT
  INIT WITH KP, KI, CYCLE;
  EX WITH HOLD, PV, SP, KP, KI, CYCLE;
END_EVENT
EVENT_OUTPUT
  INITO WITH XOUT;
  EXO WITH XOUT;
END_EVENT
VAR_INPUT
  HOLD; BOOL; (* Retenir si TRUE *)
  PV; REAL; (* Variable de processus *)
  SP; REAL; (* Valeur de consigne *)
  KP; REAL; (* Constante de proportionnalité *)
  KI; REAL; (* Constante d'intégration,1/s *)
  CYCLE; TIME; (* Période d'échantillonnage *)
END_VAR
VAR_OUTPUT
```

```
  XOUT: REAL;
END_VAR
FBS
  CALC: PID_CALC;
  INTEGRAL_TERM: INTEGRAL_REAL;
END_FBS
EVENT_CONNECTIONS
  INIT TO CALC.INIT;
  EX TO CALC.PRE;
  CALC.POSTO TO EXO;
  INTEGRAL_TERM.INITO TO INITO;
  CALC.INITO TO INTEGRAL_TERM.INIT;
  CALC.PREO TO INTEGRAL_TERM.EX;
  INTEGRAL_TERM.EXO TO CALC.POST;
END_CONNECTIONS
DATA_CONNECTIONS
  HOLD TO INTEGRAL_TERM.HOLD;
  PV TO CALC.PV;
  SP TO CALC.SP;
  KP TO CALC.KP;
  KI TO CALC.KI;
  CYCLE TO INTEGRAL_TERM.CYCLE;
  CALC.XOUT TO XOUT;
  CALC.ETERM TO INTEGRAL_TERM.XIN;
  INTEGRAL_TERM.XOUT TO CALC.ITERM;
  0 TO CALC.TD;
  0 TO CALC.DTERM;
END_CONNECTIONS
END_FUNCTION_BLOCK
(*=================================================================*)
SUBAPPLICATION PI_REAL_APPL (* une sous-application *)
EVENT_INPUT
 INIT;
 EX;
END_EVENT
EVENT_OUTPUT
 INITO;
 EXO;
END_EVENT
VAR_INPUT
 HOLD: BOOL; (* Retenir si TRUE *)
 PV; REAL; (* Variable de processus *)
 SP; REAL; (* Valeur de consigne *)
 KP; REAL; (* Gain proportionnel *)
 KI; REAL; (* Gain intégral = Pérode d'échantillonnage par temps de
réinitialisation *)
 X0; REAL; (* Sortie d'intégrateur initiale *)
END_VAR
VAR_OUTPUT XOUT: REAL; END_VAR
FBS
 ETERM: FB_SUB_REAL;
 INTEGRATOR: ACCUM_REAL;
 CALC: PI_CALC;
END_FBS
EVENT_CONNECTIONS
  INIT TO INTEGRATOR.INIT;
  INTEGRATOR.INITO TO INITO;
  EX TO ETERM.REQ;
  ETERM.CNF TO INTEGRATOR.EX;
  INTEGRATOR.EXO TO CALC.EX;
  CALC.EXO TO EXO;
END_CONNECTIONS
DATA_CONNECTIONS
  X0 TO INTEGRATOR.X0;
  HOLD TO INTEGRATOR.HOLD;
  PV TO ETERM.IN1;
  SP TO ETERM.IN2;
  KP TO CALC.KP;
  KI TO CALC.KI;
  ETERM.OUT TO INTEGRATOR.XIN;
```

```
   ETERM.OUT TO CALC.ETERM;
   INTEGRATOR.XOUT TO CALC.ITERM;
   CALC.XOUT TO XOUT;
   1 TO ETERM.QI;
 END_CONNECTIONS
 END_SUBAPPLICATION
 ================================================================
 FUNCTION_BLOCK REQUESTER
            (* Interface de demandeur de service *)
 EVENT_INPUT
  INIT WITH QI, PARAMS;     (* Initialisation de service *)
  REQ WITH QI, SD_1, SD_m;  (* Demande de service *)
 END_EVENT
 EVENT_OUTPUT
  INITO WITH QO, STATUS;       (* Confirmation d'initialisation *)
  CNF WITH QO, STATUS, RD_1, RD_n; (* Confirmation de Service *)
 END_EVENT
 VAR_INPUT
  QI; BOOL;    (* Qualificateur d'entrée d'événements *)
  PARAMS; ANY; (* Paramètres de service *)
  SD_1; ANY;   (* Données à transférer, extensible *)
  SD_m; ANY;   (* Dernier élément de données à transférer *)
 END_VAR
 VAR_OUTPUT
  QO; BOOL;    (* Qualificateur de sortie d'événements *)
  STATUS; ANY; (* Statut de service *)
  RD_1; ANY;   (* Données reçues, extensible *)
  RD_n; ANY;   (* Dernier élément de données reçu *)
 END_VAR
 SERVICE REQUESTER/RESOURCE
 SEQUENCE normal_establishment
  REQUESTER.INIT+(PARAMS) -> REQUESTER.INITO+();
 END_SEQUENCE
 SEQUENCE unsuccessful_establishment
  REQUESTER.INIT+(PARAMS) -> REQUESTER.INITO-(STATUS);
 END_SEQUENCE
 SEQUENCE normal_data_transfer
  REQUESTER.REQ+(SD_1,...,SD_m) -> REQUESTER.CNF+(RD_1,...,RD_n);
 END_SEQUENCE
 SEQUENCE data_transfer_error
  REQUESTER.REQ+(SD_1,...,SD_m) -> REQUESTER.CNF-(STATUS);
 END_SEQUENCE
 SEQUENCE application_initiated_termination
  REQUESTER.INIT-() -> REQUESTER.INITO-(STATUS);
 END_SEQUENCE
 SEQUENCE resource_initiated_termination
  -> REQUESTER.INITO-(STATUS);
 END_SEQUENCE
 END_SERVICE
 END_FUNCTION_BLOCK
 ================================================================
 FUNCTION_BLOCK XBAR_MVCA (* XBAR_MVC + Adapters *)
 EVENT_INPUT
  INIT WITH VF,VR,DTL,DT,BKGD,LEN,DIA,DIR; (* Initialiser *)
 END_EVENT
 EVENT_OUTPUT
  INITO;
 END_EVENT
 VAR_INPUT
  VF: INT;= 20;     (* Vitesse ADVANCE en +%/s *)
  VR: INT;= -40;     (* Vitesse RETRACT en -%/s *)
  DTL: TIME;= t#750ms; (* Retard LOAD *)
  DT; TIME;= t#250ms; (* Intervalle de simulation *)
  BKGD; COLOR;= COLOR#blue; (* Couleur barre de transfert *)
  LEN; UINT;= 5;   (* Longueur de barre en diamètres *)
  DIA; UINT;= 20; (* Diamètre de la pièce à travailler *)
  DIR; UINT;      (* Orientation: 0=Gauche/Droite, 1=Haut/Bas, 2=D/G, 3=B/H *)
 END_VAR
 SOCKETS
```

```
 LDU_SKT: LD_UNLD;
END_SOCKETS
PLUGS
 LDU_PLG: LD_UNLD;
END_PLUGS
FBS
 MVC: XBAR_MVC;
END_FBS
EVENT_CONNECTIONS
  INIT TO MVC.INIT;
  MVC.INITO TO INITO;
  MVC.LOADED TO LDU_SKT.UNLD;
  LDU_SKT.LD TO MVC.LOAD;
  MVC.ADVANCED TO LDU_PLG.LD;
  LDU_PLG.UNLD TO MVC.UNLOAD;
  MVC.UNLOADED TO LDU_PLG.CNF;
END_CONNECTIONS
DATA_CONNECTIONS
  LDU_SKT.WO TO MVC.WI;
  LDU_SKT.WKPC TO MVC.LDCOL;
  MVC.WO TO LDU_PLG.WO;
  MVC.WKPC TO LDU_PLG.WKPC;
  VF TO MVC.VF;
  VR TO MVC.VR;
  DTL TO MVC.DTL;
  DT TO MVC.DT;
  BKGD TO MVC.BKGD;
  LEN TO MVC.LEN;
  DIA TO MVC.DIA;
  DIR TO MVC.DIR;
END_CONNECTIONS
END_FUNCTION_BLOCK
=====================================================================
```

Annexe G
(informative)

Attributs

G.1 Principes généraux

Des *attributs* peuvent être associés aux *types de données, variables, applications* et aux *types* et *instances* de *blocs fonctionnels, équipements, ressources* et leurs éléments constitutifs. Les attributs ont des valeurs qui peuvent être modifiées et accessibles en divers points du cycle de vie du type ou de l'instance de bloc fonctionnel.

Outre les descriptions des *algorithmes* de bloc fonctionnel, des informations complémentaires sont nécessaires pour prendre en charge un bloc fonctionnel au cours de la durée de vie de son logiciel. Ces informations peuvent être données en joignant des *attributs* aux éléments constitutifs des *types* ou *instances* de bloc fonctionnel.

Des attributs peuvent être appliqués à des éléments tels que les *types de données, les variables* et les *paramètres* qui sont utilisés pour la spécification des types ou instances de bloc fonctionnel. Les éléments de langage graphiques peuvent nécessiter des attributs complémentaires pour contenir des informations telles que la position, la couleur, la taille, etc.

Les attributs peuvent aussi être appliqués directement à des types et instances de bloc fonctionnel, par exemple pour contenir la version d'une spécification du type de bloc fonctionnel.

Certains attributs peuvent être utilisés pendant tout le cycle de vie d'un bloc fonctionnel. Par exemple, un attribut lié à une spécification du type de bloc fonctionnel peut être accessible lorsque le type de bloc fonctionnel est sélectionné dans une bibliothèque, lorsqu'une instance du type de bloc fonctionnel est interrogée, etc.

D'autres attributs peuvent n'exister qu'en certains points du cycle de vie. Par exemple, le texte définissant le but d'une instance de bloc fonctionnel particulière ne pourrait être appliqué que lorsque le bloc fonctionnel est instancié, et pourrait être modifié au cours de la vie de l'instance de bloc fonctionnel.

Certains attributs de bloc fonctionnel peuvent être installés dans des *ressources* associées et être accessibles au cours de la durée de vie de l'*application* distribuée. De tels attributs sont typiquement utilisés pour prendre en charge l'accès à des valeurs de paramètres de bloc fonctionnel par des équipements externes, par exemple, pour confiner à des limites de sécurité prédéfinies les valeurs des paramètres qui peuvent être fixées à l'aide d'un configurateur manuel.

G.2 Définitions des attributs

Une définition d'attribut donne les informations spécifiées dans le Tableau G.1. Chaque attribut a un nom et un type de données de sa valeur associée. Un attribut peut avoir une valeur par défaut qui sera utilisée tant qu'une valeur n'est pas donnée à un certain stade du cycle de vie du logiciel. Dans l'exemple donné en G.1, l'attribut `DESCRIPTION` a une valeur initiale de '' (c'est-à-dire: la chaîne vide) qui peut être écrasée en écriture par une description plus explicite lorsqu'une instance de bloc fonctionnel sera configurée ou même au cours de son utilisation active.

Les attributs eux-mêmes peuvent nécessiter des informations complémentaires à celles montrées dans le Tableau G.1. De telles informations sont appelées des *sous-attributs*.

Tableau G.1 – Eléments de définitions d'attributs

Elément	Exemple	
Nom	DESCRIPTION	
Type de données	WSTRING(30)	
Valeur par défaut	""	
Élément associé	**Types de bloc fonctionnel**	**Instances de bloc fonctionnel**
Usage	Configuration	Run-time (temps d'exécution)

G.3 Exemples

NOTE Les exemples ci-après sont donnés dans le but d'illustrer l'utilisation des attributs et ne doivent pas être considérés comme des définitions normatives d'attributs normalisés.

Un exemple d'*attribut de type de données* est:

- `Max_System_Value` – Cet attribut définit la valeur maximale prise en charge d'un type de données numérique. Il est appliqué au type de données générique `ANY_NUM` et, de ce fait, tous les types numériques tels que `INT` et `REAL` hériteront de cet attribut. Noter que chaque type de données spécifique aura sa propre valeur pour cet attribut et que les valeurs normalisées pour cet attribut pour un certain nombre de types de données sont consignées dans le Tableau E.1.

Les exemples d'attributs qui s'appliquent à des *variables* sont:

- `Diagnostic_Access` – Celui-ci détermine si, oui ou non, la valeur d'une variable est accessible par un système de diagnostic en cours d'exécution.
- `Write_Access` – Celui-ci définit le niveau d'accès requis pour modifier la valeur d'une variable, par exemple, `'Operator', 'System', 'Diagnostics'`.
- `Units` – Les unités dimensionnelles qui s'appliquent à une variable, par exemple, `'l', 'm/s', 'cm'`.
- `Usage` – Une description textuelle en plusieurs lignes de l'usage de la variable associée.

Des exemples d'*attributs de type de bloc fonctionnel* sont:

- `Usage_Class` – Celui-ci décrit l'usage général du bloc fonctionnel, par exemple, `'Input', 'Output', 'Control'`.
- `Version` – Celui-ci décrit le numéro de version de la définition de type de bloc fonctionnel, par exemple, `'1.2'`.
- `Help` – Une description textuelle en plusieurs lignes qui peut être accessible en divers points du cycle de vie.

Les attributs qui sont pertinents pour la programmation d'*algorithmes* en vue de l'*exécution* comprennent:

- `ExecutionTime` – Cet attribut, de type `TIME`, spécifie le temps du cas le plus défavorable pour l'exécution d'un *algorithme* particulier d'un *type de bloc fonctionnel* spécifié dans un *type de ressource* particulier.
- `Priority` – Cet attribut est associé à une *connexion d'événements* particulière au sein d'une *ressource* et peut être hérité du *type de ressource*. Cet attribut peut être utilisé par une ressource qui prend en charge le *multitâche* préemptif pour déterminer la priorité

d'*exécution* d'un *algorithme* invoqué par une *action EC* associée à un *état EC* qui est activée par un événement avec la priorité spécifiée.

G.4 Sources d'attribut

Les attributs peuvent venir des principales sources suivantes:

- Les attributs **implicites** tels que les *noms de types* de bloc fonctionnel, *noms d'instances*, *noms de variables* et leurs *types de données*, sont définis comme partie intégrante du processus normal de *déclaration* pour le bloc fonctionnel.
- Les attributs **normalisés** sont ceux qui sont requis comme partie intégrante d'une norme, tels que les versions de types de bloc fonctionnel, la plage maximale de paramètres, les descriptions de paramètres, etc.
- Les attributs **spécifiques à un produit** sont ceux qu'un vendeur de système a fournis, tels que les codes des produits des types de bloc fonctionnel, les adresses matérielles d'instances de bloc fonctionnel, etc.
- Les attributs **spécifiques à une application** sont ceux qu'un développeur de système spécifie pour prendre en charge l'utilisation d'un type de données ou d'un bloc fonctionnel particulier d'une application, tels qu'un identificateur d'instance supplémentaire de bloc fonctionnel pour adapter un style souhaité par un client, une valeur par défaut pour les paramètres de sortie, une variante de description de paramètres en langue nationale, etc.

G.5 Héritage d'attributs

Les éléments de bloc fonctionnel hériteront des attributs provenant d'éléments plus primitifs. Par exemple, une *variable* au sein d'une *déclaration de type de bloc fonctionnel* héritera des attributs de son *type de données* associé, et une *instance* de bloc fonctionnel héritera d'attributs du *type* de bloc fonctionnel associé.

Les *types de données* hériteront d'attributs en descendant dans la hiérarchie du type générique définie dans la CEI 61131-3. Par exemple, les attributs appliqués à `ANY_REAL` s'appliqueront aussi à `LREAL` et à `REAL`.

G.6 Syntaxe de déclaration

L'attribution d'une valeur d'attribut à un élément déclaré est semblable à l'attribution d'une valeur à une *instance* d'un *type d'attribut* dans laquelle l'instance a le même nom que le type.

La déclaration d'un *type* d'attribut utilise la même syntaxe que la déclaration d'un *type de données* tel que défini dans la CEI 61131-3, avec l'exception que les mots-clés délimiteurs sont `ATTRIBUTE...END_ATTRIBUTE` en lieu et place de `TYPE...END_TYPE`. Par exemple, la déclaration du type d'attribut `DESCRIPTION` dans le Tableau G.1 serait:

```
ATTRIBUTE DESCRIPTION: WSTRING(30); END_ATTRIBUTE
```

L'attribution d'une valeur à une *instance* d'attribut utilise la même syntaxe que celle pour l'attribution d'une valeur initiale à une *variable* telle que décrite dans la CEI 61131-3, avec les extensions suivantes:

a) le nom de l'instance d'attribut est le même que le nom du type d'attribut correspondant;

b) aucun type de données n'est spécifié pour l'instance d'attribut;

c) l'attribut de valeurs est enfermé dans la construction **pragma** définie dans la CEI 61131-3;

d) plusieurs attributions de valeurs d'attributs, séparées par des points-virgules, peuvent être incluses dans la construction pragma;

e) la construction pragma doit être placée de telle manière que la déclaration à laquelle elle s'applique puisse être déterminée sans ambiguïté.

Un exemple de l'application de ces règles est:

```
FUNCTION_BLOCK PID
     {DESCRIPTION:= "Proportional + Integral + Derivative Control;
      AUTHOR:= "JHC"; VERSION:= "19990103/JHC"}
     INPUT_EVENT
      INIT WITH QI, PARAMS; {DESCRIPTION:= "Initialization Request"}
     ...etc.
```

Bibliographie

CEI 60050-351:2006, *Vocabulaire Electrotechnique International – Partie 351: Technologie de commande et de régulation*

CEI 61131-5:2000, *Automates programmables – Partie 5: Communications*

CEI 61499 (toutes les parties), *Bloc fonctionnels*

CEI 61499-2, *Bloc fonctionnels – Partie 2: Exigences pour les outils logiciels*

IEC 61499-4:2012, *Blocs fonctionnels – Partie 4: Règles pour les profils de conformité*

ISO/CEI 7498-4, *Systèmes de traitement de l'information – Interconnexion de systèmes ouverts – Modèle de référence de base – Partie 4: Cadre général de gestion*

ISO/CEI 8824-1:2008, *Technologies de l'information – Notation de syntaxe abstraite numéro un (ASN.1): Spécification de la notation de base*

ISO/CEI 8825-1:2008, *Technologies de l'information – Règles de codage ASN.1: Spécification des règles de codage de base (BER), des règles de codage canoniques (CER) et des règles de codage distinctives (DER)*

ISO/CEI 10040:1998, *Technologies de l'information – Interconnexion de systèmes ouverts (OSI) – Aperçu général de la gestion-système*

ISO/CEI/IEEE 60559, *Information technology – Microprocessor systems – Floating-point arithmetic* (disponible en anglais seulement)

ISO 2382 (toutes les parties), *Technologies de l'information – Vocabulaire*

_____

INTERNATIONAL
ELECTROTECHNICAL
COMMISSION

3, rue de Varembé
PO Box 131
CH-1211 Geneva 20
Switzerland

Tel: + 41 22 919 02 11
Fax: + 41 22 919 03 00
info@iec.ch
www.iec.ch