ISO/IEC 18012-2

Edition 1.0 2012-07

INTERNATIONAL STANDARD

colour inside

**Information technology – Home electronic system (HES) – Guidelines for product interoperability –
Part 2: Taxonomy and application interoperability model**

ISO/IEC 18012-2

Edition 1.0 2012-07

INTERNATIONAL STANDARD

colour inside

Information technology – Home electronic system (HES) – Guidelines for product interoperability – Part 2: Taxonomy and application interoperability model

INTERNATIONAL ELECTROTECHNICAL COMMISSION

PRICE CODE T

ICS 35.200

ISBN 978-2-83220-192-3

CONTENTS

INFORMATION TECHNOLOGY –

HOME ELECTRONIC SYSTEM (HES) – GUIDELINES FOR PRODUCT INTEROPERABILITY –

Part 2: Taxonomy and application interoperability model

FOREWORD

International Standard ISO/IEC 18012-2 was prepared by subcommittee 25: Interconnection of information technology equipment, of ISO/IEC joint technical committee 1: Information technology.

The list of all currently available parts of the ISO/IEC 18012 series, under the general title *Home electronic system (HES) – Guidelines for product interoperability*, can be found on the IEC web site.

This International Standard has been approved by vote of the member bodies, and the voting results may be obtained from the address given on the second title page.

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

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

The widespread development of many national and regional home automation specifications, some standard and some proprietary, necessitates a mechanism for interoperability. Interoperability ensures that products from multiple manufacturers (potentially implemented using different automation systems) can interwork. It is desirable that devices needing to interwork do so seamlessly to provide users with a variety of integrated applications without modification of their protocols used within their specific system cluster. Examples of such applications include lighting control, environmental control, energy management, audio/video equipment control, and home security.

There are two fundamental methods to enable interoperability among applications developed for different communications protocols that use different application layers. These application layers may include different syntax and semantics for command, control, eventing and data.

- Method 1: Multiple gateways

  A communications gateway is intended to interconnect two different communications protocols. Therefore, to provide interoperability among three applications (A, B and C) that each use different protocols, gateways might be specified for:

  a) A ↔ B

  b) A ↔ C

  c) B ↔ C

- Method 2: A generic gateway

  Each application developer adds an interworking function (IWF) specified in this International Standard so that the application can communicate with other applications, regardless of the underlying communications protocol.

  d) A ↔ IWF

  e) B ↔ IWF

  f) C ↔ IWF

Interoperability is achieved via the IWF. For example, for application A and C to communicate: A ↔ IWF ↔ C. The IWF is a software-based generic gateway. This is a much less complex solution than Method 1. Application developers seeking interoperability using Method 1 develop translators (gateways) to each target applications. Developers using Method 2 implement only one IWF translator.

This International Standard provides a common classification and descriptive mechanism so that there is a common way of describing applications in any individual system, and an unambiguous mapping to key implementation items (e.g., data type primitives) to allow for transparent interoperability. Application-level interoperability cannot be achieved without being able to describe applications in a common form. The term “product interoperability” should be considered synonymous with application-level interoperability, since products are developed to implement and/or participate in (distributed) applications. The value of products to the end user derives from the applications which they support or provide.

The taxonomy specified here is based on application domain classification criteria for applications in home systems, as well as a lexicon of objects, events, properties, and primitive actions to effect or otherwise propagate change in the objects (their properties). This International Standard enables the specification and implementation of distributed application functions and services within the context of home electronic systems.

Work on this International Standard began with an in-depth review of the following existing systems, to understand the various application, interaction, and implementation models in use: ISO/IEC 14543-3-x *[network based control of HES Class 1)*, ISO/IEC 14543-4-x *[network enhanced control devices of HES Class 1],* ISO/IEC 29341 *[UPnP Device Architecture UPnP)],* ANSI/CEA-600 and ANSI/CEA-709 (also known as EN-14908). From that analysis, key similarities were identified among the various approaches and implementations. Those similarities are primarily in the high-level application functions that are being implemented, with differences appearing in the details of how the functions are represented. In short, there is a great deal of semantic similarity among various automation system application functions, but significant differences at the syntactic level.

That observation is the premise for the approach used in this International Standard. In order to facilitate interoperability, it is necessary to define an application interoperability model with the following characteristics.

- It allows a rich set of application functions, properties and interactions amongst distributed components contributing to the application to be clearly described in a common format.
- It incorporates a simple but flexible interaction model abstraction to represent all interaction models adopted by various system implementations (command/response, shared variable, message-oriented, etc.).
- It establishes the minimal set of common data type primitives to support unambiguous mapping of logical application data descriptions into implementation-specific binary representations within the interoperability domain.

Interoperability in distributed application systems is defined as the ability of two or more distributed components to communicate and to co-operate in predictable ways despite differences in implementation language, execution environment, or model abstractions. Three main levels of interoperability between components in a distributed application can be distinguished, as follows:

- protocol level, where the order of message exchanges and the constraints placed on either participant in the exchange (e.g. synchronous or sequential communications), are defined, alongside the resulting behaviour and possible blocking conditions. Interoperability at the protocol level provides the foundation to support the syntactic layer;
- syntactic level, where the names, interfaces and operation of the components are defined. Interoperation at this level is a necessary condition to support interoperability at the semanctic level;
- semantic level, where the meaning of the possible interactions between the components in the distributed application system is defined, in the sense of a defined/desired effect or output being generated.

Assuming that interoperability exists at protocol and syntactic levels, semantic level interoperability clashes between two application objects belonging to two different specifications but installed in the same premises and expected to co-operate are caused by differences in the HES-lexicon (conceptual schemas) that describe the components. Simply put, they may use different units for physical variable values and different names for objects, their properties and their functions, albeit they may be addressing the same application. Therefore, a mistranslation may occur between the two systems because of incomplete shared information. The possible clashes can be classified into two main groups, described below.

- *Lossless clashes* are those that can be resolved with no loss of information. Some examples in this category include component *naming clashes*, where the same component/information is represented by different labels; *structural clashes*, where information elements are grouped in different ways in different systems and *unit clashes*, where some scalar value (e.g. distance or temperature) is represented with different units of measure.
- *Lossy clashes*, which include interoperability clashes for which any transformation available, in either direction, causes loss in the information being communicated between the two application objects. These clashes are associated with differing levels of granularity, refinement or precision of the representation of the information. Note that a lossy translation between the two application objects may be an acceptable solution, provided that it achieves the desired application behaviour and does not affect the functional safety of the application or the system as a whole. One example of a lossy clash would be a light controller with a dimmer function. This controls a light actuator

(lamp); the light actuator understands only three levels of dimming, whilst the light controller supports up to 8 different levels. Any interoperability mapping between these two devices would require the mapping of the three levels recognised by the light installation to three out of the eight levels of dimming supported by the light controller.

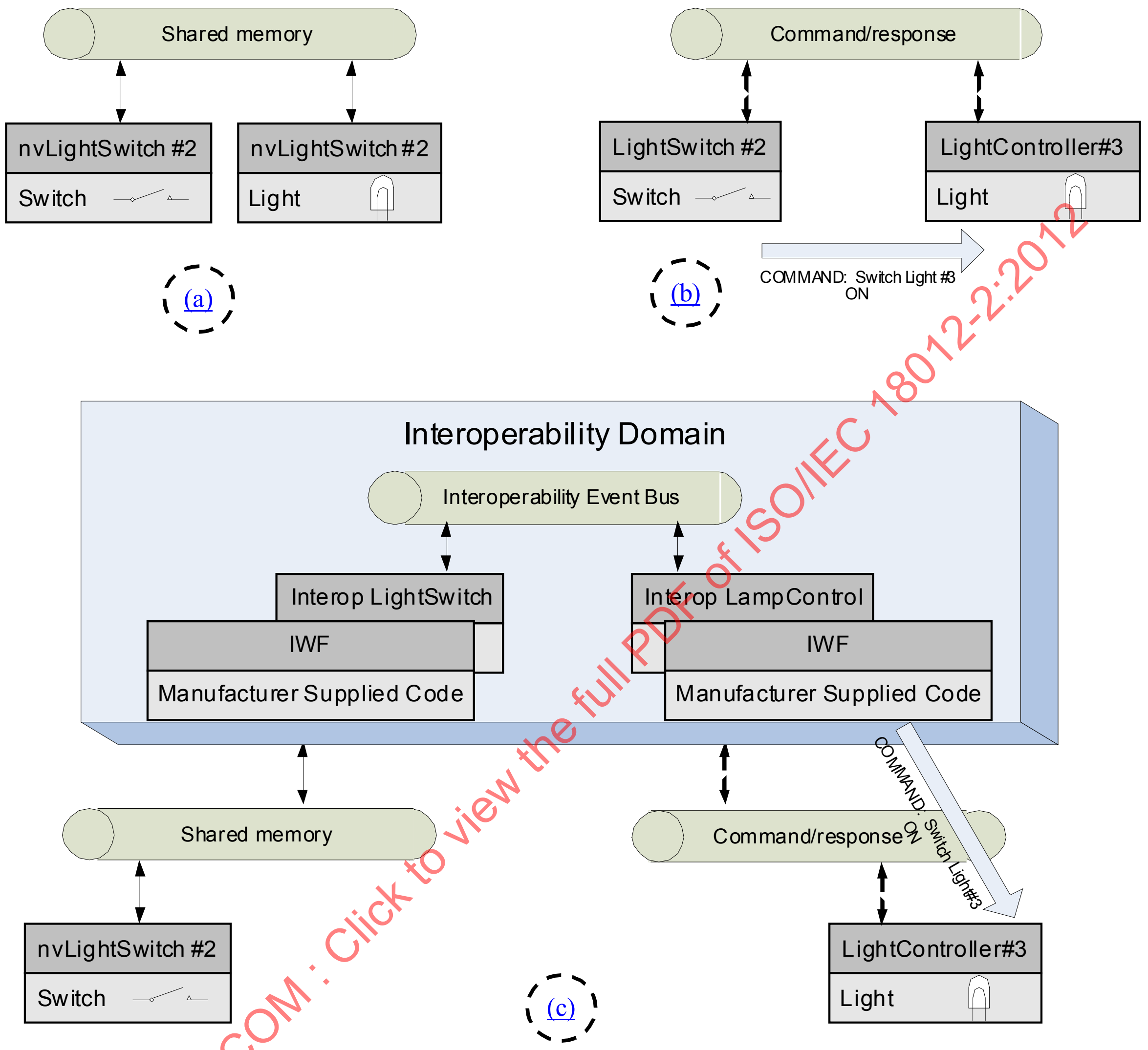


Figure 1 – Lighting application in (a) a shared memory system, (b) a command/response system and (c) an interoperating system

For example, one automation system might implement a shared variable space for communication between devices. In a simple lighting control example shown in Figure 1 (a), a user might turn a wall switch on, causing a shared variable in the switch device connected to the home system network to change from “0” (off) to “1” (on). A lighting controller component in the system might be subscribed to that shared variable, causing the automation system to notify the lighting controller of the change in the variable’s value (state). The controller could then take actions as defined by the configuration and programming of the lighting control application (in this case, switch the connected light on or off as requested).

In a complementary example, based on a command and control-based automation system, the wall switch might cause an “ON” command to be sent across the network in a message to the lighting controller component, which would then react appropriately (as per the description above).

In both examples, the behaviour of the lighting control application is the same, but the method used to implement it was quite different. This is an example of two systems having the same semantics (meaning and behaviour), but different syntax (implementation and codification or form).

This International Standard is based on the separation of the concept of action primitives from their actual implementation. An action primitive is a basic application action or device function that cannot be executed in part; it is either executed completely or not at all. A distributed application, or home electronic system aggregate function, is then thought as being provided through the execution of a sequence of such action primitives step by step, across the system. An example of an action primitive will be “Set temperature to 21 °C in Thermostat#21” (*Thermostat#21* is the name of an object.). To clarify the separation between action primitives and their implementation, let consider an example HES System-A, where devices/objects use <get>/<put> functions to implement local or remote reading and writing of variable values. This will not constitute “action primitives” in the context of this International Standard, but rather a programming mechanism used to invoke the application action primitives. These application actions are caused by <put>-ing (setting) the same variable to different values, which means that it is the variable and the value that it contains at a given time that define the application action, not specifically how the value is set (which, in this example, is through the use of a “PUT” message in System-A. This is captured later on in this International Standard by the introduction of eventing; objects (devices) notify each other with the values of their parameters, and each device (object) makes its decisions and takes actions (which contribute/constitute application actions) based on these values. During this processing each device may change these values; changes should be notified to all the interested parties. These *same actions* can be invoked in System-B by performing a (different) remote or local function call (i.e. not using <PUT>, but some form of a remote procedure call interaction). In this case it is possible for both systems to implement the same lexicon, and have the same application actions, but maintain (i.e. do not need to change) their own interaction mechanism and the corresponding protocols and syntax.

Typically, HES use different interaction modes, such as (distributed) shared memory/variable mode, command/response mode, remote procedure call mode, publish/subscribe mode (eventing) and variations of these. Using each of these gives the resulting home system certain characteristics, such as the ability to acknowledge correct execution of a remote operation (such as update of a state variable value, or the activation of a particular control). How to support a different set of operations using a common interaction model is well known in distributed control system design and implementation. However, it is beyond the scope of this International Standard to provide a detailed proof of the equivalence of such methods when translating between different interaction models, as shown in Figure 1.

The interoperable system is generic, and as such it should cater to different interactivity models (as described in the example above). It is assumed that any adaptation necessary from a system-specific interaction model to the eventing interaction adopted by the interoperability model is included in the implementation of the interworking functions provided by the manufacturers (or third party providers) interfacing *into* the interoperability model.

This interoperability standard addresses interoperability of products manufactured by manufacturers. These products can be system components in the context of a thermostat being a component of a heating system, or stand-alone as in a “device” that can collect information from such a thermostat device to be used in an application not related to heating control. The interoperability model in this International Standard addresses the requirements of two developer communities: the component developers (e.g., manufacturers), who develop individual devices and systems, and the solution developers (e.g., integrators or field installers), who provide one (or more) applications that use services provided by these components. The component developers (and manufacturers) will have a clear objective to provide a mapping onto to ensure that the services provided by their components are used outside the original single-specification system design. The solution developers will be able to choose and use from a wider range of components or even create new applications using services from components that are already installed and configured, provided they conform to this International Standard. The two communities overlap at least partially. For example, a company that develops automation devices would typically provide software routines that link their products to specific application objects. Such objects could be an application model the company defined and published or could be a set of standardised objects published as part of a standardised application interoperability model (AIM). In the former case, the object definitions would be considered to be standardised only upon their publication in an AIM registry of interoperable objects.

This International Standard comprises the following clauses:

- Clauses 1 through 4 are the scope, normative references, terms, definitions and abbreviations, and conformance clauses respectively;
- Clause 5 describes the application interoperability model;
- Clause 6 describes the interaction model in terms of an asynchronous event-bus model;
- Clause 7 describes the taxonomy used for inter-system and intra-system interoperability;
- Clause 8 describes the object schema framework;
- Clause 9 describes the application binding map schema framework;
- Annex A describes an example of an interoperable application specification;
- Annex B contains the base object schemas;
- Annex C contains examples of base object schema extensions;
- Annex D contains miscellaneous notes on interoperability and related taxonomy terms.

INFORMATION TECHNOLOGY –

HOME ELECTRONIC SYSTEM (HES) – GUIDELINES FOR PRODUCT INTEROPERABILITY –

Part 2: Taxonomy and application interoperability model

1 Scope

This part of ISO/IEC 18012 specifies a taxonomy and application interoperability model for the interoperability of products in the area of home systems. It also specifies an interoperability framework to allow products from multiple manufacturers to work together in order to provide a specific application. This standard describes an application process that exists above the OSI reference model (ISO/IEC 7498-1) stack, with sufficient detail needed to establish interoperable applications in this domain.

This International Standard is applicable to

- single implementation home electronic system networks, connected devices and applications. Such a system is created when all the home system components comply with a single standard or manufacturer specification,
- multiple implementation home electronic system networks, connected devices and applications. Such a system is created when different home system components comply with different HES standards or manufacturer specifications,
- automatically configured devices,
- manually configured devices,
- manually configured groups/clusters of devices.

This International Standard applies to application objects in operation within networks, between networks and to components located at the junction of dissimilar networks. Two (or more) dissimilar networks that conform to this International Standard, when linked by some communication system, are expected to behave as if both networks were logically the same network from an application perspective.

Interoperability considerations regarding general management processes in HES are described in ISO/IEC 18012-1. This part of ISO/IEC 18012 addresses only the management aspects related to the operation mode of interoperable HESs and does not cover the management processes of individual constituent networks.

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.

ISO/IEC 646, *Information technology – ISO 7-bit coded character set for information interchange*

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

ISO/IEC 18012-1, *Information technology – Home electronic system (HES) – Guidelines for product interoperability – Part 1: Introduction*

IEC 60050-714:1992, *International Electrotechnical Vocabulary – Chapter 714: Switching and signalling in telecommunications*

IEC 60559, *Binary floating-point arithmetic for microprocessor systems*

3 Terms, definitions, abbreviations and conventions

3.1 Terms and definitions

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

3.1.1
action primitive
fundamental unit of software invocation that results in a single defined and observable state change of the object on which it is invoked

3.1.2
application model
representation of the components, structure and interactions of a system focused on a particular domain of use

3.1.3
application interoperability model
AIM
application model specified in this standard

3.1.4
application programming interface
API
boundary across which application software uses facilities of programming languages to invoke services.

Note 1 to entry: See ISO/IEC JTC 1 Standing Document "Guidelines for API Standardization" for a complete discussion of application programming interfaces.

3.1.5
application semantics
component of the property descriptions of application objects

Note 1 to entry: See 7.4.3.

3.1.6
binding
specification of the source-end and the destination-end of an event connection between application objects

Note 1 to entry: See 6.3.

3.1.7
co-existence
two or more home networking systems co-exist when they can be used and operate without interfering with one-another

3.1.8
component
logical subunit of a larger, encompassing concept

3.1.9
concurrent events
two or more events queued for processing in the interval between any two scheduled input processing or output processing operations of an application object

3.1.10
device
distinct physical unit on a network

Note 1 to entry: It can either be an end node on the network, or an intermediate node (as in the case of a network gateway device connecting two distinct physical networks).

3.1.11
event
individual output of an application object, typically corresponding to a simple or complex state variable used in the application object

Note 1 to entry: See 6.4.

3.1.12
event bus
message path within the interoperability domain for transferring events between source and destination interoperable application objects

3.1.13
functional action
composite of one or more primitive actions from one or more application objects contained in a single functional object

Note 1 to entry: See 7.3.3.

3.1.14
functional class
collection of objects and actions on objects that model a particular application function within an application domain

3.1.15
functional object
logical grouping of some device functionality

Note 1 to entry: In this standard functional objects are used to model controller entities (hardware or logical devices) in a HES.

3.1.16
gateway
interface between dissimilar networks that may provide services up to OSI layer seven and above

3.1.17
HES gateway
specific gateway that provides protocol and language translation services above layer seven employing the ISO/IEC 18012 HES guidelines for product interoperability standards

3.1.18
interoperability
ability of logical entities to function together for applications on a network

3.1.19
interoperability
ability of two or more distributed components to communicate and cooperate in predictable ways despite differences in implementation language, execution environment, or model abstraction

3.1.20
interoperability domain
ID
logical space where interoperable objects seamlessly interact with one-another using an event bus

3.1.21
interoperability domain interface
IDI
software object that provides the translation between a generic interoperable application object instance and a corresponding network/system-specific counterpart

3.1.22
interworking
ability of two or more devices to support exchange of data and actions between them, having the same communication interface and application data types

3.1.23
interworking function
IWF
software module that provides syntactic and semantic translation services between a network/system device and its standardised interoperable representation (i.e., its interoperable application object)

3.1.24
HES lexicon
list of known HES application objects and structured information about them

Note 1 to entry: Lexicon in linguistics is an inventory of words and information about them. The term lexicon here is used to represent a listing of known HES application objects and structured information about them. This standard provides the structure to be used for the full compilation of the lexicon.

3.1.25
multiple implementation
mixed collection of two or more network implementations

Note 1 to entry: To establish interoperability, each network has a routing path to every other network in the system. This path may involve one or more hops through multiple intermediate networks.

3.1.26
network
distinct interconnection of devices that share a single physical layer implementation in terms of the OSI layered network model

Note 1 to entry: See ISO/IEC 7498-1:1994.

3.1.27
object
program or unit of software functionality

Note 1 to entry: This definition is similar to that traditionally used in object-oriented programming.

3.1.28
object binding map
information structure associated with multiple application objects that defines connections from event outputs to event inputs of the application objects

3.1.29
parameter
variable that is given a constant value for a specified application and that may denote the application

[SOURCE: IEC 60050-714:1992, 714-21-08]

3.1.30
product
device or network that may be purchased to make up a home electronic system (HES)

3.2 Abbreviations

API	Application Programming Interface
AIM	Application Interoperability Model
CM	Control Model
HAN	Home Area Network
HES	Home Electronic System
ID	Interoperability Domain
IDI	Interoperability Domain Interface
IWF	Interworking Function
OSI	Open Systems Interconnection

3.3 Conventions

Objects names consisting of two or more parts are written without space between the components to avoid confusion with normal technical English text.

4 Conformance requirements

A product supporting interoperability between devices from multiple manufacturers that claims conformance to this standard shall

- be designed according to the application interoperability model specified in Clause 5, and
- achieve interoperability by using an ID and IDIs with appropriate IWFs as described in Clause 5,
- support encapsulating software logic within application objects, as described in 6.2,
- support interaction between IWFs and the application objects, using IDIs as described in 6.2,
- support the definition of multiple bindings per application object event input or output, as described in 6.3,
- support application objects belonging to more than one binding map, as described in 6.3,
- support multiple output events generated concurrently from an application object as described in 6.4, and
- deliver those events concurrently to any other application object in the interoperability domain with bindings to more than one of those output events, as described in 6.4,
- use the taxonomy defined in Clause 7 when defining interoperable application objects and events.
- use the schema defined in Clause 8 when defining interoperable application objects, including:
  - conformance with the use of the base objects defined in 8.3 as the foundation for deriving specific application objects, and

  - assuring that all events either match data type primitives defined in 8.4, or derive from combinations of those primitives that define more complex data types,
- support a binding map mechanism that conforms to the requirements defined in Clause 9,
- design the binding map mechanism, so that it can specify logical event connections between application objects as specified in Clause 9, and
- define a logical event connection as the connection from an event output of an application object to an event input of an application object, as specified in Clause 9.

5 Application interoperability model

5.1 Overview

The application interoperability model builds on the generic functional-block structure typically adopted by different HESs. According to this structure, physical HES devices provide functions that can be classified into sensor, actuator or control functions. A device may provide, or carry out, one or more functions. Sensor functions process binary or analogue physical input signals and provide the result of that processing to other HES devices (or parts of the same device). Actuator functions process information received via their object interfaces from other functions and set binary or analogue physical output signals to act on their environment. Control functions process information received via their object interfaces from other functions and provide the result of that processing, via their object interfaces, to other functions. In this context, such functions form the building block of applications in HES. They are described through their properties, and are categorised typically in different application domains.

This International Standard describes the taxonomy to support the creation of a stable and flexible HES lexicon registry, and the mechanisms to support generic translation between different data formats and interaction models. Considering that different existing HES specifications and standards use different terminology for this functional-block approach, and to emphasise that this International Standard covers interoperability above the application layer of the Open Systems Interconnection (OSI) Reference Model, see ISO/IEC 7498-1, the application interoperability model is defined in a top-down approach.

The application interoperability model is based on the definition of one or more application domains, where each domain is identified explicitly in the HES object taxonomy (see 7.2.2). The application domain is used as a classifier, but may also form part of the object name to facilitate application configuration, access control and management.

Each domain is populated with application objects representing sensor, actuator and control devices, or related logical (software) objects that are combined to form HES applications. These application objects interact with one another by sending and receiving events across a logical event communication bus (the event bus). The event bus provides the communication services in the interoperability domain.

An application is formed as a collection of one or more application objects with specified bindings, and with a defined behaviour. A generic application example is a controller application object instance that receives sensor information from sensor application objects of a pre-defined type, and requests action from actuator application objects, again from a list of pre-defined types.

5.2 Application objects and interworking functions

The objects interact (receive information and request actions) by exchanging information as events on the event bus. Note that this view does not assume that a physical device (product) will be modelled as only one of these three classes (i.e. a product may be realised as a combined sensor/actuator/controller entity if the particular application requires it to be designed in that way). However, for the purposes of the interoperability framework each of the three functionalities will be presented by different objects when and if they are required to interoperate across systems belonging to different specifications and/or standards.

The application objects are programs running on individual devices within the interoperability domain (ID). For example a temperature sensor device has a corresponding application object that is called TemperatureSensor[1] that may be running on an HES gateway device. The association of interoperable application objects with actual devices is performed through configuration processes which are part of the interworking function (IWF) provided by the device manufacturer or an interested third party, and which utilize the functions and services of the particular network system on which the device resides. These configuration processes are outside the scope of this International Standard. Refer to ISO/IEC 18012-1, Clause 6 (Management) for a discussion of the general configuration requirements for this International Standard.

NOTE 1 Examples of these are the device address acquisition and maintenance or resource management processes (contention for gaining exclusive access to some object). Different systems define different management protocols and procedures for realising these. In this context, it is assumed that the propagation of the necessary actions to support the operational application object binding further down the protocol stack (for example, device address or network variable number), as well as the provisioning of the necessary conditions are fulfilled by separate management services not described in this part of ISO/IEC 18012. These functions are standardised or specified in the domain of the individual networks. As the interworking functions (IWFs) are used as the device interface proxy to the interoperability system and are normally provided by the manufacturer of the device (or an interested third party provider), these management functions are carried out within the IWF code and are not part of this International Standard (see also Figure 1).

NOTE 2 Further examples of management issues outside the scope of this International Standard include establishing and maintaining connectivity between devices hosting the distributed application objects independently of the fact that these devices belong to the same or different systems. Interoperability management allows for transparency to the native application management support services provided by separate, though interoperable, systems independently of the fact that these services are automatic or installer-based.

The interoperable application objects connect to the actual devices, they represent via the IWFs. When a new device is added to a particular network system, and is connected to other network systems through the ID, its IWF will be configured as mentioned above (this can be a manual or automated process, depending on the capabilities of the network system involved). The interoperable application object or objects that represent the new device in the ID shall have new instances initiated. The process of initiating those new application objects can vary between implementations of this standard, since there is no technical constraint which requires that the same process be used on all implementations to enable the ID to operate correctly. A manufacturer can choose to use a static definition of the application objects that will be initiated, or he can define a manual process to add and remove instances of interoperable application objects or can choose to implement an automatic process (for example, an IWF could initiate the required application object or objects after detecting that a new device has been connected to a particular network system). The preferred implementation would be an automatic process to allow the utmost flexibility for future evolution of the network systems involved (for example, network systems that currently require manual configuration of new devices may support automatic discovery in the future).

After one or more new interoperable application objects have been initiated within the ID they are available for use in interoperable applications by creating new binding maps or modifying existing binding maps to define bindings with the new objects. This is discussed in more detail in 6.3.

[1] Schema objects names typically do not include space to avoid confusion with English text.

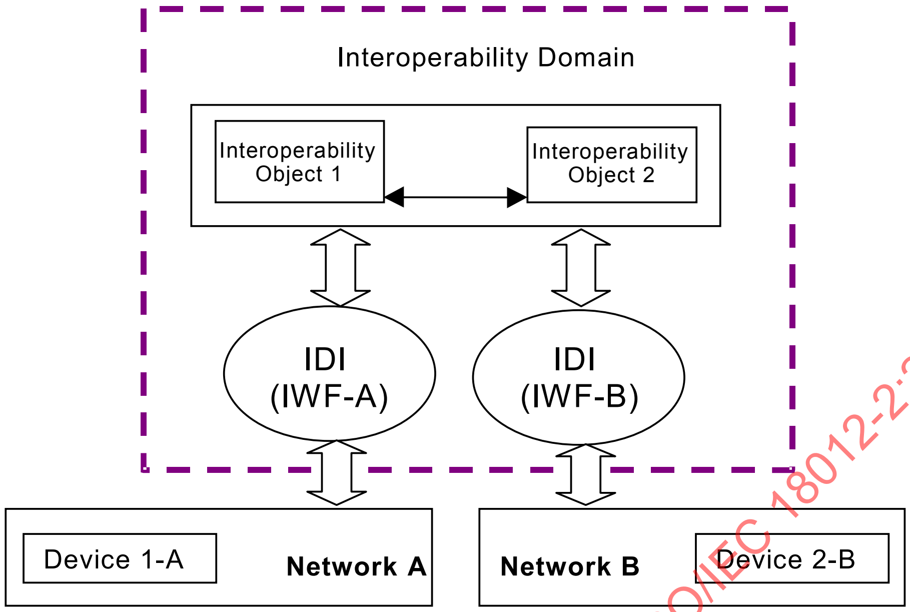


Figure 2 – Application interoperability model

Figure 2 shows an instantiation of the application interoperability model. Two devices (Device 1-A and Device 2-B) are installed in two different network systems, respectively Network A and Network B, and need to interact with each other to provide an interoperable application. The networks are linked together through a logical entity called the interoperability domain (ID). The ID connects with different network systems via a set of interoperability domain interfaces (IDIs). The IDIs are software entities containing software functions that provide any necessary translation between the generic interoperable object instance (e.g. Interoperability Object 1 within the interoperability domain) defined in this standard and the corresponding system-specific counterpart (Device 1-A in Network A). For example, they may perform data encoding translation from an 8-bit representation into a generic 16-bit representation of the value field of the specific object attribute. These translation services are called interworking functions (IWFs). The IDIs maintain the logical connection between the interoperable object instance within the interoperability domain and the corresponding device (or the system-specific counterpart). This includes maintaining object-state consistency between the interoperable object instance and the system-specific counterpart, as necessary. This is performed by the IWF.

Note that both Networks A and B may contain other devices. They will not be represented in the interoperability domain if these devices are not required to interwork in the context of this specification with devices on other different systems and/or networks.

The interoperability domain is a logical entity. It can be implemented on a single physical device/platform, distributed over several devices or its functionality may be integrated with other functional entities such as a HES gateway.

Note that this standard does not specify how two home systems simultaneously share (i.e. resolve possible contention for) a common resource or how to ensure that two home systems used within the same premises do not interfere with each other. However, note that it is required by ISO/IEC 18012-1 that two home electronic systems may share a common resource, and that they shall not interfere with one another.

6 Interaction model

6.1 Overview

An interaction model between the application objects in the interoperability domain is described here for passing application events across a logical event communication path or bus defined within the interoperability domain, known here as the event bus. Event passing is asynchronous, occurring at any time that the application objects require.

Events are originated by application objects. It should be noted that an application object is not required to produce outputs; some application objects might only be receivers of events.

Each application object can have multiple event inputs and event outputs. These correspond one-for-one to the input and output parameters of the function(s) of that application object (this standard follows the definition of input and output parameters given in IEC 60050-714:1992, definition 714-21-08). Therefore, the individual parameter values of an application object function(s) are passed across the event bus as separate events. This effectively breaks any tight association between the application object function interfaces (application programming interfaces) and the communication protocol used to move the parameters across the network. The event bus simply moves individual events from a source application object to a destination application object or objects, and is not dependent on future changes to the application object function interfaces. In other words, the event bus communication protocol is completely independent of the applications that it is supporting.

Event destinations are specified by a binding map that is part of the application interoperability framework definition. The binding map is used to define event connections (bindings) between application objects at the individual event level. These bindings are simply the declaration of a connection from an event output of an application object to one or more event inputs of an application object or objects. The binding map is generated as part of the installation and configuration process.

It would not be efficient, from a communication channel perspective, to send every individual application object function parameter as a discrete transaction on the event bus. Therefore, implementations of this standard should make use of both the application object schemas and the binding map information to enable efficient automated aggregations of multiple concurrent events. Concurrent events that have the same destination should be grouped into a single communication transaction on the event bus (e.g., aggregated into a single message payload, or a single memory transfer, etc.). Concurrent events are multiple event outputs from an application object that were generated concurrently. See 6.5 for further discussion of this concept.

This aggregation of events is independent of the application object function interfaces, and could even allow for the outputs of multiple application objects to be moved across the event bus as a single unit. Upon arriving at the destination, the grouping should be disaggregated into their individual events so that they can be brought to the appropriate application object inputs. This approach enables global system-level optimisation of communication bandwidth, rather than individual application-level optimisation.

6.2 Application objects

Application objects represent the operational components of an interoperable application. They are software representations of the sensors, actuators, control devices, and higher-level functions in a HES system that must interoperate.

Depending on their function, application objects can have zero or more event inputs and event outputs. An application object with only event outputs is considered a sensor application object. One with only inputs is considered an actuator application object. Control application objects have one or more event input(s) and one or more event output(s). For a complete description, see Clause 8.

Application objects encapsulate software logic that implements the desired interoperability function appropriate to the component they represent. For example, a temperature sensor application object in the interoperability domain will represent a corresponding temperature sensor device installed somewhere in the premises. The application object would receive temperature readings from the device in the format applicable to the specification or standard adopted by the device manufacturer, via an IDI using the IWF provided by the device manufacturer. The IWF will perform any necessary reformatting or translation on the data to put them in the form of the interoperable properties as defined in 7.4.2. The temperature sensor application object can then perform any additional interoperability processing as necessary before generating an output event on the interoperability domain event bus that contains the interoperable temperature data. This process is also illustrated in Figure 1 (c).

6.3 Binding map

A binding map corresponds to a single application in the interoperability domain. There will typically be many binding maps active, one for each application that is running.

The binding map defines the connections between the application objects that make up a complete application. A binding identifies the source-end and the destination-end of an event connection. See Figure 3 for a graphical depiction of this concept.

There can be multiple bindings from each application object event output, and there can be multiple bindings to each application object event input. In other words, an application object event output can be used as an input by more than one application object, and conversely an application object event input can receive events from more than one application object.

Binding maps may be short-lived or long-lasting, as required by the corresponding applications they represent. They may also be modified as required if devices or application objects are added, changed or removed from an application.

Also, application objects might be part of more than one application, and as such they might be included in more than one active binding map, as shown in Figure 3. In this simple example lighting controller B is part of both the manual lighting switch application and the motion-activated lighting security application.

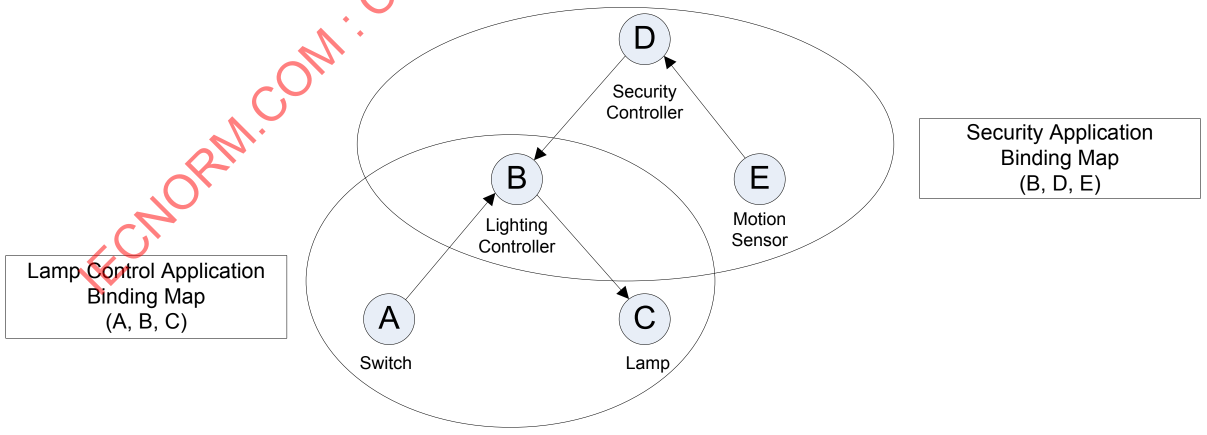

D
Security Controller
B
Lighting Controller
E
Motion Sensor
A
Switch
C
Lamp
Security Application Binding Map (B, D, E)
Lamp Control Application Binding Map (A, B, C)

Figure 3 – Binding map example

6.4 Events

Events correspond to the individual outputs of application objects, and as such the structure of their application-specific content are defined using the application object schema described in Clause 8.

It is a requirement that correct temporal associations be maintained between events passing through the interoperability domain.

NOTE 1 A possible implementation choice for maintaining temporal association is the use of timestamps on events within the interoperability domain, but other implementation approaches are also possible.

NOTE 2 One valid approach for application object interface implementation is to have a single output event, which could be a complex structure containing multiple data variables. In such an implementation temporal association is not an issue because at most one event is produced concurrently. Temporal association will continue to be required in situations where an application implementation requires multiple concurrent output events.

6.5 Event bus

The event bus provides the delivery mechanism for events to pass from an application that produces the event to any number of application objects that consume the event and potentially take some action. Events are asynchronous, occurring at any time.

The intent is that the event bus model be as simple as possible, allowing it to be implemented on the broadest possible set of underlying communication mechanisms. One implementation might be based on a shared memory for an interoperability domain that is completely contained in a single processing device such as a simple embedded gateway. A more comprehensive implementation could be a distributed event bus on top of a publish/subscribe transport spanning multiple processing devices (such as a collection of HES-Links or IDs, connected over a network), or any combination of such implementations – the event bus does not have to be implemented on a uniform underlying communication mechanism, and in fact is likely to span multiple underlying communication mechanisms as part of building an interoperable HES.

An important characteristic of the event bus is that multiple output events generated concurrently from a single application object shall be available for processing as input events concurrently by any application object that has a binding to two or more of those concurrent events. Figure 4 depicts an application described by a binding map between four application objects (A through D). For this example, assume that the events generated by Object A outputs o2 and o3 were produced at the same time (for example, concurrent sensor outputs in reaction to a change in the physical world). Object D inputs i1 and i3 are bound to those outputs from Object A, and must be delivered to Object D concurrently to maintain correct behaviour of the application.

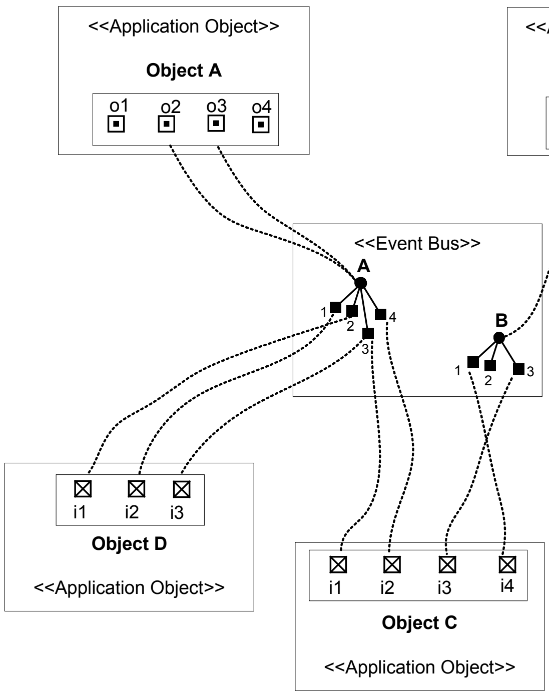


Figure 4 – Event bus example

7 Home electronic system application interoperability taxonomy

7.1 Classification

The HES application interoperability taxonomy is shown in Figure 5. For purposes of the interoperability specification the HES domain is classified into the following categories:

a) Application Domain;

b) Functional Object;

c) Application Object;

d) Functional Actions;

e) Property;

f) Property Primitive Actions.

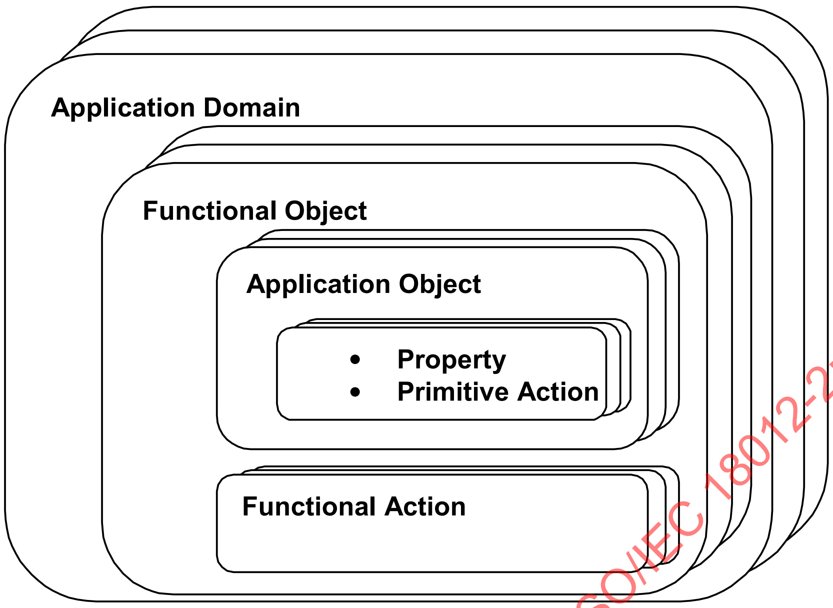


Figure 5 – Interoperable system taxonomy

Application Domains are specified in terms of Functional Objects, which represent logical grouping of some device functionality. Each Functional Object is specified as a collection of Application Objects (as well as, possibly, objects related to its configuration) that represent application atomic state and functions. This specifies the semantics of the applications and the components of the Functional Object. Objects in turn are specified as collection of properties, which define the content representation for storage, transport (syntax), and interaction purposes.

The classification of the HES components into Functional Objects, Application Objects and Properties, as logical entities in the taxonomy, is designed to allow for separation of the syntactic and semantic aspects of the interoperability. Functional Objects model controller entities (hardware or logical devices) in a home electronic system. Application Objects model sensor and actuator entities. Properties are used to specify the syntax, i.e. the data representation of the object information components. This allows for a direct mapping (data type translation and matching) between properties as specified in this standard and those specified in other specifications through the use of IWFs.

Objects, in general, represent the semantics of the application, i.e. they represent behaviour, which is exhibited through the interaction with other objects by invoking primitive actions on the properties. The use of both Functional Objects and Application Objects allows the interoperability, through translation in the IWF, to cater for both variable-oriented and device-oriented HES specifications. The separation between the semantics (object level) and the syntactic representation of the data (property) allows for flexibility in the definition of the interoperability IWFs. In the simplest case, Application Objects can be used to present a single network variable, and allow the modification of the processing of its value for representation into the Interoperability Framework space.

As an example, a Functional Object named TemperatureSensor may have an Application Object called RoomTemperature of <Temperature> data type, which is measured in degree Celsius (semantic). The Application Object in the interoperability specification is defined as having a Property called "TempValue", which is of Property unit type "Temperature" that follows the IEC 60559 specification (“float” Property data type syntax).

Detailed definitions of the framework components are given in the following subclauses.

7.2 Application domain

7.2.1 Definition

An application domain serves as context-setting description. It groups together functions that interact in semantically related applications. A non-exhaustive list of application domains is given in 7.2.2.

Application domains are enumerated entities. As part of the taxonomy they may be used in word form or represented by the corresponding enumerated value, to form part of the object name for application binding.

7.2.2 Application domain list

The application domain list provided in Table 1 provides an enumerated list of application domains. The list is not exhaustive and will expand as additional domains are identified for inclusion.

Table 1 – Application domain list

	Application domain name	**Description**
1	General	This domain groups together functions that provide generic services to other entities in a home electronic system. Examples include Time/Date, Location, User Interface, etc.
2	Audio/Video	This domain groups together functions related to the control of the distribution and consumption of audio and/or visual content in a home electronic system. Examples include Audio Amplifier, Tuner, Video-display, Speaker, etc.
3	Lighting	This domain groups together functions related to the lighting environment control in a home electronic system. Examples include Light, Light-sensor, Light-switch. etc.
4	Communications	This domain groups together functions related to the control of the communication session set up and distribution/transfer around a home electronic system. Examples include an Intercom, DataPoint, etc.
5	Heating	This domain groups together functions related to the control of heating sub-system, or parts thereof, of a home electronic system. Examples of functions include RoomController or ZoneController, TemperatureSensor, etc.
6	Ventilation	This domain groups together functions related to the control of environmental sub-systems (air-conditioning and ventilation). Examples include TemperatureSensor, HumiditySensor, VentilationZoneController, etc.
7	Utility	This domain groups together functions related to the management and/or monitoring of utilities such as electricity, water, gas, heating. Examples include UtilityMeter, LoadController, etc.
8	Security	This domain groups together functions related to the management and/or monitoring of the security sub-system. Examples include SecuritySensor, WindowSensor, SecurityZoneController, etc.
9	Appliance	This domain groups together functions related to the management and/or monitoring of domestic appliances. Examples include Washer, TumbleDryer, Cooker, Oven, …

7.3 Functional object

7.3.1 Definition

A functional object is a collection of objects and actions on objects that models a particular application function within the application domain. An instantiation of a functional object is a logical device, i.e. a software entity that provides some specific standardised control and/or information functionality in the HES in which it is installed. Functional objects model controller entities (hardware or logical devices) in a home electronic system. Functional objects form the basis of the classification in this standard.

7.3.2 Functional object structure

A functional object has properties and functional actions. Functional actions are specified as (possibly aggregate) actions on individual application objects that lead to a functional operation of the logical device in the system. For example, scene setting requires a series of (inter)actions with several sensor and actuator objects (lighting, communication, AV, etc.); a scene setting controller can be modelled as a functional object that has a functional action composed of actions on individual application objects modelling devices as described above. A functional action has contextual meaning, and is specified together with the functional object. The invocation of a functional action leads to a specified series of invocation of component (application) objects, and results in moving the application objects into a specified, and verified, end state, where applicable.

A functional object describes the application layer interface, including input and output operation objects, configuration properties, default and power-up behaviour required on devices for specific commonly used control functions. An instance of a functional object is implemented in a physical device as (part of) the application.

An instance of a functional object is a logical device. A physical device may implement more than one logical device when the physical device implements multiple functional objects.

7.3.3 Functional action

Functional action allows for modelling of functionally complex behaviour within a functional object (i.e. in a specific context). Functional actions are enumerated entities within a functional object, and have meaning only within that object.

NOTE An example of a functional action is <SwitchOn> within a LightController functional object.

Invoking a functional action means to invoke at least one primitive action on at least one of the properties in at least one application object, local or remote.

7.3.4 Functional object list

The functional object list consists of all defined functional classes for each application domain.

7.4 Application object

7.4.1 Definition

Application objects are self-describing actionable, locally addressable, entities that contain one or more properties. An application object models, through its properties and primitive actions, the most generic behaviour of sensors and actuators in a home electronic system. An object contains basic descriptive properties, configuration properties, and operational properties.

7.4.2 Application object structure

Application objects are collections of one or more properties. Each property is an internally structured self-describing entity that encapsulates some information source (in sensors) or control (in actuators) variable.

Every application object has a self-description property and an object instance identifier property.

The object instance identifier can be instantiated dynamically, or seeded with a (default) value specified in the published generic application model.

7.4.3 Property

A property is an atomic actionable structured data entity that contains a single variable. The structure of a property is given in Table 2 below. They are presented in a simple data format, and are characterised by a data type, methods, and information flow direction (in, out, in/out).

The property methods defined implicitly are given in 7.4.6.

Table 2 – Property structure

Property		**Notation**
Description	Property_Name	Character String
	Property_Action_List	Array of String
Coding	Property Data Type	[Array of] Primitives from 7.4.4
	Number of elements	Unsigned Integer
Value	Value	
	Default Value	
	Value Range	
Application Semantics	Property Unit Type	
	Support (**M**andatory / **O**ptional / **C**onditional)	
	Access (**In** / **Out** / **IO**),	

7.4.4 Property data type primitives

The following data type primitives are used.

- Character string (ISO/IEC 646)
- Boolean
- Integer (Integer8, Integer16, Integer32)
- Unsigned integer (UInt8, Uint16, Uint32,Uint64)
- Float (IEC 60559)
- Date
- Time (and Date/Time)

7.4.5 Property unit type primitives

These are semantic data units, for example degree kelvin for temperature semantic data type.

These are grouped into basic data units and derived data units.

The basic data units are given in Table 3 below, and outlined in the XML schema file in Clause B.2. Using this set of unit type primitives, any physical unit type can be derived. Derived data units will be listed and maintained in the lexicon registry as outlined in Table 3.

NOTE The information in Table 3 is intended to be available online through an ISO/IEC recognized registration authority.

Table 3 – Property unit type primitives

Property unit type	Unit
Length	Meter
Time	Second
Temperature	Kelvin
Mass	Kilogram
Electric current	Ampere
Substance amount	Mole
Luminous intensity	Candela

7.4.6 Property action primitives

The actions allowed on the properties are

- GET: the value of the property being queried is returned,
- SET: the value of the property being accessed is set to the given value,
- EVENT_REPORT: the value of the property is reported to the subscribed entities,
- SUBSCRIBE: the operational object updates the list of subscribed entities (objects) for the given/required property. If the list is empty, the property changes are not reported.

NOTE The property action primitives defined here refer to logical functionality of an application object and represent the minimum set of application-independent primitives required for the interoperability domain (ID) to interact with an application object's properties. These terms should not be interpreted as references to remote communication protocols such as get/put in HTTP, or the subscribe function of a distributed publish/subscribe message protocol.

7.4.7 Object types

Objects can be classified into two groups: application objects and application management objects (configuration). Only the application objects are addressed by this International Standard.

The object schema specifies the type of object.

8 Object schema

8.1 Descriptive methodology

The principal descriptive technology used in this standard is Extensible Markup Language (XML) Schemas. XML Schemas are used to describe the interoperability domain (ID), including the description of interoperable application objects, input and output events and object binding maps. XML Schemas are not used for the encoding of information for transmission across the event bus (i.e., they are not used for encoding as part of the ID wireline protocol).

8.2 Overview

The object schema design reflects the interaction model described in Clause 6 and emphasises describing the input and output events of objects. This includes the names, the data types, and optionally other characteristics about the inputs and outputs. These other characteristics are an open-ended set that will take the form of metadata elements that extend the input and output elements in the schema, an example of which might be maximum latency requirements on inputs.

The object schema shall also reference the executable logic that implements the object's functions. This reference is implementation-independent, and needs to be able to support any execution platform and language. The purpose is to enable an interoperability framework implementation to invoke any executable logic for an object, and present the input events that are pending (i.e., that have arrived for the object from the event bus).

There can also be additional, global metadata associated with an object. This might be for configuration parameters affecting the behaviour of the object (versus the individual input or output metadata described above).

8.3 Base objects

8.3.1 General

There are three types of primitive object types in the schema: sensors, actuators, and control objects. Although these names imply very different types of objects from a conceptual perspective, the functional difference between the three object types in this schema is simply that sensors only have event outputs, actuators only have event inputs, and control objects have both event inputs and outputs.

NOTE There can be other input and output settings for any of these objects, such as the ability to set or change metadata characteristics to adjust the object configuration, but these would be separate from the asynchronous application event flows. For example, object settings could be managed through a synchronous call interface to the object.

Using these three basic building blocks of sensors, actuators, and control objects with asynchronous input and output events, connected together across an event bus, it is possible to construct a description of any automation application or function.

Events do not contain any explicit identification of the type of object that generated the event. This means that any object with an output can be bound to any object with an input. Importantly, it means that control objects can provide inputs to other control objects (i.e., inputs do not have to come from sensor objects). This provides the basis for creating sub-automation components in an application description, allowing groups of sensors, actuators, and control objects to appear conceptually as sensor inputs or actuator outputs to higher level control objects in a complex application.

The intention is that specific types of sensor, actuator, and control object schemas will be derived from the base object schemas, and those specific types can then be further derived into vendor-specific variations, if needed. For example, a temperature sensor object schema can be derived from the base sensor schema, defining a single output of type Fahrenheit degrees. A vendor might then further refine the temperature sensor schema with vendor-specific characteristics such as additional features. These can be described by defining global configuration properties or perhaps additional outputs.

The base object XML schema definitions are shown in Annex B.

8.3.2 Control objects

Control objects have any number of inputs and outputs. Conceptually, they are the objects that will contain the core functions of an application such as the decision logic, the control loops, etc. This subclause will describe the content of the base control schema. The following subclauses discuss the sensor and actuator schemas.

The block diagram in Figure 6 shows the basic structure of the control object schema. The commons section is shared between the sensor, actuator, and control object schemas.

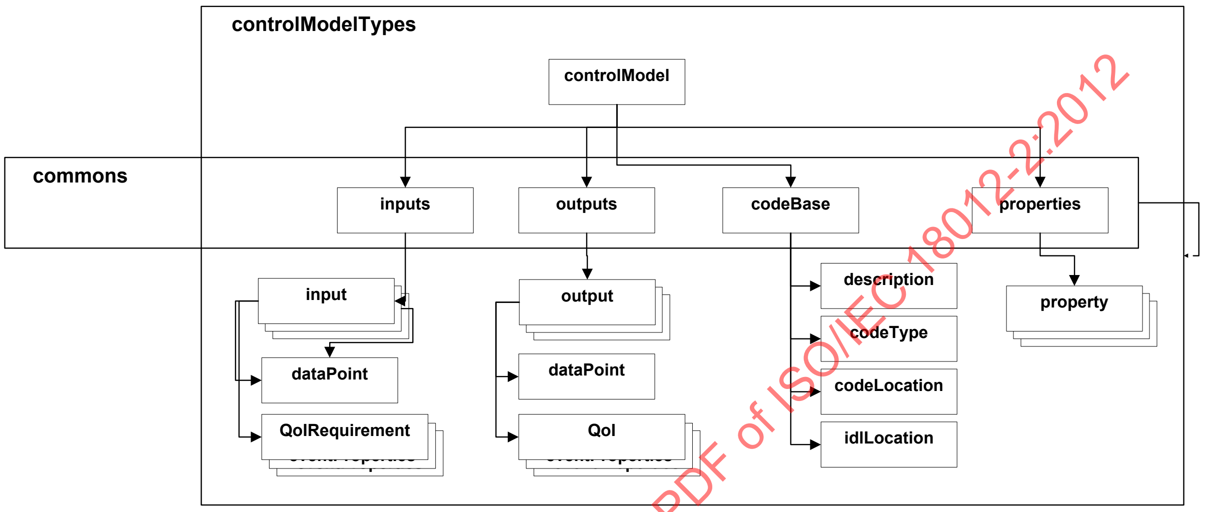


Figure 6 – Object schema structure

The *commons* section describes the basic structure of all the objects: inputs, outputs, global object properties (such as configuration settings, etc.), and a reference to the executable code that implements the logic of the control object. The *commons* section schema is listed in Clause B.1.

The inputs section of the schema is comprised of one or more input elements.

```
<xsd:element name="input">
   <xsd:complexType>
      <xsd:sequence>
         <xsd:element ref="dataPoint"/>
         <xsd:element ref="QoIRequirement" minOccurs="0"/>
      </xsd:sequence>
   </xsd:complexType>
</xsd:element>
```

It consists of a dataPoint element, which decribes the data type of an input event, and a QoIRequirement or Quality of Information Requirement element, which is used to define other characteristics of the input events, examples of which might be precision, frequency, etc. These are considered metadata that describe things about the input event, rather than describing the event itself.

The outputs section is similar in structure and content, with the exception that instead of having QoIRequirement metadata, it refers to QoI or Quality of Information metadata, which define other characteristics of the output events, and are suitable for matching or validation against input QoIRequirements.

Both the QoIRequirement and the QoI metadata are open-ended sets of information, and will be expanded as needed to support different application requirements.

The codeBase section of the schema is:

```
<xsd:complexType name="codeBase">
  <xsd:sequence>
    <xsd:element name="description" type="xsd:string" minOccurs="0"/>
    <xsd:element name="codeType" type="codebaseTypes" default="JAVA"/>
    <xsd:element name="codeLocation" type="xsd:string" minOccurs="1"/>
    <xsd:element name="codeProperties" type="properties" minOccurs="0"/>
    <xsd:element name="idlLocation" type="xsd:string" minOccurs="0"/>
  </xsd:sequence>
</xsd:complexType>
```

The codeBase section is optional and provides the elements needed to define an unambiguous reference to the executable software that implements the control object's application logic, such that the interoperability framework runtime system can invoke the executable software. The intention is that any type of executable environment can be referenced, such as C runtime code, Java code, rules engines, etc. The interoperability framework implementation would need to include appropriate language bindings for whatever runtime environments are supported (for example, the implementation might need to use the correct stack structures for invoking C executable modules in memory).

The codeLocation element would typically be a Uniform Resource Locator (URL) reference to the executable logic. This could be a dynamically referenced library routine or other type of runtime code reference.

The idlLocation element can be used to reference an optional Interface Description Language (IDL) description if needed as part of the implementation. This would typically be used to provide additional details required by the implementation to support language binding mismatches.

The properties section of the schema is comprised of zero or more property elements.

```
<xsd:element name="property">
  <xsd:complexType>
    <xsd:attribute name="name" type="xsd:string"/>
    <xsd:attribute name="value" type="xsd:string"/>
  </xsd:complexType>
</xsd:element>
```

It can be used to define properties that apply to the control object in general, rather than to individual input or output events. A property element consists of a name/value pair. This could include configuration parameters, calibration information, version tags, etc.

8.3.3 Sensor objects

Sensor objects are based on the commons section, but only have definitions of the outputs, codeBase, and properties components of the commons.

8.3.4 Actuator objects

Actuator objects are based on the commons section, but only have definitions of the inputs, codeBase, and properties' components of the commons.

8.4 Data type primitives

The purpose of the data type primitives in this standard is to support implementation-level interoperability across dissimilar automation systems that are compatible at a semantic level. In other words, to support interoperability of systems that perform the same application functions, but which are implemented differently in terms of communication mechanisms, data description methods, and other syntactic details.

The key requirement is that each property unit type listed in 7.4.5 shall map unambiguously to a single data type primitive in 7.4.4. This provides an unambiguous mapping to the underlying execution environment (for example, a C language environment), and allows unambiguous conversion between property unit types in different automation systems (for example, one system may have a temperature sensor that outputs Fahrenheit degrees in floating point representation in C, but it may need to interoperate with a control object that expects a Celsius degree input in integer representation in Java).

The baseTypes schema in Clause B.2 defines the property unit type primitives. Using these physical property unit type primitives, any physical unit can be derived (Clause C.1 contain examples of such derivative definitions). Because an implementation of this Standard shall have an unambiguous mapping from each unit type primitive (i.e., base type) to a single data type primitive in the underlying implementation programming language (e.g., integer32), the runtime representation for all unit types (base or derived) is well-defined, and can therefore support automatic conversion between unit type variations (such as between Fahrenheit and Celsius).

9 Overview of application object binding map schema

The object binding map defines the event input/output connections between the sensor, actuator, and control object input and output events of an application. A binding between two application objects shall name the two objects involved, and also the specific input and output events that are being connected.

An example of a portion of a binding map is

```
…
      <binding source="cm02" target="cm01">
            <iomapping from="o1" to="i1"/>
      </binding>

      <binding source="cm05" target="am01">
            <iomapping from="o1" to="i1"/>
      </binding>

      <binding source="sm02" target="cm06">
            <iomapping from="o1" to="i1"/>
            <iomapping from="o2" to="i2"/>
      </binding>
…
```

In this example, output o1 from control object cm02 is bound to input i1 of control object cm01 in the first example (<binding source="cm02" target="cm01">…). The next clause binds output o1 of control object cm05 to input i1 of actuator object am01, and finally the last clause binds outputs o1 and o2 of sensor object sm02 to inputs i1 and i2 of control object cm06, respectively.

As shown in the example above, it is the individual input and output events that are being bound together over the event bus, and not the sensor, actuator, and control objects. In other words, the binding is at the individual input/output event granularity, and not at the object granularity. This provides a great deal of detail to the interoperability runtime implementation, allowing such optimisations as dynamic message payload construction on the event bus communication links between interoperability domain interfaces (IDIs). Depending on the specific subset of events that need to move from one IDI to another across the event bus, the interoperability framework runtime implementation can assemble message payloads that include all the events that will be transported across the event bus to another device implementing the interoperability domain at a specific point in time (rather than transporting all output events from a sensor, actuator, or control object across the event bus communication links).

Annex A
(informative)

Example of a lighting application interoperability specification

A.1 Overview

This example shows how interoperability can be achieved between two components of a lighting control application residing on two different systems, System A and System B, using the interoperability domain (ID) specification of this standard. The particular communications protocols chosen are labelled generically to focus on the ID features of this standard.

In the following clauses it is assumed that

- Network-A is an implementation of ISO/IEC 14543-3 (KNX),
- Network-B is an implementation of ISO/IEC 29341 (UPnP),
- the interoperability domain implementation is unspecified.

A.2 Procedure

The procedure followed in this example consists of defining a generic lighting subsystem application model (simple case). The components of the system are installed on two different networks: System A and System B. The components of the framework are exemplified in (a) logical interoperable system, shown in Figure A.1 and discussed in Clause A.3, (b) respective functional/device profiles in each system, described in A.4.1 for System A, and in A.5.1 for System B, and (c) examples of the IWFs for each case, as discussed in A.4.2 for System A, and in A.5.2 for System B.

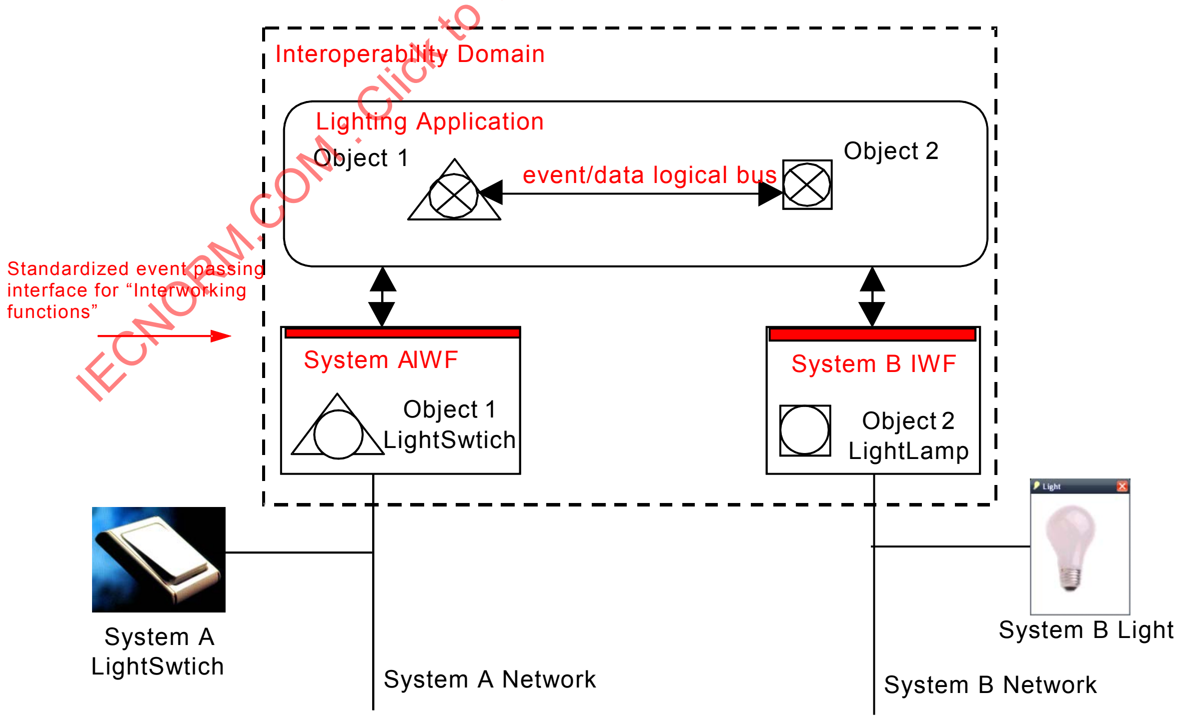


Figure A.1 – Lighting application of logical components

A.3 Generic lighting subsystem logical application model

A.3.1 Functional classes

The examples will include partial XML schemas for the individual components, and exemplify the interoperability lexicon as outlined in this standard.

A.3.2 LightSwitch

The LightSwitch object represents a switch device that includes a functional object, BinarySwitch. The BinarySwitch object has two operational objects: SwitchState and SetSwitchValue, see Figure A.2.

```
<?xml version="1.0" encoding="UTF-8"?>
<controlModel id="CM001"
         name="BinarySwitch"
         xmlns="http://<namespace URL>"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://<schemaLocation URL> controlModel.xsd">
   <category>BinarySwitch</category>
   <inputs>
      <input>
         <dataPoint id="i_01"
                 name="SwitchState"
                 xsi:type="Boolean"/>
      </input>
   </inputs>
   <outputs>
      <output>
         <dataPoint id="o_01"
                 name="SetSwitchState"
                 xsi:type="Boolean"/>
      </output>
   </inputs>
   <!—In this example System A's implementation does not require the use of the
   codebase tag -->
</controlModel>
```

Figure A.2 – LightSwitch object

A.3.3 LightLamp

The LightLamp object represents a lamp device that includes a functional object, PowerSwitch. The PowerSwitch object has two operational objects: SetSwitchState and SwitchState, see Figure A.3.

```
<?xml version="1.0" encoding="UTF-8"?>
<controlModel id="CM002"
        name="BinarySwitch"
        xmlns="http://<namespace URL>"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://<schemaLocation URL> controlModel.xsd">
 <category>PowerSwitch</category>
   <inputs>
      <input>
         <dataPoint id="i_01"
                name="SetSwitchState"
                xsi:type="Boolean"/>
      </input>
   </inputs>
   <outputs>
      <output>
         <dataPoint id="o_01"
                name="SwitchState"
                xsi:type="Boolean"/>
      </output>
   </inputs>
   <codeBase>
      <codeType>JAVA</codeType>
      <codeLocation>com.LightingCompanyB.Powerswitch</codeLocation>
      <codeProperties>
        <property name="Model_Number" value="SWT_KNX_01"/>
        <property name="A" value="2.0"/>
        <property name="V" value="110~220"/>
      </codeProperties>
      <idlLocation>http://wwww.LightingCompanyB.com/Powerswitch.wsdl</idlLocation>
   </codeBase>
</controlModel>
```

Figure A.3 – LightLamp object

A.4 System A

A.4.1 LightSwitch functional profile

The application objects LightSwitch and LightLamp in the interoperability domain are interconnected by exchanging switching events (LightLamp: direction in) and status events (LightLamp: direction out) through the event bus, see Figure A.4.

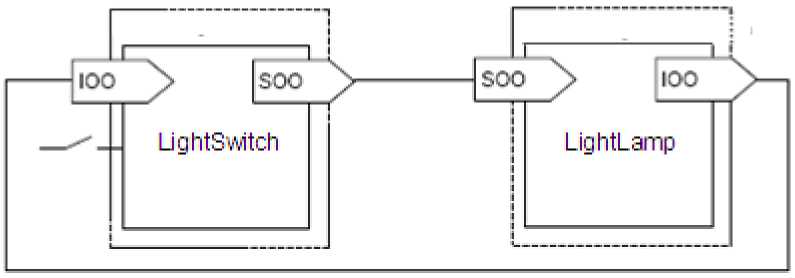
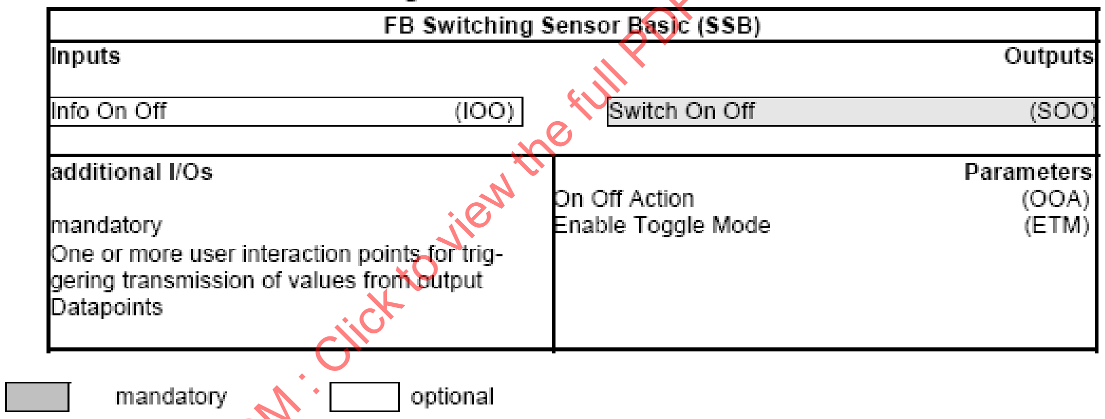


Figure A.4 – System A lighting example diagram

In Figure A.4, the application objects represent the LightSwitch in Network-A and the LightLamp in Network-B, respectively.

The functional block (FB) switching sensor basic is used in the application domain lighting to provide input data to switching actuators. It specifies the functionality, for example contained in a switch or a push button, to switch the switching actuator on or off.

Display elements can be integrated to show the status of the switching actuator, see Figure A.5.



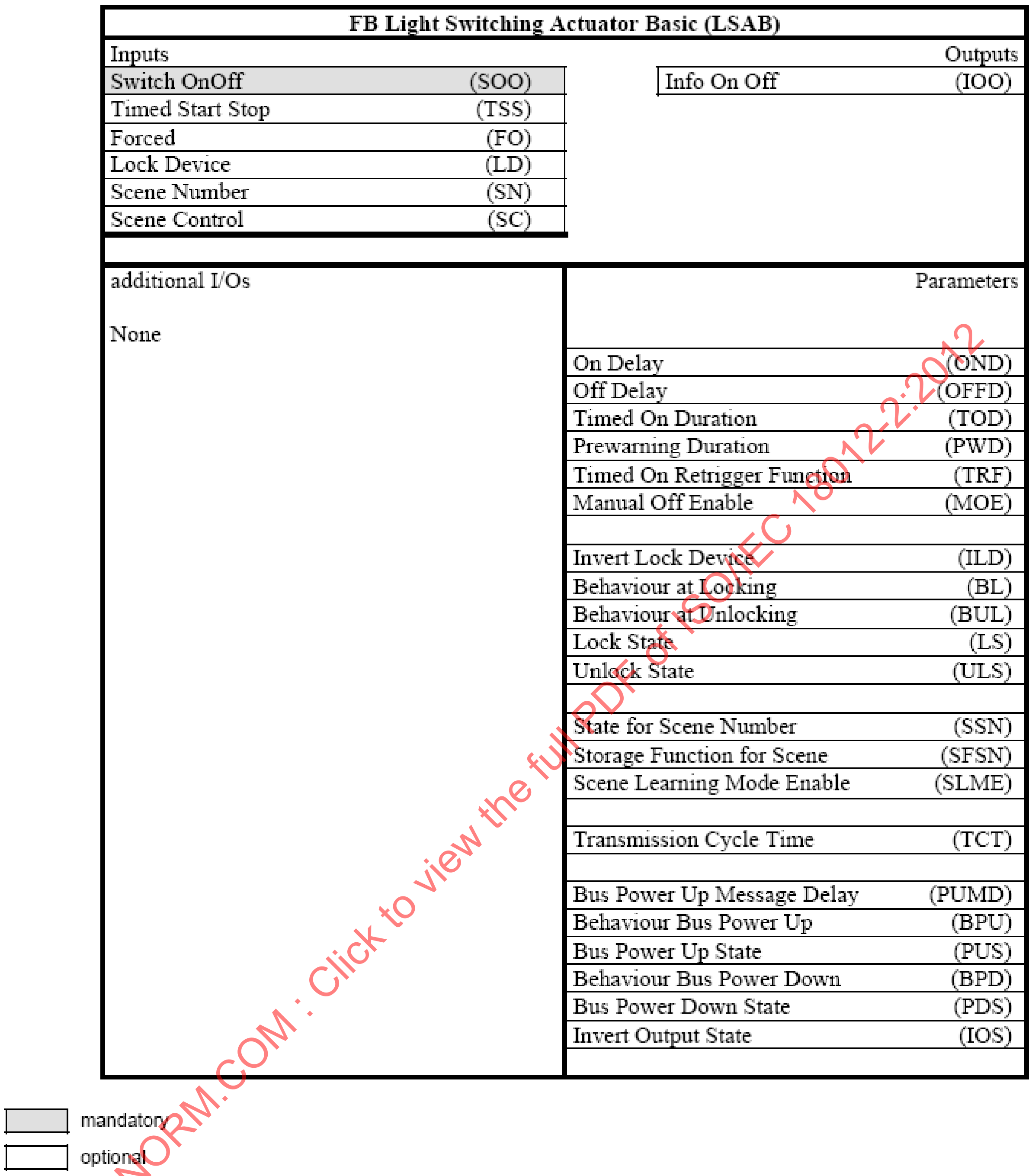
FB Switching Sensor Basic (SSB)	
Inputs	Outputs
Info On Off (IOO)	Switch On Off (SOO)
additional I/Os mandatory One or more user interaction points for triggering transmission of values from output Datapoints	Parameters On Off Action (OOA) Enable Toggle Mode (ETM)

▨ mandatory ☐ optional

Figure A.5 – Functional block diagram for FB switching sensor basic

In Figure A.5, Switch On Off (SOO) is mandatory and is used in this example to turn the lamp on or off. Info On Off (IOO) is optional and is used in this example to present the status of the lamp.

The functional block (FB) light switching actuator basic is used in the application domain lighting to contol switching actuators. It specifies the functionality, for example contained in a relay, to turn the switching actuator on or off, see Figure A.6.



FB Light Switching Actuator Basic (LSAB)

Inputs	
Switch OnOff	(SOO)
Timed Start Stop	(TSS)
Forced	(FO)
Lock Device	(LD)
Scene Number	(SN)
Scene Control	(SC)

Outputs	
Info On Off	(IOO)

additional I/Os

None

Parameters	
On Delay	(OND)
Off Delay	(OFFD)
Timed On Duration	(TOD)
Prewarning Duration	(PWD)
Timed On Retrigger Function	(TRF)
Manual Off Enable	(MOE)
Invert Lock Device	(ILD)
Behaviour at Locking	(BL)
Behaviour at Unlocking	(BUL)
Lock State	(LS)
Unlock State	(ULS)
State for Scene Number	(SSN)
Storage Function for Scene	(SFSN)
Scene Learning Mode Enable	(SLME)
Transmission Cycle Time	(TCT)
Bus Power Up Message Delay	(PUMD)
Behaviour Bus Power Up	(BPU)
Bus Power Up State	(PUS)
Behaviour Bus Power Down	(BPD)
Bus Power Down State	(PDS)
Invert Output State	(IOS)

▨	mandatory
☐	optional

Figure A.6 – Functional block diagram for FB light switching actuator basic

In Figure A.6, Switch On Off (SOO) is mandatory and is used in this example to turn the lamp on or off. Info On Off (IOO) is optional and is used in this example to present the status of the lamp.

All other I/O's and parameters in the previous two figures are not used in this example and are not described here.

A.4.2 System A ↔ Interoperability domain (ID)

In this example the Info On Off (IOO) object of the Switching Sensor Basic (SSB) functional block in the Network A (KNX) domain corresponds to the input SwitchState in the Light Switch object of the interoperability domain (ID). The Switch On Off (SOO) object of the Switching Sensor Basic (SSB) functional block in the Network A (KNX) domain corresponds to the output SetSwitchState in the Light Switch object of the interoperability domain (ID).

In the example, the Info On Off (IOO) object of the Light Switching Actuator Basic (LSAB) functional block in the Network A (KNX) domain corresponds to the output SwitchState in the LightLamp object of the interoperability domain (ID). The Switch On Off (SOO) object of the Light Switching Actuator Basic (LSAB) functional block in the Network A (KNX) domain corresponds to the input SetSwitchState in the LightLamp object of the interoperability domain (ID).

The interworking function (IWF) translates control messages between System A and the interoperability domain. The IWF translates those system-specific messages to and from lighting application model event flows within the ID, which acts as the intermediary between different networks.

Each Interworking Function (IWF) holds its own information about devices, bindings and events it needs to know about and process to fulfil its purpose.

The specific implementation of the IWF is up to the system chosen for the interoperability domain and the manufacturer of the product executing the IWF (i.e. the gateway to the interoperability domain).

A.5 System B

A.5.1 LightLamp functional profile

This is a description of the System B LightLamp device. The System B LightLamp is described as a BinaryLight device that includes a SwitchPower service, see Figure A.7.

```
<?xml version="1.0" encoding="utf-8" ?>
 <root xmlns="urn:schemas-upnp-org:device-1-0">
    <specVersion>
        <major>1</major>
        <minor>0</minor>
    </specVersion>
    <device>
        <deviceType>urn:schemas-upnp-org:device:BinaryLight:1</deviceType>
        <friendlyName>Light (UPnP)</friendlyName>
        <manufacturer>Intel Corporation</manufacturer>
        <manufacturerURL>http://www.intel.com</manufacturerURL>
        <modelDescription>Software Emulated Light Bulb</modelDescription>
        <modelName>Intel CLR Emulated Light Bulb</modelName>
        <modelNumber>XPC-L1</modelNumber>
        <modelURL>http://www.intel.com/xpc</modelURL>
        <UDN>uuid:55f52e1b-62a7-4e23-841b-283bca65a3fb</UDN>
        <serviceList>
            <service>
                <serviceType>urn:schemas-upnp-org:service:SwitchPower:1</serviceType>
                <serviceId>urn:upnp-org:serviceId:SwitchPower.0001</serviceId>
                <SCPDURL>_SwitchPower.0001_scpd.xml</SCPDURL>
                <controlURL>_SwitchPower.0001_control</controlURL>
                <eventSubURL>_SwitchPower.0001_event</eventSubURL>
            </service>
        </serviceList>
     </device>
  </root>
```

Figure A.7 – System B UPnP LightLamp device

A.5.2 System B ↔ Interoperability domain (ID)

The ID has two middleware stacks involved in the interface to System B: the System B middleware and the ID middleware, both on top of TCP/IP in the System B example in Figure A.8. The System B middleware built on SSDP/GENA/SOAP manages a generic control point. Through a generic control point, System B middleware can control a System B device. The ID middleware communicates with the IWF using lighting application model event passing on the event bus.

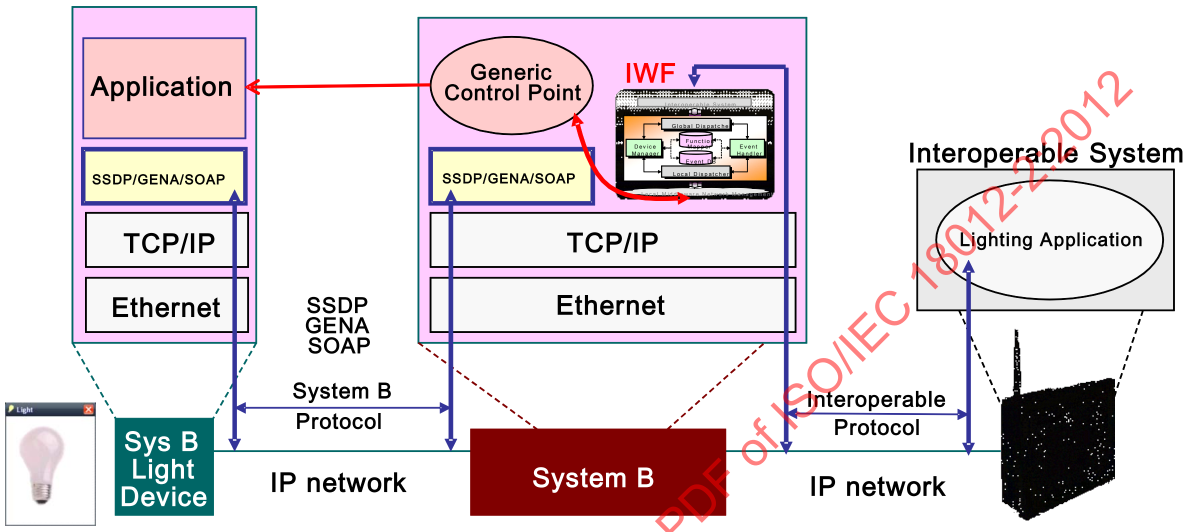
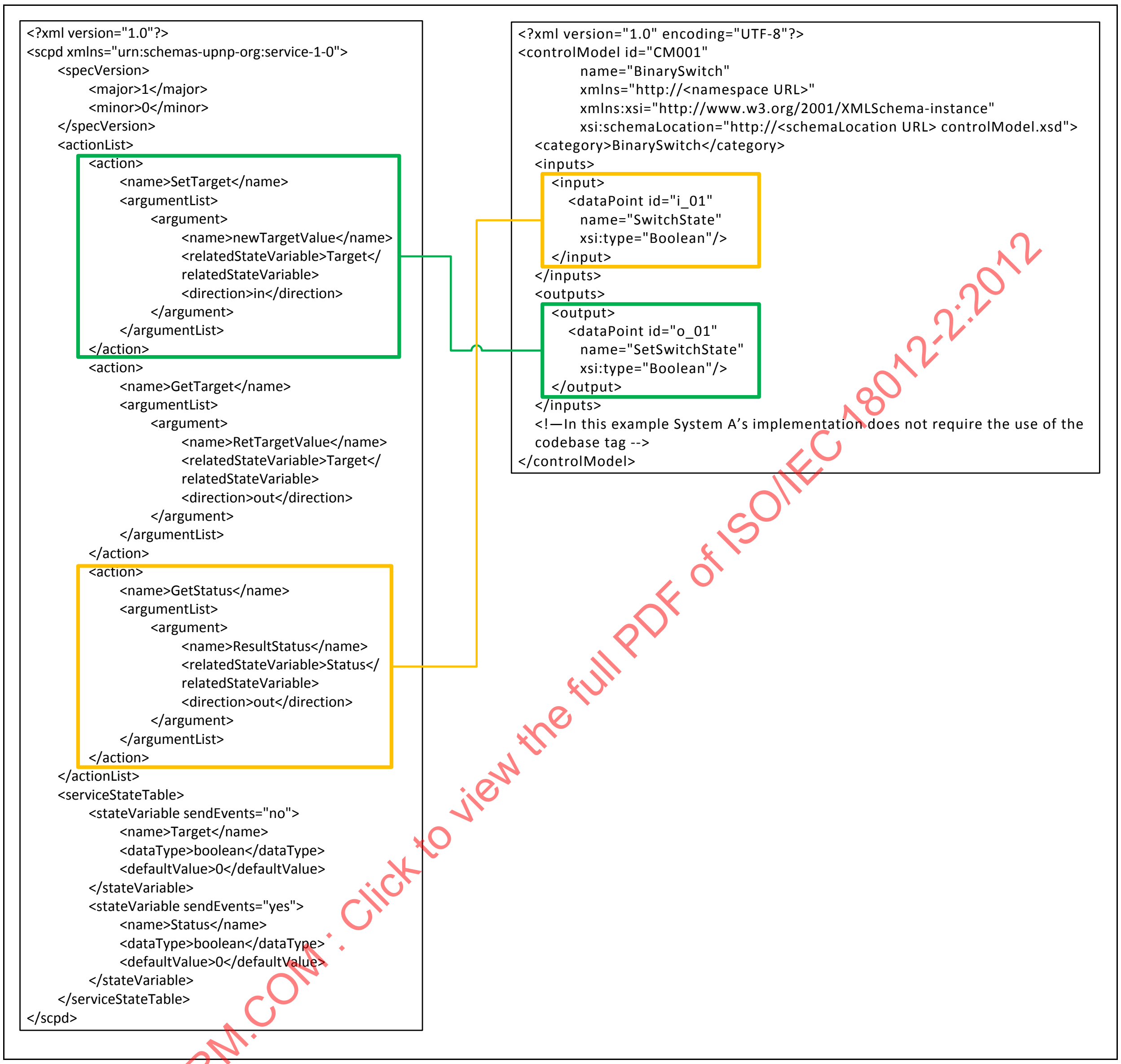


Figure A.8 – UPnP InterWorking function system

Figure A.9 describes how to map a System B device description to the generic device object. A System B light device has one service, SwitchPower, which is mapped to both SetSwitchState and SwitchState of the generic LightLamp object.

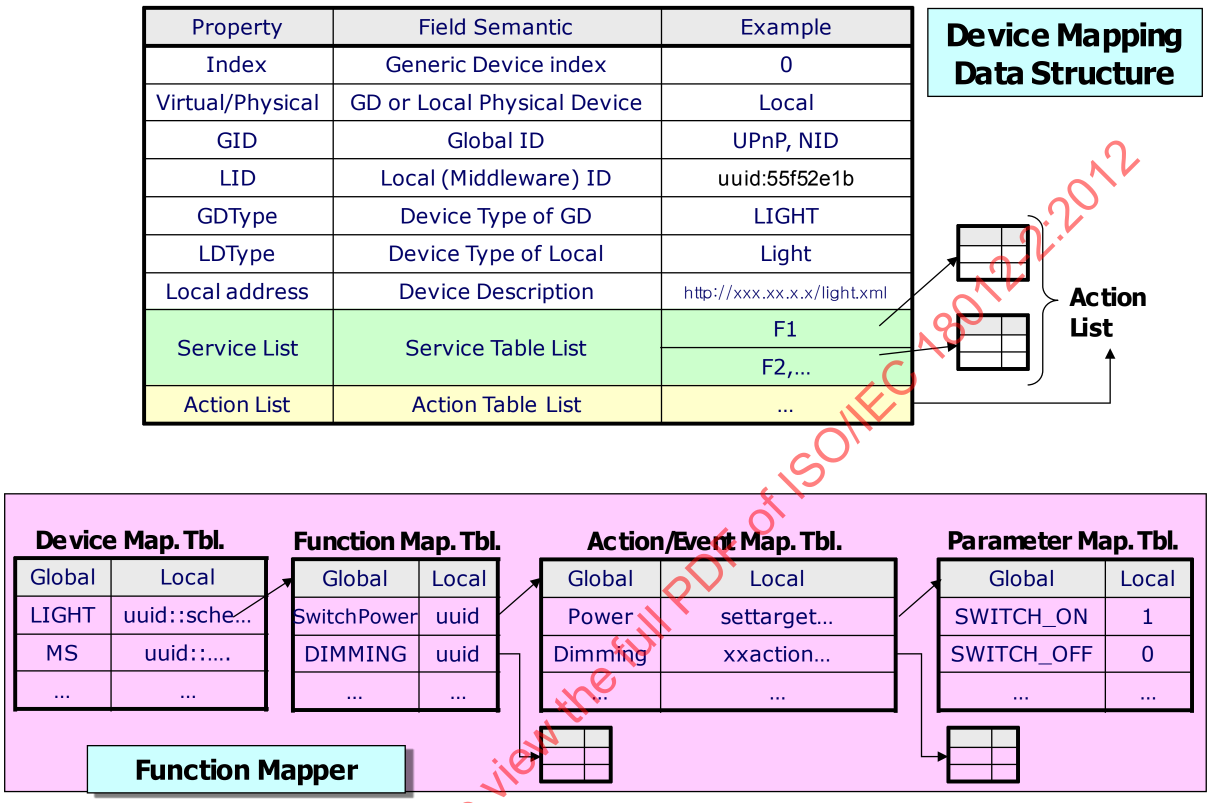


```
<?xml version="1.0"?>
<scpd xmlns="urn:schemas-upnp-org:service-1-0">
    <specVersion>
        <major>1</major>
        <minor>0</minor>
    </specVersion>
    <actionList>
        <action>
            <name>SetTarget</name>
            <argumentList>
                <argument>
                    <name>newTargetValue</name>
                    <relatedStateVariable>Target</
                    relatedStateVariable>
                    <direction>in</direction>
                </argument>
            </argumentList>
        </action>
        <action>
            <name>GetTarget</name>
            <argumentList>
                <argument>
                    <name>RetTargetValue</name>
                    <relatedStateVariable>Target</
                    relatedStateVariable>
                    <direction>out</direction>
                </argument>
            </argumentList>
        </action>
        <action>
            <name>GetStatus</name>
            <argumentList>
                <argument>
                    <name>ResultStatus</name>
                    <relatedStateVariable>Status</
                    relatedStateVariable>
                    <direction>out</direction>
                </argument>
            </argumentList>
        </action>
    </actionList>
    <serviceStateTable>
        <stateVariable sendEvents="no">
            <name>Target</name>
            <dataType>boolean</dataType>
            <defaultValue>0</defaultValue>
        </stateVariable>
        <stateVariable sendEvents="yes">
            <name>Status</name>
            <dataType>boolean</dataType>
            <defaultValue>0</defaultValue>
        </stateVariable>
    </serviceStateTable>
</scpd>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<controlModel id="CM001"
        name="BinarySwitch"
        xmlns="http://<namespace URL>"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://<schemaLocation URL> controlModel.xsd">
   <category>BinarySwitch</category>
   <inputs>
      <input>
         <dataPoint id="i_01"
           name="SwitchState"
           xsi:type="Boolean"/>
      </input>
   </inputs>
   <outputs>
      <output>
         <dataPoint id="o_01"
           name="SetSwitchState"
           xsi:type="Boolean"/>
      </output>
   </inputs>
   <!—In this example System A's implementation does not require the use of the
   codebase tag -->
</controlModel>
```

Figure A.9 – Functional mapping of SwitchPower service to the LightLamp

The System B function mapping table, shown in Figure A.10, consists of a Device Mapping Table, Function Mapping Table, Action/Event Mapping Table, and a Parameter Mapping Table. Each Table has two columns: a global value for generic object and a local value for System B device information. Whenever System B devices are detected by SSDP, an IWF searches this function mapping table and recomposes the corresponding generic application object in the ID.



Property	Field Semantic	Example
Index	Generic Device index	0
Virtual/Physical	GD or Local Physical Device	Local
GID	Global ID	UPnP, NID
LID	Local (Middleware) ID	uuid:55f52e1b
GDType	Device Type of GD	LIGHT
LDType	Device Type of Local	Light
Local address	Device Description	http://xxx.xx.x.x/light.xml
Service List	Service Table List	F1
		F2,...
Action List	Action Table List	...

Device Map. Tbl.

Global	Local
LIGHT	uuid::sche...
MS	uuid::....
...	...

Function Map. Tbl.

Global	Local
SwitchPower	uuid
DIMMING	uuid
...	...

Action/Event Map. Tbl.

Global	Local
Power	settarget...
Dimming	xxaction...
...	...

Parameter Map. Tbl.

Global	Local
SWITCH_ON	1
SWITCH_OFF	0
...	...

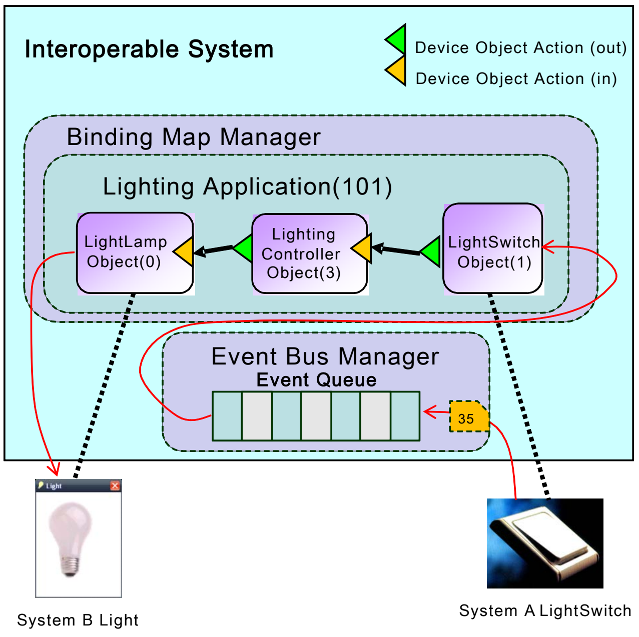
Figure A.10 – System B function mapping table

A.6 Interoperability domain (ID) – Functional model

In this example implementation, the ID has three tables, as shown in Figure A.11: Device Table, BindMap Table, and EventBus Table. Device Table contains the information about generic device objects. These objects are registered by each IWF system through interoperability protocols.

The BindMap Table is related to the Binding Maps of interoperable applications that are active in the ID. For each interoperable application its Binding Map describes the binding of two or more generic objects that will interact through the ID. Binding means connection from an output set of events in an operational object to an input set of events in one or more other operational objects.

The EventBus Table is used for managing event processing. Whenever a device changes its status, an IWF receives that change and causes an event to be generated to the EventBus Table of the ID. Note that in order to serialise processing of multiple events, the ID uses a FIFO-style EventQueue.



Device Table

GD#	Object ID	Generic Device Type
0	UPnP::LightLamp0	LightLamp
1	LonWorks::LightSwitch0	LightSwitch

Binding Map Table

Appl #	Controller Obj#	Generic Device Obj#
101	3	0, 1
...		

Event Bus Table

Obj #	out action	Obj#	in action
1	LightSwitch GetSwitchValue1	3	LightingController. SetSwitchValue1
3	LightingController. GetSwitchValue1	0	LightLamp SetSwitchValue1
...			

Event Queue

Event #	Source Obj#	Destination Obj#	Type	Value
35	1	0	LightSwitch.GetSwitchValue. SwtichValue,	On
...				

Figure A.11 – Interoperability domain (ID) work flow

Annex B
(normative)

Base object schema definitions

B.1 Commons

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" targetNamespace="http://<namespace URL>"
   xmlns="http://<namespace URL>" elementFormDefault="qualified">
  <xsd:include schemaLocation="baseTypes.xsd"/>
  <xsd:include schemaLocation="derivedTypes.xsd"/>
  <xsd:include schemaLocation="baseDefinition.xsd"/>

  <!-- Define 'Inputs' element type used by other schema such ControlModel -->

  <xsd:element name="QoIRequirement">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="frequency" minOccurs="0">
          <xsd:complexType>
            <xsd:complexContent>
              <xsd:extension base="Frequency">
                <xsd:attribute name="selector" type="LogicExpression"/>
              </xsd:extension>
            </xsd:complexContent>
          </xsd:complexType>
        </xsd:element>
        <xsd:element name="maxUncertainty" type="xsd:decimal" minOccurs="0"/>
        <xsd:element name="dataPrecision" type="xsd:decimal" minOccurs="0"/>
        <xsd:element name="dataAccuracy" type="xsd:decimal" minOccurs="0"/>
        <xsd:element name="low" type="xsd:decimal" minOccurs="0"/>
        <xsd:element name="high" type="xsd:decimal" minOccurs="0"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

  <xsd:element name="QoI">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="frequency" minOccurs="0">
          <xsd:complexType>
            <xsd:complexContent>
              <xsd:extension base="Frequency">
                <xsd:attribute name="selector" type="LogicExpression"/>
              </xsd:extension>
            </xsd:complexContent>
          </xsd:complexType>
        </xsd:element>
        <xsd:element name="maxUncertainty" type="xsd:decimal" minOccurs="0"/>
        <xsd:element name="dataPrecision" type="xsd:decimal" minOccurs="0"/>
        <xsd:element name="dataAccuracy" type="xsd:decimal" minOccurs="0"/>
        <xsd:element name="low" type="xsd:decimal" minOccurs="0"/>
        <xsd:element name="high" type="xsd:decimal" minOccurs="0"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

  <xsd:element name="input">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="dataPoint"/>
        <xsd:element ref="QoIRequirement" minOccurs="0"/>
      </xsd:sequence>
    </xsd:complexType>
    </xsd:element>

    <xsd:element name="output">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element ref="dataPoint"/>
                <xsd:element ref="QoI" minOccurs="0"/>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>

    <xsd:complexType name="inputs">
        <xsd:sequence maxOccurs="unbounded">
            <xsd:element ref="input"/>
        </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="outputs">
        <xsd:sequence maxOccurs="unbounded">
            <xsd:element ref="output"/>
        </xsd:sequence>
    </xsd:complexType>

    <xsd:element name="property">
        <xsd:complexType>
            <xsd:attribute name="name" type="xsd:string"/>
            <xsd:attribute name="value" type="xsd:string"/>
        </xsd:complexType>
    </xsd:element>

    <xsd:complexType name="properties">
        <xsd:sequence>
            <xsd:element ref="property" minOccurs="0" maxOccurs="unbounded"/>
        </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="codeBase">
        <xsd:sequence>
            <xsd:element name="description" type="xsd:string" minOccurs="0"/>
            <xsd:element name="codeType" type="codebaseTypes" default="JAVA"/>
            <xsd:element name="codeLocation" type="xsd:string" minOccurs="1"/>
            <xsd:element name="codeProperties" type="properties" minOccurs="0"/>
            <xsd:element name="idlLocation" type="xsd:string" minOccurs="0"/>
        </xsd:sequence>
    </xsd:complexType>

</xsd:schema>
```

B.2 Base types

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema elementFormDefault="qualified"
                xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                xmlns="http://<namespace URL>"
                targetNamespace="http://<namespace URL>">

<xsd:include schemaLocation="baseDefinition.xsd"/>

        <!-- 7 SI Base Quantities -->

        <xsd:complexType name="Length">
                <xsd:complexContent>
                        <xsd:extension base="AnalogPoint">
                                <xsd:attribute name="unit" fixed="meter"/>
                        </xsd:extension>
                </xsd:complexContent>
        </xsd:complexType>
        <xsd:complexType name="Time">
                <xsd:complexContent>
                        <xsd:extension base="AnalogPoint">
                                <xsd:attribute name="unit" fixed="second"/>
                        </xsd:extension>
                </xsd:complexContent>
        </xsd:complexType>

        <xsd:complexType name="Mass">
                <xsd:complexContent>
                        <xsd:extension base="AnalogPoint">
                                <xsd:attribute name="unit" fixed="kg"/>
                        </xsd:extension>
                </xsd:complexContent>
        </xsd:complexType>

        <xsd:complexType name="Temperature">
                <xsd:complexContent>
                        <xsd:extension base="AnalogPoint">
                                <xsd:attribute name="unit" fixed="kelvin" use="required"/>
                        </xsd:extension>
                </xsd:complexContent>
        </xsd:complexType>

        <xsd:complexType name="ElectricCurrent">
                <xsd:complexContent>
                        <xsd:extension base="AnalogPoint">
                                <xsd:attribute name="unit" fixed="ampere"/>
                        </xsd:extension>
                </xsd:complexContent>
        </xsd:complexType>

        <xsd:complexType name="SubstanceAmount">
                <xsd:complexContent>
                        <xsd:extension base="AnalogPoint">
                                <xsd:attribute name="unit" fixed="mole"/>
                        </xsd:extension>
                </xsd:complexContent>
        </xsd:complexType>

        <xsd:complexType name="LuminousIntensity">
                <xsd:complexContent>
                        <xsd:extension base="AnalogPoint">
                                <xsd:attribute name="unit" fixed="candela"/>
                        </xsd:extension>
                </xsd:complexContent>
        </xsd:complexType>

</xsd:schema>
```

B.3 Base definitions

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema elementFormDefault="qualified"
                        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                        xmlns="http://<namespace URL>"
                        targetNamespace="http://<namespace URL>">

    <xsd:simpleType name="LogicExpression">
        <xsd:restriction base="xsd:string">
            <xsd:enumeration value="EQT"/>
            <xsd:enumeration value="GTT"/>
            <xsd:enumeration value="GTE"/>
            <xsd:enumeration value="LTT"/>
            <xsd:enumeration value="LTE"/>
        </xsd:restriction>
    </xsd:simpleType>
```

```
<xsd:complexType name="dataType">
   <xsd:simpleContent>
      <xsd:extension base="xsd:anySimpleType">
         <xsd:attribute name="dataType" type="defined_data_types"/>
      </xsd:extension>
   </xsd:simpleContent>
</xsd:complexType>

<xsd:simpleType name="codebaseTypes">
   <xsd:restriction base="xsd:string">
      <xsd:enumeration value="C++"/>
      <xsd:enumeration value="JAVA"/>
      <xsd:enumeration value="XSL"/>
   </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="SIUnit">
   <xsd:restriction base="xsd:string">
      <xsd:pattern value="((\([0-3]\))(\((\-)?\d{1}(/\d{1})?\)){9})|((\([4-9]\))(\(0\)){9})"/>
      <!--
               field #1 : enumeration
                              0= unit is described by field #2-#10
                              1= unit is U/U where U is described by filed #2-10
                              2= unit is log10(U) where U is described by filed #2-10
                              3= unit is log10(U/U) where U is described by filed #2-10
                              4= unit is digital data, field #2-10 must be zero
                              5= arbitrary scale, field #2-10 must be zero
                              6-9= reserved

               field #2 : radian       - angle
               field #3 : steradian - solid angle
               field #4 : meter        - length
               field #5 : kilogram     - mass
               field #6 : second       - time
               field #7 : ampere       - electric current
               field #8 : kelvin       - temperature
               field #9 : mole        - amount of substance
               field #10: candela      - luminous intensity
            -->
   </xsd:restriction>
</xsd:simpleType>
<xsd:element name="SIUnit" type="SIUnit"/>

<xsd:simpleType name="defined_data_types">
   <xsd:restriction base="xsd:string">
      <xsd:enumeration value="FLOAT"/>
      <xsd:enumeration value="INT"/>
      <xsd:enumeration value="BOOLEAN"/>
      <xsd:enumeration value="STRING"/>
   </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="SIMultiple">
   <xsd:restriction base="xsd:token">
      <xsd:enumeration value="exa"/>
      <xsd:enumeration value="peta"/>
      <xsd:enumeration value="tera"/>
      <xsd:enumeration value="giga"/>
      <xsd:enumeration value="mega"/>
      <xsd:enumeration value="kilo"/>
      <xsd:enumeration value="hecto"/>
      <xsd:enumeration value="deka"/>
      <xsd:enumeration value="deci"/>
      <xsd:enumeration value="centi"/>
      <xsd:enumeration value="milli"/>
      <xsd:enumeration value="micro"/>
      <xsd:enumeration value="nano"/>
```

```
        <xsd:enumeration value="pico"/>
        <xsd:enumeration value="femto"/>
        <xsd:enumeration value="atto"/>
    </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="Multiple">
    <xsd:restriction base="xsd:float"/>
</xsd:simpleType>

<xsd:complexType name="DataValue">
    <xsd:simpleContent>
        <xsd:extension base="xsd:anySimpleType"/>
    </xsd:simpleContent>
</xsd:complexType>
<xsd:element name="dataValue" type="DataValue"/>

<xsd:complexType name="DataVector">
    <xsd:sequence>
        <xsd:element name="value" minOccurs="2" maxOccurs="unbounded">
            <xsd:complexType>
                <xsd:simpleContent>
                    <xsd:extension base="xsd:anySimpleType">
                        <xsd:attribute name="id" type="xsd:long"/>
                    </xsd:extension>
                </xsd:simpleContent>
            </xsd:complexType>
        </xsd:element>
    </xsd:sequence>
</xsd:complexType>

<xsd:element name="dataVector" type="DataVector">
    <xsd:unique name="onDataValuePerID">
        <xsd:selector xpath="dataVector/value"/>
        <xsd:field xpath="@id"/>
    </xsd:unique>
</xsd:element>

<xsd:complexType name="DataUncertainty">
    <xsd:simpleContent>
        <xsd:extension base="xsd:float">
            <xsd:attribute name="interpretation" use="optional">
                <xsd:simpleType>
                    <xsd:restriction base="xsd:token">
                        <xsd:enumeration value="ISO_GUIDE"/>
                        <xsd:enumeration value="CUSTOMER"/>
                    </xsd:restriction>
                </xsd:simpleType>
            </xsd:attribute>
            <xsd:attribute name="coverageFactor" type="xsd:float" use="optional"/>
            <xsd:attribute name="customerFactor" type="xsd:float" use="optional"/>
        </xsd:extension>
    </xsd:simpleContent>
</xsd:complexType>

<xsd:complexType name="DataPoint" abstract="true">
    <xsd:simpleContent>
        <xsd:extension base="xsd:string">
            <xsd:attribute name="name" type="xsd:Name" use="optional"/>
            <xsd:attribute name="id" type="xsd:ID" use="optional"/>
            <xsd:attribute name="isVector" type="xsd:boolean" use="optional" default="false"/>
        </xsd:extension>
    </xsd:simpleContent>
</xsd:complexType>
<xsd:element name="dataPoint" type="DataPoint"/>

<xsd:complexType name="PhysicalDataPoint">
    <xsd:simpleContent>
        <xsd:extension base="DataPoint">
```

```
            <xsd:attribute name="siunit" type="SIUnit" use="required"/>
        </xsd:extension>
    </xsd:simpleContent>
</xsd:complexType>

    <!-- Define built-in datatype array,
            currently, they are not used anywhere -->

    <xsd:simpleType name="DecimalArray">
            <xsd:list itemType="xsd:decimal"/>
    </xsd:simpleType>

    <xsd:simpleType name="IntegerArray">
            <xsd:list itemType="xsd:integer"/>
    </xsd:simpleType>

    <xsd:simpleType name="ShortArray">
            <xsd:list itemType="xsd:short"/>
    </xsd:simpleType>

    <xsd:simpleType name="ByteArray">
            <xsd:list itemType="xsd:byte"/>
    </xsd:simpleType>

    <xsd:simpleType name="LongArray">
            <xsd:list itemType="xsd:long"/>
    </xsd:simpleType>

    <xsd:simpleType name="FloatArray">
            <xsd:list itemType="xsd:float"/>
    </xsd:simpleType>

    <xsd:simpleType name="DoubleArray">
            <xsd:list itemType="xsd:double"/>
    </xsd:simpleType>

<xsd:complexType name="LogicDataPoint" abstract="true">
    <xsd:complexContent>
        <xsd:extension base="DataPoint"/>
    </xsd:complexContent>
</xsd:complexType>

<xsd:element name="logicDataPoint" type="LogicDataPoint"/>
    <xsd:complexType name="String">
            <xsd:complexContent>
                    <xsd:extension base="LogicDataPoint">
                            <xsd:sequence>
                                    <xsd:element name="value" type="xsd:string" minOccurs="0"
maxOccurs="unbounded"/>
                            </xsd:sequence>
                    </xsd:extension>
            </xsd:complexContent>
    </xsd:complexType>

    <xsd:complexType name="Boolean">
            <xsd:complexContent>
                    <xsd:extension base="LogicDataPoint">
                            <xsd:sequence>
                                    <xsd:element name="value" type="xsd:boolean" minOccurs="0"
maxOccurs="unbounded"/>
                            </xsd:sequence>
                    </xsd:extension>
            </xsd:complexContent>
    </xsd:complexType>

    <xsd:complexType name="Decimal">
            <xsd:complexContent>
                    <xsd:extension base="LogicDataPoint">
				<xsd:sequence>
					<xsd:element name="value" type="xsd:decimal" minOccurs="0"
maxOccurs="unbounded"/>
				</xsd:sequence>
			</xsd:extension>
		</xsd:complexContent>
	</xsd:complexType>

	<xsd:complexType name="Integer">
		<xsd:complexContent>
			<xsd:extension base="LogicDataPoint">
				<xsd:sequence>
					<xsd:element name="value" type="xsd:integer" minOccurs="0"
maxOccurs="unbounded"/>
				</xsd:sequence>
			</xsd:extension>
		</xsd:complexContent>
	</xsd:complexType>


	<xsd:complexType name="Int">
		<xsd:complexContent>
			<xsd:extension base="LogicDataPoint">
				<xsd:sequence>
					<xsd:element name="value" type="xsd:int" minOccurs="0"
maxOccurs="unbounded"/>
				</xsd:sequence>
			</xsd:extension>
		</xsd:complexContent>
	</xsd:complexType>

	<xsd:complexType name="Long">
		<xsd:complexContent>
			<xsd:extension base="LogicDataPoint">
				<xsd:sequence>
					<xsd:element name="value" type="xsd:long" minOccurs="0"
maxOccurs="unbounded"/>
				</xsd:sequence>
			</xsd:extension>
		</xsd:complexContent>
	</xsd:complexType>

	<xsd:complexType name="Short">
		<xsd:complexContent>
			<xsd:extension base="LogicDataPoint">
				<xsd:sequence>
					<xsd:element name="value" type="xsd:short" minOccurs="0"
maxOccurs="unbounded"/>
				</xsd:sequence>
			</xsd:extension>
		</xsd:complexContent>
	</xsd:complexType>

	<xsd:complexType name="Byte">
		<xsd:complexContent>
			<xsd:extension base="LogicDataPoint">
				<xsd:sequence>
					<xsd:element name="value" type="xsd:byte" minOccurs="0"
maxOccurs="unbounded"/>
				</xsd:sequence>
			</xsd:extension>
		</xsd:complexContent>
	</xsd:complexType>

	<xsd:complexType name="Float">
		<xsd:complexContent>
			<xsd:extension base="LogicDataPoint">
				<xsd:sequence>
```

```
                                    <xsd:element name="value" type="xsd:float" minOccurs="0"
maxOccurs="unbounded"/>
                            </xsd:sequence>
                    </xsd:extension>
            </xsd:complexContent>
    </xsd:complexType>

    <xsd:complexType name="Double">
            <xsd:complexContent>
                    <xsd:extension base="LogicDataPoint">
                            <xsd:sequence>
                                    <xsd:element name="value" type="xsd:double" minOccurs="0"
maxOccurs="unbounded"/>
                            </xsd:sequence>
                    </xsd:extension>
            </xsd:complexContent>
    </xsd:complexType>

    <xsd:complexType name="AnyURI">
            <xsd:complexContent>
                    <xsd:extension base="LogicDataPoint">
                            <xsd:sequence>
                                    <xsd:element name="value" type="xsd:anyURI" minOccurs="0"
maxOccurs="unbounded"/>
                            </xsd:sequence>
                    </xsd:extension>
            </xsd:complexContent>
    </xsd:complexType>

<xsd:complexType name="DigitalPoint">
    <xsd:complexContent>
        <xsd:extension base="PhysicalDataPoint">
            <xsd:sequence>
                <xsd:element name="probability" minOccurs="0">
                    <xsd:simpleType>
                        <xsd:restriction base="xsd:decimal">
                            <xsd:minInclusive value="0"/>
                            <xsd:maxInclusive value="1"/>
                        </xsd:restriction>
                    </xsd:simpleType>
                </xsd:element>
                <xsd:element name="value" type="xsd:boolean" minOccurs="0"
maxOccurs="unbounded"/>
            </xsd:sequence>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
<xsd:element name="digitalPoint" type="DigitalPoint"/>

<xsd:complexType name="AnalogPoint">
    <xsd:complexContent>
        <xsd:extension base="PhysicalDataPoint">
            <xsd:sequence>
                <xsd:element name="uncertainty" type="xsd:decimal" minOccurs="0"/>
                <xsd:element name="value" type="xsd:float" minOccurs="0" maxOccurs="unbounded"/>
            </xsd:sequence>
            <xsd:attribute name="unitMultiple" use="optional">
                <xsd:simpleType>
                    <xsd:union>
                        <xsd:simpleType>
                            <xsd:restriction base="SIMultiple"/>
                        </xsd:simpleType>
                        <xsd:simpleType>
                            <xsd:restriction base="Multiple"/>
                        </xsd:simpleType>
                    </xsd:union>
                </xsd:simpleType>
            </xsd:attribute>
```