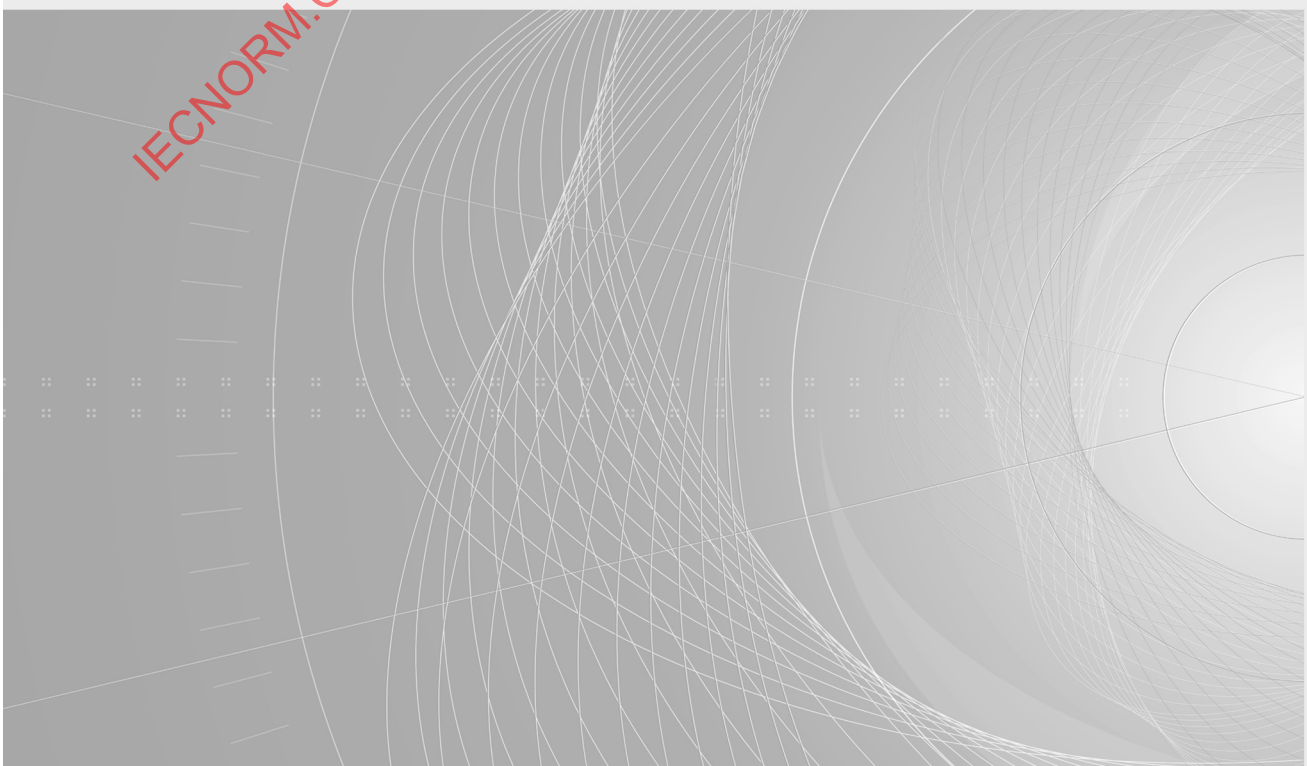
IEC TR 62453-42

Edition 1.0 2016-04

TECHNICAL REPORT

colour inside

Field device tool (FDT) interface specification – Part 42: Object model integration profile – Common Language Infrastructure



IEC TR 62453-42:2016-04(en)

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

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

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

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

IEC Catalogue - webstore.iec.ch/catalogue
The stand-alone application for consulting the entire bibliographical information on IEC International Standards, Technical Specifications, Technical Reports and other documents. Available for PC, Mac OS, Android Tablets and iPad.

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 It also gives information on projects, replaced and withdrawn publications.

IEC Just Published - webstore.iec.ch/justpublished
Stay up to date on all new IEC publications. Just Published details all new publications released. Available online and also once a month by email.

Electropedia - www.electropedia.org
The world's leading online dictionary of electronic and electrical terms containing 20 000 terms and definitions in English and French, with equivalent terms in 15 additional languages. Also known as the International Electrotechnical Vocabulary (IEV) online.

IEC Glossary - std.iec.ch/glossary
65 000 electrotechnical terminology entries in English and French extracted from the Terms and Definitions clause of IEC publications issued since 2002. Some entries have been collected from earlier publications of IEC TC 37, 77, 86 and CISPR.

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

IEC TR 62453-42

Edition 1.0 2016-04

TECHNICAL REPORT

colour inside

Field device tool (FDT) interface specification – Part 42: Object model integration profile – Common Language Infrastructure

INTERNATIONAL ELECTROTECHNICAL COMMISSION

ICS 25.040.40; 35.100.05; 35.110

ISBN 978-2-8322-3226-2

Warning! Make sure that you obtained this publication from an authorized distributor.

CONTENTS

INTERNATIONAL ELECTROTECHNICAL COMMISSION

_____

FIELD DEVICE TOOL (FDT) INTERFACE SPECIFICATION –

Part 42: Object model integration profile – Common Language Infrastructure

FOREWORD

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

The main task of IEC technical committees is to prepare International Standards. However, a technical committee may propose the publication of a technical report when it has collected data of a different kind from that which is normally published as an International Standard, for example "state of the art".

IEC TR 62453-42, which is a technical report, has been prepared by subcommittee 65E: Devices and integration in enterprise systems, of IEC technical committee 65: Industrial-process measurement, control and automation:

The text of this technical report is based on the following documents:

Enquiry draft	Report on voting
65E/439/DTR	65E/486/RVC

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

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

A list of all parts of the IEC 62453 series, under the general title *Field Device Tool (FDT) interface specification*, can be found on the IEC website.

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

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

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

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

This Part of IEC 62543, which is a technical report, is an interface specification for developers of FDT (Field Device Tool) components for function control and data access within a client/server architecture. The specification is a result of an analysis and design process to develop standard interfaces to facilitate the development of servers and clients by multiple vendors that need to interoperate seamlessly.

With the integration of fieldbuses into control systems, there are a few other tasks which need to be performed. In addition to fieldbus- and device-specific tools, there is a need to integrate these tools into higher-level system-wide planning or engineering tools. In particular, for use in extensive and heterogeneous control systems, the unambiguous definition of engineering interfaces that are easy to use for all those involved is of great importance.

A device-specific software component, called DTM (Device Type Manager), is supplied by the field device manufacturer with its device. The DTM is integrated into engineering tools via the FDT interfaces defined in this specification. The approach to integration, in general, is open for all kind of fieldbusses and thus meets the requirements for integrating different kinds of devices into heterogeneous control systems.

Figure 1 shows how IEC TR 62453-42 is related to the IEC 62453 series.

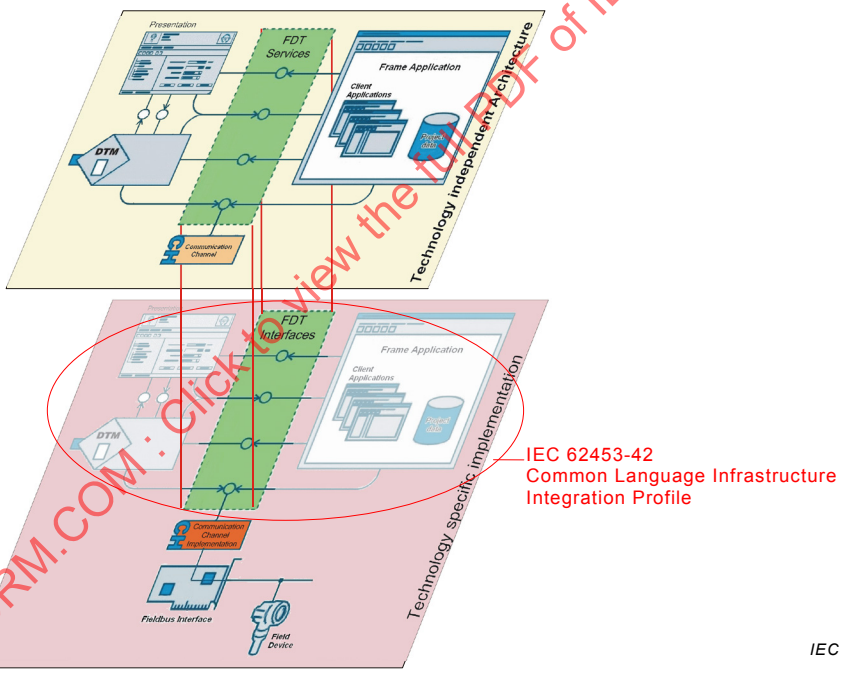


Figure 1 – Relation of IEC 62453-42 to the IEC 62453 series

The document structure is:

- Clause 3 explains the used terms, definitions and conventions
- Clause 4 introduces the general concepts of IEC 62453-42
- Clause 5 describes the technical concepts used to implement IEC 62453-42 and how FDT concepts are mapped to .NET Framework
- Clause 6 provides an overview of the FDT Objects, their interfaces and behavior
- Clause 7 presents an overview of the IEC 62453-42 datatypes
- Clause 8 shows the interaction of FDT Objects at runtime
- Clause 9 explains rules related to installation and deployment of DTMs

- Clause 10 explains how FDT life cycle concepts are implemented
- Clause 11 shows examples for Frame Application architectures

FIELD DEVICE TOOL (FDT) INTERFACE SPECIFICATION –

Part 42: Object model integration profile – Common Language Infrastructure

1 Scope

This part of IEC 62453, which is a technical report, defines how the common FDT principles are implemented based on the .NET technology, including the object behavior and object interaction via .NET interfaces.

This document specifies FDT version 2.0.

2 Normative references

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

IEC 62453-1:—[1], *Field Device Tool (FDT) interface specification – Part 1: Overview and guidance*

IEC 62453-2:—[1], *Field Device Tool (FDT) interface specification – Part 2: Concepts and detailed description*

3 Terms, definitions, abbreviations and conventions

3.1 Terms and definitions

For the purposes of this document, the terms and definitions given in IEC 62453-1, IEC 62453-2 as well as the following apply.

3.1.1
action
execution of a function which may involve several calls to interface methods of different FDT Objects

3.1.2
asynchronous methods
methods that trigger execution of asynchronous operations

Note 1 to entry: See also 5.6.7.

3.1.3
asynchronous operation
operation that is performed while the FDT object (client) that has requested the operation does not wait for the result, but the client is notified when the operation is finished

[1] To be published concurrently with this technical report.

3.1.4
bulk data
device node-specific persisted data, which is stored besides DTM instance data

Note 1 to entry: Example for bulk data: accumulated historical data, used for trend analysis.

3.1.5
bulk operation
operation to perform one or more tasks at a group of devices nodes

Note 1 to entry: Examples for bulk operation: up- or download for a group of devices, parameter adjustment for a group of devices or report generation for a group of devices.

3.1.6
clone DTM instance
process of creating a new device node in the FDT topology based on an existing device node

Note 1 to entry: This includes copying DTM instance(see 3.1.11) and resetting device node-specific DTM data.

Note 2 to entry: The identification attributes of the device are changed.

3.1.7
Communication Channel
component representing access to a fieldbus segment or to other means of communication

3.1.8
compatibility
feature of a component (hardware or software) that enables it to be interoperable with another component

3.1.9
backward compatibility
feature of a component (hardware or software) that enables it to replace an other version of the component

3.1.10
compatibility attributes
attributes used to find compatible components, to replace components or to validate compatibility after a component replacement

Note 1 to entry: Compatibility attributes are required to check whether a component is compatible with another component.

Note 2 to entry: Compatibility attributes are used to define compatibility in regard to 3.1.8.2, compatibility.

3.1.11
copy DTM instance
process of creating a new device node in the FDT topology based on an existing device node

Note 1 to entry: This includes loading the original DTM dataset to initiate the new DTM instance.

Note 2 to entry: The identification attributes of the device are not changed.

3.1.12
copy device node in FDT Topology
(see 3.1.11 copy DTM instance)

3.1.13
DataTransaction
transaction regarding the data of a DTM (persistent or device data)

3.1.14
delegate
reference type that can be used to encapsulate a method

3.1.15
device configuration
process of changing data related to device-specific characteristics/basic behavior

Note 1 to entry: Example for such characteristics may be the structure of a remote I/O or the type of measurement procedure like absolute pressure or differential pressure.

3.1.16
device data
configuration data that resides on the device

3.1.17
device parameterization
process of changing device-specific data in order to adjust application-specific behavior

3.1.18
device node
node in the topology, which represents a device

Note 1 to entry: A DtmDeviceType is assigned to a device node, which is instantiated to operate the device in online or offline modes. See Figure 14.

3.1.19
device type check
process of checking the device type when a DTM is going online with a connected physical device

Note 1 to entry: The DTM shall reject to go online if the connected physical device type is not supported. The check shall be based on same information as in DeviceIdentInfo. See 6.3.2.3.

3.1.20
DD-Interpreter DTMs
DTMs which interpret device descriptions at runtime

3.1.21
DTM
software component containing device-specific application software, including DTM Business Logic, DTM User Interface and related objects (e.g. Communication Channel)

Note 1 to entry: Older FDT specification documents used the term “DTM” for the object DTM Business Logic as well as for the whole component consisting of DTM BL, DTM UI and channels.

3.1.22
DTM Business Logic
DTM BL
part of the DTM, which contains all the functionality to access storage and communication and which manages the instance data of a DTM

3.1.23
DTM Identifier
identifier, which is used to identify a DTM (DTM BL class)

Note 1 to entry: In order to uniquely identify a DTM BL class, the property DtmInfo.Id is used.

3.1.24
DTM instance modal
prevents user interaction with other windows of the DTM instance (see modal window)

3.1.25
DTM User Interface (DTM UI)
part of the DTM, which is displayed to a human user

3.1.26
DtmType
either DtmDeviceType, DtmModuleType or DtmBlockType

Note 1 to entry: All DtmTypes provide identification in DtmInfo class.

Note 2 to entry: DtmType is described by TypeInfo.

3.1.27
DtmDeviceType
element of a DTM software supporting one or more device types

Note 1 to entry: DtmDeviceType is described by DeviceTypeInfo class.

3.1.28
DtmModuleType
element of a DTM software supporting one or more device module types

Note 1 to entry: DtmModuleType is described by class ModuleTypeInfo.

3.1.29
DtmBlockType
element of a DTM software supporting one or more block types

Note 1 to entry: DtmBlockType is described by class BlockTypeInfo.

3.1.30
FDT Object
object defined by FDT (e.g. Frame Application, DTM Business Logic, DTM User Interface, Communication Channel)

3.1.31
FDT Protocol Annex
document defining support for a communication protocol for FDT

Note 1 to entry: Examples for such documents are “PROFIBUS protocol annex” and “HART protocol annex”. Within IEC 62453 the standard parts with numbers 3xy define support for communication protols.

3.1.32
FDT Application Profile Annex
document defining support for a type of application for FDT

Note 1 to entry: An example for such a document is the “PLC Tool interface” (defined for FDT1.2). Other such documents may be defined at a later time also for FDT2.

3.1.33
fieldbus message
data in a protocol-specific telegram

3.1.34
Frame Application modal
prevents user interaction with windows of the Frame Application (see modal window)

3.1.35
hardware platform
hardware on which FDT software is executed

Note 1 to entry: Different hardware platforms may be supported that are based on different architectures and display formats, for example PC and others.

3.1.36
identification attributes
attributes which describe the identity of a component. These attributes are typically displayed to users or used to validate and ensure compatibility of components

3.1.37
incompatibility
situation where a component is not interoperable or where a component can not replace an other component

3.1.38
instance data
configuration data that resides in the DTM instance

3.1.39
lifetime of DTM instance
time span of executing a DTM BL (from state 'created' till state 'released')

3.1.40
Link
logical relation of a DTM to a physical device (not the communication connection)

3.1.41
modal window
prevents user interaction with all windows of the process

3.1.42
online data
configuration data that resides on the device and can be accessed by communicating with the device

Note 1 to entry: Online data may be a subset of device data (i.e it may be that not all device data is accessible by communicatng with the device).

3.1.43
operation
procedure that may involve one or more method calls between FDT Objects

3.1.44
pattern
a standard solution to common problems in software design

3.1.45
platform
combination of hardware platform, target platform and target CLR, that defines the environment for execution of FDT software

3.1.46
project
generic term for the sum of information related to a set of devices

Note 1 to entry: The definition of project is specific for a Frame Application.

3.1.47
proxy object
object which functions as a representative of an other object

Note 1 to entry: The proxy pattern is a often used software design pattern.

Note 2 to entry: IEC 62453-42 uses the proxy pattern for interaction between DTM BL and DTM UI, between DTM BLs, between DTM BL and Communication Channels, for supporting multiple .NET Framework versions in a Frame Application and for providing backward compatibility to FDT1.2.x.

3.1.48
reassign

3.1.48.1
reassign
<DTM replacement>
Change the TypeInfo assigned to a device node from one TypeInfo.Id to the same TypeInfo.Id in a different DTM

3.1.48.2
reassign
<device replacement**>**
change the TypeInfo.Id to another TypeInfo.Id within the same DTM or to another DTM

3.1.49
reassignment
process of assigning a different DtmType to a device node with assigned DtmType

Note 1 to entry: It is possible that for the previously assigned DtmType already a dataset exists. This dataset should be considered in the reassignment.

3.1.50
replacing installation
installation of a new version of a DTM which replaces a currently installed DTM version

Note 1 to entry: A Frame Application is notified about the installation, but the DtmDeviceTypes do not need to be reassigned. DTM Updates (see update) and DTM Upgrades (see upgrade) replace installations of older versions of the DTMs.

3.1.51
revision
identification of modification of non-FDT components, e.g. device firmware or device hardware

Note 1 to entry: Not all fieldbus specifications supported by FDT and/or device types provide a version identification which allows to derive compatibility statements.

Note 2 to entry: In contrast to a version, revisions require additional fieldbus or device type-specific knowledge to derive compatibility or interchangeability predictions.

3.1.52
SemanticInfo
identifier that provides a reference to semantics defined in a specific context

Note 1 to entry: The reference is provided by the SemanticId, the context is provided by the ApplicationDomain that accompanies the SemanticId.

Note 2 to entry: There may be several semantics provided for an information item, e.g. a parameter may be described in a fieldbus profile as well as in a device profile (e.g. for drives), that is why several semantic infos may be provided for an information item.

3.1.53
set point
target value that an automatic control system will aim to reach

Note 1 to entry: For example a boiler control system may have a temperature set point, which is the temperature the control system aims to attain.

3.1.54
Sibling DTMs
DTM instances in an FDT Project, which are classified by their relation to the same Communication Channel

3.1.55
surrogate process
process hosting an object on behalf of client processes

Note 1 to entry: A surrogate process can have other qualities than the client process. E.g. it can be used to load a different .NET Framework.

3.1.56
synchronous operation
operation that is performed while the object that requested the operation is waiting for the result

3.1.57
target CLR
common language runtime, which defines the environment for execution of FDT software

Note 1 to entry: An example for target CLR is the CLR 4.0.

3.1.58
target platform
native data size supported by the machine and operation system, on which the FDT software is executed

Note 1 to entry: IEC 62453-42 defines support for 32-bit and for 64-bit target platforms.

3.1.59
transformer
component for the translation between FDT1.2.x XML documents and IEC 62453-42 datatypes

Note 1 to entry: Transformers are provided by the FDT Group for each communication protocol specified in an FDT Protocol Annex specification.

3.1.60
update
process to replace a component with a later (up to date) revision (update revision) that includes error corrections

3.1.61
update revision
(minor) revision of a component that includes error corrections and small enhancements

Note 1 to entry: In comparison to an upgrade revision an update revision includes no major functional enhancements or new features . An Update Revision shall be backwards compatible to previous revisions of the same component.

3.1.62
upgrade
process to replace a component with a later revision that includes functional enhancements and/or new features (upgrade revision)

3.1.63
upgrade revision
revision of a component that includes functional enhancements and/or new features compared to a previous revision of the component

Note 1 to entry: An Upgrade Revision shall be backwards compatible to previous revisions of the same component.

3.1.64
version
an instance of a software product derived by modification or correction of a preceding software product instance (see [33][2])

Note 1 to entry: The format of a version is: Major.Minor[.build[.revision]] for more information see http://msdn.microsoft.com/en-us/library/hdxyt63s

Note 2 to entry: Version is used in FDT2 for identification of FDT software components and for corresponding compatibility attributes.

3.2 Abbreviations

For the purposes of this document, the abbreviations given in IEC 62453-1, IEC 62453-2 as well as the following apply.

API	Application Programming Interface
BTM	Block Type Manager
CLR	Common Language Runtime
CLS	Common Language Specification
CSS	Cascading Style Sheet
DCS	Distributed Control System
DD	Device Description
DLL	Dynamic Link Library
DOM	Document Object Model
DTM	Device Type Manager
FA	Frame Application
FDT	Field Device Tool
FDT1.2.x	FDT implementation according IEC 62453-41
FDT2	FDT implementation according IEC 62453-42
GUI	Graphical User Interface
GUID	Globally Unique Identifier (a UUID)
HART®[3]	Highway Addressable Remote Transducer
IID	Interface ID
IO	Input / Output
LCID	Locale ID
MSDN®	Microsoft Developer Network
PLC	Programmable Logic Controller
WPF	Windows Presentation Foundation
XDR	XML data reduced
XSL	eXtensible Stylesheet Language
XSLT	XSL Transformations

3.3 Conventions

The conventions for the UML notation used in this document are defined in Annex H.

---

2 Numbers in square brackets refer to the Bibliography.

3 See Annex G.

This document specifies requirements to software. Different levels of requirement may be recognized by the used wording.

Wording	Indicates
'shall', 'has to', 'have to' ,or 'Mandatory'	No exceptions allowed.
'should' or 'Recommended'	Strong recommendation. It may make sense in special exceptional cases to differ from the described behavior.
'conditional'	Function or behavior shall be provided, depending on defined conditions.
'can' or 'Optional'	Function or behavior may be provided.

Further conventions are:

Convention	Indicates
Note:	Indicates text (in small letters), which does not express requirements, but provides additional information.
<MethodName>	Angle brackets are used to indicate a reference to an asynchronous method NOTE If looking for definition in Annex B: Such methods are implemented as pair of BeginMethodName()/EndMethodName()

Code examples provided in this document are intended for illustration of the described concepts. They should not be used as is. Developers of FDT software should consider where the developed code is applied and design the software accordingly. For exact specification of protocol-specific implementations, refer to the FDT Protocol Annex documents.

4 Implementation concept

4.1 Technological orientation

The .NET Framework is a software framework by Microsoft. The Framework includes a large library and supports several programming language. Programs written for the .NET Framework execute in a software environment, named the Common Language Runtime (CLR). The class library and the CLR together constitute the .NET Framework. The CLR is a Microsoft-specific implementation of the definition provided by ISO/IEC 23271:2012 and ISO/IEC 23270:2006.

The implementation of FDT´s client/server architecture defined in this Technical Report is based on the .NET Framework.

This part of IEC 62453 specifies .NET interfaces (what the interfaces are), not the implementation (not the "how" of the implementation) of those interfaces. It specifies the behavior that the interfaces are expected to provide to client applications that use them. The FDT-specification neither specifies the implementation of DTMs nor the implementation of Frame Applications.

Included are descriptions of architectures and interfaces which seemed most appropriate for those architectures. Like all COM implementations, the architecture of FDT is a client-server model where DTMs are the server components managed by the Frame Application.

4.2 Implementation of abstract FDT object model

Figure 2 provides an overview of how the FDT Objects (defined in IEC 62453-2) are implemented in IEC 62453-42 and how their relationship to each other is implemented. The FDT Objects are implemented as .NET objects.

IEC 62453-42 defines a simplification in regard of the implementation of the object model. Within this implementation the only Channels are Communication Channels. Process Channels are mapped to ProcessDataInfo objects. Communication Channels may not have User Interfaces. The objects "Project" and "Host Channel" are considered as Frame Application-specific implementations and are not defined within this Technical Report.

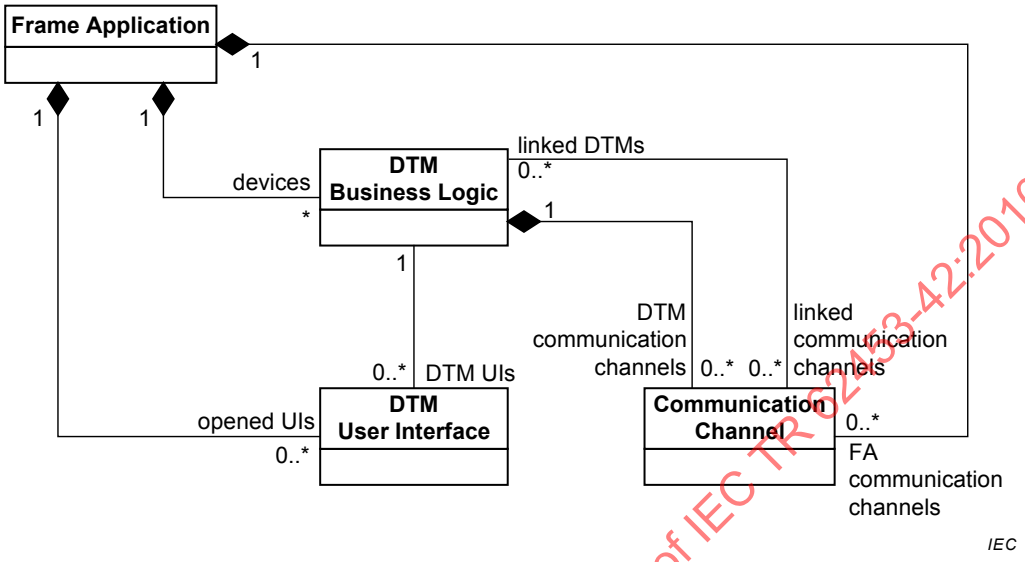


Figure 2 – IEC 62453-42 Object Model

If the Frame Application is a distributed software system, the Frame Application is responsible to organize the instantiation of the objects (based on a vendor-specific implementation).

4.3 FDT Frame Application (FA)

A Frame Application is the runtime environment for the DTMs and provides interfaces which enable the DTM Business Logic and the DTM User Interfaces to interact with its environment. In addition, the Frame Application manages the interaction between the DTM Business Logic and the DTM User Interface by providing a standard messaging interface (see Figure 3).

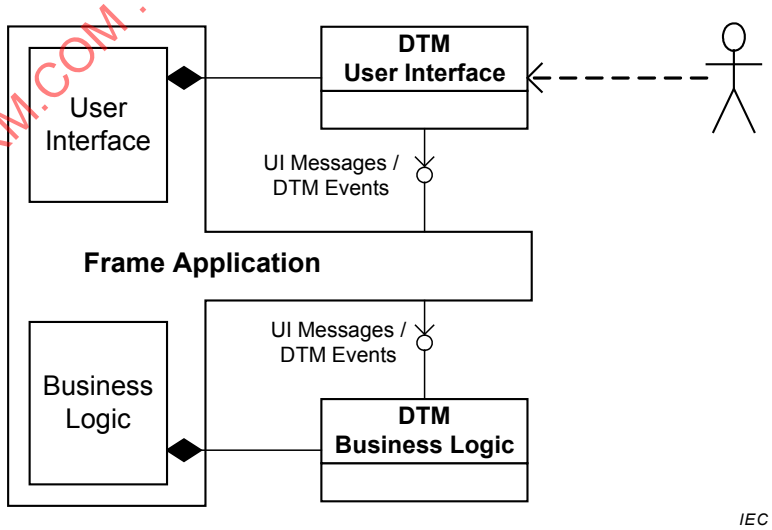


Figure 3 – Frame Application

The messaging interface is used for the transport of DTM-specific messages and events. Contents and format of the messages are proprietary and not understood by the Frame Application.

The Frame Application User Interface represents all functions of the frame related to user interface. The Frame Application Business Logic represents all frame functions related to business logic. Both are not specified by this Technical Report, but are implementation specific parts of the Frame Application (e.g. functional parts or structural parts). These two parts can be comprised in one single application or in separated applications, for example in a server and client application.

Frame Applications can have no, one or multiple Frame Application User Interfaces.

The Frame Application Business Logic part is responsible to execute the DTM Business Logic. It provides services which enable the DTM Business Logic to:

- persist data in the Frame Application persistence storage (see 4.11.1),
- communicate with associated device,
- request displaying of further user interfaces (e.g. user dialogs, additional DTM User Interface),
- browse the FDT topology and interact with other DTMs,
- inform the Frame Application regarding events (error / trace messages, progress etc.),
- interact with the DTM User Interface.

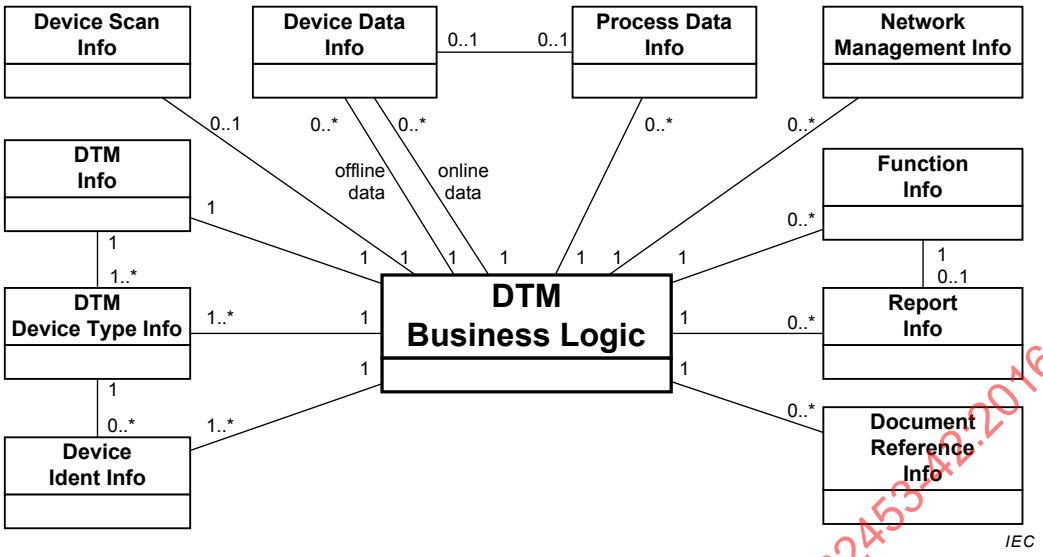
The Frame Application user interface part makes the DTM services available to the users, for example the functions and user interfaces supported by a DTM (see 4.5). It hosts the DTM User Interfaces as part of its own user interface and provides services to:

- interact with the DTM Business Logic (see 4.4)
- request displaying of further user interfaces (e.g. user dialogs, additional DTM User Interface)
- browse the FDT topology and interact with other DTMs
- inform the Frame Application regarding events (error / trace messages, progress etc.)

4.4 DTM Business Logic

4.4.1 General

The Frame Application interacts with the DTM Business Logic through defined interfaces (see Annex B). Figure 4 shows the information objects which are used by the interface definitions implemented by the DTM Business Logic.



NOTE Since this subclause describes the general concept of FDT, the actual implementation in a DTM may differ. (E.g. if a network protocol uses network management, it will be mandatory to provide network management information.)

Figure 4 – DTM Business Logic

4.4.2 Implementation of DTM, DTM Device Type, and Device Ident Info

In order to increase performance in creation of libraries and selection of DTMs, the service GetTypeInformation shall be provided by a separate class (the so called DtmInfoBuilder class). The DtmInfoBuilder is installed together with the DTM. It implements the same interface to provide information as defined for a DTM (see IDtmInformation in Annex B). The main advantage of DtmInfoBuilder is, that it can be used without instantiating the DTM.

NOTE 1 By using the DtmInfoBuilder it is possible to adapt the available TypeInfos depending on various conditions (e.g. DD-Interpreter DTMs may provide TypeInfos depending on installed DD files).

Figure 5 shows the information datatypes that are provided by a DTM in order to support these Frame Application function and how this info datatypes are used to describe the supported devices (see 7.4 and 7.5 for detailed description).

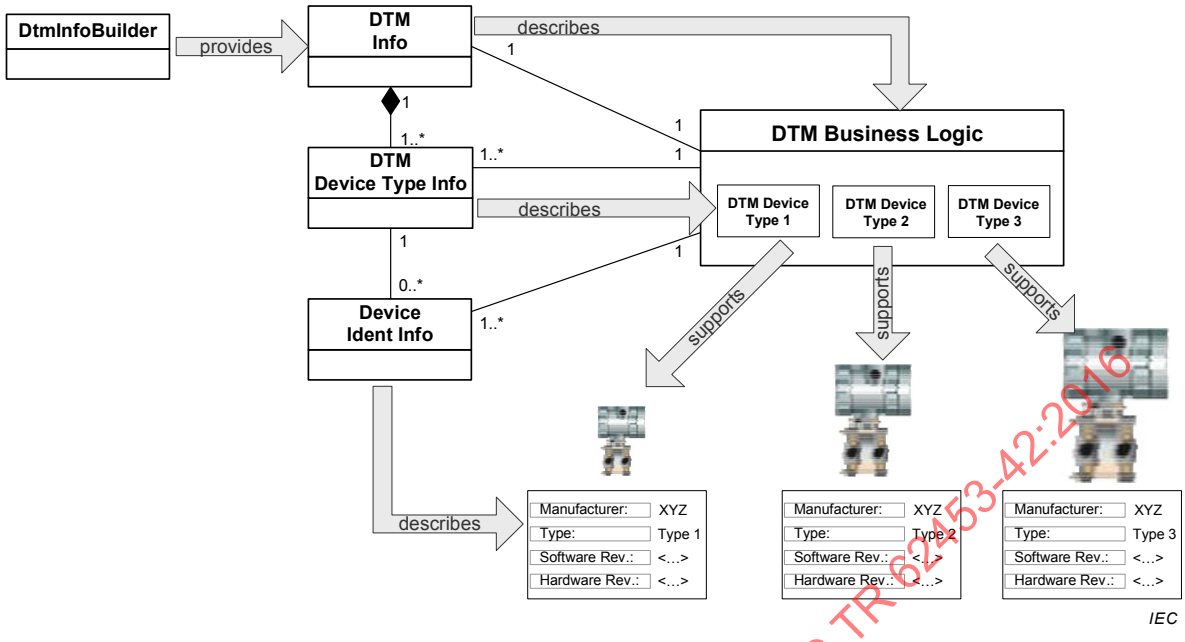


Figure 5 – DTM, Device Type and Device Ident Info

NOTE 2 Since this subclause describes the general concept of FDT, the actual implementation in a DTM may differ.

The representation for a particular physical device type within the DTM is called DTM Device Type. A DTM may provide one or more DTM Device Types. The concrete design and implementation of the DTM Device Types is not in scope of FDT.

Information about physical device types, which can be handled by the DTM Device Types is returned by the service IDtmInformation:GetDeviceIdentInfo (see definition of IDtmInformation in Annex B). Such information is for example manufacturer, type, hardware and embedded software version of the device. The DTM may even return regular expressions for some specific device identification elements to signal that the DTM Device Type can be used for all devices for which the expression matches (e.g. the character asterisk '*' for the hardware version may signal that the DTM Device Type supports all hardware versions).

The information returned by service IDtmInformation:GetDeviceIdentInfo is fieldbus-specific and therefore defined by the document describing the protocol profile integration in FDT2. However, FDT defines the means to transform the information into a protocol-independent format to enable Frame Applications without protocol-specific knowledge to use it.

4.4.3 Implementation of DTM device parameter access

A DTM supports interfaces to read and write device parameters stored in the DTM instance data (offline data) and directly in the connected device (online data). This data is represented by DataInfo (see 4.12.1 for detailed description).

A DTM has to expose a defined set of device parameters which are publicly available (see 4.12 for detailed description). Parameters are provided in a bus neutral structure allowing their use without knowledge of the fieldbus protocol.

4.4.4 Process Data Info

Process data provided by devices (e.g. IO signals) are integrated into the functional planning of the control system. The process data related information for the integration of the device into the control system like datatype, signal direction, engineering units, and ranges is provided by the DTM Business Logic for each DTM Device Type (Figure 6), but may also depend on the device instance configuration.

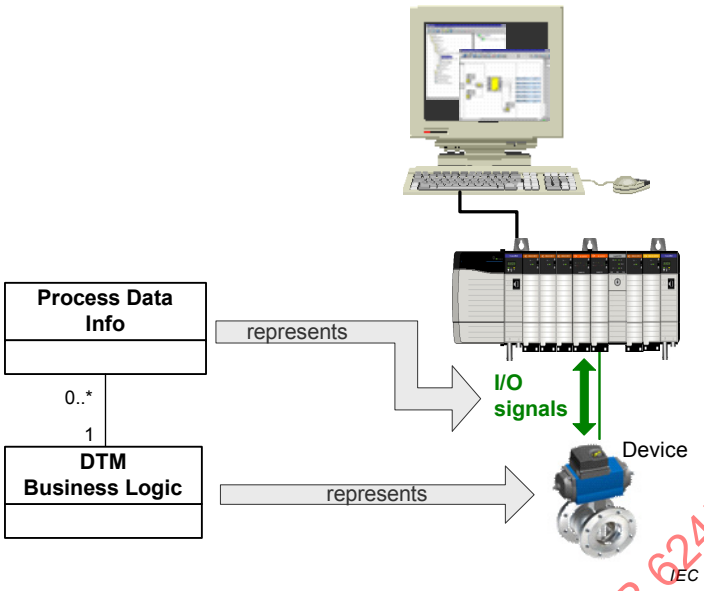


Figure 6 – Process Data Info

The process values provided by a device, both the number and type, may depend on the configuration of the DTM. Thus the number of available Process Data Info objects may also depend on the device configuration. A Frame Application is able to inform the DTM that further configuration changes shall be prohibited because the process data is already integrated into functional planning of the control system. In this case the DTM shall not allow any changes which will affect the definition of the available process values.

The process information is protocol-specific. Each FDT Protocol Annex defines a derived class defining which information shall be contained. However, the properties in the base class Process Data Info provide common information (e.g. IO signal name, tag etc.) in a protocol-independent format to enable Frame Applications without protocol-specific knowledge to integrate the process variables into the system (see 7.11.1 for detailed description).

If the process data value is also available as Device Data Info, then the corresponding element is referenced by the Process Data Info element (see IOSignalRefs in 7.11.1).

A Process Data Info element may have a relation to a Communication Channel. This is a typical case for a Gateway DTM for a remote IO (see IEC 62453-2:−, 4.2.3.2.4). The relation is represented within the Process Data Info element (see IOSignalRef in 7.11.1).

NOTE The Process Data Info replaces Process Channel as defined in FDT1.2.x.

4.4.5 Diagnostic Data Info

A DTM provides a service to retrieve the status of the related device. The status is encoded in a protocol independent way, according to [7].

4.4.6 Network Management Info

The DTM supports an interface to read and write network management information which can for example be used for address management and bus master configuration (see 8.11.1).

The Network Management Info is protocol-specific. It may contain device bus-address, tag and additional bus-specific configuration settings. Each FDT Protocol Annex defines a derived class defining the protocol-specific record. However, the base class Network Management Info provides common information (e.g. bus-address, tag) in a protocol-independent format to enable Frame Applications without protocol-specific knowledge to use it (see 7.13 for detailed description).

4.4.7 Function Info

Each DTM may provide a set of functions comprised by

– DTM User Interfaces,
– DTM functions without DTM User Interface (CommandFunctions) and
– references for external documents.

The Function Info object provides the information about the functions such as name, status (enabled / disabled), etc. (see 7.14 for more detailed description).

A function should provide printable information. A Frame Application may call the documentation interface (see [4], 7.2.7.1) of the DTM to retrieve printable information. This interface returns a corresponding Report Info object which holds the printable information. The Report Info object may indicate the relation to the function.

4.4.8 Report Info

If a DTM provides instance data and/or online data, then the DTM shall implement a device type-specific reporting of the provided data for documentation and archiving purposes. The DTM BL may implement different types of reports that each cover a distinct subset of the instance or online data of a device.

The Report Info exposes a list of report types supported by a DTM (see 7.19). The list may be grouped. The list is static over the lifetime of a DTM instance, there are no dependencies on the current application context.

4.4.9 Document Reference Info

A DTM may provide references to external documents, which are displayed by the Frame Application. These references may be provided as part of the TypeInfo (as static information) or as part of the Function Info (may change dynamically). The Frame Application may provide an own viewer for the documents or rely on the condition that external software is installed for the document type. It is recommended, that DTMs use common document formats (recommended formats are PDF, CHM and HTML). Otherwise DTMs should provide the necessary viewers.

4.5 Implementation of DTM Functions

4.5.1 DTM User Interface

A DTM User Interface (DTM UI) may be a graphical control which is integrated into the user interface of the Frame Application or a proxy object handling the interaction with an external program provided together with the DTM.

A DTM UI may be modeless or modal. A modal DTM UI behaves modal only in respect to the DTM instance. A DTM instance should expect that the modal DTM UI blocks only activity in regard to its own UI (DTM instance modal) and that other UI (Frame Application, other DTMs) still may trigger actions. Modal DTM UIs should be used as sparingly as possible.

A Frame Application may also implement Frame Application modal behavior for modal DTM UIs.

The contents and layout of the DTM User Interfaces is device-specific, but shall follow the DTM Style Guide [6].

NOTE 1 A DTM may provide DTM User Interface for different platforms (e.g different display formats e.g. PC, mobile device etc.). Which platform is supported by a DTM User Interface is described by corresponding UI Function Info element returned by the DTM (see 7.14). The FDT specification will define which platforms shall be supported.

NOTE 2 The intention is for IEC 62453-42 to allow later extensions in regard to User Interface technologies and platform support.

4.5.2 Function access control

The Frame Application can restrict the invocation of functions provided by a DTM, disable actions (buttons) of the DTM UI (e.g. Apply) and restrict DTM transition to certain states (e.g. restrict transition to state Online by not calling the method EnableCommunication()).

The Frame Application will get the list of the functions provided by the DTM in IFunction.FunctionInfo property. The DTM shall expose all functions available in the DTM in all modes. When the functions are not applicable for the current mode of operations, FunctionItem.Enabled property will indicate that. A DTM shall expose always the same set of functions for a device node (even if the DTM instance was terminated and loaded again). The number and type of listed functions shall not change, but the status of the functions in regard to availability and visibility may change. Frame Application should hide the functions with Hidden property set to TRUE.

The DTM shall indicate when the state of Enable is changed, by IFunction.FunctionsChanged event.

When a user interface for the DTM is invoked, the DTM shall not allow switching the context from within the DTM user interface.

For Example:
If a Diagnostic Function is invoked by the Frame Application, the DTM will present the diagnostic information in the user interface. The user should not be allowed to invoke the configuration screen from within the DTM user interface for the diagnostic function without the permission from the frame. The user shall be able to invoke the configuration screen from the functions exposed to the Frame Application if the permissions in the Frame Application allow it.

If the MainOperation Function is invoked by the Frame Application, the DTM will present all available functions within the user interface. The user may invoke the Configuration screen from within the DTM user interface as well as the Diagnostic function or any other integrated function.

4.5.3 Handling of standard UI elements in modeless DTM UI interfaces

Modeless DTM UIs shall delegate the presentation and handling of their standard dialog elements to the Frame Application. The standard dialog elements are:

– actions with standardized semantics (Apply / Close / Online Help) (see Action Area in [6]) and
– DTM UI-specific status information (data source, summary parameter modification state, UI operation mode, activation of service mode) (see Status Bar in [6]).

To ensure a consistent user interface appearance between the different DTM vendors, a DTM UI may delegate presentation and handling of additional DTM application-specific actions to the Frame Application. Nonetheless DTM UIs are allowed to implement non-standard dialog actions within their own UI area (see Application Area in [6]).

The set of standard dialog actions and their respective semantics is fixed. However, the availability of these actions may change at any time depending from the internal state of the DTM UI. The set of application-specific actions including their individual availability is not fixed. A DTM UI may add, remove, rename, enable or disable application-specific actions at any time depending from its individual requirements. A DTM UI shall inform a Frame Application whenever the availability of its standard actions or the set or availability of its custom actions changes (see events IStandardActions.StandardActionItemSetChanged and IApplicationSpecificActions.ApplicationSpecificActionItemSetChanged in Annex B).

A Frame Application may use dedicated UI elements, e.g. button controls, to provide direct access to the standard dialog actions, as well as indirectly invoke them in the context of user interaction with other Frame Application UI elements. A Frame Application shall always show all custom actions exposed by a DTM UI with dedicated UI elements. These shall be unambiguously associated with the DTM UI as described in the DTM Style Guide [6].

4.5.4 Command functions

Command functions are used to execute actions (commands) either on the DTM BL or within the context of the graphical user interface. Command functions in context of the DTM BL shall not have a GUI, but UI Command functions may show a GUI (see 6.4).

A Command function may have parameters. The information about the parameters, which is provided by the DTM BL, may include default values of the parameters. The actual parameter values are passed when the Command function is executed.

4.6 User management

4.6.1 General

FDT does not define a standard system for user management. The user management is part of product-specific definitions and may be implemented differently for different Frame Applications. However it is still necessary to define a common handling for access permissions, access rules and how components from different vendors communicate information regarding access permissions.

4.6.2 Multi-user access

Some Frame Applications provide multi-user capability. Such a system provides access for multiple users at the same time and may be distributed over several computers. This specification considers the distributed environment as one Frame Application. The Frame Application and the DTM are equally responsible to provide the multi-user access and to ensure consistency of data.

If, within one Frame Application, multiple users access the same device or the same device dataset, the Frame Application shall start a separate DTM instance for each user. All these DTM instances shall have same DTM type, shall be instantiated for the same DtmDeviceType and for the same physical device. Each DTM instance manages a separate instance dataset. These instance datasets are synchronized by means of the persistent dataset (see 4.11.5).

NOTE An example architecture for multi-user scenarios is found in 11.4.

4.6.3 User levels

4.6.3.1 Introduction

DTMs may be integrated in different Frame Applications, which may have varying requirements to restrict visibility and accessibility of device and persistent data, for example for plant safety reasons or to present a customized view to the user. The grade of restriction varies with the types of users supported by a system. Examples for users requiring data access restrictions are:

- a user assigned to observe a plant shall not have access to calibration-specific device parameters and consequently shall not see related DTM functionality,
- a device commissioning specialist needs to have access to calibration data and functions,
- a user assigned to operate a plant shall be able to change (write) set point values and be offered appropriate functionality while a user assigned to observe a plant is not allowed to execute such changes.

4.6.3.2 Access Control Concept

FDT uses a role based access control concept. A Frame Application initializes a DTM Business Logic and the associated User Interface with the same FDT-specific user level (see definition of IDtm.Init() and IDtmUIFunction.<Init()> in Annex B). The user level is immutable over the lifetime of the Business Logic/User Interface instances.

In terms of access control, every actor in an FDT system may have one of the following three user levels at the interface of a DTM (Table 1):

Table 1 – FDT User levels

User Level Name	Description
Observer	This user level stands for an actor that observes the current process only.
Expert	This user level stands for an actor who has to execute specific use cases, e.g operation use cases (operation expert) or device maintenance use cases (maintenance expert). This user level allows the Frame Application to configure access and privileges.
Engineer	This user level stands for an actor that has to do the plant planning, device configuration/parameterization and plant maintenance.

The user levels allow a stepwise extension of permissions. The Observer typically has a minimum permission set, the Expert has an intermediate permission set (which is configured by the Frame Application) and the Engineer has a full permission set.

The Expert user level may be considered as a super-set of the actors “Operator” and “Maintenance” as defined in IEC 62453-2. Since a Frame Application may configure the access permissions for the Expert, it is possible to apply fine grained permissions, that can be adapted to different application scenarios.

NOTE For an explanation of the fundamental user levels and use cases that were considered for the design of the FDT specification see Annex A. A Frame Application may support only a subset of these use cases or additional use cases not defined in the annex.

According to the role set by the Frame Application, the DTM Business Logic and User Interface shall control access to device and persistent data (see definition of interfaces IInstanceData and IDeviceData in Annex B) as well as adapt its user interface appearance. This includes to hide some data or display it as read-only, but also to partially disable DTM-specific functionality (see definition of IFunction.FunctionInfo in Annex B), if it requires data access rights that are not associated with the specified user level.

It is mandatory for a DTM to implement a safe and read-only usage for the “Observer” user level. It is also mandatory for a DTM to implement unlimited usage for the “Engineer” user level. It is optional for a DTM to implement configurable custom usage for “Expert” user level. If “Expert” level is not implemented by a DTM but is set by the Frame Application, the DTM shall use the behavior for “Observer” user level.

Table 2 gives an overview about the user interfaces and functions that are expected to be available for the individual user levels. The data access rights should be defined to allow for the execution of these use cases.

The assignment of roles to individual users is Frame Application-specific. The Frame Application may implement an own user management sub-system or use fixed user levels.

Table 2 – Role dependent Access Rights and User Interfaces for DTMs

Use Case	Sub Cases	User level		
		Engineer(M)	Expert(O)	Observer(M)
System Planning	Network Management	If a DTM implements these use case, it shall expose all related commands and user interfaces.	If the DTM implements these use cases and supports the user role "Expert", it shall allow Frame Application to configure access to the exposed data, commands and user interfaces	-*)
	Busmaster Configuration			-
	Channel Assignment			-
System Generation	Network Management			O {r}
	DTM matching			O {r}
Device Configuration	-			-
Simulation (Force)	-			-
Offline Operation	Offline Parameterization			-
	Persistent Data Comparison			-
Online Operation	Online Functions (reset + other functionality that requires online device connection)	If a DTM implements these use cases, it shall expose all related commands and user interfaces.	If the DTM implements these use cases and supports the user role "Expert", it shall allow Frame Application to configure access to the exposed data, commands and user interfaces	-
	Online Parameterization			-
	Calibration			-
	Device/Persistent Data Comparison			-
	Adjust SetValues			O {r}
	Upload			-
	Download			-
Bulk Operation	Upload			-
	Download			-
Online View	Network Scan			O {r}
	Online Status			M {r}
	Online Trend			M {r}
	Device Identification			M {r}
	Online View Parameter Set			O {r}
Report Generation	-			M {r}
Device-specific Operations	Device vendor-specific (or extended) DTM functions after DTM/Device-specific OEM Service login			-

M Mandatory (if a DTM implements this use case, it shall expose all related commands and user interfaces in the specified user level)

O Optional (a DTM may expose the related commands and user interfaces in the specified user level)

r User level shall have read access to all data related to the use case

w User level shall have write access to all data related to the use case

- Use case not supported in this user level (DTM shall not expose any related commands/user interfaces)

*) A DTM shall allow all user levels to set the device address in the DTM with the method SetAddressInfo().

4.6.3.3 Frame Application configured access control

It is very difficult and may even be impossible for the DTM vendor to provide correct access control settings for all occasions. The data, which can be accessed, and the functions, which can be used, are changed by the user; depending on where the DTM is used or what is the operational phase of the plant. Here are some examples:

a) The same user may have different permissions for the same device controlled by the same DTM when the device is connected in the plant or in the instrument shop. In the instrument shop, the user may have all equipment to calibrate the instrument and the corresponding privileges should be granted. Little or no changes may be allowed when the instrument is connected later to the actual running control system.

b) The same user may have full control when the plant is being engineered, but the changes to the device may be significantly restricted, when the plant is in running state.

c) The same group of users may have different permissions for different instruments – some of the maintenance personell may be trained to work with transmitters, other may be specialized in valve maintenance.

d) In a small application one person may be responsible for the entire application and he may have unlimited access to all device maintenance procedures, but in a big application, often the access is controlled according to the individual experience of the technical staff.

To address the different cases, an “Expert” user level is provided. When the DTM supports the “Expert” user level, the data which can be accessed and the function which can be invoked are restricted by the Frame Application depending on the rules in the plant, on the operational phase, the individual user or team experience and other factors.

The Frame Application can use the Expert user level to create additional levels of access to the DTM data and functions for individual user or for a group of users. For example, when the access control is configured for the Operation Expert, the Frame Application may enable the access to Set Point Values, to the Tuning parameters and to Diagnostic functions. In another example, the Frame Application may enable the access to the Calibration parameters, to the calibration functions and to the Online Parameter View when the Device DTM is invoked in the instrument shop environment.

NOTE Bbe aware that the user level “Operator” as defined in FDT1.x specification is not supported in IEC 62453-42. The term Operator in this document is used to describe an expert for plant operations.

4.7 Implementation of FDT and system topology

4.7.1 General

IEC 62453-42 differentiates two topology views: logical topology and physical topology (see Figure 7).

A logical topology is created by a hierarchy of DTMs. Child DTMs are connected to Parent DTMs via the Communication Channel of the Parent DTM. A Parent DTM may have multiple children. This relation is managed by the Parent DTM. This means that a Parent DTM knows all its Child DTMs.

A Child DTM may be assigned to multiple parents (e.g. if different network paths may be used to access a device). A Child DTM is not notified if it is assigned to a Parent DTM, but it may request a list of parents from the Frame Application by using the method ITopology.GetParentNodes().

A Child DTM can use only one communication path at a time to access the respective device. The Parent DTM providing this communication path will be marked by the Frame Application as ‘primary parent’.

This means, that the logical topology describes the logical relations between the devices on an abstraction level that supports managing the communication between DTMs and devices.

A physical topology is created by defining physical connections between DTMs. Connections are defined between Ports of the DTMs. This means the physical topology describes the actual hardware installation. The connections are managed by the Frame Application. It is possible to use these connections for representation of all kind of network structures.

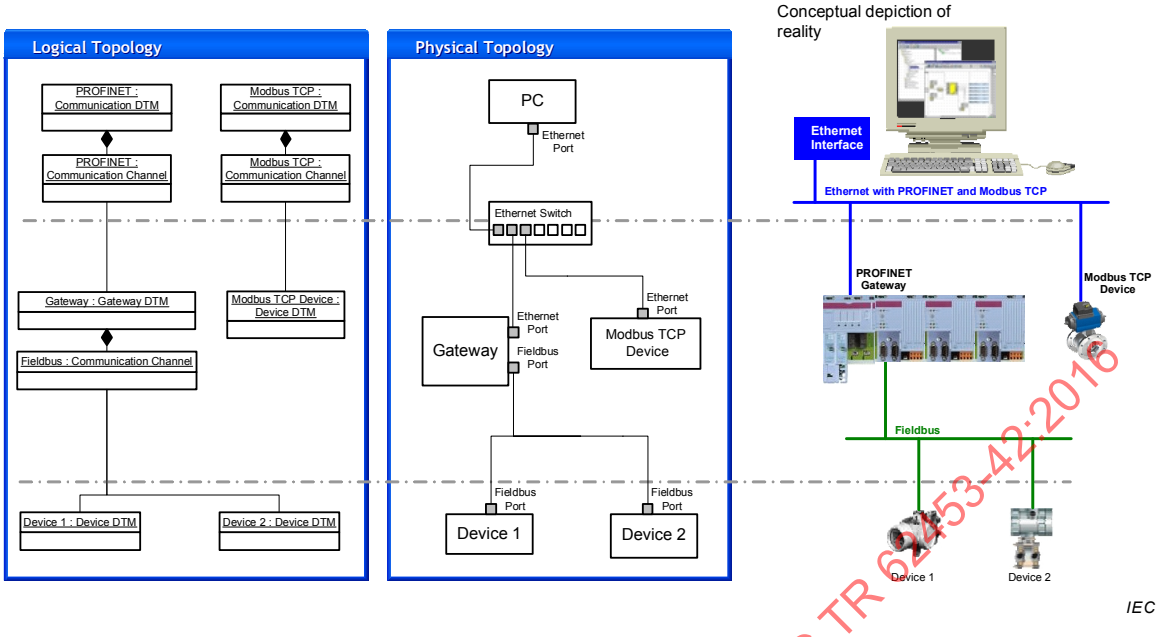


Figure 7 – Logical topology and physical topology

4.7.2 Topology management

4.7.2.1 Logical topology

The Frame Application is responsible for managing the logical topology – it is mandatory to support the logical topology. That means the Frame Application shall organize the routing of data for accessing a device in the plant. Some Frame Applications may require user interactions; others may support automatic operations such as topology import or fieldbus scanning. The sum of all links between DTMs according the logical topology is called FDT topology and further described in IEC 62453-2.

A DTM exposes all required information (see 4.4.2) which enables the Frame Application (and the user) to choose the appropriate DTM for a device, for example name, vendor, version of supported device types and corresponding identification properties.

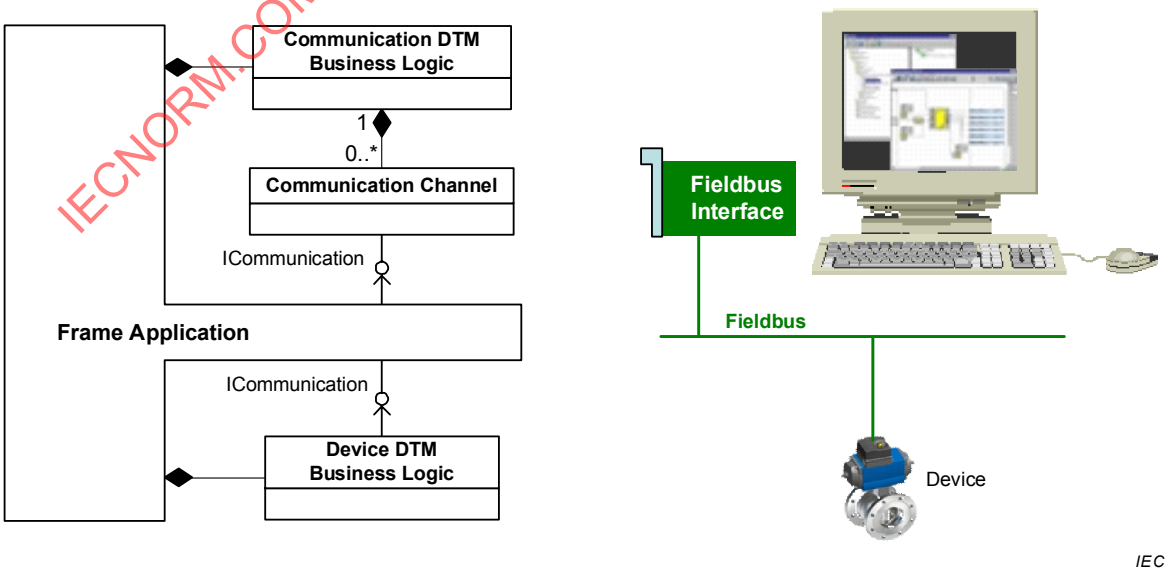


Figure 8 – FDT and logical topology

As shown in Figure 8 a Communication Channel is used as the linking element between Communication DTM and Device DTM. The Communication Channel provides access to the fieldbus.

The link between a Communication Channel and a DTM is created by the Frame Application. However, final decision whether a DTM shall be linked or not shall be made by the Communication Channel. The Frame Application has to call the method ISubTopology.<ValidateAddChild()> (see definition in Annex B) before link is created. The Communication Channel shall at least check whether the required network protocol of the DTM to be linked fits to its own supported protocol. If this is not the case, then the linking shall be rejected. In addition, the Communication Channel may perform further checks, for example whether the number of linked DTMs exceeds a limit.

Neither the Communication Channel (or corresponding DTM) nor the linked DTM shall need to manage topology information in order to access the respective physical device. The Frame Application supports to request topology information by the methods ITopology.GetParentNodes(), ITopology.GetSiblingNodes() (and ITopology.GetChildNodes() (for all see definition in Annex B).

The rules for identification of DTMs and devices are described in 4.10.

For some communication protocols the order of the devices linked to the network affects the configuration of the network itself. This order is defined when inserting the corresponding DTM into the logical topology (ITopology.BeginAddChild()) and can be modified later via ITopology.BeginRepositionChild(). The Frame Application always has to maintain this order when returning collections of DTMs in ITopology.GetChildNodes() and ITopology.GetSiblingNodes().

If the Frame Application provides a view on the channel, it shall show all Child DTMs in their respective order. In this view, the Frame Application shall allow the user to insert a new DTM at a specific position between the existing Sibling DTMs or to change the position of an existing DTM in regard to its Sibling DTMs.

4.7.2.2 Physical Topology

Advanced topology management requires the additional planning of cable bound or wireless connections between devices. This capability is provided by the Physical Topology.

The management of the Physical Topology is the responsibility of the Frame Application. It is optional for a Frame Application to support the Physical Topology. It is mandatory for a DTM to expose all information which is required to determine whether a physical connection is possible or not. The Physical Topology may not have dependencies to the Logical Topology and shall be handled separately as shown in Figure 9.

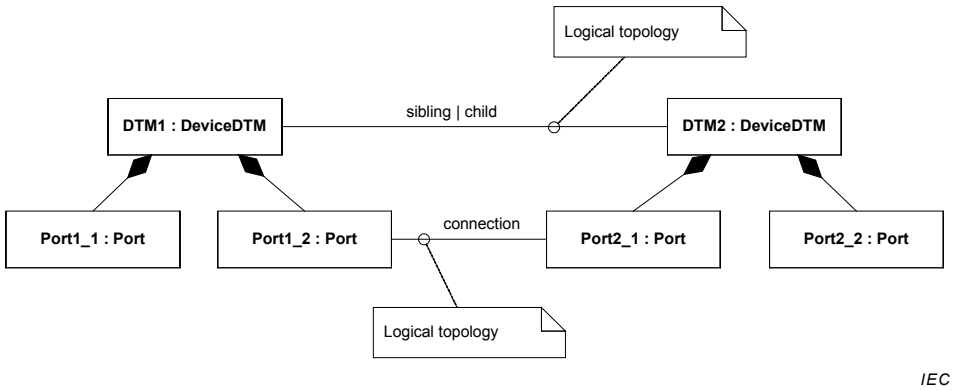


Figure 9 – DTMs and physical topology

The connections are managed by the Frame Application. Information regarding connections may be accessed with the interface IPhysicalTopology. See Annex B for a detailed description.

4.7.2.3 Communicating and non-communicating devices

An automation system integrates communicating devices,as well as devices which do not communicate and therefore are not configurable via communication (e.g. power supplies and other network infrastructure elements). Information about such devices may be essential during the planning phase of the communication system and can be used to verify integrity of the network, for instance in regard to bus power overload, communication distance limitations, validity of the design (e.g. correct termination). In order to integrate such devices in an FDT-based system, a DTM may be provided to represent such a 'passive device'. The DTM provides information about the device/equipment to the Communication DTM, which is capable to use this information.

Different protocols require specific information to be provided and may have different limitations to be enforced. Information provided from DTMs for passive devices has to be provided in a standard way and format (in Network Management Info), so that different Communication DTMs can use it in a standard way. DTMs for communicating devices may need to provide similar information. Protocol-specific extensions have to define the information provided by the communicating devices and by the non-communicating devices and also how this information is used. For example, Communication DTMs for a bus powered protocol can use information from non-communicating devices (defining the power source) and communicating devices (defining the power consumption) in order to balance the power on the network.

4.7.3 Data exchange between Frame Applications

The interaction between different Frame Applications is not in the scope of FDT, but the FDT specification defines datatypes (see TopologyImportExport definition in Annex B) which can be used for this purpose. These datatype classes may be used by one Frame Application to export the FDT topology information to an XML file which then can be imported in another Frame Application.

4.8 Implementation of Modularity

Different fieldbus protocols use different device models. FDT supports the following different approaches to describing the structure of the device:

– monolithic DTM with topology description in NetworkDataInfo, ProcessDataInfo, ProcessImageInfo, DataInfo and CommunicationChannelInfo

– Module DTM, and

– BTM.

4.9 Implementation of FDT communication

4.9.1 Handling of communication requests

In order to optimize the communication, the interface of a Communication Channel allows passing multiple transaction requests in one call to <CommunicationRequest> (as a list).

The Communication Channel is expected to process the transaction requests in the order they are provided in the list. The results of the transaction requests may be passed back to the client of Communication Channel sequentially as part of the Progress callback (partial results) and the complete result shall be passed back at the end of the <CommunicationRequest> according to the extended AsyncResult pattern (see 5.6.7.2).

The relation between communication requests and communication responses can be managed by the IAsyncResult handle that is passed to a client in the call to

<CommunicationRequest>. The transaction responses for these specific transaction requests will be received by that specific IAsyncResult handle. Each transaction can be identified by an ID, the same ID is provided in the transaction response.

The cancel of <CommunicationRequest> stops execution of the transaction requests. The results of already executed transactions shall be provided back to the client. For each transaction request, that has been not executed a CommunicationError “Cancelled” shall be provided to the client.

4.9.2 Handling of communication errors

Since it is possible to pass multiple transaction requests within one call to <CommunicationRequest>, multiple transaction responses will be provided in the result of <CommunicationRequest>. This set of transaction responses may contain a mix of positive communication results (e.g. communication data) and negative communication results (CommunicationError).

4.9.3 Handling of loss of connection

After sending an Abort notification the Communication Channel shall not send any further CommunicationResponses to the communication client. For all pending requests the exception FDTConnectionAbortedException shall be thrown.The communication client should ignore any CommunicationResponse received after receiving an Abort notification.

4.9.4 Point–to-point communication

The Frame Application manages the interaction between the DTM Business Logic and the Communication Channel. The Frame Application passes a communication interface (see interface ICommunication in Annex B) to the DTM, which provides in each case a point-to-point connection between a DTM Business Logic and a device.

It is under the control of the Frame Application to enable a DTM to communicate with its device. The Frame Application has to provide the communication interface to be used to the DTM by calling the method IDtm.EnableCommunication()(see definition in Annex B) and thus allowing communication access.

In order to access the device, the DTM uses this interface as shown in Figure 10.

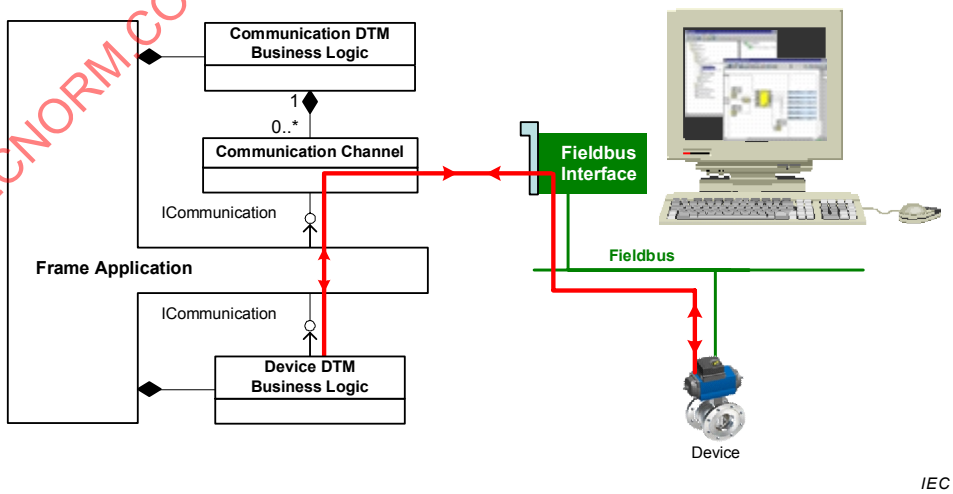


Figure 10 – Point–to-point communication

The Frame Application starts the corresponding Communication DTM and forwards the communication requests to the Communication Channel which then communicates with the hardware.

A DTM shall call the Communication Channel method ICommunication.<Connect()> (see definition in Annex B) in order to establish a communication connection to the device. After the connection has been established the DTM is able to communicate to the device by calling the ICommunication.<CommunicationRequest()> (see definition in Annex B). It is general expectation, that a DTM tests if it is connected to the intended device. See also 8.6.2.

4.9.5 Nested communication

In a nested communication scenario the Frame Application manages the interaction between the Device DTM Business Logic and the Gateway Communication Channel as well as between the Gateway DTM Business Logic and the fieldbus Communication Channel. (see Figure 11).

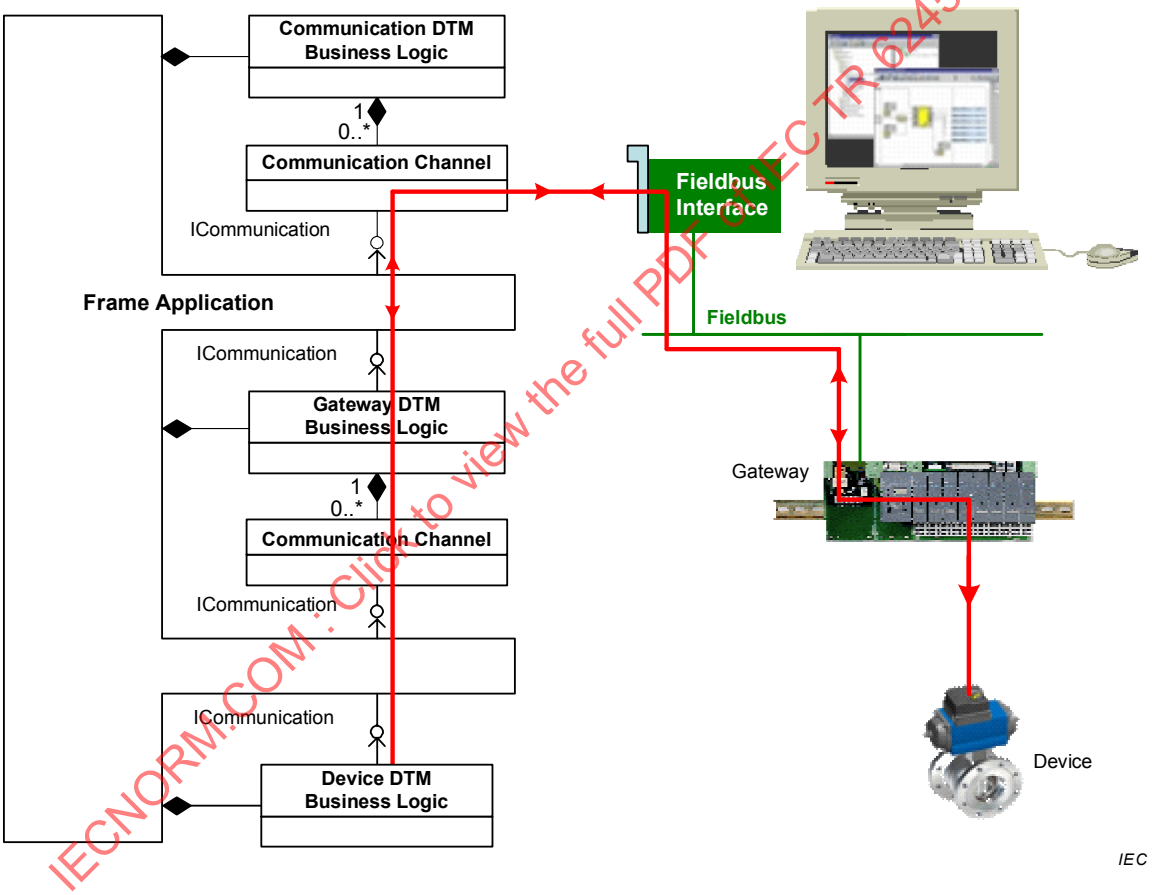


Figure 11 – Nested communication

Like in the point–to-point communication all DTMs simply use the communication interface with their devices without the awareness of the nested communication

See 8.7 for sequences related to nested communication.

4.9.6 Dynamic changes in network

Many fieldbusses provide a mechanism for temporarily disconnecting devices or switching between distinct groups of devices during operation (e.g. tool change for roboters, docking/undocking of transportation vehicles). Such mechanisms lead to changes in the communication network (called “dynamic configurations”) – devices may be disconnected. The

Frame Application (and the network configuration tool as part of the Frame Application) are able to manage the current device states at the DTM (see NetworkDataInfo.DeviceMayBeDisconnected and NetworkDataInfo.DeviceIsDisconnected in Annex B).

4.10 Identification

4.10.1 DTM instance identification

4.10.1.1 System Tag

An FDT Frame Application shall assign a unique identifier for each DTM instance. This unique identifier is referred to as “System Tag ”. The System Tag is used by DTMs

– for navigation in the FDT topology;
– for the management of Child DTMs in the FDT topology (e.g. address setting);
– to identify a DTM instance at the event interface of the Frame Application.

The System Tag is defined as GUID.

4.10.1.2 Assignment of System Tag

Following rules apply to assignment and use of System Tag:

– A Frame Application shall not change the value of the System Tag of a DTM instance during the complete lifecycle of a DTM instance. This means the same system_tag_value is used to identify the DTM instance in all interfaces (e.g. IChildDtmEvents, ITopology and ISubTopology).
– When a project is persisted the Frame Application shall save the System Tag of the DTMs such that they will be the same system_tag_value when loading as before.
– A Frame Application shall use the same system_tag_value only for DTM instances associated to the same Device Node (see Figure 14). This means it is not allowed to reuse system_tag_values for other DTM instances.
– A DTM shall not persist the value of its own System Tag.

Since the System Tag uniquely identifies a DTM instance, it is possible that DTMs store System Tags as references to other DTMs. For example if a Parent DTM needs to keep track of its children and the data they expose (e.g. for Address Setting or Busmaster Configuration), then the Parent DTM may cache information published by its children. The Parent DTM can store the cached information using the System Tag as a key.

NOTE If multiple users work on the same Device Node, each user has an own instance of the DTM, but all DTM instances use the same System Tag.

4.10.2 Hardware identification

A DTM supports the method IHardwareInformation.<HardwareScan()> (see definition in Annex B) that enables to read device information online from the connected device (see Figure 12).

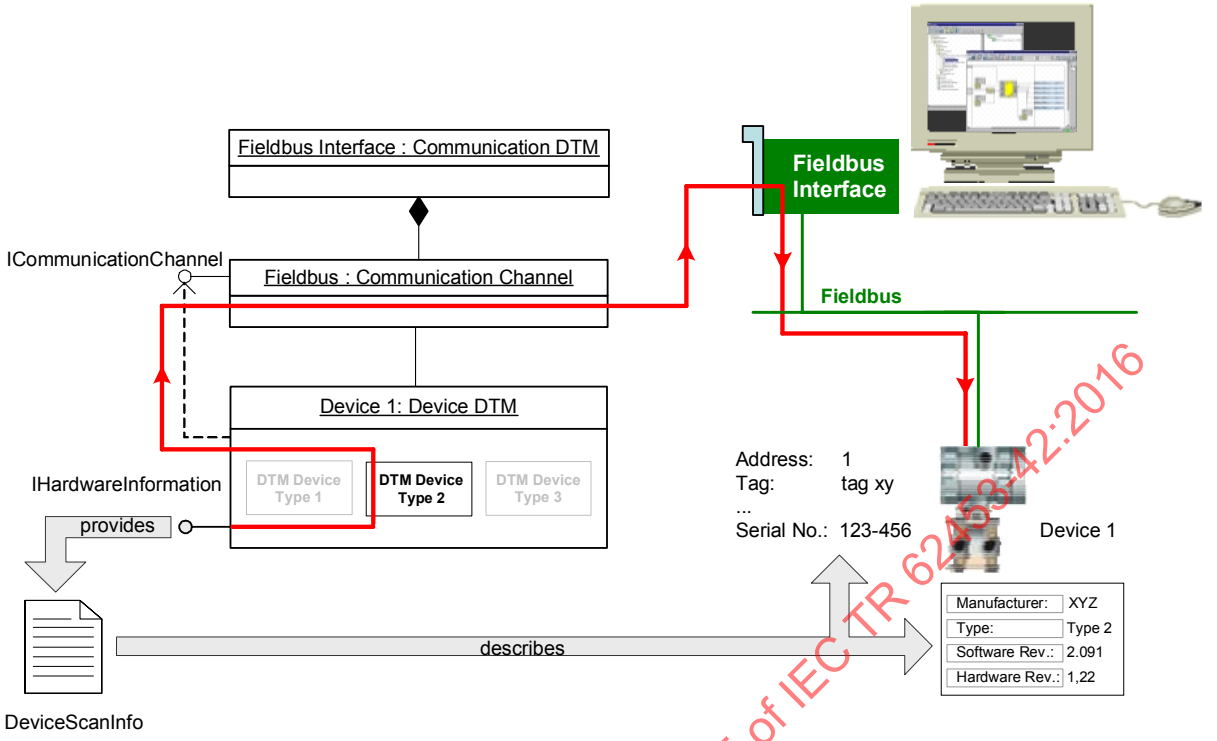


Figure 12 – Identification of connected devices

The method IHardwareInformation.<HardwareScan()> returns device type related information, which is fieldbus-specific like the information returned by IDtmInformation.GetDeviceIdentInfo() (see definition in Annex B). The transformation to protocol-independent format is implemented by the protocol-specific datatypes. See examples in 7.5.

4.11 Implementation of DTM data persistence and synchronization

4.11.1 Persistence overview

The Frame Application is responsible for the persistent storage of data (data persistence). This includes topology information as well as data managed by the DTM itself (e.g. device parameters). IEC 62453-42 only defines the interfaces, which shall be used by the DTM for data persistence (see Figure 13). While the implementation of the persistent storage system is specific for a Frame Application, the format of stored data is specific for each DTM. Both are not in scope of the FDT specification.

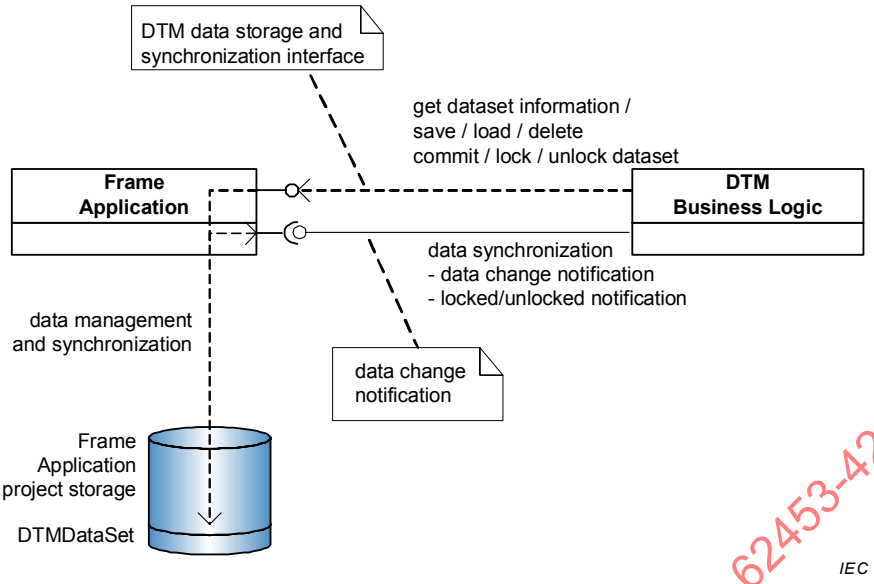


Figure 13 – FDT storage and synchronization mechanism

The Frame Application storage interface provides the DTM Business Logic methods to access its dataset (called DTMDataSet) in the Frame Application storage implementation, e.g. in a database or file persistence.

The Frame Application has to guarantee the data consistency for multi-user and multi-client data access and provides corresponding methods and events to the DTM Business Logic.

4.11.2 Relations of DTMDataSet

The Frame Application manages for each physical device one DTMDataSet and the related DTM instance as shown in Figure 14.

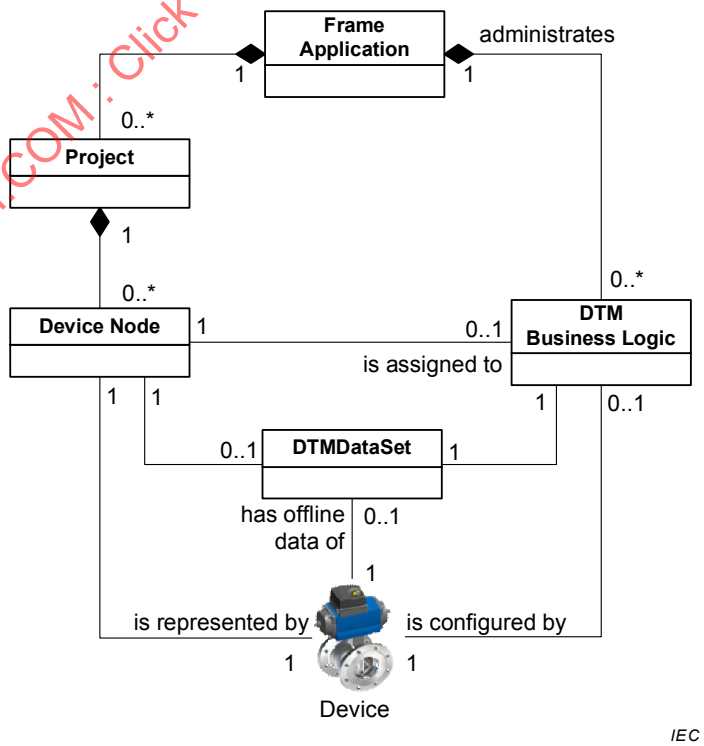


Figure 14 – Relation between DTMDataSet, DTM instance, and device

NOTE For multi-user scenarios, the multiplicities on the DTM Business Logic all have an upper limit of 'many' (see 11.4).

The Project is part of internal model of the Frame Application. It is an abstract, logical object used here to describe the management of device-instances. FDT does not define any interfaces for the Project object, since it is a pure Frame Application internal object and may have different specific implementations.

The Device Node also is part of internal model of the Frame Application. It is an abstract, logical object used here to represent a physical device in the Frame Application. It controls the lifetime and data of device-instances within a Frame Application. FDT does not define any interfaces for the Device Node object, since it's a pure Frame Application internal object and may have different specific implementations.

A Frame Application typically (vendor-specific) saves DTMInfo and TypeInfo information of the corresponding DTM (see 7.4) together with the DTMDataSet to be able to start the DTM Business Logic, which originally saved the data.

4.11.3 DTMDataSet structure

Figure 15 shows the structure and content of a DTMDataSet.

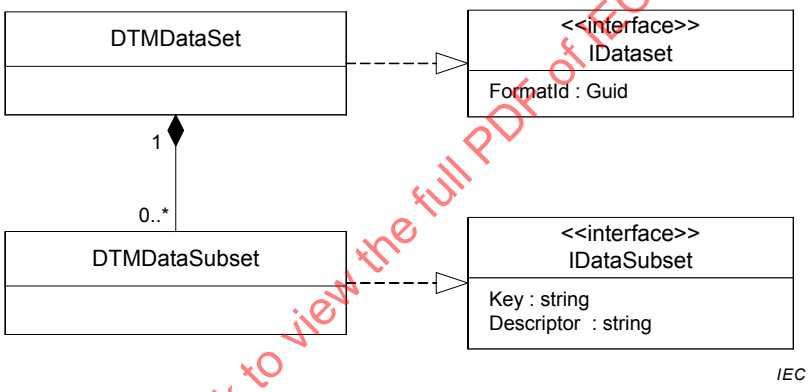


Figure 15 – DTMDataSet structure

The DTMDataSet has a property FormatId, which is a unique identifier for the format of the data. This ID is created by the device (DTM) vendor. The DTM Business Logic can use this information to decide how to load the data, e.g. to migrate the data from an older version.

A DTM always writes the DTMDataSet in one specific format, but may be able to read also other data formats. In such a case a DTM can declare to support more than one FormatId. If different DTMs declare to support the same FormatId the following scenario can be supported:

A DTM vendor can provide a scenario to migrate the data from an old version of a DTM to a newer DTM version. The new DTM version declares to support the old FormatId as well as the new FormatId. The Frame Application detects the old FormatId and creates the new DTM. The new DTM loads the DTMDataSet, migrates the data and saves the data with the new format (identified by a new FormatId).

NOTE This scenario may work for DTMs of one vendor or for DTMs from different vendors. However the definitions necessary to support such a scenario are out of scope of this specification.

A DTMDataSet can have one or more DTMDataSubsets. The DTMDataSubsets contain the persistent data of a DTM. The DTM Business Logic can explore the DTMDataSet and add or remove DTMDataSubsets to/from the DTMDataSet. The DTMDataSubsets are identified by an ID which is created by the DTM. The DTM can use the IDs to read or write the data.

Which data is stored in one DTMDataSubset is DTM-specific. The DTM should group data in one DTMDataSubset if it belongs to one functional unit and needs to be loaded together. In order to improve the system performance a DTM should avoid loading of unnecessary data whenever possible, especially at start-up of the DTM. The following grouping should be considered:

- Basic data which is needed during the complete lifetime of a DTM instance (e.g. Network Management Info)
- Device parameter group information which is needed if corresponding DTM User Interface is opened (e.g. a page in a dialog) or if the Frame Application requests data (e.g. DeviceDataInfo objects (see 7.9))
- Process data information which is needed if Frame Application requests ProcessDataInfo objects (see 7.11.1)
- etc.

The DTMDataSubset data format is DTM-specific. Any serializable datatype can be used. The Frame Application is not allowed to modify the data.

4.11.4 Types of persistent DTM data

Two types of DTM related data are considered:

- Instance-related data (called “instance data”). Instance-related data belongs to the DTM itself. It is specific for a DTM which data it stores but the DTM has to guarantee that it is able to represent the stored device instance by loading these data;
- Bulk data. DTM-specific data, for example historical data. A DTM can save bulk data as separate DTMDataSubsets in the DTMDataSet in the same way as instance-related data (each in a separate collection). Configuration data shall not be stored in the bulk data. DTM shall be prepared to be loaded without previously stored bulk data.

Instance-related data and Bulk data may be stored in separate storages in order to allow a Frame Application to distinguish instance related DTMDataSubsets and bulk DTMDataSubsets for management purposes.

4.11.5 Data synchronization

If multiple users access the same device, systems shall start several DTM instances of the same DTM type and for the same physical device (see 4.6.2). The different DTM instances access the same DTMDataSet. This is for example the case in a distributed system where multiple users access the same Frame Application (see 11.4).

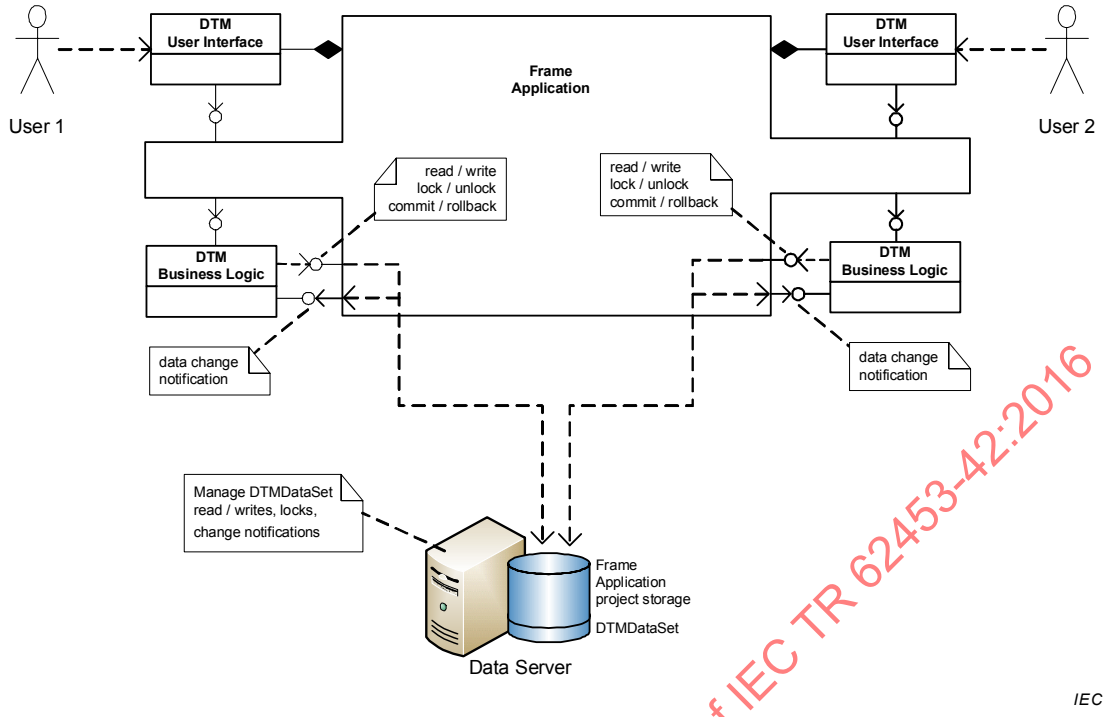


Figure 16 – Data Synchronization

To support such a scenario the Frame Application shall support interfaces which allow the realization of a dataset locking and changing notifications concept (see Figure 16).

FDT2 uses a pessimistic locking concept on DTMDataSet (device) level. The concept works as following:

– A DTM shall try to lock its DTMDataSet before execution of an operation that may lead to data changes (e.g. opening of a parameterization user interface).
  - If locking was successful, then data changes are allowed
  - If locking failed, e.g. because another DTM instance has already locked the data, then no data changes are allowed (e.g. opened user interface shall disable input fields)
– A DTM that has no lock can only read the last committed data from the DTMDataSet
– A DTM that has the lock can read and write the data in the DTMDataSubsets.
– Changes in the DTMDataSet are only visible to the DTM that holds the lock until DTM commits the changes and until the Frame Application sends TransactionCommitted to other DTMs.
– Uncommitted changes are automatically discarded if the DTM unlock the DTMDataSet
– The Frame Application notifies all other DTM instances working with the same DTMDataSet if
  - Data in a DTMDataSubset has changed and changes are committed (DTM should re-read and display the data)
  - DTMDataSet is locked or unlocked (DTM should change the state of UIs, e.g. input fields are enabled / disabled)

4.12 Implementation of access to device data and IO information

4.12.1 Exposing device data and IO information

In addition to device-specific functions and user interfaces a DTM provides access to device data and to instance data via the programming interface(see definition of IInstanceData, IDeviceData, DataInfo, Read-Write Request and Read-Write Response in Annex B).

It is recommended, that a DTM exposes all parameters, which are accessible in the user interfaces of the DTM, also by IInstanceData and IDeviceData. A DTM shall expose at least all parameters defined in applicable profiles of FDT Protocol Annexes and FDT Application Profile Annexes. If a DTM is supporting a device with a device description (e.g. EDD or EDS), parameters should be exposed with the same name and label as in the corresponding DD (for example EDD: parameter name should be the identifier of the corresponding EDD-Variable).

Device data and instance data may expose different sets of parameters. Device data may expose dynamic data like process value, device status and operating hours, whereas instance data should not expose such dynamic data.

The DTM shall adapt the list of exposed parameters according to the user level (e.g. restrict access to parameters). DTMs shall update the list of exposed parameters during runtime, for instance when parameters become inaccessible due to a changed configuration (e.g. changed measurement principle). Parameters, which are only available if the DTM is in an OEM service mode, shall not be exposed.

The DTM should expose the data in DataGroups in the same organization as presented in DTM-user interfaces. Parameters shall be exposed in a way (format and semantic information) which allows the processing of the data without fieldbus knowledge. For example, instead of raw data in hex format, a parameter shall be exposed as readable value with a numeric datatype and provide additional information like unit and range.

FDT Protocol Annex specifications may define additional requirements regarding the exposed device data.

A Frame Application may use the exposed data for various use cases, for instance for comparison. For examples see the following sections.

4.12.2 Data access control

The Frame Application can use IDeviceCustomConfiguration/IInstanceCustomConfiguration interfaces to read the description for all data and all data groups exposed by the DTM independent of the current settings of the device and independent of the mode of operations. The Frame Application can use the information to present the list of exposed data, the name, label, descriptor, read/write status and semantic information to the user and let the Administrator create custom access permissions for each user or group of users. The Frame Application can enable access to individual data using the method <EnableParameters> with a list of IDs for all parameters that shall be changeable.

Each data item is represented by an object of class AccessibleData. This class defines properties related to data access control (Table 3).

Table 3 – Description of properties related to data access control

AccessibleData Properties	Description
IsReadable	Specifies whether the value can be read from the device or DTM instance. The value may change depending on the internal business logic of the device / DTM.
IsWritable	Specifies whether the value can be written to the device or to the DTM instance. The value may change depending on the internal business logic in the device / DTM.
IsChangeEnabled	This attribute is applicable only when the DTM has been initialized with Expert user level. It specifies whether the FA has enabled changes to the data for the current user. This property controls what can be changed directly by the user through the DTM-UI or through the methods called by the Frame Application. TRUE: Allows the parameter to be changed by the FA using IDATA or by the user through the DTM User Interface FALSE: Parameter access is restricted and the value cannot be changed by the FA using IDATA or by the DTM User Interface The Frame Application has to verify that the values of IsChangeEnabled and IsWritable attributes are both set to TRUE for the parameter to be writable in the DTM.

The “IsChangeEnabled” property value is provided by the DTM and can be set only by the Frame Application. It cannot be changed by the DTM. The IsChangeEnabled property shall be set to FALSE by default by the DTM for the user with Expert user level. The value of IsChangeEnabled property shall be ignored by the DTM and by the Frame Application when the user level is Observer or Engineer.

The Frame Application can enable the change of a data item by setting the “IsChangeEnabled” flag to TRUE. Setting the IsChangeEnabled flag to TRUE is required to allow the change of the data item in the DTM/ device. The device/DTM may have additional restrictions, e.g. the data or data group may remain read only, the value of a data item may be restricted by the value of other data items, the data item may be read only in the device, etc.

When “IsChangeEnabled” attribute for a data item is FALSE, the data item cannot be modified by the Frame Application or by the user through the UI of the DTM. It is not expected that “IsChangeEnabled” attribute will change the visibility of a parameter in the user interface of the DTM, but the DTM shall present the value as read only if “IsChangeEnabled” is set to FALSE.

When a parameter value is set in the DTM BL, it may apply additional internal logic and modify the values of the related parameters even if the “IsChangeEnabled” flags for those related parameters are set to FALSE. The user will be able to see the modified values for the related parameters, but will not be able to modify their values since the “IsChangeEnabled” flag is set to FALSE. In a similar way, if the parameter value is set in the device, the device may change multiple parameter values of dependent parameters even if these parameters cannot be modified directly. This means that the “IsChangeEnabled” flag is only used to control the modification of data items by the user or by the FA, not by the DTM or the device itself.

When the FA wants to set the “IsChangeEnabled” flag to TRUE for a data group, it has to set the “IsChangeEnabled” flag to TRUE for each of the data items in the respective group. If a DTM has a user interface that shows a group of parameters, it is recommended to create the user interface in a way, which allows to control which parameter in the group is changeable and which is non-changeable. If the DTM cannot control the access to parameters of a group individually, then the entire user interface may be enabled for change for all parameters of the group if one of the parameters in the group is changeable.

Note that IData exposes the list of parameters according to the actual status or device mode. However, the DTM has to expose all parameters independent of state or device mode or role

through <GetAllDataInfo()> method. The Frame Application will provide the list of changeable parameters to the DTM by calling the <EnableParameters()> method. The DTM shall apply the "IsChangeEnabled" values set by this method to the individual data groups and data items. The settings shall be applied to all parameters, independent of the device mode.

The <EnableParameters()> methods (for instance data items and for device data items) shall be called only once in running state before any function or any other method is invoked in the DTM. Once set, the DTM shall preserve the settings for "IsChangeEnabled" flag during the lifetime of the DTM instance and shall reject any other request to <EnableParameters()>.

The DTM shall not save the value of the IsChangeEnabled flag in its instance data set. It shall initiate the flag to the default state ("IsChangeEnabled" = FALSE) any time a new DTM instance is created and initialized with Expert user level. The Frame Application shall invoke <EnableParameters()> each time a new instance of the DTM is initialized with Expert user level.

There might be device data or instance data that cannot be exposed as parameter and thus ICustomConfiguration interface cannot be used to modify the "IsChangeEnabled" property in the Expert user level. By default, the DTM is expected to create this data as non-changeable and the Frame Application will not be able to make it changeable.

4.12.3 Routed IO information

If a device (for instance a gateway device) delivers IO signals that originated from a connected device, then the IO Signal Info items of the ProcessDataInfo(see 7.11.1) returned by corresponding Gateway DTM Business Logic shall describe this relation (see Figure 17).

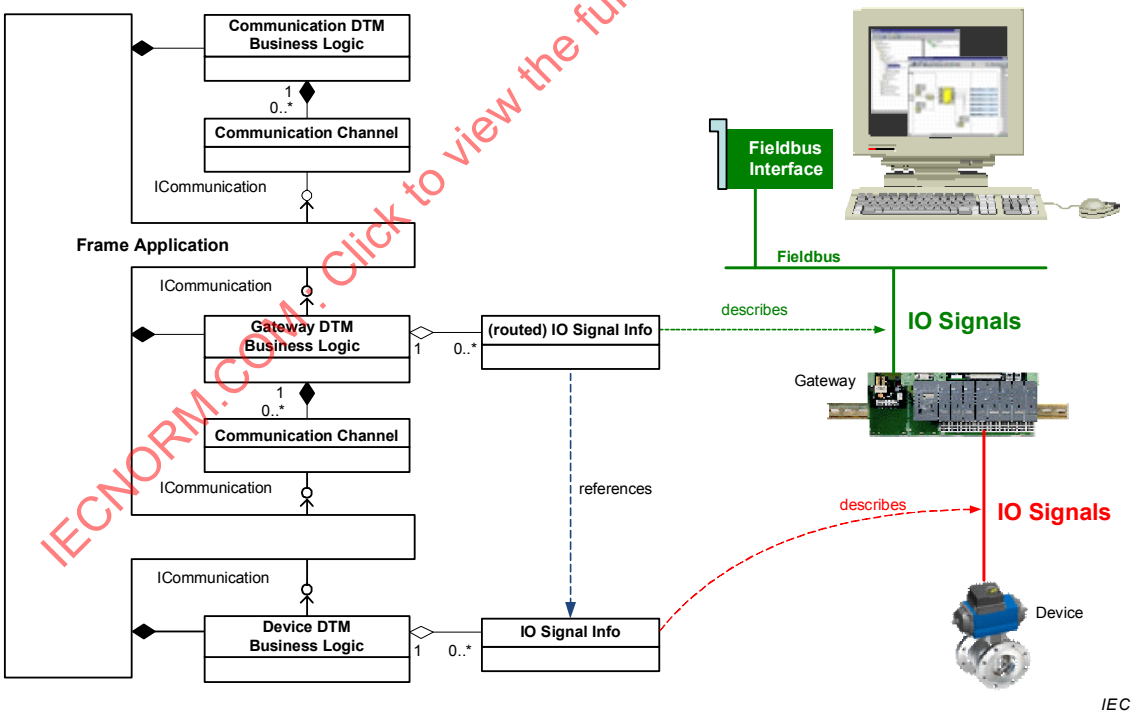


Figure 17 – Routed IO information

The IO Signal Info items of the Gateway DTM shall reference the IO Signal Info items of the Device DTM by the SystemTag of the Device DTM and the Id of the IO Signal Info.

4.12.4 Comparison of DTM and device data

FDT supports comparison of DTM and device data, for example:

- Comparison of persisted data with data in the device
- Comparison of historical data with current data
- Comparison of data from different devices

In order to support these scenarios, FDT defines two alternative comparison concepts:

a) DTM publishes all data in the corresponding interfaces. In this case the Frame Application is responsible to perform the comparison (see 5.13.1).

b) DTM provides the comparison interface. In this case the Frame Application shall call this interface for the comparison. (see 5.13.2)

4.12.5 Support for multirole devices

4.12.5.1 General

Next to Master/Slave Gateways, which are described by Gateway DTMs, Slave/Slave Gateways do exist. As these kinds of devices do not open a new type of communication, they are modeled as Device DTMs or Module DTMs which may be part of more than one logical topology (see Figure 18). Both slave roles may support the same or different bus protocols.

Furthermore, some bus protocols allow sharing of devices and/or modules between multiple masters. These shared devices are part of multiple topologies, too.

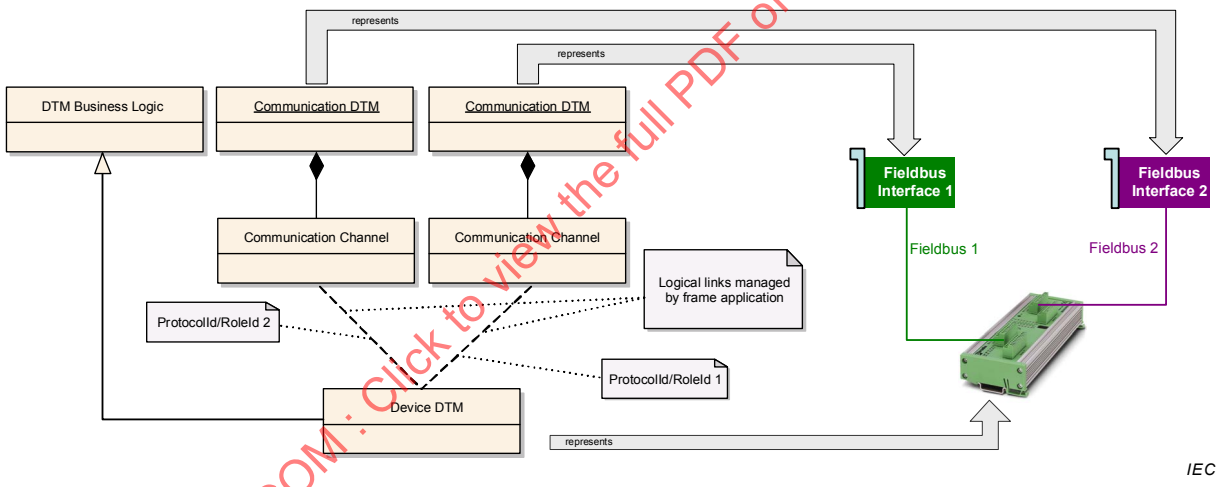


Figure 18 – Multirole Device

Different roles are assumed in different topologies, e.g. the same device may act in one topology as slave and may act as master in an other topology. In one topology only data relevant for a certain protocol or for the respective role is of interest to the Frame Application.

It is the responsibility of the Frame Application to handle the instantiation and release of one DTM in multiple topologies.

4.12.5.2 Accessing multirole related data

The support of multirole devices is optional for Frame Application and DTMs. If both support multirole devices, instead of direct access to interfaces at the DTM, role related data can be accessed by way of the IDtmRoleAccess and IDtmProxyRoleAccess interfaces (see definition in Annex B). It is in the responsibility of the DTM to provide role related data only when accessed by the IDtmRoleAccess or IDtmProxyRoleAccess interface.

If a DTM accesses Sibling DTMs, which provide role related data, the accessing DTMs should support access via role access interfaces.

4.13 Clone of DTM instances

4.13.1 General

A Frame Application may offer the functionality to copy a part of the FDT topology (i.e. multiple DTMs) e.g.for “copy and paste”.

If a part of the FDT topology is copied, then the System Tag for all cloned DTMs of the copy shall be changed by the Frame Application. Otherwise the System Tag would not be unique any more.

To create a cloned DTM instance the Frame Application shall perform following steps:

- Copy the DTMDataset to the new device node;
- Create a new DTM instance using the same DTM (unique class identifier).

Depending on the use case the Frame Application should ask the user to set the correct fieldbus address in the DTM, to set a correct TAG and to adjust DTM offline parameters before the dataset is downloaded, for example:

- device position-specific parameters like settings related to mounting related settings
- device instance-specific parameters like device calibration, linearization

A DTM shall reset cached online parameters (e.g. device serial number, operating hours etc.) and consider removing bulk data subsets when LoadData() is called with argument isCloned set true. The Frame Application applies the argument isCloned to all DTMs involved in the cloning operation.

If a Parent DTM is storing the System Tags of its children then these are invalid after the Parent DTM was cloned.

If a Parent DTM instance is cloned and has cached the System Tags of its children, then it shall rebuild its internal data structure based on the list of changed topology nodes passed to IDtm.LoadData().

4.13.2 Replicating a part of topology with Parent DTM and a subset of its Child DTMs

Cloning of a DTM with only some of its children is not supported. A Frame Application should not offer this function to the user. This restriction is to avoid inconsistencies. If a Frame Application offered this functionality, then the rules which are implemented in ISubTopology.<ValidateRemoveChild()> could not be applied.

4.13.3 Cloning of a DTM without its children

If a DTM which has children is cloned without its children, then the internal data structure used to manage children most likely is invalid. If IDtm.LoadData() has an empty list of changed topology nodes, then a Parent DTM shall release the complete set of data associated to its children (See 8.18.1 for the workflow).

4.13.4 Delayed cloning

If a Frame Application allows delayed cloning (“copy” the DTM, then make changes to the topology, then paste the DTM) then a Parent DTM is responsible for ensuring the consistency of its internal data structure used to manage children. This is done by keeping track of the topology via ISubTopology.<ValidateAddChild()> and ISubTopology.<ChildAdded()> (see 8.8.2 for the workflow).

4.14 Lifecycle concepts

Automation systems in process industry typically have a life time of 10 to 15 years or more. Over time hardware and software components in a system will be exchanged, which may require updates or upgrades of FDT related components.

The FDT2 life cycle concepts rely on identification and versioning of components which may change during the plant lifetime.

The concept defines rules to identify software and hardware components and rules to ensure backward compatibility of a component from one version to another.

The general lifecycle guidelines are described in [23]. The implementation of lifecycle concepts with IEC 62453-42 is described in Clause 10 of this document.

4.15 Audit trail

4.15.1 General

Audit trail is about recording who has accessed an automation system and what operations were performed during a given period of time. FDT defines Frame Application services which shall be used by the DTM to record operations performed on the associated device.

Frame Applications can use this information for:

- recording the information, date and time of operator entries and actions
- generating the records, e.g. for inspection and reviews
- evaluating the system

These features are for example needed for a Frame Application to comply with FDA [22] guidelines.

4.15.2 Audit trail events

A DTM shall send an Audit Trail notification to the Frame Application to record any changes in the device. DTMs shall only send notifications for changes and not for internal state transitions (e.g. the instantiation of a user interface shall not trigger Audit Trail events).

The following notifications are defined:

- Function Notification: Notifies the Frame Application that a function was called (e.g. self test functionality of the device or download was executed).

  A function notification shall indicate the start of a function and the end of a function. A notification about the end of a function shall contain the information if the function was executed successfully, cancelled or executed with a failure. A DTM shall also fire notifications for operations which are triggered by the Frame, e.g. Download parameters.

  The notifications related to functions are:

  – 'Function_name' started
  – 'Function_name' finished successfully
  – 'Function_name' finished with error 'error_reason'.
  – 'Function_name' cancelled by user

- Parameter Change Notification: Notifies the Frame Application that a parameter was changed. Contains the old value of the parameter as well as the new value.

  A DTM shall group notifications which belong to one logical operation (changes set) into one single notification. This means that there shall be e.g. one single notification for the complete set of parameters which are part of a download.

The notification related to parameter change is:

– Parameter ‘Parameter_name’ changed from ‘old_value’ to ‘new_value’.

If the DTM supports different cultures and languages (see 5.8), then the Audit Trail notifications also shall be localized.

It is up to Frame Application to request an additional comment from the user e.g. to document the reason of a performed action. The Frame Application may request this comment when an operation is started on a DTM, for example

- Upload/Download,
- DTM functions are started, or
- DTM User Interface is started.

The Frame Application shall not disturb the user interaction with the DTM by requesting a comment during execution of the DTM action. If a comment is needed the user should be asked for the comment before the DTM action is started or after the action is finished.

The DTM does not need to provide any Audit Trail Information for the changes in the list of changeable parameters. The Frame Application may provide the notifications for the modifications in the list of changeable parameters without invoking the DTM.

5 Technical concepts

5.1 General

FDT Objects shall be build upon the Microsoft .NET Framework[8] and executed in the .NET Common Language Runtime (see 5.2).

The services, specified in the IEC 62453-2 specification [3], [4], are modeled as .NET interfaces passing .NET datatype arguments (see chapter 7). These interfaces and datatypes are used for FDT Object interaction and data exchange. In addition, .NET exception classes (see 5.6.9.4) are defined for returning error information in an interface method call.

The FDT .NET interfaces, argument datatypes, and exception classes are defined in three different .NET assemblies (FDT core assemblies). Figure 19 shows the assemblies and their dependencies to each other.

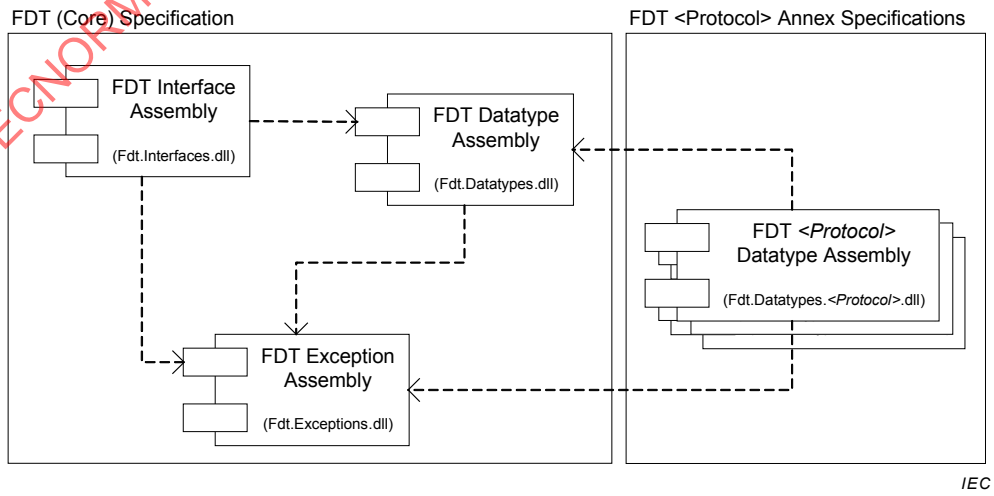


Figure 19 – FDT .NET Assemblies

The assemblies are provided together with this specification and shall be used for the development of Frame Applications and DTMs.

Some of the interface methods have to exchange protocol specific information. These methods work with abstract base classes defined in the FDT Datatype assembly (e.g.: communication interface, see ICommunication interface definition in Annex B). Protocol-specific classes defining the protocol-specific data to be exchanged are derived from these base classes. These classes are defined in separated .NET assemblies, which are provided together with corresponding FDT Protocol Annex specifications or by the DTM vendor in case of a vendor-specific protocol.

All FDT assemblies (FDT core assemblies and FDT protocol assemblies) are strong named (see 5.16.2) and additionally signed with an authenticode key (see 5.16.3) owned by FDT Group, installed into the Windows Global Assembly Cache (see 9.3 and 9.4), and shared between the different FDT Objects.

The DTM Business Logic, Communication Channels and User Interfaces shall be realized by classes and controls implemented in separate .NET assemblies (see Figure 20), which are installed and registered by the DTM setup (see 9.5).

In order to increase performance in loading the GUI, it is recommended to provide the different DTM User Interfaces in different assemblies.

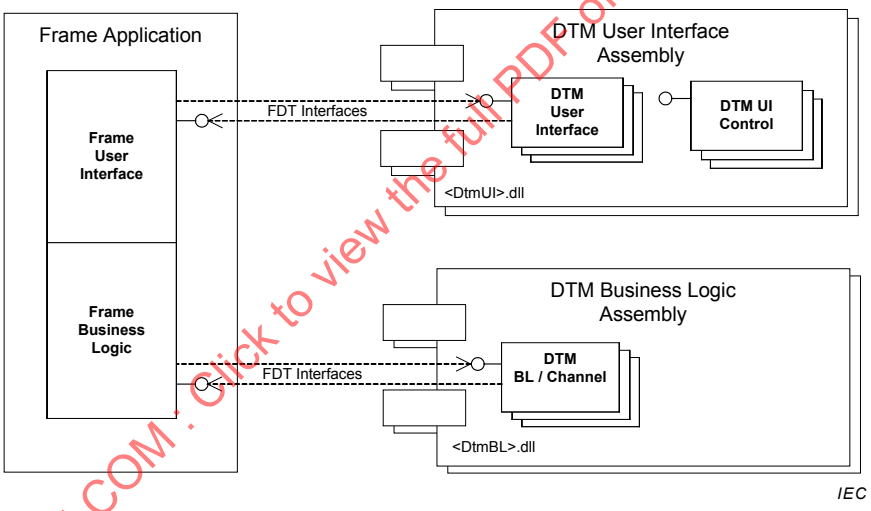


Figure 20 – FDT Object implementation

The DTM-specific assemblies shall be signed with a vendor-specific key.

The DTM Business Logic and Communication Channels shall be simple .NET classes implementing the interfaces defined in the FDT Interface assembly (Fdt.Interfaces.dll).

The implementation of the DTM User Interfaces depends on the type (see 5.10). User Interface controls which can be embedded into the Frame Application User Interface shall be implemented as pairs of two objects, a DTM UI class (.NET class) and a DTM UI control (.NET WinForms control or Windows Presentation Foundation control). A User Interface which cannot be embedded shall be implemented as DTM UI class (.NET classes), which handles the interaction between the actual DTM User Interface (i.e. an external application) and the Frame Application.

The DTM Business Logic and User Interface classes / controls shall be “creatable”:

- marked as public (non abstract)

- provide a public default constructor with no arguments

The implementation of the Frame Applications is not in scope of FDT. FDT only defines the interfaces which shall be provided to the DTM Business Logic and User Interface for callbacks.

5.2 Support of .NET Common Language Runtime versions

5.2.1 General

Specific .NET CLR (Common Language Runtime) versions are released for execution of software components built with specific .NET Framework versions. The .NET CLR version 4 is for example used to execute software components built with .NET Framework 4

Different .NET CLR versions are not fully compatible. That means software components built with a specific .NET Framework version may not execute correctly in a different CLR version. For example a .NET Framework 3.5 software component may not execute correctly in the .NET CLR 4.

FDT Group defines the .NET CLR versions which shall be supported by FDT Software.

This version of the FDT standard supports the CLR version as shown in Table 4.

Table 4 – Supported CLR versions

Supported CLR version
CLR4.0

In future, FDT Group may define support for additional CLR versions. That is why this document describes support for multiple CLR versions.

The use of other CLR versions is not allowed until the standard FDT .NET assemblies (see 5.1) are released for these versions. To enforce this rule, the standard datatype classes throw exceptions if executed in an unsupported CLR.

5.2.2 Rules for FDT .NET assemblies

In order to support interoperability with FDT 1.2.x (see [31]) the FDT .NET Assemblies and the FDT protocol assemblies are compiled for CLR2. They work in both CLR2 and CLR4. They are compiled for the “any” platform in order to support 32bit and 64bit target platforms.

5.2.3 DTM rules

A DTM shall support at least one of the CLR versions listed in Table 4. The supported version(s) shall be exposed in the DTM manifest (see 7.6.2). Support in this context means that the DTM vendor guarantees the correct function of the DTM in this CLR (e.g. verified by tests).

5.2.4 Frame Application rules

A Frame Application shall support all CLR versions listed in Table 4. This also means that the Frame Application is responsible for installation of the supported CLR runtime versions. The Frame Application shall check the CLR versions supported by the DTM before a DTM Business Logic or DTM User Interface is started. If the same CLR version which is used by the Frame Application is supported, then the DTM Business Logic and the DTM User Interface may be loaded and executed directly in the Frame Application main process. Otherwise the Frame Application shall execute the DTM in a separate process with corresponding CLR version loaded.

A Frame Application shall support an extension concept for the support of further CLR versions, for example for versions released after the Frame Application has been developed. For an explanation of the concept see 5.2.5.

In order to support backward compatibility, a Frame Application may need to support CLR2.

5.2.5 FDT CLR extension concept

This section describes a concept for CLR extension support for Frame Applications.

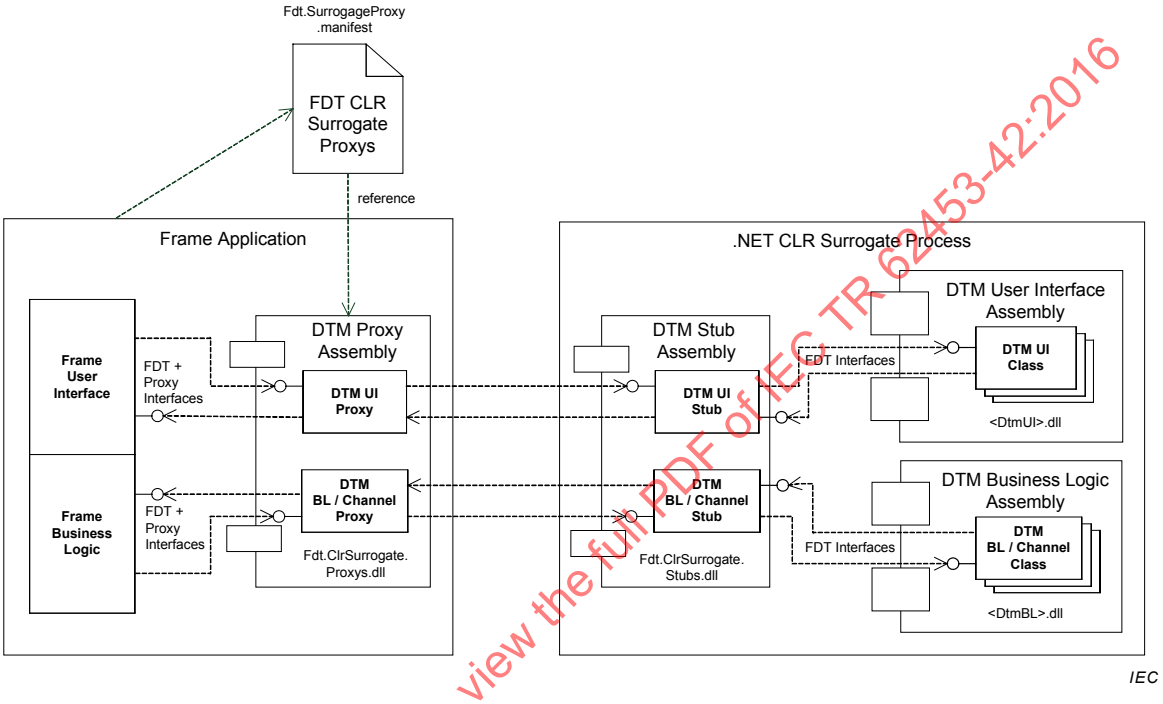


Figure 21 – FDT CLR extension concept

If a Frame Application detects that a DTM Business Logic or a DTM User Interface supports only CLR versions not supported by the main process of the Frame Application, then the Frame Application utilizes the proxy classes supporting the CLR version which is used by the Frame Application itself (see Figure 21).

The Frame Application loads the proxy .NET assembly, creates an instance of the proxy class, and delegates the execution of the DTM Business Logic or DTM User Interface to this proxy. The proxy starts a process with the required CLR (surrogate process) and executes the DTM Business Logic or DTM User Interface in this process. The proxy classes provide the standard FDT interfaces. The Frame Application can use these interfaces to interact with the DTM Business Logic or with the DTM User Interface executed in the surrogate process.

5.3 Support for 32-bit and 64-bit target platforms

DTMs should support 32-bit operating system as well as 64-bit. This means that they should be compiled using the “any”- platform target.

If it is not possible to support both platforms, then a DTM shall support at least one of the platform targets. For instance, if dependent dlls are not available as 64-bit target, then a DTM may be available in 32-bit only.

A DTM shall expose the information whether it supports 32-bit, 64-bit or both target platforms in the corresponding setup manifest (see 9.6.2).

64-bit Frame Applications shall support 32-bit DTMs (e.g. if the DTMs do not have a 64-bit variant). This can be implemented, for instance, by using a surrogate process.

5.4 Object activation and deactivation

5.4.1 General

A Frame Application needs to find and load the DTM-specific .NET assemblies dynamically into the memory and execute the contained DTM Business Logic and User Interface classes by calling corresponding FDT interfaces. Finally, the created objects need to be destroyed and unloaded from the memory.

This chapter describes the means which shall be utilized for object activation and deactivation and the corresponding rules that shall be followed by Frame Applications and by DTMs.

5.4.2 Assembly loading and object creation

The DTM-specific .NET assemblies are installed by the DTM setup. The setup also registers the DTM by installing “DTM manifest” file(s) in an FDT-defined directory (see 9.5). A manifest file contains the information where to find the .NET assemblies and which DTM classes and/or UI controls are contained (see 7.6.1 and 7.6.2).

The Frame Application shall use this information for loading and execution of the DTM classes and UI controls by using the LoadFrom mechanism. The .NET Framework provides following classes / methods for this purpose:

- Assembly.LoadFrom() and .CreateInstance() (namespace System.Reflection)
- AppDomain.CreateInstanceFrom() (namespace System)

Figure 22 outlines the use of the methods provided by the Assembly class as an example:

```
public object CreateFdtObject(string filePath, string fullClassName)
{
    Assembly assembly = Assembly.LoadFrom(filePath);
    return assembly.CreateInstance(fullClassName);
}
```

IEC

Figure 22 – Example: Assembly.LoadFrom()

NOTE The method System.Reflection.Assembly.LoadFrom() behaves as following:

1. LoadFrom() loads the assembly addressed with the file path and also the referenced assemblies in same directory.
2. If an assembly is loaded with LoadFrom(), and later an assembly in the “load context” attempts to load the same assembly by display name, then this load attempt fails.
3. If an assembly with the same identity is already loaded (e.g.: by another DTM), then LoadFrom returns the loaded assembly, even if a different file path was specified.
4. If an assembly is loaded with LoadFrom(), and the probing path includes an assembly with the same identity (e.g.: in Global Assembly Cache, application directory), then this assembly is loaded, even if a different file path was specified.
5. LoadFrom() requires the permissions FileIOPermissionAccess.Read and FileIOPermissionAccess.PathDiscovery, or WebPermission, on the specified path.
6. If a native assembly image (generated by ngen.exe) exists for the specified file path, then it is not used. The assembly cannot be loaded as domain neutral (assembly cannot be shared between ApplicationDomain, each loads its own copy).

Because of this behavior FDT defines the following rules:

a) Rules regarding assembly dependencies (see 5.4.3)

b) Only LoadFrom shall be used in the context of FDT. The use of other .NET

c) assembly loading / object creation means is not allowed.

d) Rules regarding shared assemblies (see 5.4.4).

e) DTM assemblies shall be installed to a path which is browseable and readable.

f) DTM assemblies shall not be precompiled using ngen.exe (or similar tools).

The security aspects regarding loading and execution of assemblies are described in chapter 5.16.

The next steps after creation depend on the object type:

- Steps for the DTM Business Logic: 6.3.2.
- Steps for the DTM User Interfaces: 5.10.

5.4.3 Assembly dependencies

5.4.3.1 Introduction

DTM-specific .NET assemblies may depend on other .NET assemblies, for example on a device vendor-specific library or on a 3rd party library as outlined in Figure 23.

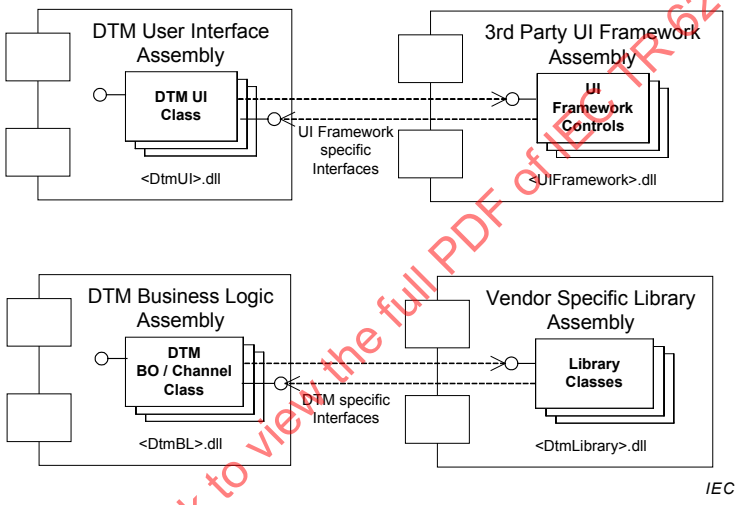


Figure 23 – Example: Assembly dependencies

These dependencies and the interaction between the classes / controls contained in the assemblies are DTM-specific, but the DTMs have to follow some rules in order to function correctly and to avoid problems in conjunction with other DTMs executed in a Frame Application.

5.4.3.2 Loading of dependent assemblies

The Frame Application loads the .NET assemblies – containing the DTM main class / control – by calling the .NET Framework LoadFrom() method (see 5.4.2).

Referenced assemblies which are stored in the same directory or in the GAC are automatically loaded together with this .NET assembly.

Referenced assemblies which are stored in other locations (e.g. in a sub-directory) have to be loaded specifically by the DTM. The DTM shall load such assemblies also by using the LoadFrom() method provided by the .NET Framework. Loading assemblies with other.NET Framework methods is not allowed (see 5.4.2).

5.4.4 Shared assemblies

Special attention is necessary for assemblies which are shared. Shared in this context means that an assembly with the same identity is used by another software on the computer (see

LoadFrom() behavior description 3.and 4. In the Note to 5.4.2). This applies to assemblies containing DTM BL, DTM UI as well as all other used assemblies.

NOTE 1 The identity of strong named assemblies consists of the assembly simple name, version, culture and public key token.

NOTE 2 The behavior described here applies to all shared assemblies independent of the location of the assembly.

If a shared assembly is used, then following rules apply:

a) Any incompatible change to the shared assembly shall lead to a new identity (e.g. different version number).

b) Shared assemblies shall not presume to be loaded from a specific installation path (e.g. rely that some files are stored in the same directory or in a sub-directory).

c) Static variables in shared assemblies are also shared if the assembly is loaded into the same ApplicationDomain. Thus static variables shall not have side effects in such scenarios. It's strongly recommended not to use static variables in a shared assembly.

If the rules above cannot be ensured by a DTM vendor, then the assembly shall not be used as a shared assembly. That means either the assembly gets a DTM-specific identity or it shall not be used at all.

5.4.5 Object deactivation and unloading

5.4.5.1 Introduction

Destroying of DTM Business Logic and User Interfaces and unloading of corresponding .NET assemblies have to be considered separately.

5.4.5.2 Destroying of objects

Steps to destroy an object depend on the object type. The procedure for the DTM Business Logic is defined in 6.3.2. The procedure for the DTM User Interfaces is defined in 8.5.6 and the following sections.

For all object types providing the interface IDisposable, the method IDisposable.Dispose() shall be called by the Frame Application at the instance. This call shall be used to free all used resources (e.g. close opened files, stop running threads) and release the references to other objects (set to null). The instance is not destroyed. This happens sometimes later by the NET garbage collector.

5.4.5.3 Unloading of assemblies

A .NET assembly which is loaded into a process respectively into an ApplicationDomain is never unloaded, except if the ApplicationDomain is destroyed. That means if the Frame Application loads a DTM-specific assembly into the default ApplicationDomain, then these assembly and all dependent assemblies are never unloaded unless the application is closed.

The DTM assemblies shall be developed with this .NET Framework behavior in mind. To reduce the memory consumption it's recommended

- to minimize the use of static variable, because these increase the memory consumption of the assembly.
- to move DTM functionality which is not always (or rarely) needed to separate assemblies. These assemblies are loaded only (automatically or manually) (see 5.4.2) if corresponding code is executed.
- to use shared assemblies whenever possible (see 5.4.4).

Frame Application should consider the execution of .NET assemblies in a separate ApplicationDomain in order to have the ability to unload them.

5.5 Datatypes

5.5.1 General

.NET data classes (datatypes) are used for the data exchange between the different FDT objects. Instances of these classes are passed as arguments in the FDT interface methods, properties, and events.

The datatypes are defined in the .NET assembly Fdt.Datatypes.dll, which is distributed together with this specification document. This assembly shall be used for the development of FDT Objects.

The datatypes are designed as so called “Data Contract” classes. These are classes using the attributes defined in the .NET Framework System.Runtime.Serialization namespace. The actual data is provided by properties with corresponding [DataMember] attribute set as shown in Figure 24.

```
using System.Runtime.Serialization;
using Fdt;

/// <summary>
/// Description of SomeDatatype
/// </summary>
[DataContract]
public class SomeDatatype : FdtDatatype<SomeDatatype>
{
    /// <summary>
    /// Description of data provided by the property
    /// </summary>
    [DataMember(IsRequired = true)]
    public string DataProperty1 { get; set; }

    /// <summary>
    /// [Optional] Description of data provided by the property
    /// </summary>
    [DataMember(IsRequired = false)]
    public FdtList<SomeSubDatatype> DataProperty2 { get; set; }

}
```

IEC

Figure 24 – Example: Datatype definition

The attributes control the serialization / deserialization of the instances (see 5.5.2) and also defines which properties are mandatory and optional (see 5.5.4).

All data classes are directly or indirectly derived from the base class FdtDatatype (see 7.1), which provides methods to verify (see 5.5.5) or clone instances (see 5.5.6).

5.5.2 Serialization / deserialization

The data classes support serialization / deserialization of data in different formats over the DataContractSerializer class provided in the .NET Framework System.Runtime.Serialization namespace (e.g. binary format and XML) [9]. This may for example be used by the Frame Application to transport the data classes in WCF interfaces (Windows Communication Foundation) or for remote interaction in a network, but such use cases are out of scope of this specification.

5.5.3 Support of XML

FDT defines W3C compliant XML Schemas defining the format for XML serialization / deserialization. The name of the Schema is defined in the [DataContract] attribute assigned to the corresponding data class (see Figure 24). This may for example be used by the Frame Application to exchange device related data with other applications running in a non Windows operating system, but this is out of scope of FDT.

NOTE The interaction between FDT Objects is based on .NET datatypes (see 5.5.1) and is not based on XML.

5.5.4 Optional elements

Properties with [DataMember(IsRequired = true)] attribute assigned are mandatory (see DataProperty1 in Figure 24). That means they shall not be set to null (or string.Empty) if the instance is passed over an FDT interface.

Properties with [DataMember(IsRequired = false)] attribute assigned are optional (see DataProperty2 in Figure 24). That means they may be set to null if the instance is passed over an FDT interface.

For better distinction optional properties are marked with an "[Optional]..." comment. Additionally, all data classes provide a constructor for mandatory properties, which may be used to initialize a class instance with required data. The use of this constructor is optional. The mandatory properties can also be set later, but before the instance is passed over an FDT interface.

5.5.5 Verify

All data classes provide a Verify() method, which checks the rules defined for the data. Dependent on the class this may only be the basic mandatory / optional rules or additional rules defined in the data class description.

The FDT object receiving data from another object may use this method to check whether data is valid. However, the use of this method is optional. The receiving object may use other means to handle invalid data (e.g. check used properties whether they are null manually) or provide a specific mode which allows to switch verification on/off. This may be a good option to reach maximum performance during runtime, but to provide a fallback strategy for trouble-shooting.

5.5.6 Clone

All data classes provide a Clone() method, which creates a new object that is a deep-copy of the called instance. That means all objects are duplicated – the top-level objects in the properties provided by the data class itself, as well as all lower level objects in properties of the sub-classes.

The cloning of data class instances is mandatory if an FDT object class member variable is passed over an FDT interface as argument or return value. This rule applies to methods, events and properties of interfaces.

This is necessary because of two reasons:

a) The receiving object may change the property values of received data instance. This would also affect internal data if only a reference is passed.

b) The receiving object may keep a reference to the received data instance. Further changes to the original data instance after the call returned may lead to unexpected results and threading issues.

If references are passed (e.g. interface reference or AsyncResult objects), no cloning shall be used.

Figure 25 shows two examples where cloning is necessary.

```
public class MyDtm
{
    private SomeDatatype _someData = new SomeDatatype(/* init with data */);

    public SomeDatatype DoSomething1()
    {
        return _someData.Clone();
    }

    public void DoSomething2()
    {
        MyOtherObject anotherObj = new MyOtherObject();
        anotherObj.DoSomethingWithMyData(_someData.Clone());
    }
}
```

IEC

Figure 25 – Example: Data cloning

If data class instances are created each time a method is called and no internal instances are referenced, then passing of instance references is allowed as shown in Figure 26.

```
public class MyDtm
{
    public SomeDatatype DoSomething1()
    {
        SomeDatatype someData = new SomeDatatype(/* init with data */);

        return someData;
    }

    public SomeDatatype DoSomething2()
    {
        SomeDatatype someData = new SomeDatatype(/* init with data */);

        AnotherObject anotherObj = new AnotherObject();
        anotherObj.DoSomethingWithMyData(someData);

        return someData;
    }
}
```

IEC

Figure 26 – Example: Methods without data cloning

5.5.7 Equals

The Equals() method compares the identity of objects, it can not be used to compare the contents of different objects.

In order to compare the contents of objects, developers need to implement the comparison.

5.5.8 Lists

The generic class FdtList<> is used for listing of data class instances (see 7.1). Like FdtDatatype this class provides methods to verify or clone the content of the FdtList<> instances itself and all contained elements.

If an FdtList is passed over an FDT interface, then the instance shall never be empty. If the corresponding property is optional, then the property shall be set to null instead.

5.5.9 Nullable

Nullable represents an object whose underlying type is a value type to which also ‘null’ can be assigned. (like a reference type)

5.5.10 Enumeration

Enumeration is a distinct type consisting of a set of named constants.

5.5.11 Protocol-specific datatypes

5.5.11.1 General

Protocol-specific datatypes shall be defined in .NET assemblies which are provided either together with the corresponding FDT Protocol Annex specifications or by DTM vendors in case of vendor-specific protocols.

The protocol-specific assemblies shall contain datatypes derived from the corresponding base classes in the FDT Datatype assembly (see Figure 27).

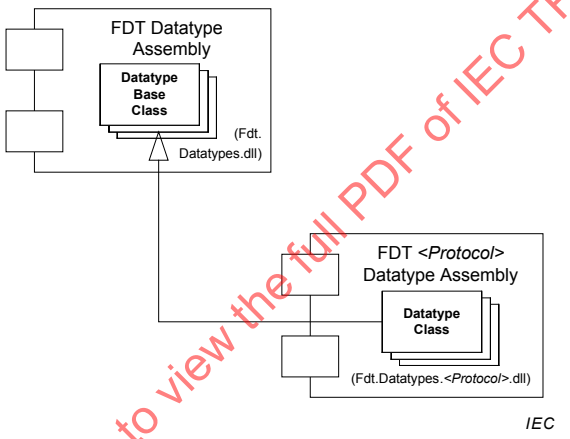


Figure 27 – Protocol-specific datatypes

Some of the FDT interface methods exchange protocol-specific information. These methods are defined with the protocol neutral base classes.

Protocol-specific assemblies shall support 32-bit platforms as well as 64-bit platforms. This means they shall be built using the “any”- platform target.

5.5.11.2 Interaction DTM – Frame Application

Typically, the DTMs create instances from the protocol-specific classes and pass them to the Frame Application over an FDT interface. The Frame Application then works with the properties and methods in the base classes. Thus, the Frame Application is able to handle any DTM independent of the protocol. Subclause 7.5 provides examples for using the DeviceIdentInfo classes with protocol neutral data and protocol-specific data.

5.5.11.3 Interaction DTM – DTM

If protocol-specific datatypes are used for a DTM to DTM interaction, then one DTM typically creates an instance of the protocol-specific classes and passes it over the corresponding FDT interface. The DTM which receives the data then casts the reference back from the base class to the protocol-specific datatype. Subclause 7.7 contains examples for using the protocol-specific Communication classes.

5.5.11.4 Installation and registration

The protocol-specific .NET assemblies shall be installed and registered by the DTMs using the protocol (see 9.4).

The protocol-specific .NET assemblies are installed in the Windows Global Assembly Cache. The DTM-specific assemblies can use static references in order to load the protocol assembly automatically together with itself (see 5.4).

In some cases the Frame Application also needs to load the protocol-specific assemblies and create instances from the contained classes, e.g. for deserialization of a protocol-specific datatype. In order to support such scenarios the protocol-specific assemblies shall be registered with a corresponding manifest file (see 9.4.3). The Frame Application can evaluate the provided information and then load corresponding assembly specifically (see Figure 28).

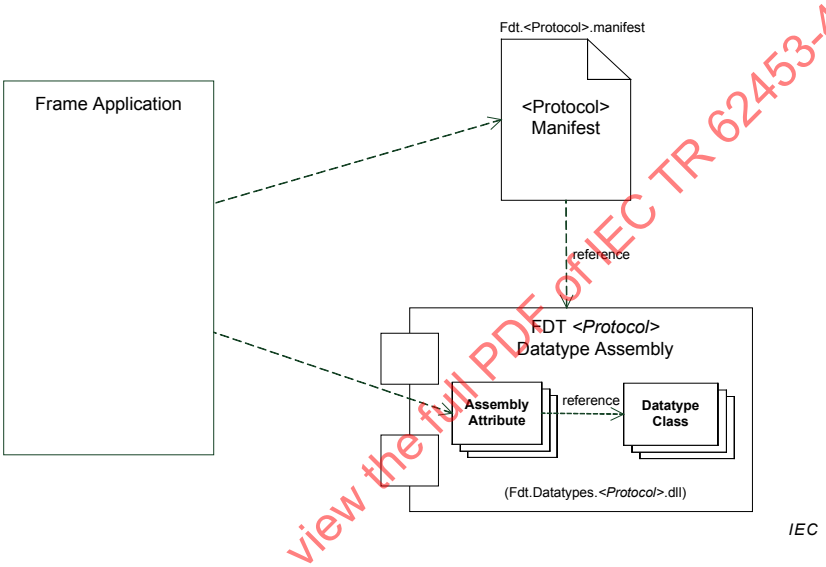


Figure 28 – Protocol manifest and type info attributes

In addition, the assembly shall expose type information as attributes assigned to the assembly itself. The Frame Application can use this information to create instances from the protocol-specific classes. The attribute classes are defined in the FDT Datatype assembly.

Following attributes (and corresponding datatypes) shall be supported by a protocol-specific assembly:

- ProtocolInfo attribute(see 7.3)
- DeviceIdentInfoType attribute(see 7.5)
- CommunicationType attribute (see 7.7)
- IOSignalInfoType attribute (see 7.11.1)
- DeviceAddressType (see 7.12)
- NetworkDataType attribute (see 7.13)

The example in Figure 29 shows the attributes assigned to the HART-specific datatype assembly (Fdt.Datatypes.Hart.dll).

```
[assembly: ProtocolInfoAttribute(ProtocolId = Hart.ProtocolId, ProtocolName =
Hart.ProtocolName)]

[assembly: DeviceIdentInfoType(
                        DeviceIdentInfoType = typeof(DeviceIdentInfo<HartDeviceIdentInfo>),
                        ProtocolDeviceIdentInfoType = typeof(HartDeviceIdentInfo),
                        DeviceScanInfoType = typeof (DeviceScanInfo<HartDeviceScanInfo>),
                        ProtocolDeviceScanInfoType = typeof(HartDeviceScanInfo))
]

[assembly: CommunicationType(AbortMessageType = typeof(HartAbortMessage),
                             ConnectRequestType = typeof(HartConnectRequest),
                             ConnectResponseType = typeof(HartConnectResponse),
                             DisconnectRequestType = typeof(HartDisconnectRequest),
                             DisconnectResponseType = typeof(HartDisconnectResponse),
                             SubscribeRequestType = typeof(HartSubscribeRequest),
                             SubscribeResponseType = typeof(HartSubscribeResponse),
                             UnsubscribeRequestType = typeof(HartUnsubscribeRequest),
                             UnsubscribeResponseType = typeof(HartUnsubscribeResponse))
]

[assembly: IOSignalInfoType(IOSignalInfoType = typeof(IOSignalInfo<HartIOSignalInfo>),
                            ProtocolIOSignalInfoType = typeof(HartIOSignalInfo))
]
```

IEC

Figure 29 – Example: Protocol assembly attributes

Protocol-specific datatypes shall support the serialization/deserialization mechanisms as defined in section 5.5.2. The example in Figure 30 shows how the Frame Application can load a protocol-specific assembly and create an instance of a datatype class by using the DataContractSerializer.

```
public DeviceIdentInfo DeserializeDeviceIdentInfo(ProtocolManifest manifest, Stream stream)
{
    string longName = manifest.AssemblyInfo.Name + ", " +
                  "Version=" + manifest.AssemblyInfo.Version + ", " +
                  "PublicKeyToken=" + manifest.AssemblyInfo.PublicKeyToken;
    Assembly assembly = Assembly.LoadFrom(longName);

    Type attributeType = typeof(DeviceIdentInfoTypeAttribute);
    DeviceIdentInfoTypeAttribute deviceIdentAttrib =
        (DeviceIdentInfoTypeAttribute)assembly.GetCustomAttributes(attributeType, false)[0];

    DataContractSerializer serializer =
        new DataContractSerializer(deviceIdentAttrib.DeviceIdentInfoType);
    return  (DeviceIdentInfo)serializer.ReadObject(stream);
}
```

IEC

Figure 30 – Example: Handling of protocol-specific assemblies in Frame Application

5.5.12 Custom datatypes

The FDT datatypes are not intended for customization, because they are used in cooperation of software from different parties. That is why most FDT-datatypes are sealed (protected against changes/inheritance).

The only datatypes that can be extended are the base classes for protocol-specific datatypes and for UI-messaging datatypes. If extending such datatypes, following rules shall be applied:

– Use the [DataMember] attribute for all newly declared class members.

– All class members must have serializable type. (I.e. it is not allowed to use reference types, for instances interfaces.)

Protocol-specific datatypes (as described in 5.5.11) also shall be sealed.

5.6 General object interaction

5.6.1 General

All FDT Objects interact with each other exclusively via the interfaces defined by this specification. These interfaces are defined according to the services specified in IEC 62453-2 [4].

The interfaces define properties and methods of the server object as well as events, that may be received by the client object. In order for a client object to receive those events, the client object has to register delegates for these events. If not explicitly defined otherwise for an interface it is optional for the client object to register for the events of an interface.

5.6.2 Decoupling of FDT Objects

IEC 62453-42 decouples the FDT Objects from each other. The Frame Application is the one and only component that directly interacts with the DTM Business Logic, User Interfaces and Communication Channels via the IEC TR 62453-42 interfaces corresponding to the services defined for the objects in IEC 62453-2 [4].

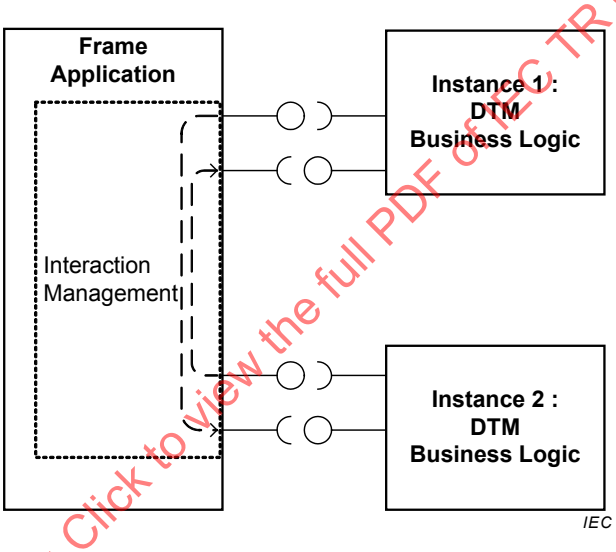


Figure 31 – Decoupled FDT Objects in IEC 62453-42

All component interactions are passed through the Frame Application or proxy components (see Figure 31). The Frame Application shall not change interactions or inject interaction requests.

This addresses the following objectives:

a) Interoperability
The decoupling of the FDT objects by the Frame Application shall improve interoperability.

b) Tracing
The Frame Application is able to observe the complete interaction between FDT objects. Thus it can implement a system wide tracing which is useful for diagnosis and trouble-shooting.

c) Testing
Each component shall also be testable in a component test environment. The test support can be achieved by tracing, replaying recorded sequences or by error injection. In the testing use case the Frame Application is allowed to change interactions.

d) Threading / Synchronization
The Frame Application is responsible for the assignment of FDT Objects to processes. An FDT Object shall not expect to be executed in the same thread, process or host like other

related FDT Objects. The Frame Application can enforce rules in regard to method calls, whereas the rules may differ between the different FDT Objects (see 5.7).

e) Remoting
The Frame Application can pass the messages to a different process or a remote computer (see 11.3)

An example for components, which are used for decoupling of FDT Objects, are proxy objects (e.g. DTM UI-proxy or channel proxy) that are used to interact with the respective FDT Object.

NOTE The Frame Application part handling the interaction between the different FDT objects – called Interaction Management in Figure 31 – could be separated from the actual Frame Application implementation. It could be realized as shared component, which is then used by different vendors. This would reduce the implementation effort for the different Frame Application vendors and increase the interoperability with DTM.

5.6.3 Parameter interchange with .NET datatypes

The arguments of interface methods are defined as .NET datatypes. The definition of these .NET datatypes includes:

- Type definition (e.g. definition as .NET class/structure)
- Definition of standard methods for
  - Serialization to/from XML
  - Serialization to/from binary stream
  - Verification

The XML format and the format for the binary stream are well defined formats, specified in this technical report. The XML format is based on W3C schemas and may be used for backward compatibility to FDT1.2.x and for interaction with external applications

In order to ensure interoperability for FDT components, the .NET interfaces and datatypes specified by this technical report are implemented in primary assemblies, which are provided by FDT Group. It is mandatory for all FDT components to use this primary assembly.

5.6.4 Interaction patterns

In this technical report, the following interaction patterns are used:

– Properties
– Synchronous methods
– Asynchronous methods
– Events

These patterns and their usage is explained in the following sections.

5.6.5 Properties

Properties are used for simple get or set operations on simple data objects that are performed synchronously. Other interfaces of an FDT object are also provided by properties.

5.6.6 Synchronous methods

Synchronous methods are used for simple operations that can be performed synchronously within the calling thread. The called object shall not block the calling thread, e.g. by waiting on asynchronous operations to finish or waiting on events.

Examples for synchronous methods are:

– Information Requests (e.g. IDtmInformation.GetDeviceIdentInfo())

– Simple state machine operations (e.g. IDtm.Init(), IDtm.EnableCommunication())
– Frame Application calls that do not require nested calls (e.g. ITopology.GetParentNodes())

5.6.7 Asynchronous methods

5.6.7.1 Introduction

Asynchronous methods are typically used to perform operations that may take a relatively long time to complete, such as I/O or database operations, communication requests. Such an asynchronous operation executes in a separate thread. When an application starts an asynchronous operation, the application can continue execution while the asynchronous operation is performed. Asynchronous methods are implemented using the IAsyncResult pattern.

5.6.7.2 IAsyncResult pattern

The IAsyncResult pattern as defined in [14] is used for asynchronous calls to services.

Using this pattern an asynchronous operation is implemented as a set of methods:

– The Begin*OperationName*() method starts the asynchronous operation OperationName. The BeginOperationName method shall return control to the calling thread immediately. If the BeginOperationName method throws exceptions, the exceptions are thrown before the asynchronous operation is started and the *OperationName*Completed() callback method is not invoked.
– The End*OperationName*() method ends the asynchronous operation OperationName and retrieves the results of the operation. If the operation has not completed when EndOperationName is called, End*OperationName* blocks until the operation is finished. Exceptions which occurred during the asynchronous operation are thrown from the EndOperationName method.
– The Callback delegate *OperationName*Completed() (implemented by the client) is provided only for a specific service call that triggers one specific event type that can be received.

For further information on how to implement the IAsyncResult pattern see F.1.

One of the advantages of the IAsyncResult-Pattern is, that the client may choose to use the service in a blocking or in a non-blocking way.

If a service is used in a blocking way, the client calls the Begin() method and immediately the End() method (see Figure 32). The calling thread of the client will be blocked, until the service execution is finished.

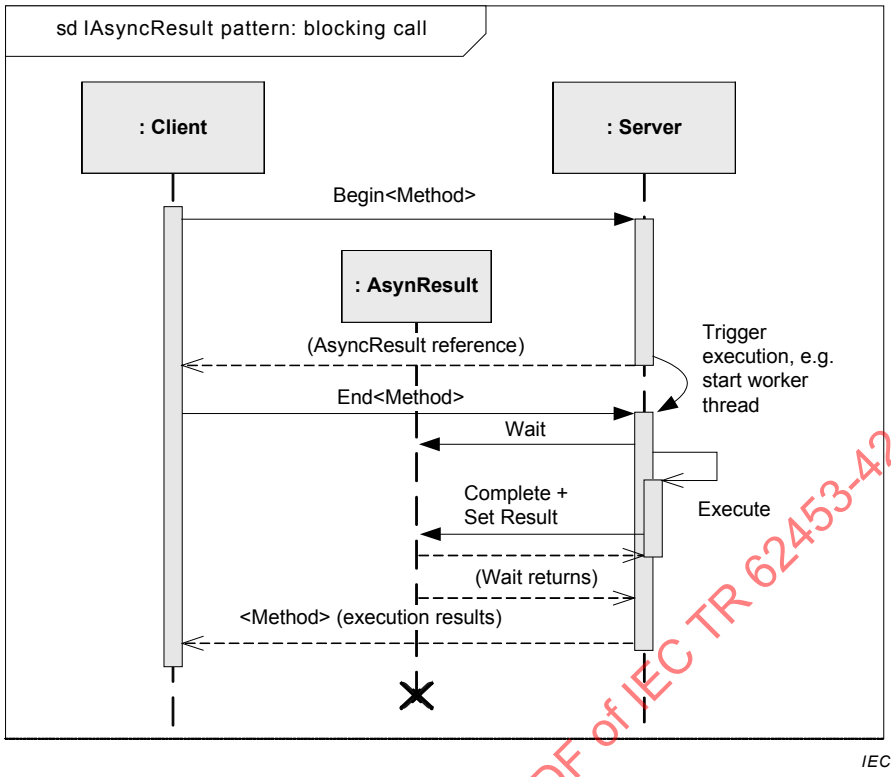


Figure 32 – IAsyncResult pattern: blocking call

Figure 33 show an example how blocking use of asynchronous operation may be implemented.

```
void SyncUpload(IDtm dtm, ICommunicationChannelProxy channelProxy)
{
    // go online and stay connected (synchronous)
    dtm.EnableCommunication(channelProxy, ConnectMode.StayConnected);

    // perform upload from device (synchronous)
    try
    {
        IOnlineOperation onlineOperations = (dtm as IOnlineOperation);
        IAsyncResult result =
            onlineOperations.BeginReadDataFromDevice(null, null, null);
        onlineOperations.EndReadDataFromDevice(result);

        MessageBox.Show("Upload finished");
    }
    catch (Exception e)
    {
        MessageBox.Show("Upload failed! " + e.Message);
    }

    // go offline (synchronous)
    IAsyncResult offline_result = dtm.BeginStopCommunication(null, null, null);
    dtm.EndStopCommunication(offline_result);
    dtm.DisableCommunication();
}
```

IEC

Figure 33 – Example: Blocking use of asynchronous interface

In order to simplify the presentation of interactions based on the IAsyncResult pattern, a simplified presentation for blocking call is used throughout the document. Figure 34 shows the simplified depiction of IAsyncResult pattern with blocking call:

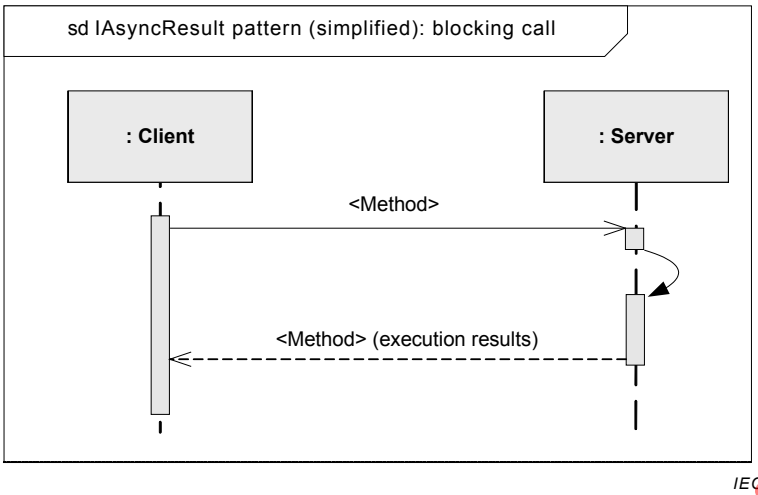


Figure 34 – IAsyncResult pattern (simplified): blocking call

Rule: If the client follows the pattern for blocking execution, it shall provide no callback.

If a service is used in a non-blocking way, the client calls the BeginOperationName() method and provides a callback delegate for OperationNameCompleted() (see Figure 35). The EndOperationName() method is called as part of handling the OperationNameCompleted() callback.

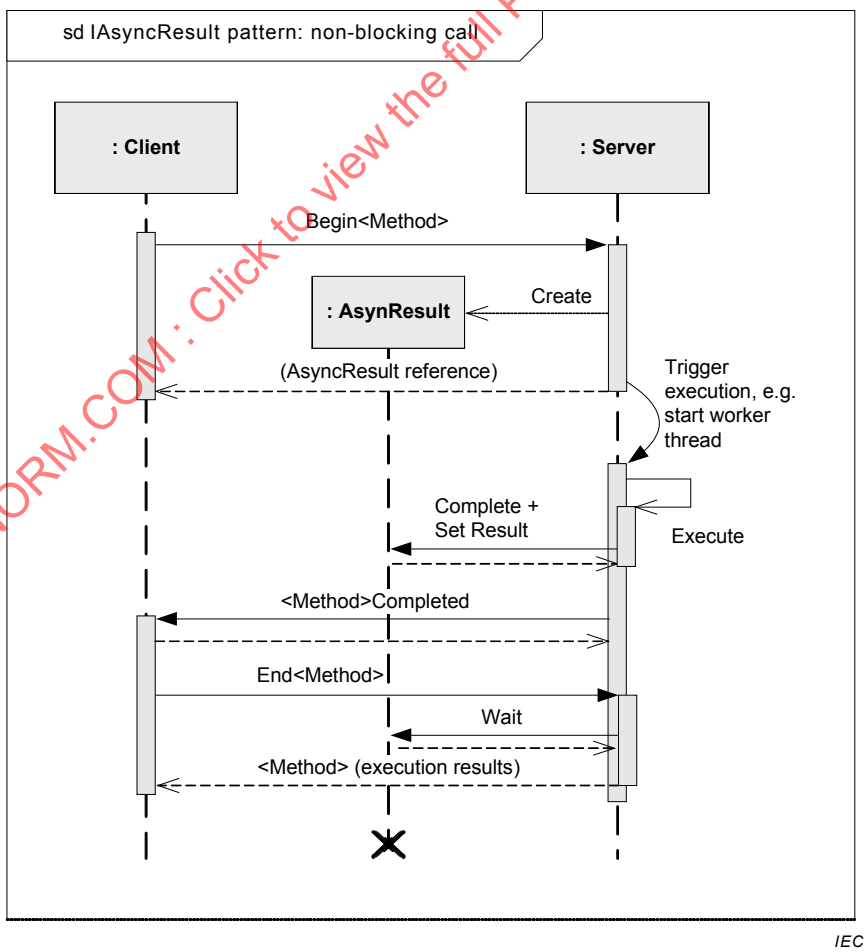


Figure 35 – IAsyncResult pattern: non-blocking call

Rule: If the callbacks are provided, the client shall follow the pattern for non-blocking execution.

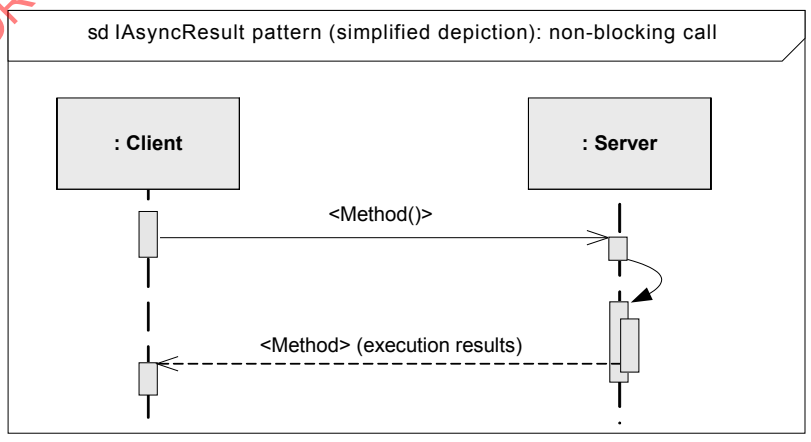
```
void AsyncUpload(IDtm dtm, ICommunicationChannelProxy channelProxy)
{
    // go online and connect only if necessary
    dtm.EnableCommunication(channelProxy, ConnectMode.OnDemand);

    IOnlineOperation onlineParam = (dtm as IOnlineOperation);
    IAsyncResult result = onlineParam.BeginReadDataFromDevice(UploadProgress,
                                                              UploadComplete, dtm);
}

void UploadProgress(ProgressInfo progressInfo)
{
    UpdateProgressBar(progressInfo.PercentComplete, progressInfo.Message);
}

void UploadComplete(IAsyncResult result)
{
    IOnlineOperation onlineParameter = result.AsyncState as IOnlineOperation;

    try
    {
        onlineParameter.EndReadDataFromDevice(result);
        SignalUploadFinishedToUI();
    }
    catch (Exception e)
    {
        SignalUploadErrorToUI();
        throw;
    }

    // go offline (not waiting for results..)
    _stop_result = dtm.BeginStopCommunication(StopCommunicationProgress,
                                              StopCommunicationComplete, dtm);
}
```

IEC

Figure 36 – Example: Non-blocking use of asynchronous interface

In the example given in Figure 36, the UploadProgress() delegate is decoupled in order to avoid blocking of the server.

In order to simplify the presentation of interactions based on the IAsyncResult pattern, a simplified presentation for non-blocking call is used throughout the document. Figure 37 shows the simplified depiction of IAsyncResult pattern with non-blocking call:



IEC

Figure 37 – IAsyncResult pattern (simplified depiction): non-blocking call

NOTE Throughout the document the simplified depiction of IAsyncResult pattern is used to show how methods are using the IAsyncResult pattern. The patterns for blocking and non-blocking calls can be used equivalently. The use of one of the call pattern in a workflow does not prohibit the use of the other call pattern if not stated explicitly otherwise.

5.6.7.3 Extended IAsyncResult pattern (Progress pattern)

In addition to the IAsyncResult pattern, the extended IAsyncResult pattern provides the possibility to cancel an asynchronous operation and to receive progress notifications and intermediate results during the processing of the operation. This pattern is used for operations that may have long execution times.

For each operation a set of methods is provided:

– The Begin*OperationName*() method starts the operation.

– The End*OperationName*() method retrieves result of the operation. If the operation is not finished, the method blocks until the operation is finished. If an error occurred during execution of the operation, this method will throw an exception with the error information.

– The Cancel*OperationName*() method stops the operation. If the operation was cancelled, then the End*OperationName*() method shall always throw the FdtOperationCancelledException.

– The Callback delegates (implemented by the client) are provided only for a specific operation. Possible delegates are: *OperationName*Progress(), *OperationName*Completed().

Figure 38 shows how the method Cancel*OperationName*() may be used.

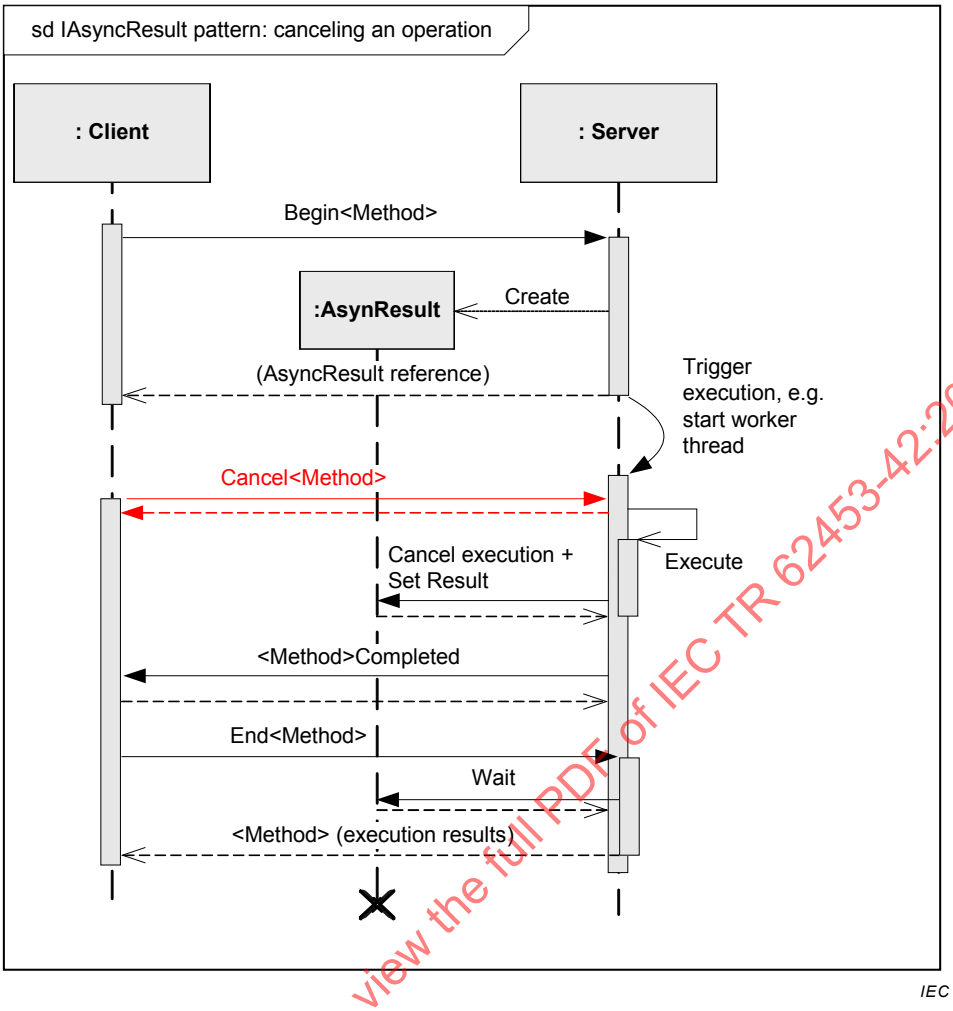


Figure 38 – IAsyncResult pattern: canceling an operation

If Cancel() can not be executed, it may result in an FdtCancelFailedException. It may also occur that the operation finished at the same time as Cancel() was called. This may lead to the caller receiving a positive result.

The Completed() callback shall not be called within a call to Cancel() (avoid call-stacks).

If Cancel() is called for an asynchronous operation, the End<Method> may throw a corresponding exception. See the documentation for each asynchronous operation.

After a call to Cancel() has succeeded it may occur in an exceptional case that the operation finishes successfully. Therefore the caller shall be prepared to receive a positive result instead of the corresponding exception.

Figure 39 shows how the progress callback may be used.

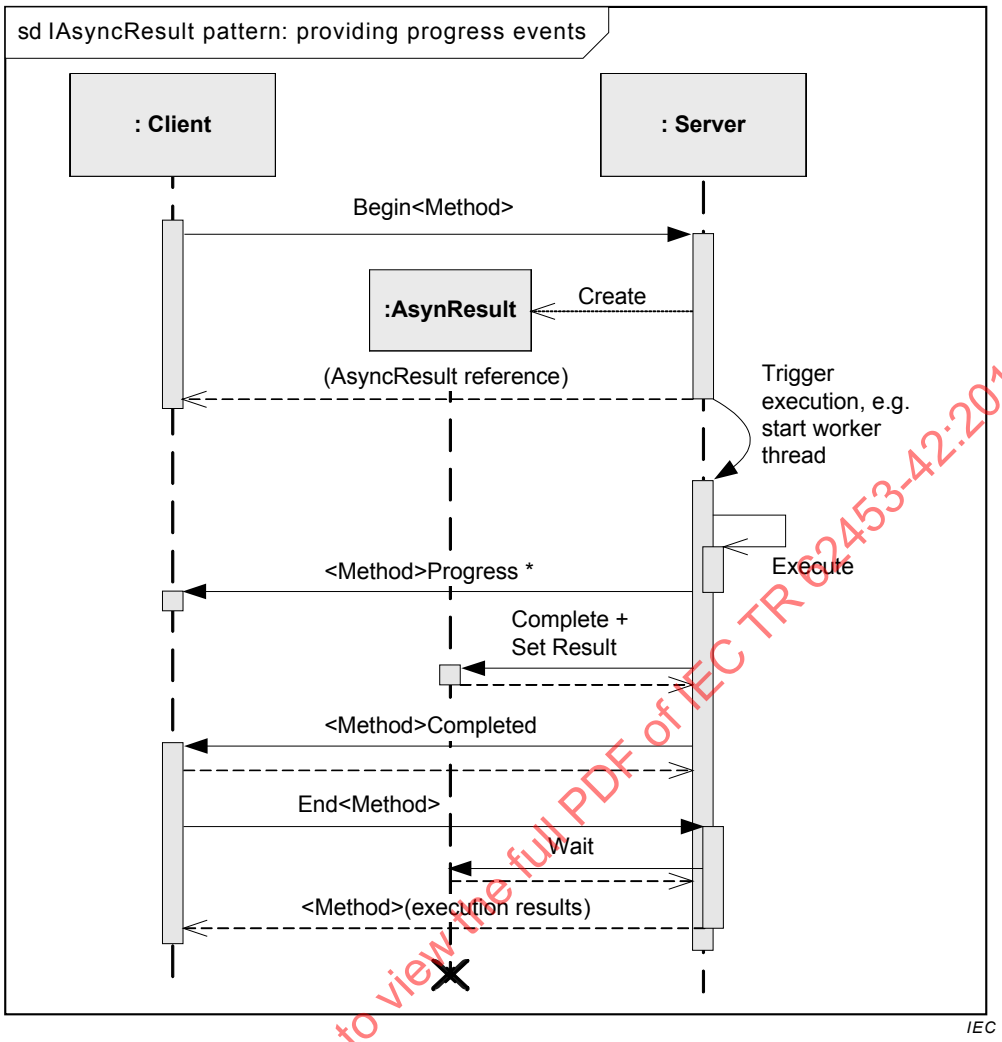


Figure 39 – IAsyncResult pattern: providing progress events

The Progress() delegate not only allows to pass progress information, but also can be used to transport partial results of the service execution. The transport of partial results is designed for each service specifically. Even if the progress delegate is used to transport partial results, the End*OperationName*() method will provide the complete result of the operation.

Rule: If the client follows the pattern for blocking execution, it shall provide no callback. For non-blocking execution both callbacks shall be provided.

If the callbacks are provided, the client shall follow the pattern for non-blocking execution.

5.6.8 Events pattern

An event is a message sent by an object to signal the occurrence of a condition.

This technical report uses the Events pattern as defined in the .NET Framework which is based on delegates (see [21]).

Clients provide delegates for receiving events (without trigger). A client registers with a server for receiving a specific event. Multiple clients may register with one server for receiving the same event.

One advantage of the events pattern is that events are defined in the same interface like the methods that may trigger those events. This allows to define the events methods in the same context in which they are used.

5.6.9 Exception handling

5.6.9.1 General

Exceptions are the primary means of reporting errors in the .NET Framework. (refer to [10], Clause 7]). They are used for both hard errors (e.g. passing of invalid arguments) as well as logical errors (e.g. connection aborted).

An exception provides two pieces of information:

- the exception message, explaining to the developer what went wrong (and how to fix it). Exception messages should be human readable text in English (not just an error number) that describes what went wrong
- the exception type that is used by exception handlers to decide what programmatic action to take

NOTE In general error codes are not used as they can always be replaced by corresponding exception messages and exception types. However, there is one exception from the rule: communication errors. Communication errors that occur during communication requests with a device are reported within the communication response. However, if the server fails to perform the transaction itself, this will throw an appropriate exception.

The FDT specification defines the exceptions which shall be thrown if specific error situations occur when calling an FDT interface method or accessing a property. This shall be considered as part of the contract between the client and server of an interface.

5.6.9.2 Throwing exceptions

Exceptions shall be thrown in cases of execution failures. An execution failure occurs whenever an interface method or property can not do what it was designed for. For example, if the <ReadDataFromDevice()> method cannot retrieve data from the device, it is considered an execution failure and an exception shall be thrown. Exceptions are the primary means of reporting errors in the .NET Framework. Error codes shall not be used.

If an FDT method invokes other FDT methods it shall handle all FDT exceptions that are defined for these methods. When the FDT method fails because of an FDT exception from an invoked method, then the FDT method shall throw the most appropriate FDT exception defined for the FDT method and include any caught exception as an inner exception. One example for this is if setting IO signal information via the interface IProcessImage to the DTM fails because the dataset can not be locked. In this case an FdtOperationFailedException shall be thrown by the IProcessImage:SetIOSignalInfo() method. This shall include the inner exception, e.g. FdtNoWriteAccessException.

DTM-specific exceptions shall also be included in FDT exceptions as inner exceptions when they occur within an FDT method.

Event-handlers are not allowed to throw exceptions. If an event-handler calls other methods that may throw exceptions, the implementation of the event-handler shall catch those exceptions in order to protect the event-source from those exceptions.

5.6.9.3 Handling exceptions

If an exception is handled, a rich and meaningful message should be provided to the end user. The message should explain the cause of the problem and describe what could be done to avoid the problem.

Since the exception message is targeted to developers, the user message should be based on the exception type and the context of the caught exception.

If an exception is just caught in order to re-throw the exception, no user message should be provided. The goal here is to avoid multiple user messages for a problem that occurred.

5.6.9.4 FDT exception types

For each FDT method a set of exceptions is defined that may occur on invocation of the method. All FDT exceptions are derived from the serializable class FdtException that is derived from System.Exception. Exceptions shall be serializable in order to work correctly across application domain and process boundaries.

The following is the list of FDT Exceptions:

FdtInvalidStateException
This exception shall be thrown when a property can not be set or when a method can not be executed, because the FDT object is not in an appropriate state (e.g. IDtm:EnableCommunication() is called in DTM state 'initialized'). Each implementation of an FDT interface member shall check whether the called object is in an appropriate state to execute the requested operation. If this is not the case, FdtInvalidStateException shall be thrown. For asynchronous operations this exception shall be thrown in the EndOperationName method.

Example: IDtm:EnableCommunication() is called in state "initialized"

FdtOperationFailedException
This exception shall be thrown when an operation can not be performed or completed successfully. For all asynchronous operations this exception may be thrown by the EndOperationName method. If more specific exceptions are available, always the most specific exception shall be used.

FdtOperationFailedException should not occur under normal operating conditions.

Example: IDtm:LoadData() is called with a valid dataset but still fails. In this case an FdtOperationFailedException is thrown. However, if IDtm:LoadData() is called with an invalid dataset, FdtInvalidDatasetException is thrown.

FdtOperationCancelledException
This exception shall be thrown if an asynchronous operation has been cancelled by CancelOperationName and the EndOperationName method is called. This happens under normal operating conditions. The client shall handle this exception and abort its own operation.

FdtCancelFailedException
This exception shall be thrown when a CancelOperationName method fails, e.g. because the operation has been finished already or can not be cancelled for other reasons. Note that this may happen under normal conditions because of the asynchronous execution of the operation. The client shall handle this exception and finish the calling operation. If the user has triggered the cancel operation, the user should be informed that the operation could not be cancelled.

FdtConfigurationErrorException
This exception shall be thrown when an operation can not be performed because of a wrong configuration.

Example: A DTM performs a connect request. The Parent DTM can not perform the request as the communication driver is not properly configured.

FdtCommunicationErrorException
This exception shall be thrown when a communication error occurs. Communication errors that occur within a communication request are reported with the communication response.

Example: The Device DTM tries to establish a connection by calling ICommunication:BeginConnect() on the provided Communication Channel proxy. The Communication Channel (or the device) is not able to establish the connection with the device because of a communication error and throws FdtCommunicationErrorException in method ICommunication:EndConnect().

NOTE Protocol-specific communication error exceptions are not defined. However, subclasses may be defined by protocol annexes if required.

FdtConnectRefusedException

This exception shall be thrown when an online operation can not be performed because the connect request has been refused.

Example: IDtm:EnableCommunication() is called on a Device DTM. The Device DTM tries to establish a connection by calling ICommunication.BeginConnect() on its Communication Channel proxy. The Communication Channel (or the device) refuses the connect and throws FdtConnectRefusedException in method ICommunication.EndConnect().

FdtConnectionAbortedException

This exception shall be thrown when an online operation can not be performed because the connection has been aborted.

Example: The connection is aborted by the Communication Channel during a download to the device. IOnlineOperation.EndWriteDataToDevice() throws an FdtConnectionAbortedException.

FdtDeviceTypeNotSupportedException

This exception shall be thrown when an online operation can not be performed because the type of the connected device is not supported by the DTM.

Example: A download operation is started with IOnlineOperation.BeginWriteDataToDevice(). The DTM is in state notConnected and connects to the device. It checks the device type and detects an unsupported device type. The operation is aborted and IOnlineOperation.EndWriteDataToDevice() throws an FdtDeviceTypeNotSupportedException.

FdtInvalidUserPermissionsException

This exception shall be thrown when an operation can not be performed because the operation is not allowed with the current user permissions.

Example: A function is started with ICommandFunction:BeginExecute(). The user is logged in as Observer and has no access rights to perform this function. The DTM aborts the operation. ICommandFunction:EndExecute() throws an FdtInvalidUserPermissionsException.

FdtInvalidValueException

This exception shall be thrown when an invalid value was given as an argument in the request.

Example: A client tries to write a value via IInstanceData/IDeviceData that is out of the valid range.

FdtInvalidTypeIdException

This exception shall be thrown when an invalid type id was given as an argument in the request.

Example: IDtm:InitData() is called with a typeid that is not supported by the DTM and throws an FdtInvalidTypeIdException.

FdtInvalidDataObjectException

This exception shall be thrown when an invalid data object was given as an argument in the request.

Example: An invalid DtmSystemTag is given in ITopology:BeginGetDtm().

FdtInvalidReferenceException

This exception shall be thrown when an invalid reference to another object was given as an argument in the request. FdtInvalidReferenceException should not occur under normal operating conditions.

Example: An invalid IAsyncResult object is given in an EndOperationName or in a CancelOperationName method

FdtInvalidCommunicationChannelException

This exception shall be thrown when an invalid Communication Channel is set.

Example: The argument parentCommunicationChannel is set to a channel that is not supported (e.g. protocol is not supported).

FdtInvalidDatasetException

This exception shall be thrown when an invalid dataset was given as an argument in the request. FdtInvalidDatasetException should not occur under normal operating conditions.

Example: IDtm:LoadData() is called with a dataset that is not supported by the DTM type.

FdtNoReadAccessException
This exception shall be thrown when a read operation can not be performed because the data object is not readable.

Example: A client tries to read a data object that is classified as write only. EndRead() throws an FdtNoReadAccessException.

FdtNoWriteAccessException
This exception shall be thrown when a write operation can not be performed because the data object is not writable.

Before writing any data, the DTM shall initiate a transaction on the dataset with IDataset:StartTransaction() if this fails, the operation shall be aborted and an FdtLockDatasetException shall be thrown in this case.

Example: A client tries to write a data object that is classified as read only. EndWrite() throws an FdtNoWriteAccessException.

FdtLockDatasetException
This exception shall be thrown when the dataset can not be locked in order to perform transactions on the dataset or device.

Example: IOnlineOperation.BeginReadDataFromDevice() is called. The dataset can not be locked as it is currently locked by another instance. IOnlineOperation.EndReadDataFromDevice() throws an FdtLockDatasetException.

FdtCommitTransactionFailedException
This exception shall be thrown when a commit transaction fails. This may happen e.g. when the database is located on a remote computer and the network connection is disrupted.

FdtCommitTransactionFailedException should not occur under normal operating conditions.

FdtCannotCloseUiException
This exception shall be thrown if a user interface can not be closed. The user interface may have changed data items that have not been committed yet or some active actions with the device that need to be finished.

In this case, the Frame Application shall inform the user that he needs to finish active actions with this user interface before it can be closed.

5.6.9.5 Standard exception types

In general FDT exceptions shall be used where applicable (please refer to the corresponding FDT interface definitions). Following .NET standard exception types should be used in situations where no FDT exceptions are applicable.

InvalidOperationException
InvalidOperationException shall be thrown if the object is in an inappropriate state. If the object is a defined FDT object use FdtInvalidStateException.

ArgumentException, ArgumentNullException, ArgumentOutOfRangeException
ArgumentException or one of its subtypes shall be thrown if bad arguments are passed to an interface member. The most derived exception type should be used where applicable. The ParamName property represents the name of the parameter that caused the exception to be thrown. Note that the property can be set by one of the constructor overloads. Use “value” for the implicit value parameter of property setters.

5.6.9.6 Other standard exceptions

The following exceptions shall not be thrown by FDT objects. Argument checking shall be performed to avoid throwing these exceptions.

- NullReferenceException,
- IndexOutOfRangeException,

- AccessViolationException

The following exceptions shall not be thrown explicitly by FDT objects:

- StackOverflowException,
- OutOfMemoryException,
- InteropException,
- ComException,
- SEHException,
- ExecutionEngineException

5.7 Threading

5.7.1 Introduction

5.7.1.1 General

Multi-threading as supported by .NET runtime solves several problems with regard to throughput and responsiveness, but in doing so it introduces new problems such as races and deadlocks. In order to avoid such problems in FDT2 some rules are defined that shall be applied by all FDT components.

NOTE The threading terms (e.g. apartment model) as used within the context of COM do not apply in .NET.

Additional information about Multi-threading and concurrency can be found in [30].

5.7.1.2 Races

A race is a failure which occurs because of improper synchronization between threads. Depending on which of two or more threads reaches a particular block of code first the result of a program (or a particular piece of code) cannot be predicted. When different threads access common memory concurrently, the computing result may be correct or not.

There are four conditions required for a race to be possible:

a) There are memory locations that are accessible from more than one thread. Typically, locations are global/static variables or are heap memory reachable from global/static variables.

b) There is a property (invariant) associated with these shared memory locations that is needed for the program to function correctly. Typically, the property needs to hold true before an update occurs for the update to be correct.

c) The property does not hold during some part of the actual update.

d) Another thread accesses the memory when the invariant is broken, thereby causing incorrect behavior.

5.7.1.3 Locks

The most common way of preventing races is to use locks to prevent other threads from accessing shared memory associated with an invariant while it is broken. This removes the fourth condition mentioned above, thus making a race impossible.

The most common kind of lock is called a monitor (sometimes the same basic functionality is named a critical section, a mutex, or a binary semaphore). A monitor provides Enter and Exit methods, and once a thread calls Enter, all attempts by other threads to call Enter will cause the other threads to block (wait) until a call to Exit is made. The thread that called Enter is the owner of the lock, and it is considered a programming error if Exit is called by a thread that is not the owner of the lock. Locks provide a mechanism for ensuring that only one thread can execute a particular region of code at any given time.

5.7.1.4 Deadlocks

A deadlock is a situation wherein two or more concurrent operations are each waiting for the other to finish, and thus neither can make any further progress.

There are four conditions required for a deadlock to be possible:

a) Mutual exclusion. Only a limited number of threads may utilize a resource concurrently.

b) Hold and wait. A thread holding a resource may request access to other resources and wait until it gets them.

c) No preemption. Resources are released only voluntarily by the thread holding the resource.

d) Circular wait. There is a set of {T1, …, TN} threads, where T1 is waiting for a resource held by T2, T2 is waiting for a resource held by T3, and so forth, up through TN waiting for a resource held by T1.

Since multiple threads can access an FDT object concurrently, it is necessary to synchronize the access to internal data objects or to user interface objects by mutual exclusion (see condition #1). Condition #2 is hard to avoid since multiple resources are often required to perform an operation. Resources that are locked can not be pre-empted from the current owner, so condition #3 can not be avoided as well. The most common and actionable condition in FDT is condition #4, circular waits.

5.7.2 Threading rules

5.7.2.1 Implementation rules

The following rules shall be applied in order to allow multithreading and using locks to avoid races:

a) Software shall be prepared to receive calls in any thread. Each FDT object shall be able to receive calls in any thread. Operations that need to be performed in a dedicated thread (e.g. user interface thread) shall be synchronized to this thread internally.

b) Software shall protect internal data against parallel access. Make static data and instance data thread safe. Ensure that all thread-shared, read-write data is protected by locks.

c) Each lock shall be assigned to a specific region of memory (not a region of code!). This assignment shall be well documented in the developer documentation.

d) Each lock shall provide mutual exclusion for the region of memory that it is assigned to. No writes to that memory can occur without entering the same lock. Data structure invariants have to hold any time the lock protecting the data structure is not held.

e) If two data structures are related, locks for both structures shall be entered before using that relationship.

Implementers should also consider the following recommendations:

– Memory regions (e.g. data structures) that are protected by locks should not overlap. Consider that mutual exclusion may not be guaranteed if you have overlapping regions with different locks. If it would always be required to enter two or more locks, a single lock would be more appropriate to protect the overlapping regions.

– Use as few locks as possible. The complexity grows quickly with the number of locks in the system, so it is best to have few locks that protect large regions of memory and only split them when lock contention is shown to be a bottleneck on performance. Generally, the finer the granularity of the locks, the more of them that can be held at once—and the longer they are held, the higher the risk of deadlock.

– Check whether read locks are required. Entering locks is not only required when writing to memory but also when reading from shared memory. In general, when code needs a program invariant, all locks associated with any memory involved with the invariant shall be entered.

5.7.2.2 Avoiding deadlocks

For the reasons explained in 5.7.1.4, the following rules are defined to avoid deadlocks (list is continued from previous subclause):

a) FDT objects are not allowed to call any FDT interface method or wait on incoming FDT calls, callbacks or events while holding any locks. This avoids a circular wait across multiple FDT objects.

   If it is necessary to call other FDT objects in order to perform a complex operation, then implementation as a state machine should be considered, because parallel requests may be refused or queued until the running operation is finished.

   Exceptions: It is allowed to call ITrace methods and asynchronous BeginXXX methods within locked code areas. The interaction management will ensure that these calls are decoupled and processed in a safe way.

b) Each FDT object shall avoid that a circular wait can happen within a single FDT object. Multiple threads accessing the object at the same time may perform different tasks and require multiple resources that need to be locked. Avoiding the circular wait condition is usually done by acquiring locks always in a specific order or by lock leveling. This is a strategy where each lock is assigned a level. A thread can only acquire locks with the same or lower level that it already holds.

5.7.2.3 FDT Object interaction rules

All FDT objects shall apply the following rules (list is continued from previous subclause):

a) Do not call FDT interfaces in the user interface thread.
   The user interface thread of a process shall be dedicated to receive user inputs and perform drawing tasks only. FDT objects shall not use the user interface thread to call FDT interface methods, perform callbacks or events.

b) Do not block the user interface thread.
   The user interface shall always stay responsive. The user interface thread is shared between the different FDT (user interface) objects for user input and drawing operations. If one object blocks this thread in order to perform some processing, this would affect the responsiveness of other objects.

c) Do not block a BeginMethodName method call.
   A BeginMethodName method shall only start an asynchronous operation. Therefore it shall not block the caller.

d) Do not block a synchronous method call.
   A thread calling a FDT synchronous method shall not be blocked. It is not allowed to call any EndMethodName within a synchronous method or to wait on events, because this will block the calling thread.

e) Process events and callbacks asynchronously.
   No FDT operations shall be performed within an event handler or callback method. A work item shall be created that is processed asynchronously. The calling thread shall not be blocked.

5.8 Localization support

5.8.1 General

There are two main processes for developing software that supports different languages and cultures.

– Globalization is the first process to design software that is capable of running with different cultures and languages. This process is realized by separating the executable code that is culture or language independent from those parts that are culture or language dependent. Language dependent parts for example are such as user interfaces, calendars, numbers, several string manipulation and comparison algorithms. The .NET Framework as technological basis for IEC 62453-42 supports this process through a

number of classes that are packaged in the .NET Framework under the namespace System.Globalization.

– Localization is the second process to customize the software to a specific culture and language. This is primarily achieved by translating the user interface. For .NET Framework based applications, this results in a primary assembly that contains only culture-neutral and language-neutral executable code and resources. Each additional culture, region or language is provided in a separate satellite assembly.

The .NET Framework provides infrastructure to access the hierarchically organized resources. First the framework classes attempt to access the resources that belong to the specified region or country. If this access fails, the framework classes attempt to access the resources that belong to the specified language. If this access also fails, the framework classes attempt to access culture-neutral or language-neutral resources.

It is recommended to utilize the infrastructure provided by the .NET Framework. There are several translation tools on the market and the translation agencies know how to deal with the XML based as well as the binary resource files.

5.8.2 Access to localized resources and culture-dependent functions

The .NET Framework provides two different ways to access the localized resources and culture-dependent functions.

The preferred way is to use the automatic culture handling. Each thread within an application provides information about the currently used culture and language setting. It can be retrieved by reading the properties CurrentCulture and CurrentUICulture from class System.Threading.Thread. All user interface related functions rely on the property CurrentUICulture whereas other culture-dependent functions use the property CurrentCulture. The value of these two properties are initialized to the default values defined in the system settings “Control Panel – Regional Settings”. The values of the properties can be overridden by writing the property values.

The second way to access localized resources and culture-dependent functions is needed only in some rare cases. All culture-dependent functions provide an overloaded variant where the culture can be specified explicitly.

5.8.3 Handling of cultures

As described in 5.8.2, the .NET Framework handles the culture settings for each thread separately and derives the start value from the system settings. The Frame Application is responsible to synchronize the culture settings for all threads with outgoing function calls.

There are two properties defining the used language: Thread.CurrentUICulture and Thread.CurrentCulture.

The Frame Application indicates the currently used language by setting the property Thread.CurrentUICulture before the DTM is started. During initialization the DTM Business Logic and DTM User Interface shall read this property in order to display the DTM User Interface or any other output that is provided to the user (e.g. labels and descriptors). If the DTM implementation relies on the resource management classes that are contained in the .NET Framework class library, no additional implementation will be necessary. Otherwise, the language-dependent resources need to be handled explicitly. Switching the UI culture by the Frame Application shall not trigger any notifications from the DTM (e.g. DataInfoChanged).

If a DTM uses additional threads with outgoing function calls or if it raises events from additional threads, it is responsible to synchronize the language settings of newly created threads with the settings from the original thread.

The property Thread.CurrentCulture shall not be changed by any FDT component in order to reflect the culture settings of the operating system.

The described mechanism results in the following behavior: Texts, pictures and similar user interface elements will be displayed according to the language that was selected by the user at the Frame Application (in Thread.CurrentUICulture). Whereas input direction, sort orders, comparison, number formatting are determined by the culture settings of the operating system (in Thread.CurrentCulture).

5.8.4 Switching the User Interface language

A Frame Application may provide a mechanism that allows switching the language of the user interface. It is specific to the Frame Application, whether switching is realized during runtime or whether it needs the restart of the Frame Application. The Frame Application shall not switch the user interface language as long as any DTM or DTM component is instantiated. The Frame Application may need to shutdown and restart the DTM objects in order to switch the language. The DTMs will start with the new language setting.

At the Frame Application the user might select a language that is not available for certain DTMs. The DTM Business Logic and DTM User Interface shall operate as expected independent of the selected language. If a DTM does not support the selected language, it shall switch to a commonly used fallback language, which is English.

A DTM User Interface may provide a menu, where the user can override the language setting for this user interface. The DTM Business Logic and DTM User Interface shall not change the properties Thread.CurrentUICulture and Thread.CurrentCulture, because this would influence the behavior of other FDT components that share this thread. The language remains active until the language setting is changed again or until the user interface is closed.

5.9 DTM User Interface implementation

5.9.1 General

A DTM Business Logic is operated by DTM User Interface. Different kinds of user interfaces for one DTM Business Logic may exist. FDT supports following DTM User Interface types:

- DTM WPF controls [11] can be embedded into the user interface of the Frame Application. These controls shall derive from the standard .NET WPF UserControl class (namespace System.Windows.Controls).
- DTM WinForms controls can be embedded into the user interface of the Frame Application. These controls shall derive from the standard WinForms UserControl class (namespace System.Windows.Forms).
- DTM Applications are external DTM-specific user interfaces (e.g. executable applications) which cannot be embedded into the Frame Application. These external applications are represented in the Frame Application by simple .NET classes (named “DTM UI Application”, see Figure 65) which manage the interaction between the external user interface and the Frame Application.
- DTM UI CommandFunctions are similar to the CommandFunctions which can be executed at the DTM Business Logic (see 7.14), but UI CommandFunctions are allowed to open private user interfaces (e.g. dialog boxes etc.). Such functions are represented by simple .NET classes which contain the code to execute.

All four DTM User Interface types implement the same interface (see 6.4).

5.9.2 Resizing

The DTM WPF and WinForms Controls should be implemented in a resizable way. That means the controls are responsible for supporting re-arrangement of the inner graphical elements. In addition, a control shall specify its minimum size.

The Frame Application may use the minimum size as a hint for the initial size to show the control. If the Frame Application displays the control's host window smaller than the minimum size of the control, then the Frame Application has to provide scrollbars.

When the Frame Application allows the user to resize the host window to a size smaller than the minimum size of the control then the Frame Application has to show scrollbars for the DTM UI control (see Figure 40).

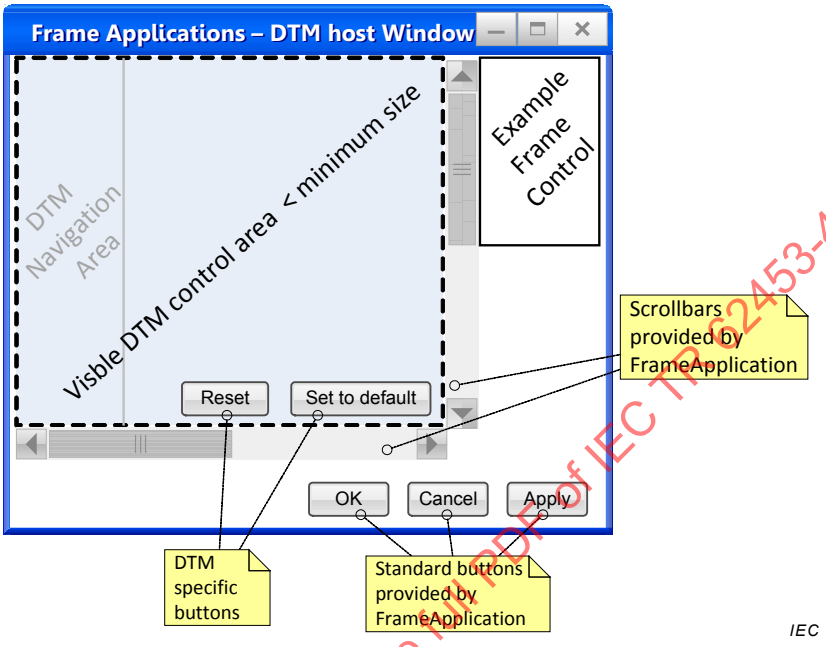


Figure 40 – Frame Application's host window providing scroll bars

Independent of the scrollbars shown by the Frame Application the control itself may show additional scrollbars if appropriate. This is needed for example if the application area requires more space than available (see Figure 41).

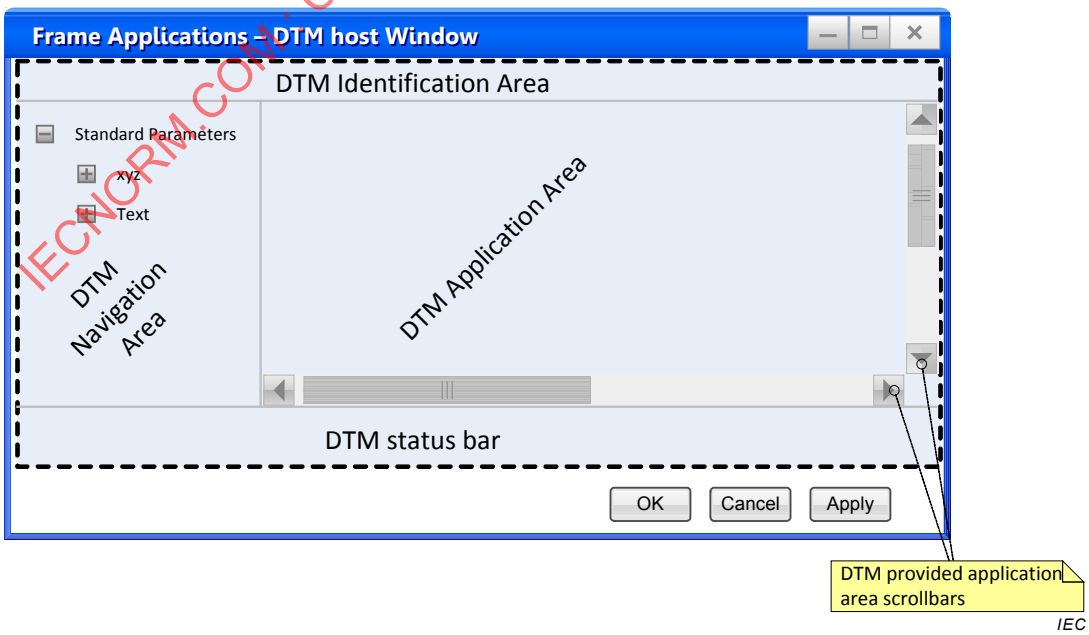


Figure 41 – Control using internal scrollbars

5.9.3 Private dialogs

Private dialogs are all kinds of graphical user interfaces such as:

- message boxes (i.e. standard message box);
- file or printer selection dialogs (i.e. provided by operating system);
- (default) web browsers;
- (default) mail clients;
- help file view;
- manual viewer;
- splash screens;
- external stand-alone applications,
- any other windows.

Any DTM UI (control, application and UI command function)is allowed to show private dialogs, but should prefer use of the services supported by the Frame Application. If a Frame Application needs to ensure that private DTM UIs are not overlapping critical Frame Application UIs, then this needs to be implemented Frame Application-specific (e.g. do not run DTM UIs on such PCs, display on a second screen etc.).

A DTM Business Logic is not allowed to open private dialogs or user interfaces. A DTM Business Logic shall always use the Frame Application user interface services (see IFrame.Ui property). This rule is necessary since operations on the DTM Business Logic may be executed unattended (e.g. batch processing).

5.10 DTM User Interface hosting

5.10.1 General

The Frame Application can dynamically load the DTM User Interface.

For different DTM User Interface types the activation and initialization is similar (see 5.4). The sequence diagrams in 8.5 describe these operations in more detail.

The DTM WPF controls and DTM WinForms controls additionally need to be embedded in a Frame Application user interface element. The Frame Application shall implement following general sequence:

a) Load the assembly and create the control
b) Check the type of the control (UiControlFunctionInfo.Type)
c) Host the control in a parent user interface element
d) Initialize the control (<Init()>)
e) Make the control visible and size it to the parent window size

The Frame Application shall initialize the controls before they are made visible. This enables the control for example to load the correct device picture before it's displayed. The initialize method of an DTM provided control shall return immediately. If the control need to perform operations taking a longer time (e.g. communicate with the device), then this shall be done asynchronously.

5.10.2 Hosting DTM WPF controls

A WPF Frame Application shall embed DTM WPF controls in a layout element like Grid control or Panel controls (see Figure 42).

```
void HostWPFControlInWPFApp(string dtmUiAssemblyPath, string controlClassName)
{
    // Load DTM User Interface assembly
    Assembly dtmUiAssembly = Assembly.LoadFrom(dtmUiAssemblyPath);

    // Create WPF control
    Type controlType = dtmUiAssembly.GetType(controlClassName);
    UIElement wpfControl = Activator.CreateInstance(controlType) as UIElement;

    // Host WPF control in layout element (e.g. Panel, Grid)
    Panel parent = this.clientArea;
    parent.Children.Add(wpfControl);

    // Initalize
    (wpfControl as IDtmUiFunction).Init(/* parameters */);

    // Make visible
    parent.Visibility = Visibility.Visible;
}
```

IEC

Figure 42 – Example: Hosting a DTM WPF control in a WPF Frame Application

A WinForms Frame Application shall embed DTM WPF controls in the ElementHost control (System.Windows.Forms.Integration namespace) (see Figure 43).

```
void HostWPFControlInWinFormApp (string dtmUiAssemblyPath, string controlClassName)
{
    // Load DTM User Interface assembly
    Assembly dtmUiAssembly = Assembly.LoadFrom(dtmUiAssemblyPath);

    // Create WPF control
    Type controlType = dtmUiAssembly.GetType(controlClassName);
    UIElement wpfControl = Activator.CreateInstance(controlType);

    // Host WPF control in ElementHost
    ElementHost host = new ElementHost();
    host.Dock = DockStyle.Fill;
    host.Child = wpfControl;
    parent.Controls.Add(host);

    // Initalize
    (wpfControl as IDtmUiFunction).Init(/* parameters */);

    // Make visible
    parent.Visible = true;
}
```

IEC

Figure 43 – Example: Hosting a DTM WPF control in a WinForms Frame Application

5.10.3 Hosting DTM WinForms controls

A WinForms Frame Application shall embed DTM WinForms controls in layout elements like Forms or Panels. (see Figure 44)

```
void HostWinFormControlInWinFormApp(string dtmUiAssemblyPath, string controlClassName)
{
    // Load DTM User Interface assembly
    Assembly dtmUiAssembly = Assembly.LoadFrom(dtmUiAssemblyPath);

    // Create WinForm control
    Type controlType = dtmUiAssembly.GetType(controlClassName);
    Control winFormControl = Activator.CreateInstance(controlType) as Control;

    // Host WinForm control in a WinForm layout element
    Control parent = this;
    winFormControl.Dock = DockStyle.Fill;
    parent.Controls.Add(winFormControl);

    // Initalize
    (winFormControl as IDtmUiFunction).Init(/* parameters */);

    // Make visible
    parent.Visible = true;
}
```

IEC

Figure 44 – Example: Hosting DTM WinForms controls in a WinForms Frame Application

A WPF Frame Application shall embed DTM WinForms controls in a WindowsFormsHost. (see Figure 45)

```
void HostWinFormControlInWPFApp(string dtmUiAssemblyPath, string controlClassName)
{
    // Load DTM User Interface assembly
    Assembly dtmUiAssembly = Assembly.LoadFrom(dtmUiAssemblyPath);

    // Create WinForm control
    Type controlType = dtmUiAssembly.GetType(controlClassName);
    System.Windows.Forms.Control winFormControl = Activator.CreateInstance(controlType)
as System.Windows.Forms.Control;

    // Host WinForm control in a WPF layout element (grids, panel)
    System.Windows.Controls.Panel parent = this.clientArea;
    WindowsFormsHost host = new WindowsFormsHost();
    host.Child = winFormControl;
    parent.Children.Add(host);

    // Initalize
    (winFormControl as IDtmUiFunction).Init(/* parameters */);

    // Make visible
    parent.Visibility = Visibility.Visible;
}
```

IEC

Figure 45 – Example: Hosting a DTM WinForms control in a WPF Frame Application

5.11 Static Function implementation

Static Functions are implemented as .NET functions. Each function shall be implemented in a separate assembly.

A DTM provides information about the available Static Functions (see StaticFunctionInfo in Annex B) with the method IStaticFunctionInformation.GetStaticFunctions. The information provided by the DTM describes the supported use case and the arguments of the function.

Since a Static Function can process all .NET datatypes, including FDT-specific datatypes, it is possible that a Static Function processes CommunicationResponses from a device. In such cases the description of the input argument contains the CommunicationRequest that is needed to retrieve the respective CommunicationResponse from the device.

A StaticFunction and related communication requests are not allowed to change the status of the device.

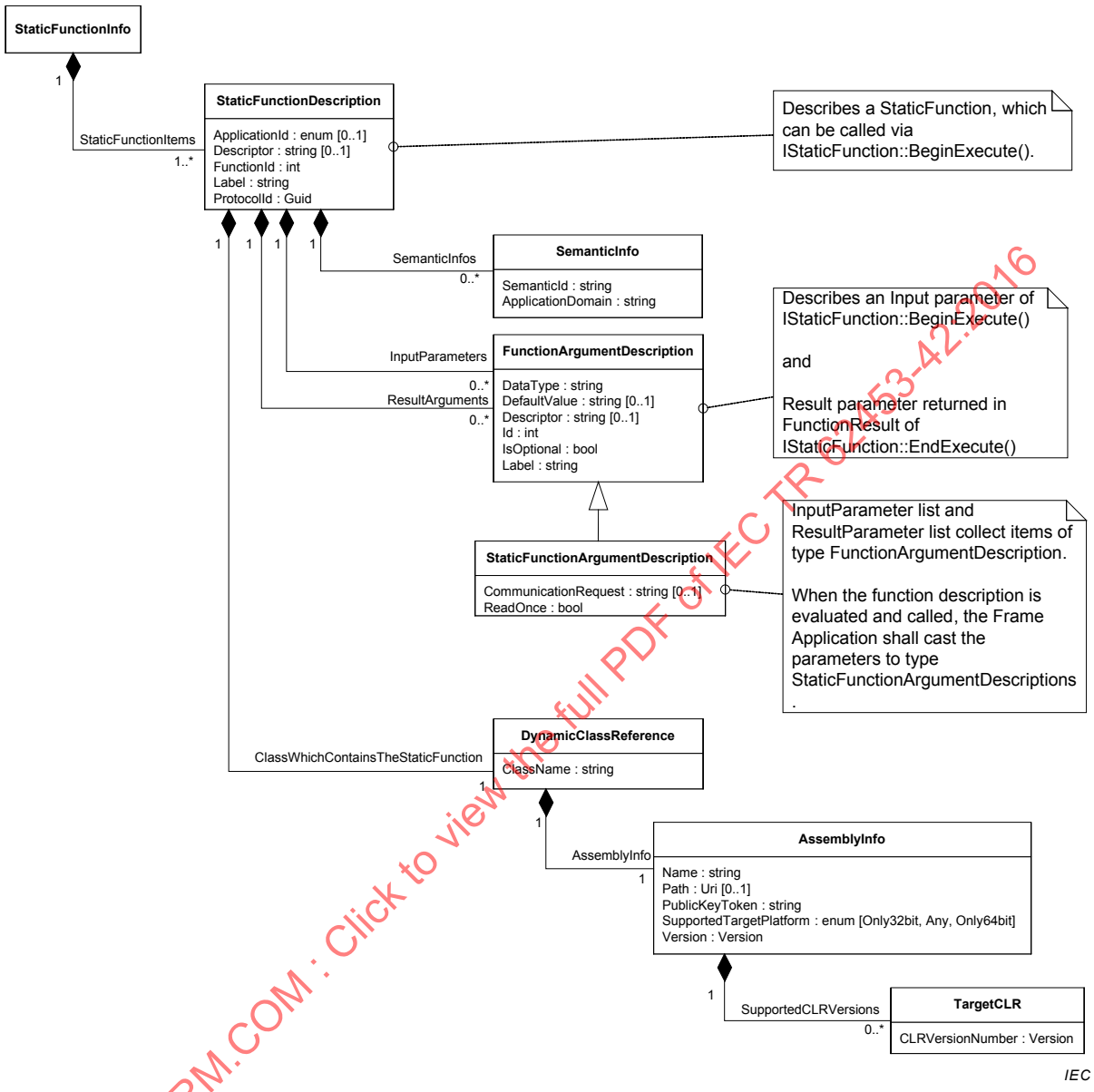


Figure 46 – Relation of StaticFunctionDescription to Static Function

Figure 46 shows the relation between description of a static function and the actual function that may be invoked. The StaticFunctionDescription describes a StaticFunction together with its input arguments and its result arguments. For each argument of the StaticFunction there is a corresponding description.

The input and result arguments of a static function may be of any datatype. The values of the arguments are provided as string. The string contains the serialized value of the datatype.

It is possible to define that a StaticFunction is using a communication response (TransactionResponse) as an input argument. In such a case the CommRequest attribute of the corresponding StaticFunctionArgumentDescription contains the communication request (TransactionRequest) necessary to retrieve the communication response.

In order to optimize the communication access the DTM may use the flad ReadOnce. If ReadOnce is set to "TRUE", then the FA may issue the CommunicationRequest only one time

(e.g. when starting an observation). The resulting CommunicationResponse can be used as multiple times when executing the static function. If ReadOnce is set to "FALSE", then the FA shall retrieve the CommunicationResponse each time when executing the static function.

5.12 Persistence

5.12.1 Overview

In the call to InitData() or LoadData() the DTM receives a reference to the IDataset interface provided by the Frame Application.

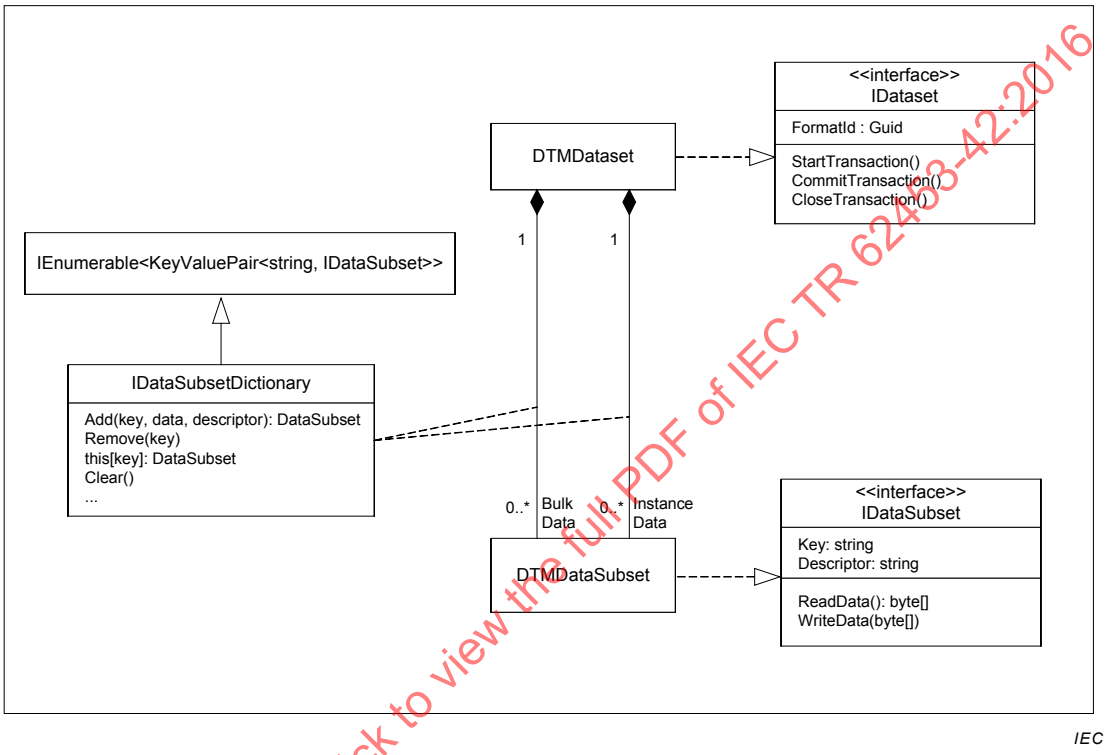


Figure 47 – DTMDataset structure

The DTMDataset contains two DTMDataSubset dictionaries for the actual persistence of data (see Figure 47): InstanceData dictionary and BulkData dictionary. Each dictionary contains DTMDataSubsets. The DTMDataSubsets shall be used for grouping of persistence data. The number and content of the DTMDataSubsets is DTM-specific. In order to improve the system performance the DTM shall group data which need to be loaded and stored together in one DTMDataSubset. Furthermore, a DTM shall avoid unnecessary loading of data whenever possible, especially when starting the DTM Business Logic.

The InstanceData dictionary shall be used for data which is directly related to the represented device instance, for example the device parameters, network information, etc. The DTM has to guarantee that it is able to represent the device by loading this data. Following DTMDataSubsets should be considered:

– Basic data which is needed during the complete lifetime of a DTM instance(e.g. represented device type information, device tag and address and other identity information).
– Device parameter information that is needed if corresponding DTM User Interface is opened (e.g. a page in a dialog) or if the Frame Application requests data (e.g. DeviceDataInfo (see 7.9))
– IO signal information which is needed if Frame Application requests ProcessDataInfo (see 4.4.4)

The BulkData dictionary shall be used for further device instance-specific data, for example for bulky trend or historical data, which is only needed in special scenarios. The Frame Application may implement a special storage mechanism for bulk data, which might be optimized for handling of big amount of data, but may be slower than the implementation for the instance data storage.

Beside these data separation and grouping rules the DTM shall also follow the rules defined for data searching (see 5.12.4) to support a maximum system performance.

5.12.2 Data format

The format of the data persisted in the DTMDataSubsets is DTM-specific, but the DTM shall store information about the used format in the IDataset.FormatId property. The FormatId is a unique identifier created by the DTM vendor, it shall correspond to the used FormatId exposed in the IDtm.ActiveType property.

The DTM shall use the FormatId information to decide how to load the data, e.g. to load data stored in a different format by an older DTM version.

A DTM may also expose further supported FormatIds in its TypeInfo. This information may be used by the Frame Application to migrate data stored by a different DTM (see 8.17).

5.12.3 Adding / reading / writing / deleting of data

By default the two DTMDataSubset dictionaries contained in the DTMDataset are empty. The DTM itself is responsible for adding the needed DTMDataSubsets to the dictionaries, for example at first start-up in the call to InitData() (see 8.2.1).

The DTMDataSubset dictionaries also provide methods to remove data from the persistence storage managed by the Frame Application. However, in case of deleting of the DTM instance the Frame Application itself is responsible to remove the data from the storage after the call to IDtm.BeginRelease(deleteInstance=true) returned.

The IDataSubset interface provides methods to read and write binary data. The DTM itself is responsible to serialize / deserialize the data.

Figure 48 shows an example implementation on how a DTM can initialize a DTMDataSubset with binary data by using the .NET Framework BinaryFormatter class for serialization.

```
public void InitData(Guid dtmDeviceTypeId, IDataset dataset)
{
    // initialize class members
    _dataset = dataset;
    _activeDtmDeviceType = _supportedTypes.Find((item) =>
        item.Id == dtmDeviceTypeId);
    _deviceAddress = new DeviceAddress<HartDeviceAddress>(1,
                        new HartDeviceAddress(0, "SHORTTAG", "Long Tag ....",
                            HartDeviceAddress.AddressingModeSelection.ShortAddress,
                            new HartLongAddress()));
    // start transaction (needed for adding of DataSubsets)
    _dataset.StartTransaction();

    // create binary formatter needed for serialization of data
    MemoryStream stream = new MemoryStream();
    BinaryFormatter binaryFormatter = new BinaryFormatter();

    // serialize ActiveType(Id) and DeviceAddress
    binaryFormatter.Serialize(stream, _activeDtmDeviceType.Id);
    binaryFormatter.Serialize(stream,
                              _deviceAddress.ProtocolSpecificDeviceAddress.ShortAddress);
    stream.Close();

    // create data subset for "basic" DTM data and initialize with default data
    _dataset.InstanceData.Add("basicData", stream.GetBuffer());

    // store used format and close transaction with auto commit = true
    _dataset.FormatId = _activeDtmDeviceType.DatasetFormats_Used;
    _dataset.CloseTransaction(true);
}
```

IEC

Figure 48 – Example: Initialization of DTMDataSubset with DTM data

The DTM has to provide a unique key for the DTMDataSubset when adding it to the dictionary. The DTM can use the key to access DTMDataSubset, for example for reading and writing of the binary data.

Figure 49 shows an example on how a DTM can write binary data into a DTMDataSubset by using the .NET Framework BinaryFormatter class for serialization.

```
protected void SaveBasicData()
{
    // start transaction (needed for writing of DataSubset data)
    _dataset.StartTransaction();

    // create binary formatter needed for serialization of data
    MemoryStream stream = new MemoryStream();
    BinaryFormatter binaryFormatter = new BinaryFormatter();

    // serialize ActiveType(Id) and DeviceAddress
    binaryFormatter.Serialize(stream, _activeDtmDeviceType.Id);
    binaryFormatter.Serialize(stream,
                              deviceAddress.ProtocolSpecificDeviceAddress.ShortAddress);
    stream.Close();

    // create datasubset for "basic" DTM data and initalize with default data
    _dataset.InstanceData["basicData"].WriteData(stream.GetBuffer());

    // close transaction with auto commit = true
    _dataset.CloseTransaction(true);
}
```

IEC

Figure 49 – Example: Writing of DTM data in DTMDataSubset

Figure 50 shows an example on how a DTM can read data from a DTMDataSubset by using the .NET Framework BinaryFormatter class for deserialization.

```
public void LoadData(IDataset dataset)
{
    _dataset = dataset;

    // read persisted "basic" data
    byte[] data = _dataset.InstanceData["basicData"].ReadData();

    // create binary formatter which is insensitive regarding assembly version
    // in which serialized classes have been defined
    MemoryStream stream = new MemoryStream(data);
    BinaryFormatter binaryFormatter = new BinaryFormatter();
    binaryFormatter.AssemblyFormat = FormatterAssemblyStyle.Simple;

    // deserialize ActiveType and DeviceAddress data
    Guid dtmDeviceTypeId = (Guid)binaryFormatter.Deserialize(stream);
    _activeDtmDeviceType = _supportedTypes.Find((item) =>
        item.Id == dtmDeviceTypeId);

    // deserialize DeviceAddress data
    int shortAddress = (int)binaryFormatter.Deserialize(stream);
    _deviceAddress = new DeviceAddress<HartDeviceAddress>(1,
        new HartDeviceAddress(shortAddress, "SHORTTAG", "Long Tag ...",
                HartDeviceAddress.AddressingModeSelection.ShortAddress,
                new HartLongAddress()));
    stream.Close();
}
```

IEC

Figure 50 – Example: Reading of DTM data from a DTMDataSubset

The DTM vendor shall consider loading of data created by an “older” version of the DTM. Even if the format of data has not changed also the deserialization of data to new class versions shall be considered. In the example in Figure 50 this is achieved by setting the BinaryFormatter in an Assembly version insensitive mode.

5.12.4 Searching for data

The DTMDataSubset dictionaries provide several methods to find a particular DTMDataSubset:

– By Key

– By Descriptor

The optional DTMDataSubset.Descriptor property can be utilized by the DTM to implement advanced searching algorithms.

The content of the Descriptor property is DTM-specific. A DTM can use this property to provide further information about the DTMDataSubset content. Figure 51 shows an example how a DTM may save some trend data in the BulkData dictionary with additional descriptor information.

```
protected void SaveTrend(SomeTrendData someTrendData, DateTime createdAt)
{
    // start transaction (needed for adding of DTMDataSubset)
    _dataset.StartTransaction();

    // create binary formatter and serialize TrendData
    MemoryStream stream = new MemoryStream();
    BinaryFormatter binaryFormatter = new BinaryFormatter();
    binaryFormatter.Serialize(stream, someTrendData);
    stream.Close();

    // create trend data datasubset with Descriptor containing current date / time
    byte[] data = stream.GetBuffer();
    _dataset.BulkData.Add(Guid.NewGuid().ToString(), data,
                          "TrendData - " + createdAt.ToString("yyyy:MM:dd hh:mm:ss"));

    // close transaction with auto commit = true
    _dataset.CloseTransaction(true);
}
```

IEC

Figure 51 – Example: Creation of a BulkData.DTMDataSubset with descriptor

The Descriptor property shall be used by the DTM to search for specific DTMDataSubsets without reading the binary data. This enables the Frame Application to read the binary data from the persistence storage only if really needed by the DTM. Thus the searching algorithm is fast and has a low memory footprint.

Figure 52 shows an example on how a DTM can search for DTMDataSubsets with-specific Descriptors by using a .NET LINQ query.

```
protected List<SomeTrendData> GetTrendsOfDay(DateTime date)
{
    // (LINQ) query for all DTMDataSubsets containing trend data for a specific day
    IEnumerable<IDataSubset> dataSubsets = from item in _dataset.BulkData
                    where item.Value.Descriptor.Contains("TrendData - " +
                          date.ToString("yyyy:MM:dd"))
                    select item.Value;

    // deserialize found trend data and return list to caller
    List<SomeTrendData> trends = new List<SomeTrendData>();
    BinaryFormatter binaryFormatter = new BinaryFormatter();
    binaryFormatter.AssemblyFormat = FormatterAssemblyStyle.Simple;
    foreach (IDataSubset dataSubset in dataSubsets)
    {
        MemoryStream stream = new MemoryStream(dataSubset.ReadData());
        trends.Add(binaryFormatter.Deserialize(stream) as SomeTrendData);
    }
    return trends;
}
```

IEC

Figure 52 – Example: Searching for DTMDataSubsets with specific descriptor

5.13 Comparison of DTM and device data

5.13.1 Comparison of datasets using IDeviceData / IInstanceData

If a DTM does not provide the IComparison interface, then it shall publish all data relevant for comparison in the IDeviceData / IInstanceData interfaces (at least).

Some of the published data may not be relevant for comparison, for example dynamic data or process data. Therefore the Frame Application should provide means (e.g. user interface) to select data which is relevant for comparison.

The Frame Application shall read the data via the IDeviceData and/or the IInstanceData interface and compare the values of data items with the same identifier. If a data item with the same identifier is missing, then this shall be evaluated as not equal.

5.13.2 Comparison of datasets using IComparison

DTMs which do not publish all data shall implement the interface IComparison. If a DTM implements this interface, then the Frame Application shall use this interface for comparison.

The IComparison interface provides methods to compare:

- the currently persisted dataset with the data in the device (Online Comparison)
- the currently persisted dataset with another persisted datasets (Offline Comparison)

Be aware that a DTM can only compare a dataset which has a supported format (format ID is equal the current format ID or to a supported format ID). The comparison shall include only the dataset of the DTM. Related FDT Objects (e.g. Child DTMs or Parent DTMs) are not included in the comparison provided by IComparison.

If it is necessary to compare multiple DTMs, the Frame Application is responsible to execute the comparison method on all respective DTMs. For example the comparison of a Composite Device DTM may require also the comparison for the attached Module DTMs.

5.14 Tracing

For troubleshooting or debugging trace information (logging) is essential. Whenever multiple components need to interact it is of advantage if all components have a common place to put the trace information. This makes it easier to detect and resolve problems where several components are involved.

An FDT Frame Application shall implement a dedicated interface ITrace (see 6.2), which is used by DTMs to send trace messages.

A trace message can be either a human readable description or data as an array of objects. An array of objects is useful if a DTM developer wants to trace a complete exception object and not only a description. If a DTM sends a trace message with an array of objects it shall also provide a corresponding additional message with a human readable description.

A trace message includes an assessment of severity (e.g. verbose, warning or error) and a classification. To limit the amount of trace messages sent by a DTM an FDT Frame Application can set the minimum trace level using IDtm and IDtmUifunction.

How messages are collected, stored or displayed to the user is Frame Application-specific. It is not in the scope of this specification.

A trace message is not intended to be shown to the user directly. It is dedicated to debugging and troubleshooting. If a message is intended to be displayed to the user one of the message box methods of interface IFrameUi shall be used.

A trace message shall be in English. It shall not contain a timestamp, because the timestamp is provided by the Frame Application if necessary.

5.15 Report generation

5.15.1 General

Due to the shared responsibilities for data management in an FDT system, the generation of a comprehensive report requires the compilation of report fragments delivered by different components in the system. While the topological information is managed by the Frame

Application, all the device-specific information is to be delivered by the constituent DTMs in a project. To generate a report, a Frame Application uses the IReporting interface provided by DTMs to request report fragments with the device-specific presentation of configuration or parameterization data from each DTM.

5.15.2 Report types

Complex devices may have a huge amount of configuration and parameter information. Frame Applications shall be able to access only a subset of this data for the generation of context-specific reports, e.g. a report only of network management related data, offline or online data, IO signal information, bulk data, etc.

A DTM shall offer different types of reports, each covering a distinct subset of its device data. If the report corresponds to a DTM function (a Function ID or Application ID), it shall be referenced in the report type. A DTM may provide additional report types for specific purposes without reference to specific functions. The DTM informs the Frame Application by means of its ReportInfo property about the available report types. The list of available reports shall not change over the lifetime of a DTM BL instance.

A Frame Application may use only the report types with an associated Application ID to generate a standardized report on a specific aspect of a system, collect the data of all report types from all DTMs to create a full report or offer a user interface based on the ReportInfo properties to let the end user decide about which data to include in the report.

5.15.3 DTM report data format

A DTM shall deliver its report fragment in form of a strictly conforming XHTML (XTHML 1.0 strict) document as specified in [24] (see Figure 53).

NOTE Since XHTML 1.0 is a reformulation of HTML 4 conforming to the XML 1.0 standard, documents with this type of markup can be processed by any XML compliant tool or library. This includes the XSL transformation to paginated output formats like XSL-FO, that can be postprocessed for example to the ISO 19005 1 (PDF/A 1) or Rich Text Format (RTF). The final report format of a Frame Application is out of scope of this specification.

The report fragment shall not contain any script or style (CSS) elements nor frames. It shall be self contained, that is it shall contain all the mandatory parts of an XHTML document, so that any XHTML compliant rendering engine can display it without further modification:

- XML Prolog with character encoding declaration. The default character encoding is UTF 8. Note: different from the XHTML specification, this element is declared mandatory here to simplify the postprocessing with standard XML tools and libraries.
- Document type declaration (DOCTYPE) according to the XHTML 1.0 strict standard.
- root <html> element with XHTML 1.0 namespace declaration and declaration of the content language. The content language shall be the same the DTM uses in its BL and UI; xml:lang and lang-attributes shall always have the same value.
- <head> section with content type declaration including the character encoding. The encoding shall be equal to the encoding defined in the XML Prolog; the declaration in the prolog takes precedence. This declaration is for compatibility with older XHTML rendering engines.
- <body> section with presentation of the device-specific data. As any other XHTML document, report fragments may reference external resources, e.g. images.

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE html PUBLIC
   "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">

  <head>
    <meta http-equiv="content-type" content="text/html; charset=utf-8" />
    <title>Title of report fragment</title>
  </head>

  <body>
    … (Device specific data presentation here)
  </body>

</html>
```

IEC

Figure 53 – Skeleton of a DTM-specific report fragment

5.15.4 Report data exchange

DTM and Frame Application exchange report fragments by means of a file system folder which can be accessed by the DTM BL. When a Frame Application requests the report fragment for a device with a call to <GenerateReport()> on the IReporting interface of a DTM BL, it specifies a destination folder path, the Base-URI. A DTM BL shall store the report result with an arbitrary filename in the destination folder and return the filename to the Frame Application as the result of the asynchronous call.

If the report fragment needs to reference external resources, e.g. images, the DTM shall store them likewise under the specified Base-URI. The DTM is free to create subfolder under the Base-URI to organize the external resources. A report fragment shall always use relative references to link to its external resources. A DTM shall assume that the Base-URI is a temporary identifier – it is only valid until the DTM returns completion of the <GenerateReport()> call and may change between subsequent calls to <GenerateReport()>.

Frame Applications have to take appropriate measures to prevent name clashes between the URIs of report fragments and accordingly external resources of different DTMs; e.g. use different base URIs for the reports of individual instances. Furthermore they are responsible for disposal of the report fragments and external resources when they are not used any more. A typical implementation of a Frame Application creates an individual subfolder as Base-URI for each DTM to be included in the report.

5.16 Security

5.16.1 General

A Frame Application hosts DTM BL or DTM UI which are, from the Frames perspective, external components provided by third parties. Therefore the system is exposed to possibly unknown code. The system shall be protected against security threads originating from unknown code.

5.16.2 Strong naming of assemblies

Strong naming of assemblies allows for checking if an assembly was tampered after it was published.

All assemblies which are part of a DTM Setup shall have a strong name.

5.16.3 Identification of origin

Microsoft Authenticode [25] is a digital signature format to sign executable code, which allows i.e. checking the origin of an assembly using a public-key cryptography approach. Authenticode shall be used to allow for verification of origin and genuineness of a DTM.

DTM vendors shall obtain a code-signing certificate issued by one of the certification authorities (CAs) that are trusted by default in Windows. Such a CA is referred to as “Windows root certificate program member”. Updates of the trusted root certificates in Windows are automatically installed during Windows updates or can be downloaded from the Microsoft website (see [26]).

The following DTM assemblies (DTM binaries) shall be signed using a code-signing certificate:

- DTM BL assembly
- DTM UI assemblies
- DtmInfoBuilder if implemented in a separate assembly
- StaticFunction assemblies (if available)
- Installer application

Figure 54 shows how a Frame Application can verify the origin of a DTM assembly using the .NET Framework namespace System.Security.Cryptography.X509Certificates.

```
// Create an X.509 certificate from the signed DTM assembly
X509Certificate x509Certificate = X509Certificate.CreateFromSignedFile(
                                    @"C:\...\AuthenticodeSignedDemo.dll");

// Create a new X506Certificate2 instance by passing the previously created
// X506Certificate instance
X509Certificate2 x509Certificate2 = new X509Certificate2(x509Certificate);

// Check if the chain of the created X.509 certificate (represented by the
// x509Certificate2 instance) is valid
bool isX509ChainValid = x509Certificate2.Verify();
```

IEC

Figure 54 – Example: Authenticode check

5.16.4 Code access security

The .NET Framework provides a security mechanism referred to as “Code Access Security (CAS)” [27]. This mechanism allows to limit the access permissions (e.g. to file system, registry or network) of an assembly.

As this could mean limiting essential capabilities of a DTM, a Frame Application is not allowed to limit code access permissions. That means Code Access Security shall not be used.

5.16.5 Validation of FDT compliance certification

The FDT Group defines an FDT compliance certification procedure for DTMs.

FDT supports the means to enable a Frame Application to validate the compliance certification of a DTM. Certified DTMs shall install a conformity record file, which is generated by an authorized FDT certification laboratory and signed using a public-key cryptography approach.

The conformity record file is digitally signed using a private key which is only known to the FDT Group and authorized certification labs. Frame Applications can check the conformity record file by checking the signature using the corresponding public key of the FDT Group.

The certification record file shall be signed according to the W3C XML Signature Syntax and Processing recommendation [28].

A DTM certification record shall use the exactly same DTM vendor name as used in the Authenticode signatures of DTM BL and UI assemblies included in the DTM deployment package.

Figure 55 shows an example for a conformity record file. This is an xml serialized instance of the datatype “ConformityRecord”.

```
<?xml version="1.0" encoding="utf-8"?>
<ConformityRecord xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://schemas.datacontract.org/2004/07/Fdt.Deployment">
  <TestedDtmName>Name of DTM</TestedDtmName>
  <TestedDtmVersion xmlns:d2p1="http://schemas.datacontract.org/2004/07/System">
    <d2p1:_Build>0</d2p1:_Build>
    <d2p1:_Major>1</d2p1:_Major>
    <d2p1:_Minor>0</d2p1:_Minor>
    <d2p1:_Revision>1</d2p1:_Revision>
  </TestedDtmVersion>
  <TestedDtmId>6d0ffd65-0936-420e-9e40-42d039fd8a98</TestedDtmId>
  <TestedTypeId>00000000-0000-0000-0000-000000000000</TestedTypeId>
  <DateOfTest>2010-04-29T00:00:00</DateOfTest>
  <TestedOSVersion>
    <OSVersionNumber xmlns:d3p1="http://schemas.datacontract.org/2004/07/System">
      <d3p1:_Build>6002</d3p1:_Build>
      <d3p1:_Major>6</d3p1:_Major>
      <d3p1:_Minor>0</d3p1:_Minor>
      <d3p1:_Revision>131072</d3p1:_Revision>
    </OSVersionNumber>
    <ServicePack>Service Pack 2</ServicePack>
  </TestedOSVersion>
  <VendorName>Vendor Ltd.</VendorName>
  <TestLabName>AccreditedLabName</TestLabName>
  <Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
    <SignedInfo>
      <CanonicalizationMethod Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315" />
      <SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1" />
      <Reference URI="">
        <Transforms>
          <Transform Algorithm="http://www.w3.org/2000/09/xmldsig#enveloped-signature" />
        </Transforms>
        <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
        <DigestValue>JfVQR8CV/MuCpxyqItJCVNBIeM8=</DigestValue>
      </Reference>
    </SignedInfo>

<SignatureValue>IwCJLPq6r3zLc12Auk3ast8KrXXICLWmxHjWSYE611LpqQQPGFwgQP2aGhL38jNir9OwnKep1NX5gIZGLPMGw
SGsPq3giczAf6QN3aK0eJ28TDLXxXDKvz6f5HDSXT7iCjWwGwYi9JtkJxwKRmi1hOpURXxdN1NGkeaykl2ELTM=</SignatureVal
ue>
  </Signature>
</ConformityRecord>
```


IEC

Figure 55 – Example: Conformity record file

Figure 56 shows how a Frame Application can check a certification record file.

```
public static Boolean VerifyXmlFile(String fileName, X509Certificate2 certificate)
{
    // Create a new XML document.
    XmlDocument xmlDocument = new XmlDocument();

    // Load the passed XML file into the document.
    xmlDocument.Load(fileName);

    // Find the "Signature" node and create a new
    // XmlNodeList object.
    XmlNodeList nodeList = xmlDocument.GetElementsByTagName("Signature");
    if (nodeList == null || nodeList.Count != 1)
    {
        return false;
    }

    // Create a new SignedXml object and pass it
    // the XML document class.
    SignedXml signedXml = new SignedXml(xmlDocument);

    // Load the signature node.
    signedXml.LoadXml((XmlElement)nodeList[0]);

    // Check the signature and return the result.
    return signedXml.CheckSignature(certificate, false);
}
```

IEC

Figure 56 – Example: checking conformity record file

6 FDT Objects and interfaces

6.1 General

The FDT interface specification includes the following FDT Objects:

- DTM Business Logic
- Presentation objects
  - WPF Control
  - WinForms Control
  - Standalone Application
  - UI Command Function
- Communication Channel
- Frame Application

The behavior of these objects and their interfaces are described in this clause. Developers implementing DTMs or parts of Frame Application like storage or communication objects shall implement the functionality as defined in this clause.

This clause also references and defines expected behavior of FDT specific interfaces that FDT-compliant objects shall implement.

In order to describe the availability of interfaces for the different FDT objects, following abbreviations are used:

M: mandatory – the interface shall be provided

C: conditional – the interface shall be provided depending on conditions

O: optional – the interface may be provided based on product decisions

-: not allowed – the interface shall not be provided

6.2 Frame Application

The class diagram shown in Figure 57 shows the interfaces which shall be implemented by a Frame Application.

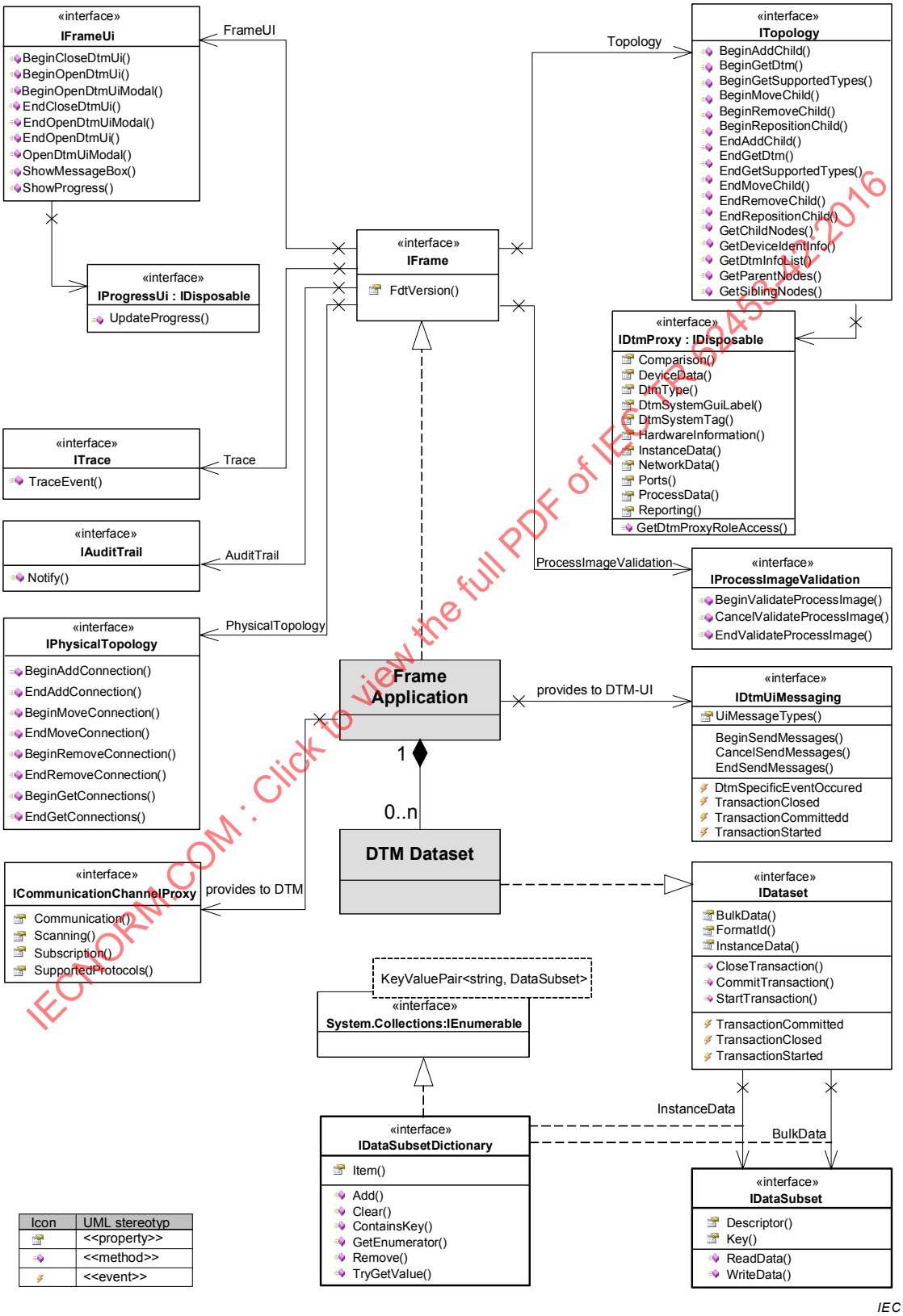


Figure 57 – Frame Application interfaces

The Frame Application implements the IFrame interface which is passed to the DTM Business Logic and the DTM User Interface. The properties of IFrame interface, named FrameUi, Topology, Trace and AuditTrail, provide access to the corresponding interfaces IFrameUi, ITopology, ITrace and IAuditTrail (see Table 5).

IFrameUi can be implemented in a separate user interface part of the Frame Application.

ICommunicationChannelProxy is implemented as part of the Frame Application and is provided to the DTM in IDtm.EnableCommunication().

DTMDataSet instances are provided from Frame Applications internal persistent storage for each device node (DTM instance) in the topology. The corresponding IDataset interface is implemented by frame-specific instances (shown as DTMDataSet class in the diagram) for each device node and passed to the DTM Business Logic. The DTMDataSubsets collected in DTM InstanceData or in DTM BulkData implement the interface IDataSubset.

Table 5 – Frame Application interfaces

Interface	Availability	Description
IAuditTrail	M	Interface used to receive audit trail events from DTMs in order to record changes and actions performed on a device.
ICommunicationChannelProxy	M	Proxy interface which enables a DTM to interact with the linked Communication Channel provided by the Parent DTM in the FDT topology.
IDataset	M	Interface used to read and store DTM instance-specific data in a dataset
IDataSubset	M	The DTMDataSubsets contain the actual DTM persistent data.
IDataSubsetDictionary	M	Represents a collection of data subsets.
IDisposable		.NET interface for disposable objects.
IDtmProxy	M	This interface is provided by DTM proxy objects. These objects enable a DTM to interact with another DTM instance (represented by the proxy object).
IFrame	M	The IFrame interface is the main interface of a Frame Application. It includes the services that shall be provided by the Frame Application to the DTM Business Logic and the DTM User Interfaces. The reference to this interface is passed to the DTM Business Logic and to the DTM User Interface in the call. The interface provides references to further interfaces.
IFrameUi	O	This interface provides access to the Frame Application user interface. A Frame Application that provides a user interface shall provide this interface. If the Frame Application does not provide this interface, the DTM knows explicitly that no GUI is available. A DTM shall be able to adapt to the situation where it can not show a user interface.
IPhysicalTopology	O	Interface used to manage physical connections between DTMs. The ability to manage physical connections depends on the availability of the IPorts interface at a DTM.
IProcessImageValidation	O	In some automation systems it is a requirement to apply changes to the process image while the PLC is running. This interface provides the methods needed to validate whether a potential change can be applied while the PLC is running.

Interface	Availability	Description
IProgressUi	O	Interface to a Frame Application progress user interface opened by IFrameUi.ShowProgress. A Frame Application that provides a user interface shall provide this interface. The interface can be used by DTM UI to show progress information. For asynchronous operations that are executed in the DTM BL the progress mechanism of extended AsyncResult pattern shall be used.
ITopology	M	This interface provides the access to the FDT topology. A DTM can request and release references to other DTM instances as well as create and remove Child DTMs.
ITrace	M	Trace interface that shall be used by DTMs to inform a Frame Application about trace message.

6.3 DTM Business Logic

6.3.1 DTM BL interfaces

The class diagrams shown in Figure 58 and Figure 59 show the interfaces, which shall be implemented by a DTM Business Logic class. IDtm is implemented by the DTM Business Logic and provides access to all other interfaces by corresponding properties.

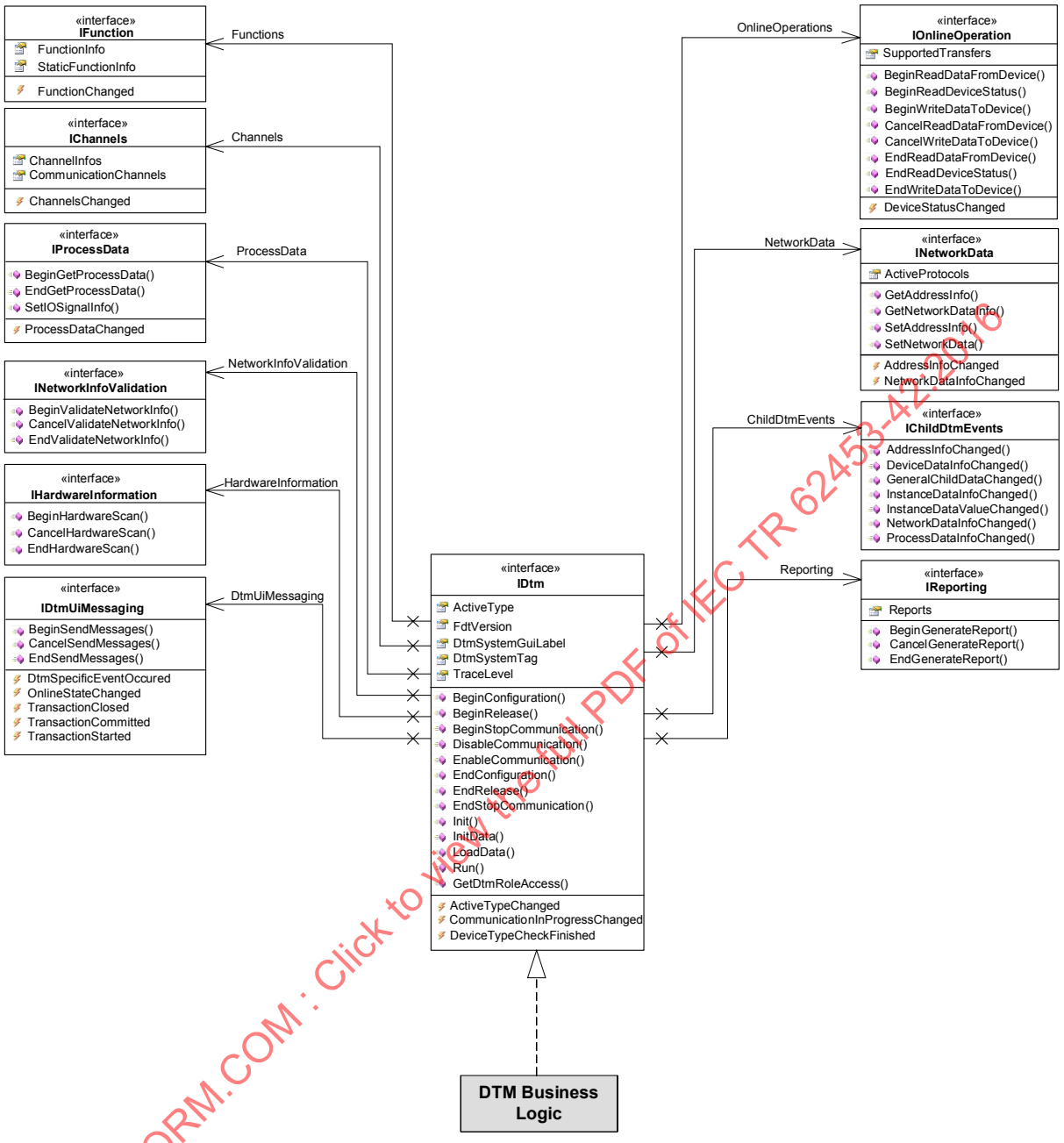


Figure 58 – DTM Business Logic interfaces (Part 1)

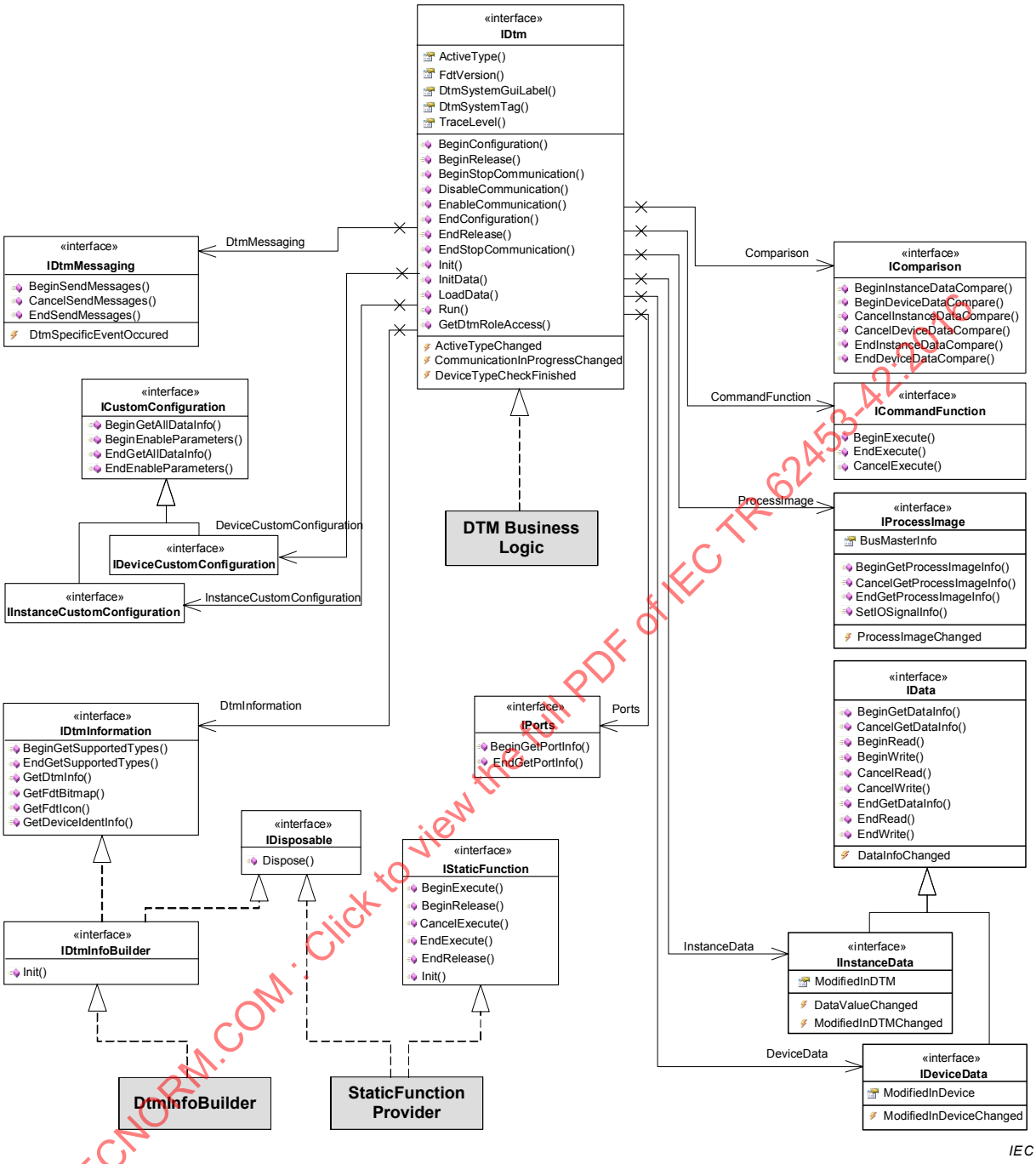


Figure 59 – DTM Business Logic interfaces (Part 2)

There is no state machine defined for DtmInfoBuilder instances. DtmInfoBuilder objects are created with new() and destroyed with Dispose().

Table 6 provides an overview on DTM interfaces, while Table 7 defines under which conditions interfaces shall be implemented.

Table 6 – DTM Business Logic interfaces

Interface	Description
IChannels	This interface is used for accessing the Communication Channel objects of a DTM.
IChildDtmEvents	Interface used by the Frame Application to inform the DTM about events occurred in a Child DTM in the FDT topology.
ICommandFunction	This interface is used to execute command functions.
IComparison	This interface allows a Frame Application to request the DTM to compare the dataset with another dataset or with the data in the physical device.
IDeviceData	This interface provides online access to specific parameters of a device.
IDtm	This is the main interface of a DTM. It defines the methods to control the DTM state-machine and general properties.
IDtmInformation	This interface provides general information about the DTM itself and the supported device types.
IDtmMessaging	This interface is used for interaction between the DTM Business Logic of two DTMs (Composite and Module DTM).
IDtmUiMessaging	Interface used for interaction between the Business Logic and DTM User Interfaces.
IFunction	This interface provides access to functions, user interfaces and documents provided by a DTM.
IHardwareInformation	This interface is used by Frame Application to request hardware information from a device.
IInstanceData	This interface provides access to DTM instance data parameters.
INetworkData	This interface provides network management relevant information
INetworkInfoValidation	In some automation systems it is a requirement to apply changes to the Network Info (which leads to a change in process image) while the PLC is running. This interface provides the needed methods to validate if a potential change can be applied while the PLC is running.
IOnlineOperation	This interface allows a Frame Application to request the DTM to exchange online data with the device.
IPorts	The interface allows to request a list of ports from the DTM.
IProcessData	This interface provides information related to process data of a field device for the integration of the device into the control system like datatype, signal direction, engineering units, and ranges etc.
IProcessImage	This interface provides access to the description of the process image provided by a fieldbus master.
IReporting	This interface is used to report the current instance or device dataset of a DTM (online data allowed here), e.g. for printing or documentation. The Frame Application may generate reports using IInstanceData/IDeviceData interfaces.
IStaticFunction	This interface is used to execute static functions independently of the DTM.
IDeviceCustomConfiguration IInstanceCustomConfiguration	These are optional interfaces . Only DTMs that allow customization of parameter access for User Level “Expert” need to implement these interfaces. These interfaces are supported only when the DTM is in the running state, before any function is invoked on the DTM. In all other states, the DTM shall restrict access to these interfaces.

Table 7 – Availability of interfaces depending of type of DTM

Interface	Condition	Device DTM	Communication DTM	Gateway DTM	Composite Device DTM	Module DTM	BTM
IChannels	Interface shall be provided by all DTMs that provide communication access to other DTMs.	O *)	M	M	M	O *)	-
ICommandFunction		O	O	O	O	O	O
IComparison	Interface shall be provided if not all parameters of the DTM/device can be accessed by IInstanceData /IDeviceData interfaces.	C	C	C	C	C	C
IDeviceData	Interface shall be provided for all devices which have online data.	C	C	C	C	C	C
IDtm		M	M	M	M	M	M
IDtmInformation		M	M	M	M	M	M
IDtmMessaging	May be implemented in case a tight coupling between two DTMs of the same vendor is required.	O	O	O	O	O	O
IDtmUiMessaging	Interface shall be provided for DTMs with user interfaces.	C	C	C	C	C	C
IFunction		M	M	M	M	M	M
IHardwareInformation		M	M	M	M	M	M
IInstanceData		M	M	M	M	M	M
INetworkData		M	M	M	M	M	M
INetworkInfoValidation	Only implemented by DTMs which represent a fieldbus Master and which are used in automation systems with specific requirements.		O	O			
IOnlineOperation	Interface shall be provided for all devices which have online data and shall be loaded during commissioning.	C	O	O	C	C	C
IPorts	The protocol-specific specification annex defines the rules for this interface.	M	M	M	M	M	-
IProcessData	The protocol-specific specification annex, defines the rules for this interface. If the protocol supports process data and the respective device provides process data the DTM shall provide this interface.	C	C	C	C	-	-
IProcessImage	Interface shall be provided by Communication /Gateway-DTMs that provide the layout of a process image of a master device.	-	C	C	-	-	-
IReporting	Interface shall be provided to support advanced reporting capabilities.	M	M	M	M	M	M

Interface	Condition	Device DTM	Communication DTM	Gateway DTM	Composite Device DTM	Module DTM	BTM
IDeviceCustomConfiguration IInstanceCustomConfiguration		O	O	O	O	O	O
*) Optional for DeviceDTM and ModuleDTM for instance because of possible BTM support.							

For a DTM, which has set the flag TypeInfo.CommunicationSupport to value 'PassiveDevice', all interfaces related to communication (i.e. IChannels, IDeviceData, IHardwareInformation, IOnlineOperation, IProcessImage) shall not be supported.

6.3.2 State machines related to DTM BL

6.3.2.1 General

The following state machines describe the behavior of the DTM in regard to its interfaces. The states are defined mainly to describe how a DTM is guided through different stages by a Frame Application and which interface methods can be used at a specific stage of the lifetime of a DTM instance. The state machine is based on the general state machine as defined in IEC 62453-2. It is extended to accommodate the specific needs of .NET based implementation. The state machines provided here are intended as a general specification for all types of DTM and are not intended as implementation design (e.g. in order to implement the "{enter state}" triggers an implementation might define additional states).

For information on which interface methods can be used at specific states refer to 6.6.

6.3.2.2 DTM state machine

The DTM State Machine in Figure 60 shows the states and transitions that are controlled by a Frame Application (the Frame Application has full control).

The diagram uses following notation:

- Method(): denotes a method used as a trigger for a state transition, the transition taken only, when the respective method returns
- [condition expression]: denotes a condition (guard) that has to evaluate true for the transition to be taken
- <method_name>: denotes an asynchronous operation

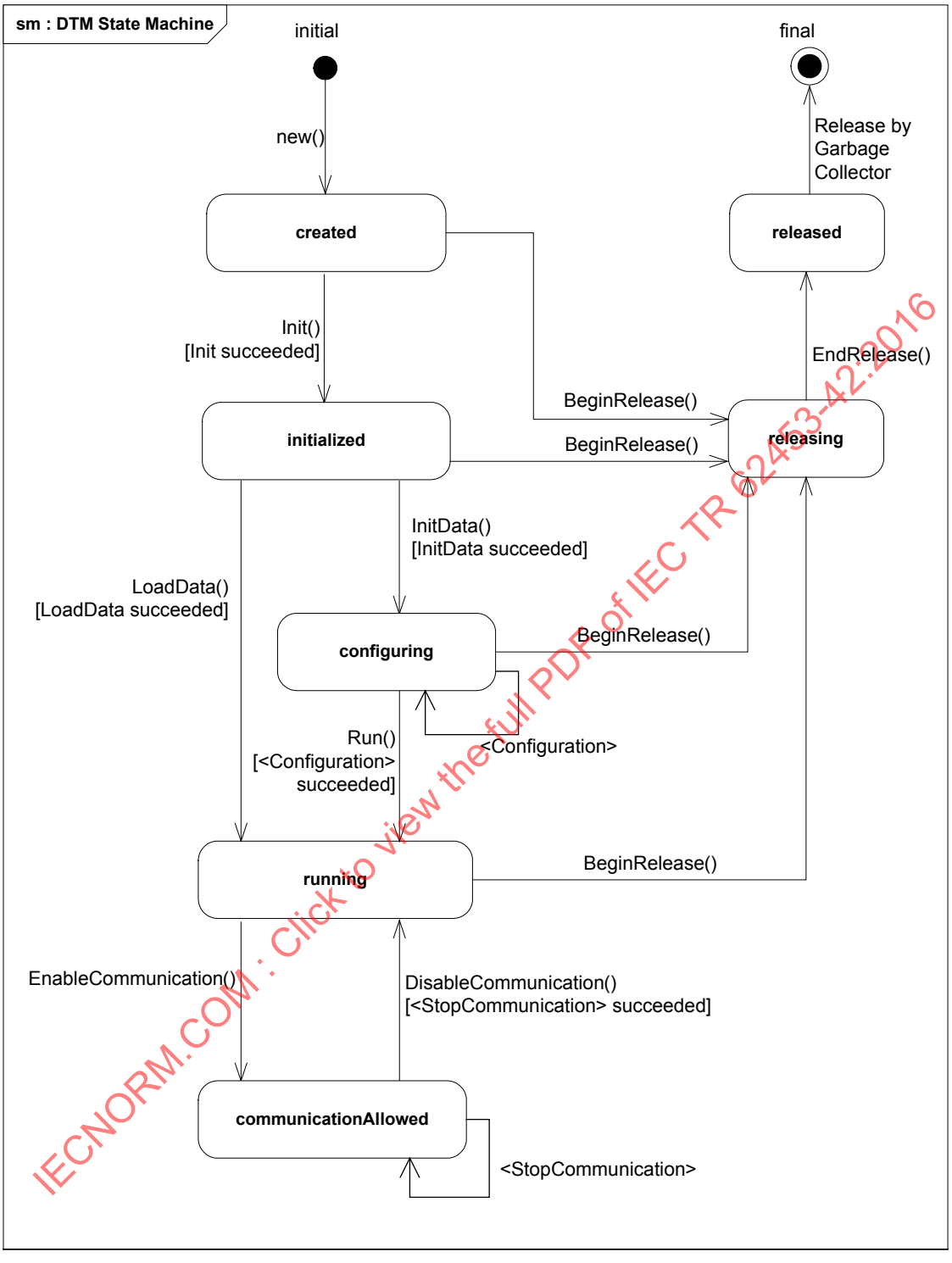


Figure 60 – State machine of DTM BL

Table 8 provides a description of the state transitions with their conditions and actions.

Table 8 – Definition of DTM BL state machine

#	Start state	End state	Trigger	Condition
1	initial	created	new()	new() succeeded
2	created	initialized	Init()	Init() succeeded
3	created	releasing	BeginRelease()	BeginRelease() succeeded
4	initialized	running	LoadData()	LoadData() succeeded
5	initialized	configuring	InitData()	InitData() succeeded
6	initialized	releasing	BeginRelease()	BeginRelease() succeeded
7	configuring	configuring	BeginConfiguration()/ EndConfiguration()	<Configuration> succeeded
8	configuring	running	Run()	<Configuration> succeeded and Run() succeeded
9	configuring	releasing	BeginRelease()	BeginRelease() succeeded
10	running	communication Allowed	EnableCommunication()	Reference to parent CommunicationChannel is valid and EnableCommunication() succeeded
11	running	releasing	BeginRelease()	all user interfaces are closed, all operations are finished
12	communication Allowed	communication Allowed	BeginStopCommunication()/ EndStopCommunication()	<StopCommunication> succeeded
13	communication Allowed	running	DisableCommunication()	<StopCommunication> succeeded and DisableCommunication() succeeded
14	releasing	released	EndRelease() *)	
	released	final	Removal by .NET GarbageCollector	
*) This transition is taken, even if the method failed.				

6.3.2.3 Online state machine

The following state machine (Figure 61) shows the internal states of state “communicationAllowed”. The DTM controls the internal states according to this state machine. The DTM does not expose the substate, but fires events which inform the Frame Application about internal state transitions (OnlineStateChanged event). In order to prepare an exit from the state “communicationAllowed”, the Frame Application performs the asynchronous <StopCommunication()> operation. The actual exit from state “communicationAllowed” is triggered by DisableCommunication() (see Table 8)

The state machine is used to define state dependent interface and method availability in 6.6.

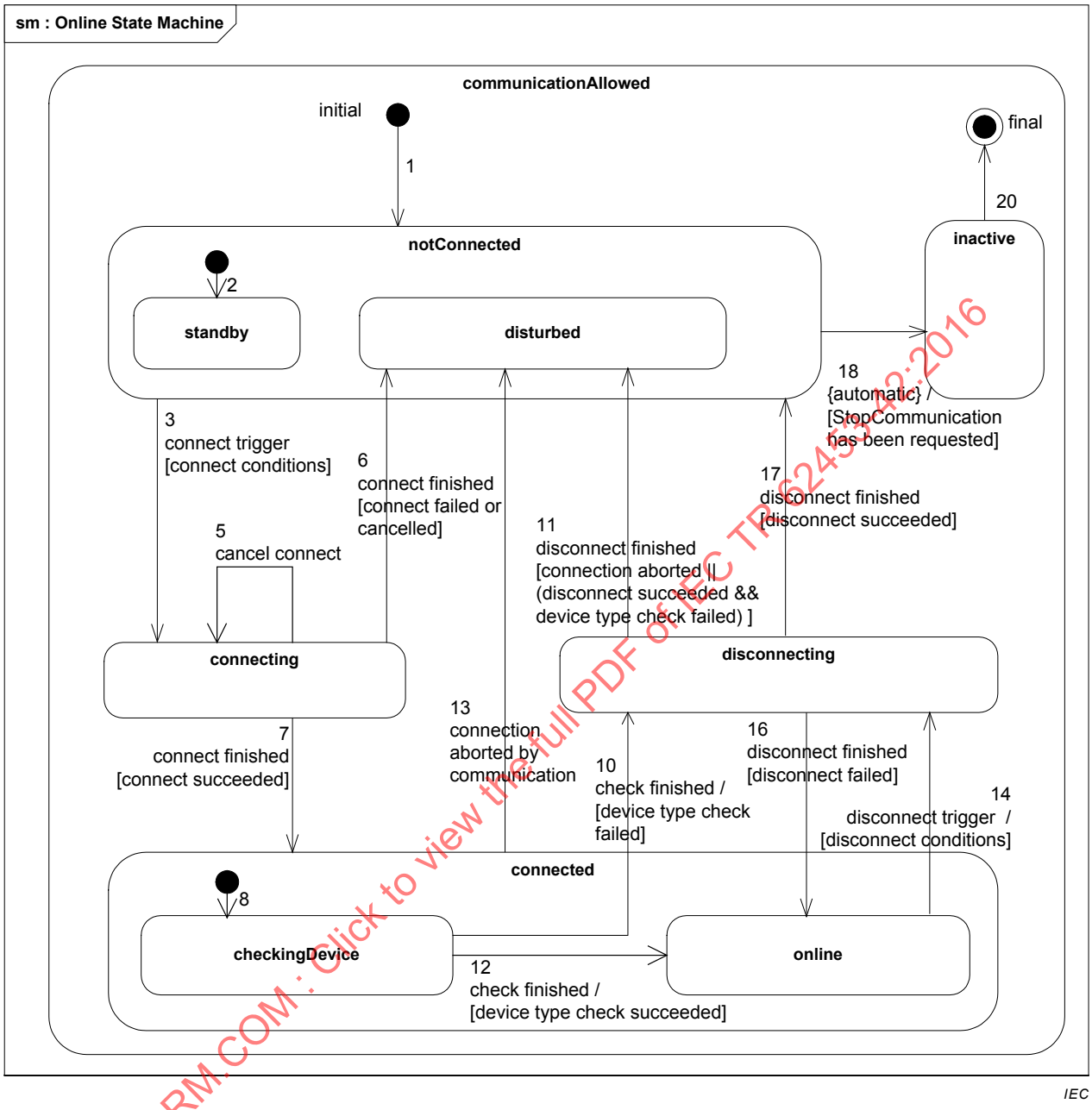


Figure 61 – Online state machine of DTM

Additional to the diagram notation explained above, Table 9 also shows triggers, that occur automatically.

The trigger {automatic} is a trigger, that activates automatically after the start state of the transition has been reached (spontaneous transition).

The trigger {enter state} fires when a state is reached.

These triggers are not associated to specific transitions, but fire every time, when a transition leads into the state. That is why those triggers are shown in the table without number and only with the start state.

Table 9 – Definition of online state machine

#	Start state	End state	Trigger	Condition	Action
1	communication Allowed	notConnected	{automatic}		
2	notConnected	standby (notConnected Standby)	{ automatic }		raise OnlineStateChanged(notConnectedStandby)
3	notConnected	connecting	connect trigger Several triggers possible: For Child DTMs: - Immediate connect because “StayConnected” was requested in EnableCommunication() - online function started(e.g. Download or Online-GUI) - reconnect after lost connection For CommDTM: - Immediate connect because “StayConnected” was requested in EnableCommunication() - online function started(e.g. Scan, Download or Online-GUI) - Child DTM requested connection+)	<StopCommunication> has not been called	
4	connecting++)		{enter state}		raise OnlineStateChanged (connecting) initiate connection: For Child DTMs: Call <Connect()>on parent channel For Comm DTMs+): Use driver API to connect.
5	connecting	connecting	cancel connect		CancelConnect()+)
6	connecting	disturbed (notConnected Disturbed)	connect finished	connect failed or cancel succeeded	raise OnlineStateChanged(NotConnectedDisturbed)
7	connecting	connected	connect finished	connect succeeded	
8	connected	Checking Device (connectedCheckingDevice)	{automatic}		raise OnlineStateChanged(ConnectedCheckingDevice)
9	checkingDevice		{enter state}		If device has not been checked, perform device type check. *)
10	checkingDevice	disconnecting	device type check finished	device type check failed	raise OnlineStateChanged (Disconnecting) raise DeviceTypeCheckFinished(UnsupportedDevice)

#	Start state	End state	Trigger	Condition	Action
11	disconnecting	Disturbed (notConnected Disturbed)	disconnect finished	connection aborted by communication || (disconnect succeeded && device type check failed)	
12	checkingDevice (connectedCheckingDevice)	online (connected Online)	device type check finished	device type check succeeded	raise OnlineStateChanged(ConnectedOnline) raise DeviceTypeCheckFinished(SupportedDevice)
13	connected	notConnected	connection aborted by communication Abort notification received from parent Communication Channel+)		For Child DTMs: Handle pending transactions or active online functions. **) For ParentDTMs: Abort all child connections
14	connected	disconnecting	disconnect trigger Several triggers possible: - all online functions finished (and DTM is in ConnectionMode “OnDemand”) - call to <StopCommunication> has been received - Child DTM calls <Disconnect>		
15	disconnecting		{enter state}		raise OnlineStateChanged(Disconnecting) terminate connection: For Child DTMs: Call <Disconnect()> on parent channel For Comm DTMs+): Use driver API to disconnect.
16	disconnecting	online (connected Online)	disconnect finished	disconnect failed	raise OnlineStateChanged(ConnectedOnline)
17	disconnecting	notConnected	disconnect finished	disconnect succeeded	
18	notConnected	inactive	{automatic}	<StopCommunication> has been requested	
19	inactive		{enter state}		raise OnlineStateChanged(Inactive)
20	inactive	final	{automatic}		<StopCommunication> completed callback

NOTES:

*) Device type check means that the DTM checks if it is connected to the correct device type. Device type check shall be performed at least once when state “checkingDevice” is entered the first time. The Frame Application receives an event DeviceTypeCheckFinished, after the device type check has been performed. For some devices a device type check may not be feasible. In this case, the DTM shall raise DeviceTypeCheckFinished event with ‘NotChecked’ value.

**) Asynchronous operations shall always be finished by calling the ‘Completed’ callback method. If a connection is aborted, each aborted transaction will raise an FdtConnectionAbortedException in its ‘End’-method.

+)	Communication DTMs do not call <Connect> or <Disconnect> as they do not have a parent Communication Channel. Instead Communication DTMs work on a driver API. For the same reason, the abort notification is not valid for Communication DTMs, but a Communication DTM may receive a similar notification from the driver.
++)	When <StopCommunication> is called in state 'connecting', then the connection establishment is finished and <StopCommunication> is handled in the following state.

6.3.3 State machine of instance data

6.3.3.1 General

A DTM BL shall expose the state of the actual instance data to the Frame Application in order to support Frame Applications in synchronizing DTM datasets with their respective devices. Two properties reflect the possible states of the data (instance data and online data) in regard to modifications (see Figure 62):

- modification in DTM: IInstanceData.ModifiedInDtm (see state machine in Figure 63) reflects changes in the instance data and
- modification in device: IDeviceData.ModifiedInDevice (see state machine in Figure 64) reflects changes in the online data.

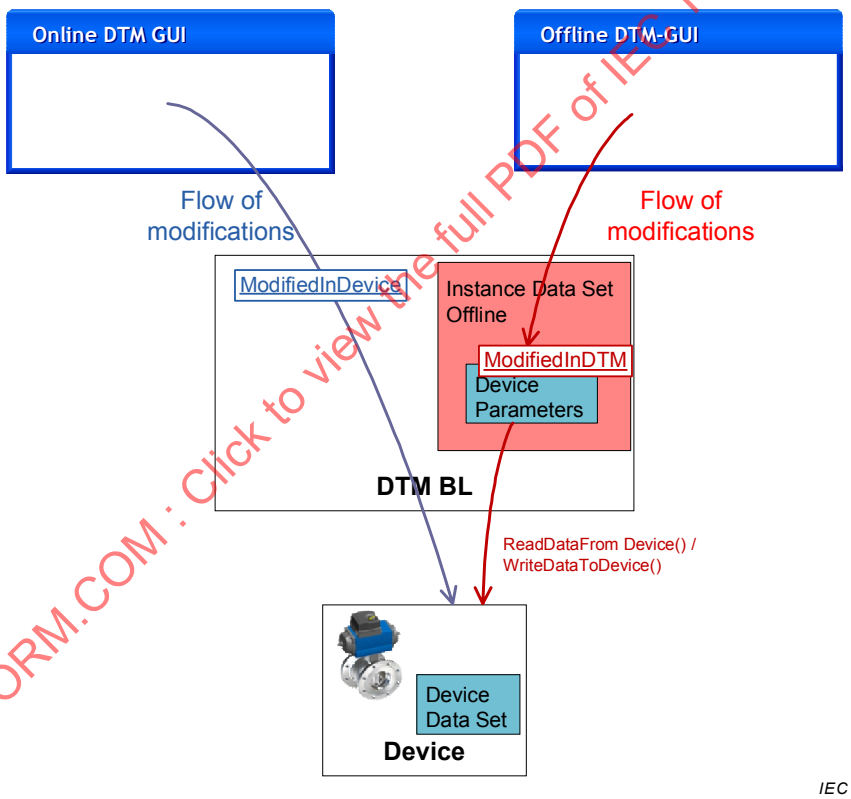


Figure 62 – Modifications of data through a DTM

NOTE For description of the concept of Instance Data and Device Data see 4.12.1.

If the DTM supports the methods <ReadDataFromDevice()> and <WriteDataToDevice()>, then the Frame Application may use these methods for synchronization of Instance Data Set and Device Data Set. A DTM indicates in the property IOnlineOperation.SupportedTransfers whether the respective device supports these methods.

6.3.3.2 Modifications in DTM

The property ModifiedInDtm can be used by a Frame Application to detect which DTMs have modification of offline data that are not synchronized with the respective device.

The state “default” indicates the initial status of the dataset (after InitData()).

Any offline modification of device parameters will lead to a state not equal to “default” (device parameters here means subset of offline data that is synchronized with the device).

<ReadDataFromDevice()> or <WriteDataToDevice()> change the state to “dataLoaded”.

The state shall be exposed in IInstanceData property ModifiedInDtm and shall be read only (see Figure 63). The DTM shall include the state in its persisted instance dataset and set the state accordingly in LoadData(). When the state changes, the DTM fires an IInstanceData.ModifiedInDtmChanged() event.

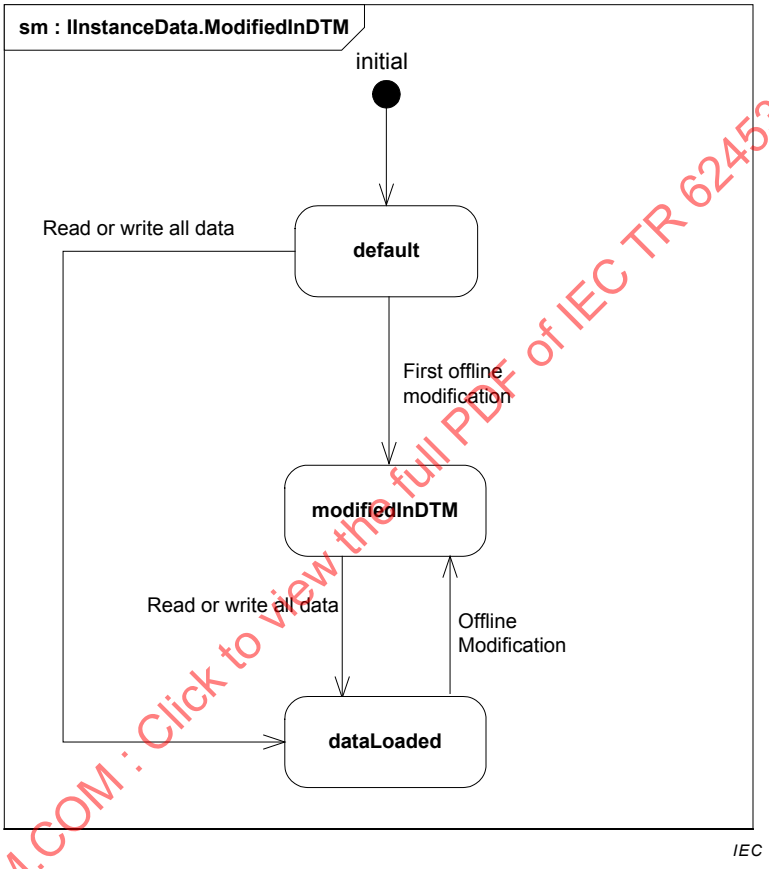


Figure 63 – ModifiedInDtm: State machine of instance data

The meaning of the different states can be seen in Table 10.

Table 10 – Description of instance dataset states

State	Meaning
default	This state is set after creation of a new instance dataset in InitData(). The state is only valid if the newly created dataset contains enough information to establish a proper communication.
modifiedInDTM	The offline instance dataset is modified and not synchronized with the device.
dataLoaded	The offline instance dataset has been synchronized with the device. No further change has been executed on the instance dataset since the synchronization.

6.3.3.3 Modifications in device

The property ModifiedInDevice can be used by a Frame Application to detect which DTMs have modified the data in the device and have not synchronized their DTM dataset. Any change to the device, which is performed or recognized by the DTM will lead to a state “modifiedOnline”.

NOTE The status "modifiedOnline" is intended to indicate all changes in data intended to configure the device. It is not intended to reflect changes in dynamic values (e.g. operating hours).

The state shall be exposed in IDeviceData property ModifiedInDevice and shall be read only. (see Figure 64) The state shall be included in the persisted instance dataset. When the state changes, an IDeviceData.ModifiedInDeviceChanged() event is fired.

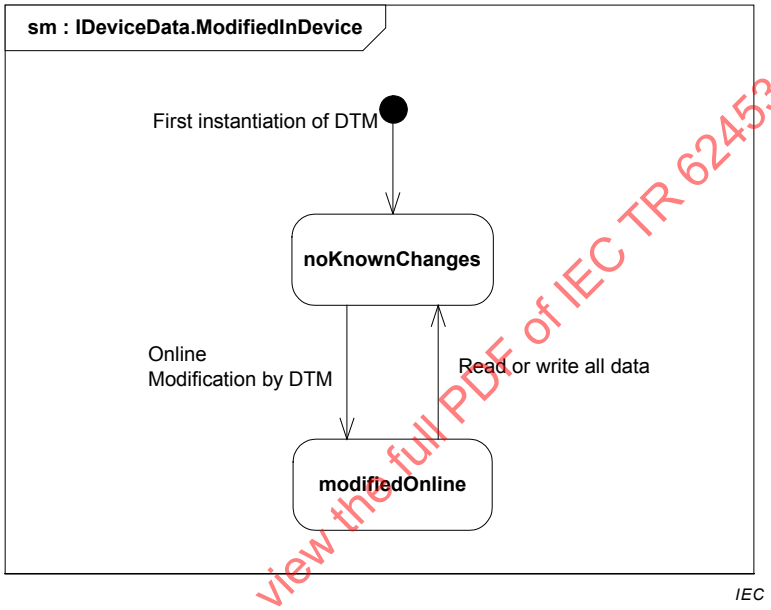


Figure 64 – ModifiedInDevice: State machine related to device data

The meaning of the different states can be seen in Table 11.

Table 11 – Description of dataset states regarding online modifications

State	Meaning
noKnownChanges	The dataset state regarding the device is unknown because the DTM was not connected to the device or the DTM has synchronized at some point of time with the device. The dataset has been uploaded (<ReadDataFromDevice()> or downloaded (<WriteDataToDevice()>). No further change has been executed on the device by the DTM. But there may be changes on the device, which were triggered from other sources.
modifiedOnline	Parameters have been changed in the device but not in instance dataset (E.g.: see use case Online parameterization, IDeviceData interface definition) ‘modifiedOnline’ status shall be set only once in case the data in the device has been changed by the DTM. In case of successful Upload or Download of complete dataset, the state shall be set to “noKnownChanges”.

Data in the device can also be modified directly by a tool out of the scope of the FDT. In this case, it is recommended not to set the status to ‘modifiedOnline’.

If an application is started, which may need to change the state in ModifiedInDevice (the property is part of the instance dataset and can not be changed when the dataset is not locked), then the dataset shall be locked (StartTransaction()).

For special operations it is useful to keep the device configuration and the instance dataset in sync. Therefore it is strongly recommended that the DTM should ask the user whether the data should be synchronized. This is necessary for user interface functions like Online Parameterization and Offline Parameterization (see Table A.4).

IDeviceData methods shall not modify the instance dataset, but shall set the state in ModifiedInDevice.

6.4 DTM User Interface

The class diagram shown in Figure 65 shows the interfaces, which shall be implemented by the different DTM User Interface classes and controls.

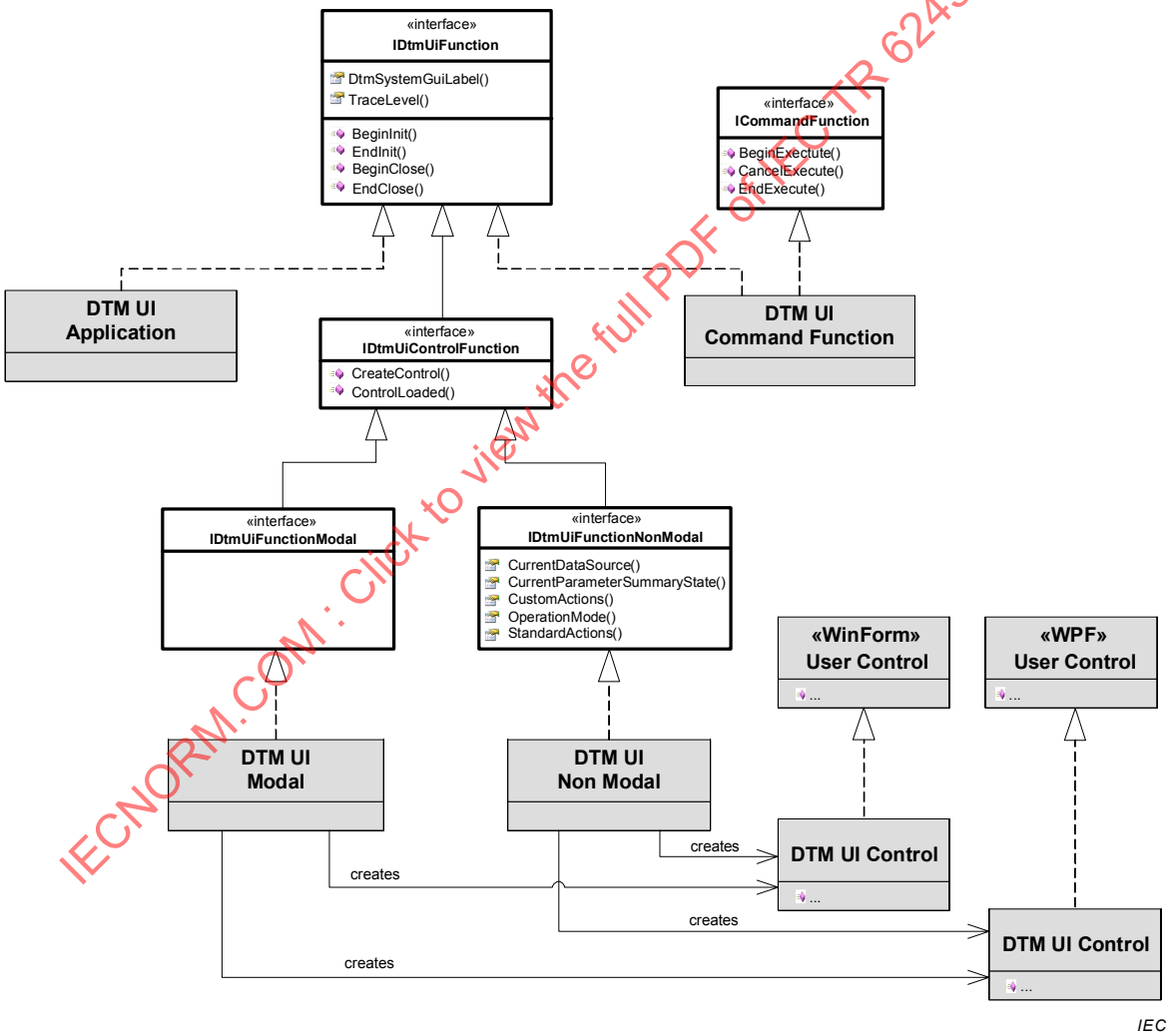


Figure 65 – DTM UI interfaces

FDT supports following DTM User Interface types:

- WPF Controls can be embedded into the user interface of the Frame Application. These controls shall derive from the standard .NET WPF User Control class (namespace System.Windows).

- WinForms Controls can be embedded into the user interface of the Frame Application. These controls shall derive from the standard WinForms User Control class (namespace System.Windows.Forms).
- Applications are external DTM-specific user interfaces (e.g. executable applications) which can not be embedded into the Frame Application. These are represented by simple .NET classes (called “DTM UI Application”) which may be used by the Frame Application to control the external user interface via the interface IDtmUiFunction.
- UiCommandFunctions are similar to the command functions which can be executed at the DTM Business Logic, but UiCommandFunctions are allowed to open own user interfaces (e.g. dialog boxes, private dialogs etc.). Such functions are represented by simple .NET classes which contain the code to execute.

The different DTM UI types can be accessed by API interfaces (Table 12):

Table 12 – DTM UI interfaces

Interface	Availability	Description
ICommandFunction	O	This interface is used to execute command functions (same interface as implemented by the DTM Business Logic)
IDisposable	M	.NET interface for disposable objects.
IDtmUiFunction	M	This is the main interface of a DTM UI function.
IDtmUiFunctionModal	O	This interface is implemented by DTM UIs which are executed modal.
IDtmUiFunctionNonModal	O	This interface is implemented by DTM UIs which are executed modless.

6.5 Communication Channel

The following class diagram (Figure 66) shows the interfaces, which shall be implemented by a Communication Channel. A Communication Channel implements the main interface ICommunicationChannel and the interfaces ICommunication, ISubscription, IScanning ,and ISubTopology, which are accessible by corresponding properties of ICommunicationChannel.

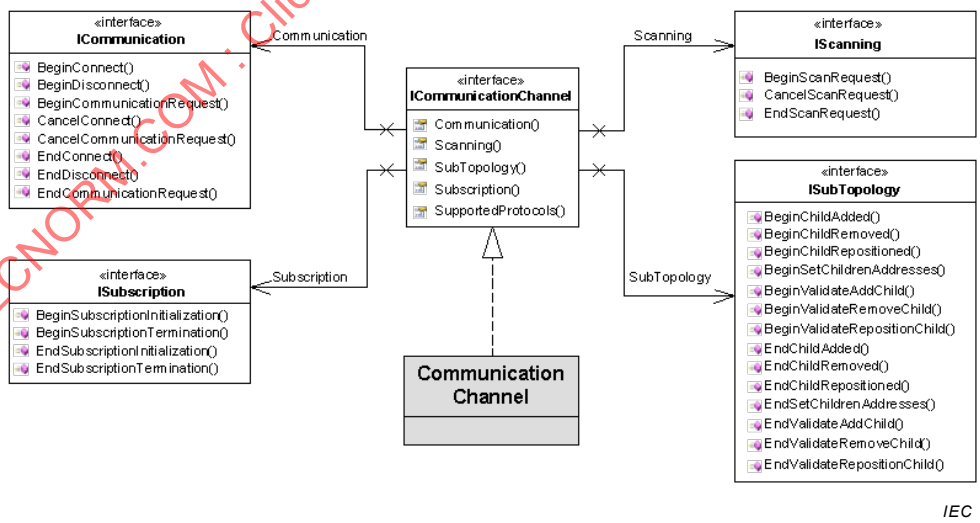


Figure 66 – Communication Channel interfaces

Table 13 provides an overview on the interfaces of a Communication Channel.

Table 13 – Communication Channel interfaces

Interface	Availability	Description
ICommunicationChannel	M	This is the main interface of a Communication Channel. It provides access to all other channel interfaces and to channel related information (e.g. supported protocols).
ICommunication	M	This interface is the communication entry point of a channel.
IScanning	C	This interface is used to request a scan of the sub-topology of a Communication Channel. This interface shall be provided for communication protocols that support scanning. The corresponding protocol annex shall define whether this interface is mandatory or not.
ISubscription	C	This interface extends the communication entry point of a channel with device initiated data transfer functionality. This interface should be provided for communication protocols that allow for device initiated data transfer. The corresponding protocol annex shall define whether this interface is mandatory or not.
ISubTopology	M	This interface provides methods for management of the sub-topology for a Communication Channel.

6.6 Availability of interface methods

Frame Application interfaces can always be called from other FDT objects as soon as the Frame Application provides access to these interfaces.

The availability of interface methods of the DTM-related objects may depend on the state of the DTM instance.

Table 14 defines the interfaces of a DTM BL which can be used by a Frame Application at the shown states.

Table 14 – Availability of DTM BL methods in different states

Interface / Method	created	initialized	configuring	running	communicationAllowed	releasing	released
IChannels [+])			X	X	X		
ICommandFunction				X	X		
IComparison:<InstanceDataCompare>				X	X		
IComparison:<DeviceDataCompare>					X		
IDeviceData							
<GetDataInfo()>				X	X		
(all other methods)					X		
IDtm [+])							
Init()	X						
BeginRelease()	X	X	X	X			
LoadData()		X					

Interface / Method	created	initialized	configuring	running	communicationAllowed	releasing	released
InitData()		X					
BeginConfiguration()/ EndConfiguration()			X				
Run()			X				
DtmSystemGuiLabel, DtmSystemTag, FdtVersion, TraceLevel		X	X	X	X	X	
All other methods / interface properties				X	X	X	
IDtmInformation [+])	X	X	X	X	X		
IDtmMessaging				X	X		
IDtmUiMessaging				X	X		
IFunction				X	X		
IHardwareInformation					X		
IInstanceData				X	X		
INetworkData				X	X		
INetworkInfoValidation				X	X		
IOnlineOperation					X		
IPorts				X	X		
IProcessData				X	X		
IProcessImage				X	X		
IReporting				X	X		
IDeviceCustomConfiguration IInstanceCustomConfiguration				X			

[+]) The Frame Application shall not subscribe to events before the DTM is in state 'running'

Communication Channel interfaces can always be called from other FDT objects as soon as the Communication Channel provides access to these interfaces.

DTM UI interfaces can always be called from other FDT objects as soon as the DTM UI provides access to these interfaces.

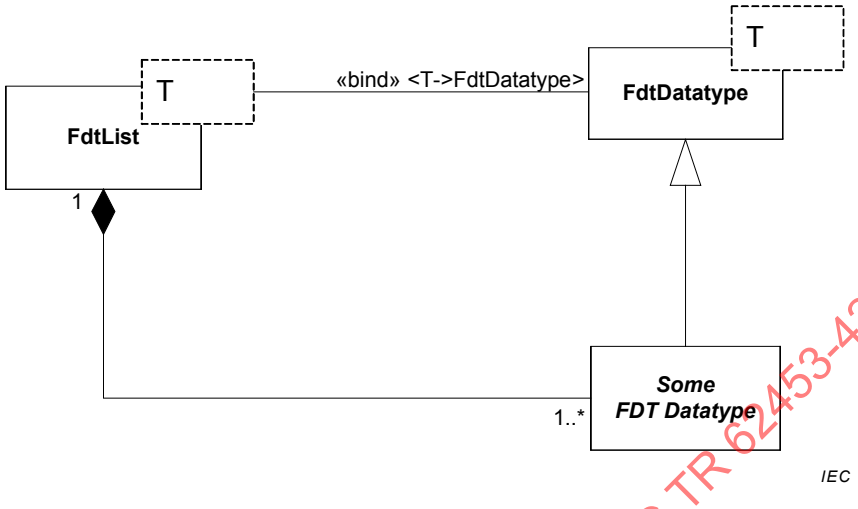
7 FDT datatypes

7.1 General

Datatypes are defined in Annex B. This clause provides an overview on top-level datatypes and how they are used. This clause (figures and tables) does not provide the complete datatype definition; please refer to Annex B for a complete datatype definition.

7.2 Datatypes – Base

FDT defines two basic datatypes: FdtDatatype and FdtList<>. Figure 67 shows examples how FdtDatatype and FdtList<> are used.



Used in:

-/-

Figure 67 – FdtDatatype and FdtList

Table 15 describes the base datatypes.

Table 15 – FDT base datatypes

Datatype	Description
FdtDatatype	Base class for all FDT datatype classes. The class provides the base implementation for the Verify() and Clone() methods. The type parameter T is always set to the derived class and is used to control these methods: • Verify() checks whether all properties are valid (e.g. mandatory properties have a value etc.). • Clone() creates a new object that is a deep-copy of the instance. All objects are duplicated – the top-level objects are duplicated as well as all the lower levels.
FdtList<>	Generic list of FdtDatatypes. The type parameter T defines the type of the list elements, FdtList<> is derived from System.Collections.Generic.List. and provides all functions of List. The Verify() method enforces the rule that the list shall not be empty. If an empty list shall be represented, the respective member shall return 'null'. Like FdtDatatype the FdtList<> also provides the methods Verify() and Clone().

7.3 General datatypes

General FDT datatypes are used in various other FDT datatypes.

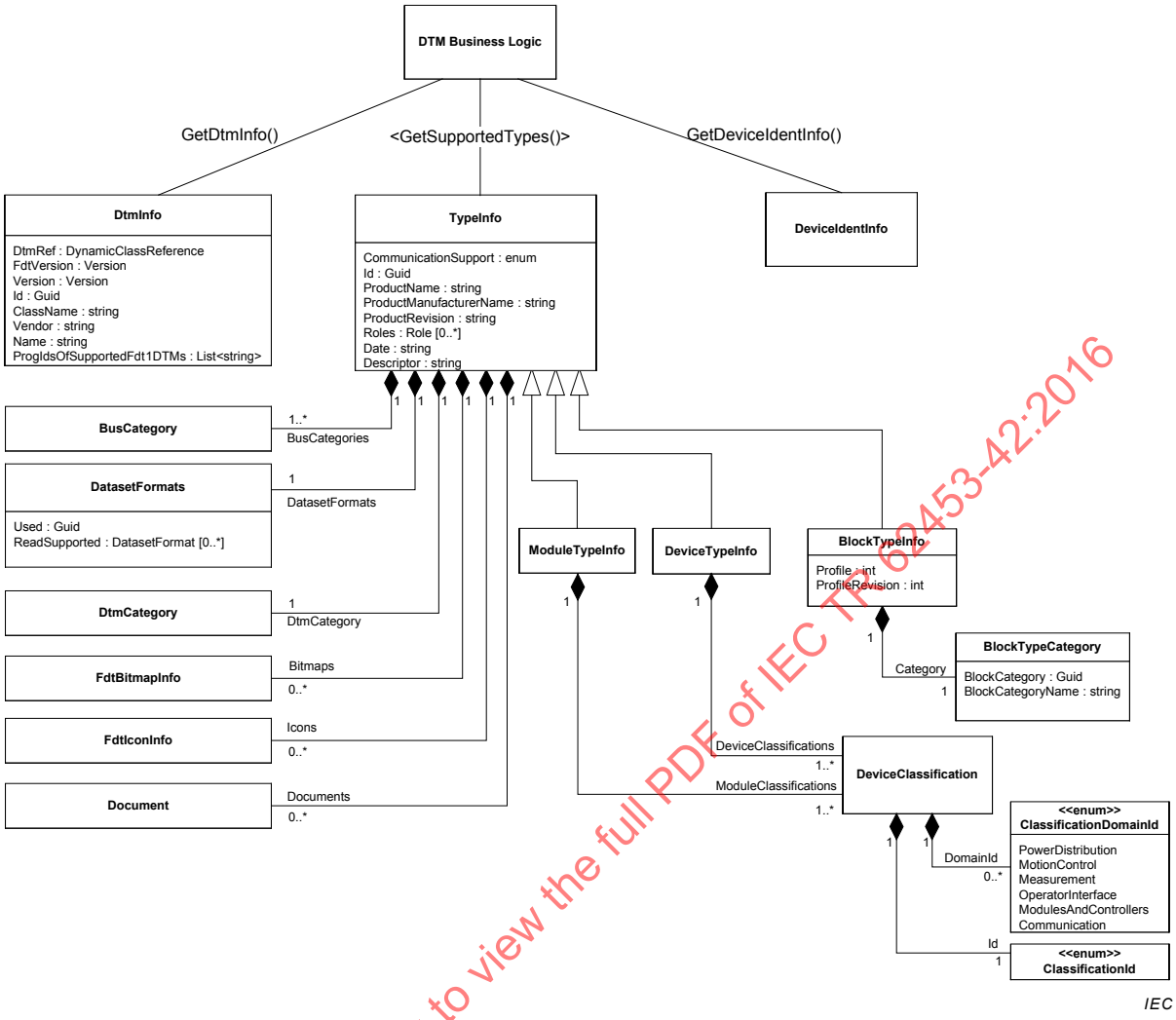
Table 16 lists and describes the general FDT datatypes

Table 16 – FDT General datatypes

Datatype name	Description
BusCategory	A bus category is a Universally Unique Identifier for a fieldbus protocol (or a point-to-point communication protocol). A property indicates whether the bus category is 'supported' or 'required'.
ChannelReference	Unique identifier of a Communication Channel provided by a DTM.
InvokeId	Unique identifier for an opened user interface.
PhysicalLayer	Unique identifier for a physical layer of a fieldbus like PROFIBUS PA.
PortReference	Unique identifier of a Port provided by a DTM.
ProgressInfo	Information about progress of an operation.
ProgressInfo<T>	Intermediated result and information about progress of an operation.
ProtocolInfoAttribute	This attribute class exposes general information about a protocol-specific assembly.
SemanticInfo	This class provides semantic information for a data object. For a range of predefined SemanticIds see Annex J.
UserInfo	Description of the user level including information about permissions, current session etc.

7.4 Datatypes – DtmInfo / TypeInfo

The class diagram shown in Figure 68 describes the relations of DtmInfo, TypeInfo and associated classes.



Used in:

IDtmInformation.GetDtmInfo()

IDtmInformation.BeginGetSupportedTypes() / IDtmInformation.EndGetSupportedTypes()

IDtmInformation.GetDeviceIdentInfo()

IDtm.ActiveType

Figure 68 – DtmInfo / TypeInfo – datatypes

Table 17 describes datatypes related to DtmInfo.

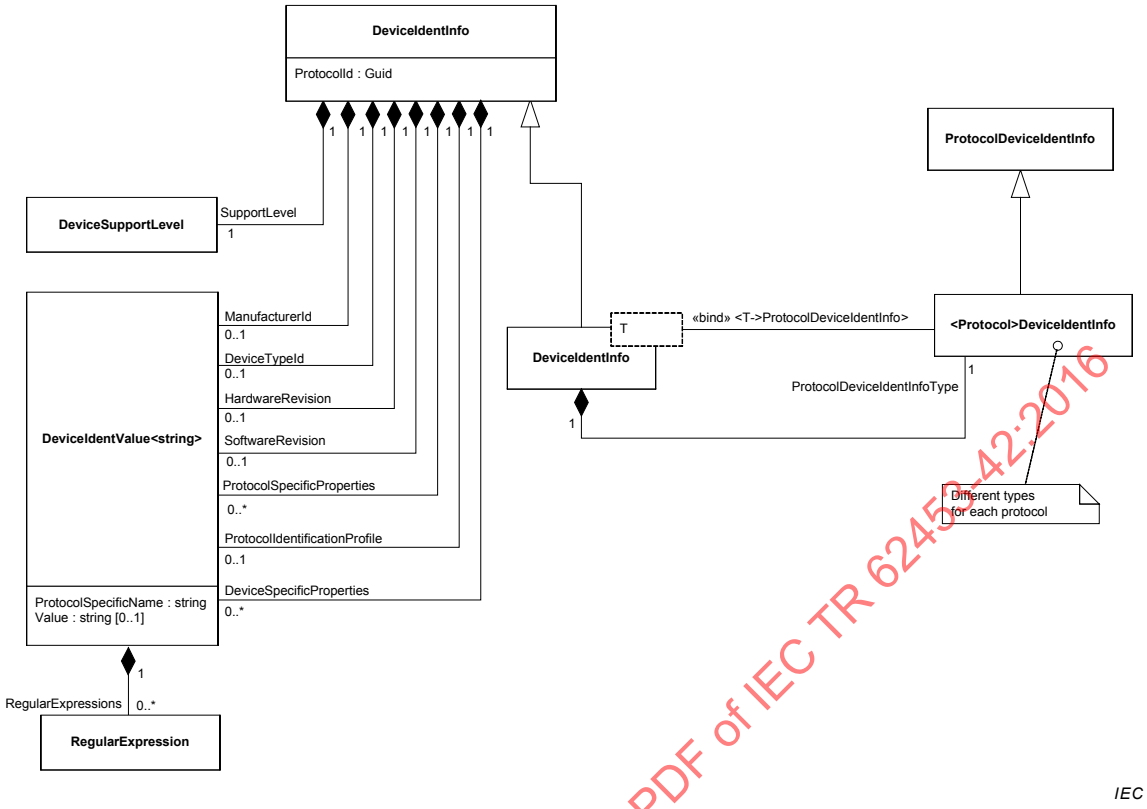
Table 17 – DtmInfo datatype description

Datatype	Description
BlockTypeCategory	A block type category is a Universally Unique Identifier for a block category (e.g. Analog Input, Digital Output).
BlockTypeInfo	The representation for a particular block type within the DTM is called DTM Block Type. A DTM may contain one or more DTM Block Types. The concrete design and implementation of the DTM Block Types is not in scope of FDT. This class provides only information about these pieces of software like name, version, vendor, supported protocols etc.
BusCategory	A bus category is a Unique Identifier for a fieldbus protocol (or a point-to-point communication). See also 7.8.
ClassificationId	Unique identifier according to its primary measurement (IEC 62390 AnnexG).

Datatype	Description
ClassificationDomainId	Device classification domain groups (IEC 62390 AnnexG).
DatasetFormats	Dataset format identifiers of persisted data, used and supported by a DTM
DeviceClassification	Classification of a device according IEC 62390, Annex G
DeviceIdentInfo	This class is used to describe physical device types which are supported by a Type. It contains identification elements of a physical device type or device type group.
DeviceTypeInfo	The representation for a particular physical device type within the DTM is called DTM Device Type. A DTM may contain one or more DTM Device Types. The concrete design and implementation of the DTM Device Types is not in scope of FDT. This class provides only information about these pieces of software like name, version, vendor, supported protocols etc.
Document	Information about documents on hard disk or in the Web. This could be any device manual, help file, spare part list etc. which is installed together with the DTM or available on the Web. A Document may also provide protocol-specific information for a DeviceType (e.g. EDS). In such cases the document shall be categorized as 'Technical Document' and be marked with an appropriate protocol-specific SemanticId.
DtmInfo	DtmInfo contains general information about a DTM such as name, version, identifier and vendor of the software, the FDT version to which the DTM complies.
FdtBitmapInfo	Description of a bitmap for representation of a device, module or block in BMP format (high resolution, 24 bit color and 8 bit transparency info (alpha channel))
FdtIconInfo	Information about device, module or block icon.
ModuleTypeInfo	The representation for a particular physical module type within the DTM is called DTM Module Type. A DTM may contain one or more DTM Module Types. The concrete design and implementation of the DTM Module Types is not in scope of FDT. This class provides information about DTM Module Types like name, version, vendor, supported protocols etc.
TypeInfo	Abstract base class used for definition of device type, block type or module type. A DTM shall contain one or more TypeInfo objects.

7.5 Datatypes – DeviceIdentInfo

The class diagram shown in Figure 69 describes the relations of the DeviceIdentInfo class. DeviceIdentInfo can be requested from IDtmInformation.GetDeviceIdentInfo() for a given TypeIdent and BusCategory.



Used in:

IDtmInformation.GetDeviceIdentInfo()

Figure 69 – DeviceIdentInfo – datatypes

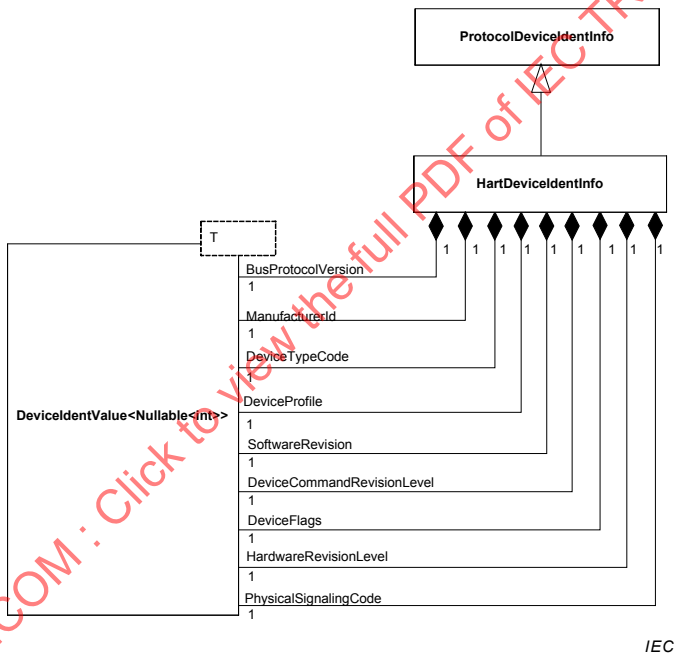
Table 18 describes datatypes related to DeviceIdentInfo.

Table 18 – DeviceIdentInfo datatype description

Datatype	Description
DeviceIdentInfo	This class is used to describe physical device types which are supported by a DTM Device Type. It contains identification elements of a physical device type or device type group. Remark: This class provides a protocol neutral access to the information , therefore it typically will be used by the Frame Application if no protocol-specific handling is needed.
DeviceIdentInfo<T>	The derived class DeviceIdentInfo<T> provides a protocol-specific access to the information which is more type-safe. This class should be used whenever possible. The generic type parameter T defines the type of the protocol-specific class which defines the protocol-specific identification properties. These protocol-specific properties are mapped to the properties: • ManufacturerId • DeviceTypeId • SoftwareRevision • HardwareRevision • ProtocolIdentificationProfile • ProtocolSpecificProperties which are defined in the base class DeviceIdentInfo.
DeviceIdentValue<T>	Represents a single identification element of a physical device type or group. For example: Device Type Id, Manufacturer Id etc.

Datatype	Description
DeviceSupportLevel	Enumeration which defines the support level of a DTM Device Type for a physical device.
ProtocolDeviceIdentInfo	Abstract base class for protocol-specific device identification properties. Protocol-specific classes derive from this class and define the-specific device identification properties. However, these protocol-specific properties can be accessed in a protocol neutral way by accessing the corresponding properties in the DeviceIdentInfo class.
ProtocolId	Universally Unique Identifier for a fieldbus protocol (or a point-to-point communication).
ProtocolIdentificationProfile	Defines the protocol-specific identification profile which is used for device identification. (examples for PROFIBUS: I&M, PA, DP). If a protocol does not support multiple identification profiles then this property shall be empty.
RegularExpression	Regular expression that defines which physical device types are supported by a DTM Device Type.

The class diagram in Figure 70 shows the protocol-specific datatype DeviceIdentInfo<T> for the example HART protocol.



Used in:

Protocol-specific Device DTM providing values for HART-specific DeviceIdentInfo

Figure 70 – DeviceIdentInfo – Example for HART

Table 19 describes HART datatypes related to DeviceIdentInfo.

Table 19 – DeviceIdentInfo – Example for HART

Datatype	Description
DeviceIdentValue<T>	Represents an identification element of a device type for a physical device type or group. For example: Device Type Id, Manufacturer Id etc. The generic type parameter T defines the type of the identification value (e.g. int, float, string etc.) corresponding to the format defined in the protocol. The identification element can either be a specific value or a regular expression (e.g. defining a range of supported identification values).
HartDeviceIdentInfo	HART-specific device identification information.
ProtocolDeviceIdentInfo	Abstract base class for protocol-specific device identification properties. Protocol-specific classes derive from this class and define the specific device identification properties. However, these protocol-specific properties can be accessed in a protocol neutral way by accessing corresponding properties in the DeviceIdentInfo class.

The example in Figure 71 demonstrates how a (HART) Device DTM creates and returns a DeviceIdentInfo instance:

```
public DeviceIdentInfo GetDeviceIdentInfo()
{
    // Create the HART specific identification properties first

    // Manufacturer code of the device vendor is 17
    HartDeviceIdentInfo hartSpecificInfo = new HartDeviceIdentInfo();
    hartSpecificInfo.ManufacturerId = new DeviceIdentValue<int?>(17);

    // The ID of the supported device is 123
    var identVal = new DeviceIdentValue<int?>();
    hartSpecificInfo.DeviceTypeCode = new DeviceIdentValue<int?>(123);

    // Device is a HART 5 Device
    hartSpecificInfo.BusProtocolVersion = new DeviceIdentValue<int?>(5);

    // This DTM is able to handle the software versions 1,2 and 3 of the device
    identVal = new DeviceIdentValue<int?>();
    identVal.RegularExpressions = new FdtList<RegularExpression>(
                                        new RegularExpression("1|2|3"));
    hartSpecificInfo.SoftwareRevision = identVal;

    // This DTM is able to handle only the command revision level 5 of the device
    hartSpecificInfo.DeviceCommandRevisionLevel = new DeviceIdentValue<int?>(5);

    // This DTM is able to handle all hardware versions of this device
    identVal = new DeviceIdentValue<int?>();
    identVal.RegularExpressions = new FdtList<RegularExpression>(
                                        new RegularExpression(".*"));
    hartSpecificInfo.HardwareRevisionLevel = identVal;

    // Physical Signaling Code is not relevant for identification
    identVal = new DeviceIdentValue<int?>();
    identVal.RegularExpressions = new FdtList<RegularExpression>(
                                        new RegularExpression(".*"));
    hartSpecificInfo.PhysicalSignalingCode = identVal;

    // Device Flags are not relevant for identification
    identVal = new DeviceIdentValue<int?>();
    identVal.RegularExpressions = new FdtList<RegularExpression>(
                                        new RegularExpression(".*"));
    hartSpecificInfo.DeviceFlags = identVal;

    // Device ident information (protocol neutral)
    DeviceIdentInfo<HartDeviceIdentInfo> deviceIdentInfo =
                            new DeviceIdentInfo<HartDeviceIdentInfo>();

    // This DTM is designed to support a specific device
    deviceIdentInfo.SupportLevel = DeviceSupportLevel.SpecificSupport;

    // Set the protocol specific info
    deviceIdentInfo.ProtocolSpecificIdentInfo = hartSpecificInfo;

    return deviceIdentInfo;
}
```

IEC

Figure 71 – Example: DeviceIdentInfo creation

The example in Figure 72 demonstrates how a Frame Application requests and uses the DeviceIdentInfo instance created in Figure 71. The protocol-specific properties shown in Figure 71 are mapped automatically to the protocol-independent properties which are used in Figure 72.

```
public void ShowDeviceIdentInfo(IDtmInformation dtm, DeviceTypeInfo deviceTypeInfo)
{
   FdtList<DeviceIdentInfo> deviceIdentInfo = dtm.GetDeviceIdentInfo(deviceTypeInfo.Id,
                                 deviceTypeInfo.BusCategories[0]);
   // Standard FDT2 ident properties
   MessageBox.Show("Manufacturer ID = " + deviceIdentInfo[0].ManufacturerId.Value + "\n" +
                   "Device Type ID = " + deviceIdentInfo[0].DeviceTypeId.Value + "\n" +
                   "Software Rev. = " + deviceIdentInfo[0].SoftwareRevision.Value + "\n" +
                   "Hardware Rev. = " + deviceIdentInfo[0].HardwareRevision.Value + "\n");

    // Ident properties only defined in the protocol
    // (for HART: DeviceCommandRevisionLevel and Device Flag)
    foreach (DeviceIdentValue<string> identValue
        in deviceIdentInfo[0].ProtocolSpecificProperties)
    {
        MessageBox.Show(identValue.ProtocolSpecificName + " = " + identValue.Value);
    }
}
```

IEC

Figure 72 – Example: Using DeviceIdentInfo

The example in Figure 73 demonstrates how the HART-specific datatype assembly (Fdt.Datatypes.Hart.dll) exposes the type information over the DeviceIdentInfoTypeAttribute:

```
[assembly: DeviceIdentInfoType
(
DeviceIdentInfoType = typeof(DeviceIdentInfo<HartDeviceIdentInfo>),
ProtocolDeviceIdentInfoType = typeof(HartDeviceIdentInfo),
       DeviceScanInfoType = typeof(DeviceScanInfo<HartDeviceScanInfo>),
       ProtocolDeviceScanInfoType = typeof(HartDeviceScanInfo)
)
]
```

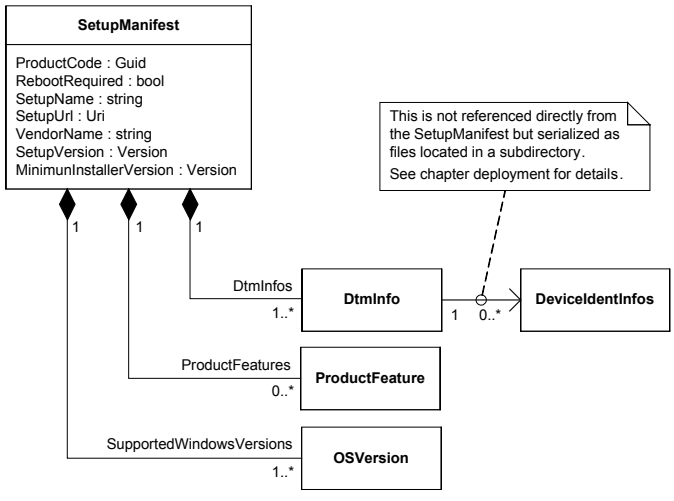
IEC

Figure 73 – Example: DeviceIdentInfoTypeAttribute

7.6 Datatypes for installation and deployment

7.6.1 Datatypes – SetupManifest

A SetupManifest describes the setup of a DTM. It is used for installation and deployment. (see 9.6). Figure 74 shows a class diagram with related classes of SetupManifest.



IEC

Figure 74 – SetupManifest – datatypes

Table 20 describes SetupManifest class and its related classes.

Table 20 – SetupManifest datatype description

Datatype	Description
DtmDeviceIdentManifest	A DtmDeviceIdentManifest describes additional physical device parameters that are required for device identification.
DtmInfo	DtmInfo contains general information about a DTM such as name, version, identifier and vendor of the software, the FDT version to which the DTM complies.
OSVersion	This class represents a version of the operating system.
ProductFeature	This class represents a product feature for installation.
SetupManifest	A setup manifest describes the setup of a DTM, including identification of the product, the vendor, version and included DTMs.

7.6.2 Datatypes – DtmManifest

A DtmManifest describes the components of a DTM (see 9.5.3). Figure 75 shows a class diagram with related classes of DtmManifest.

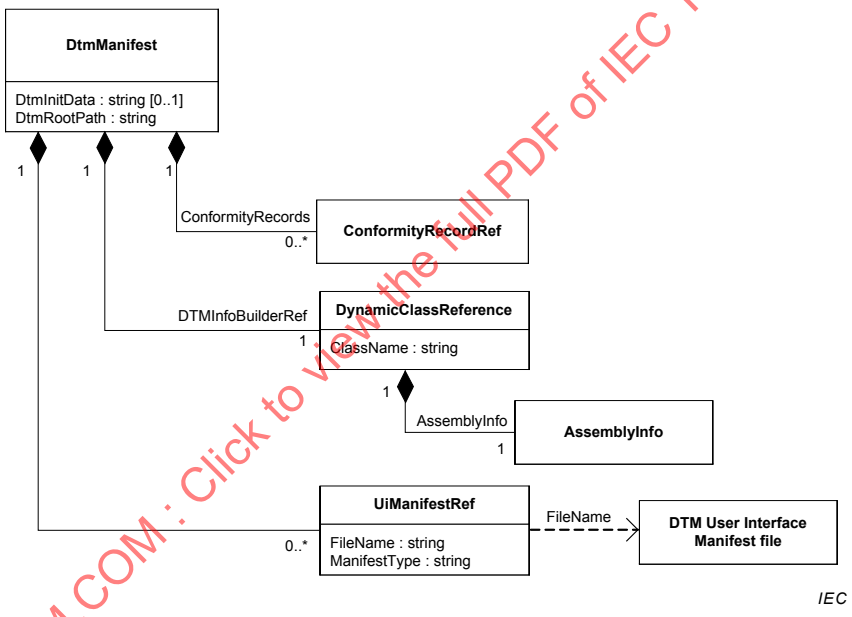


Figure 75 – DtmManifest – datatypes

Table 21 describes DtmManifest datatype and its related classes.

Table 21 – DtmManifest datatype description

Datatype	Description
AssemblyInfo	Information about a .NET assembly.
ConformityRecordRef	Reference to a conformity record file.
DtmManifest	A DTM manifest describes the assembly of a DTM and the included DTM itself. The manifest is used to register an installed DTM in order to enable Frame Applications to find it.
DynamicClassReference	Information about a class e.g. a DtmInfoBuilder or a DTM.
DTM User Interface manifest file	A DTM User Interface manifest file is used to register a DTM User Interface in the system in order to enable Frame Applications to find it. The file contains a DtmUiManifest (see 7.6.3).

7.6.3 Datatypes – DtmUiManifest

This manifest describes a DTM UI assembly and the included DTM User Interface functions. The manifest is used to register installed DTM User Interface functions in order to enable the Frame Applications to find and execute them. Figure 76 shows a class diagram with related classes of DtmUiManifest.

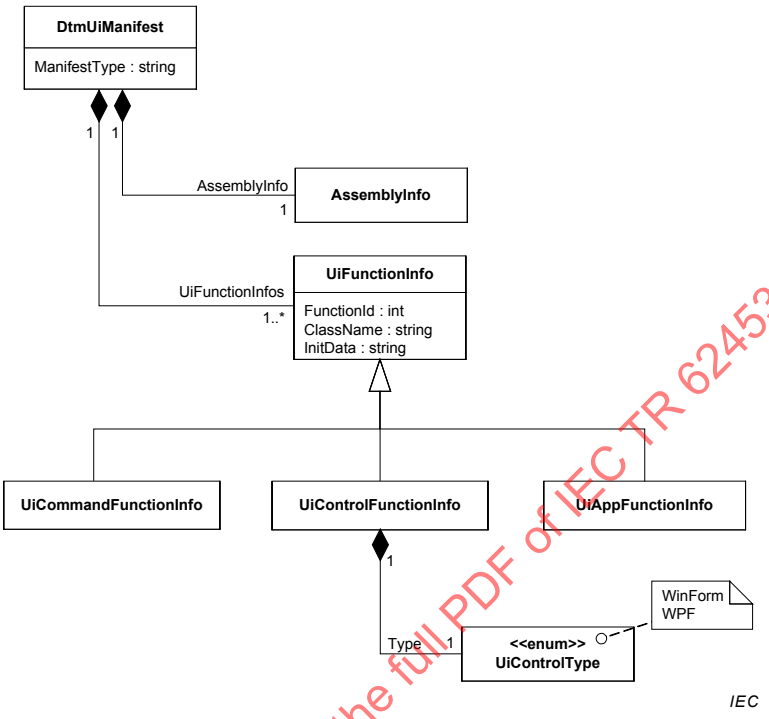


Figure 76 – DtmUiManifest – datatypes

Table 22 describes DtmUiManifest class and its related classes.

Table 22 – DtmUiManifest datatype description

Datatype	Description
DtmUiManifest	This manifest describes a DTM User Interface function. The manifest is used to register installed DTM User Interface functions in order to enable the Frame Applications to find and execute them.
AssemblyInfo	Information about a .NET assembly.
UiFunctionInfo	Abstract base class for a DTM User Interface description. Frame Applications shall use this information to find the user interface function for a specific function.
UiCommandFunctionInfo	Information about a command function which is provided by a DTM User Interface class.
UiControlFunctionInfo	Information about a WinForms control or WPF control that can be embedded into the Frame Application user interface.
UiAppFunctionInfo	Information about an application which can be started by a DTM User Interface class.
UiControlType	Enumerates possible user interface control types (WinForms, WPF etc.)

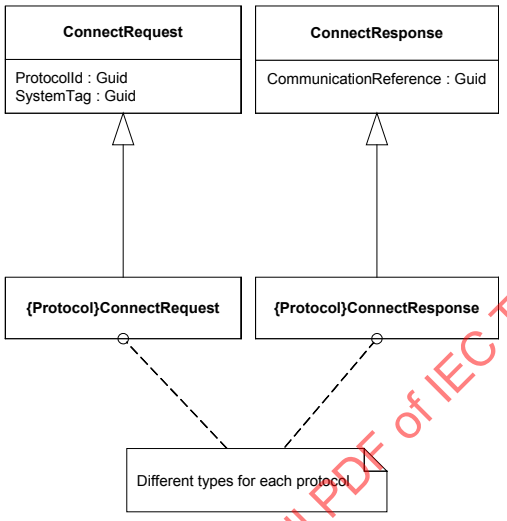
7.7 Datatypes – Communication

The communication datatypes are used to exchange data between a DTM and its parent Communication Channel in order to:

– Establish a connection to the device

– Perform data exchange transactions with the device
– Release the connection
– Subscribe device initiated data transfer between a DTM and its parent Communication Channel
– Request scanning of bus topology
– Request address setting of Child DTM

Figure 77 shows a class diagram with datatypes used to establish a connection to the device.



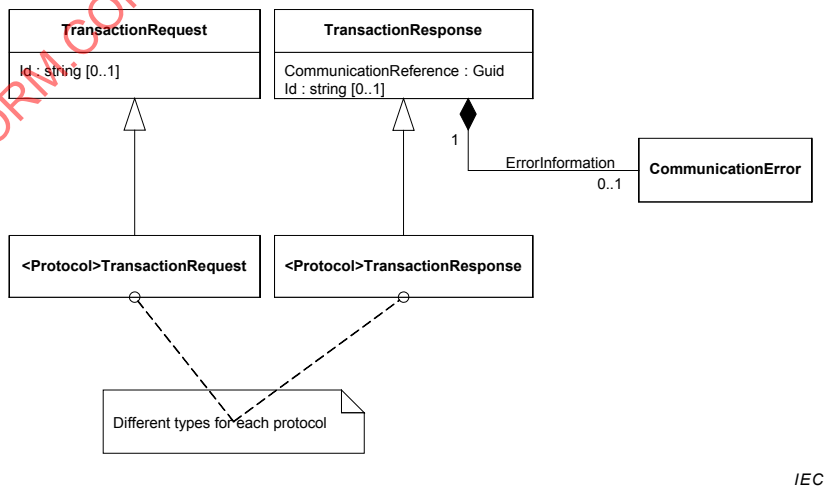
Used in:

ConnectRequest: ICommunication.BeginConnect()

ConnectResponse is returned in ICommunication.EndConnect()

Figure 77 – Communication datatypes – Connect

Figure 78 shows a class diagram with datatypes used to exchange data with the device.



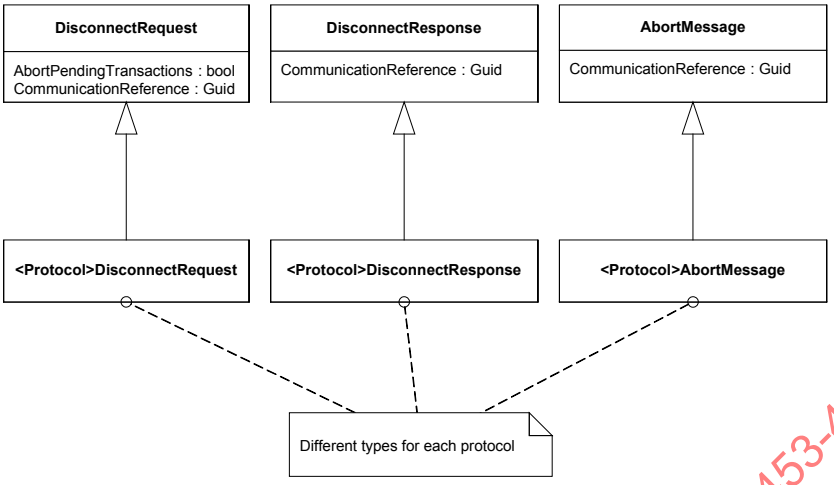
Used in:

TransactionRequest: ICommunication.BeginCommunicationRequest()

TransactionResponse is returned in ICommunication.EndCommunicationRequest()

Figure 78 – Communication datatypes – Transaction

Figure 79 shows a class diagram with datatypes used to release a connection to the device.



Used in:

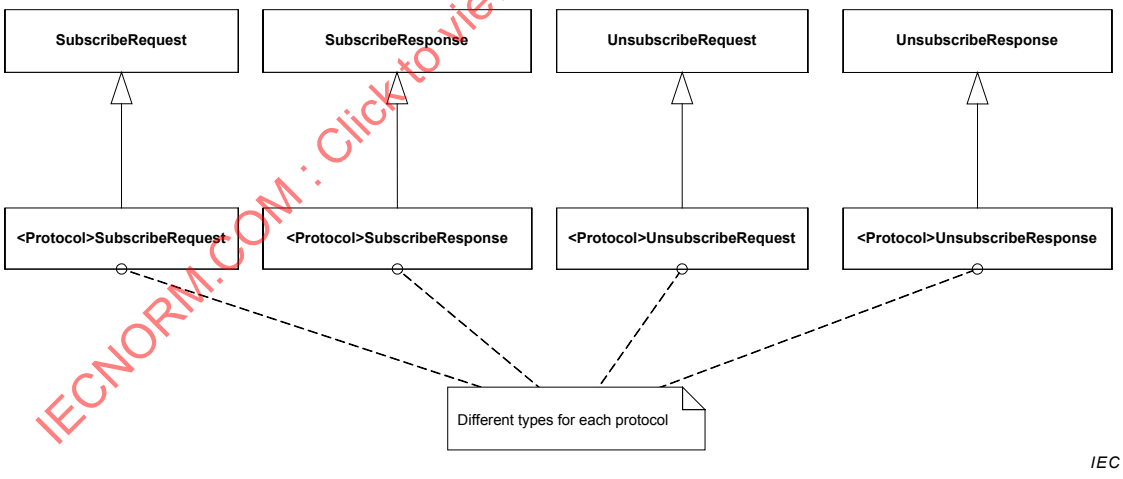
DisconnectRequest: ICommunication.BeginDisconnect()

DisconnectResponse is returned in ICommunication.EndDisconnect()

AbortMessage: AbortCallback()

Figure 79 – Communication datatypes – Disconnect

Figure 80 shows a class diagram with datatypes used to subscribe and unsubscribe device initiated data transfer.



Used in:

SubscribeRequest: ISubscription.BeginSubscriptionInitialization()

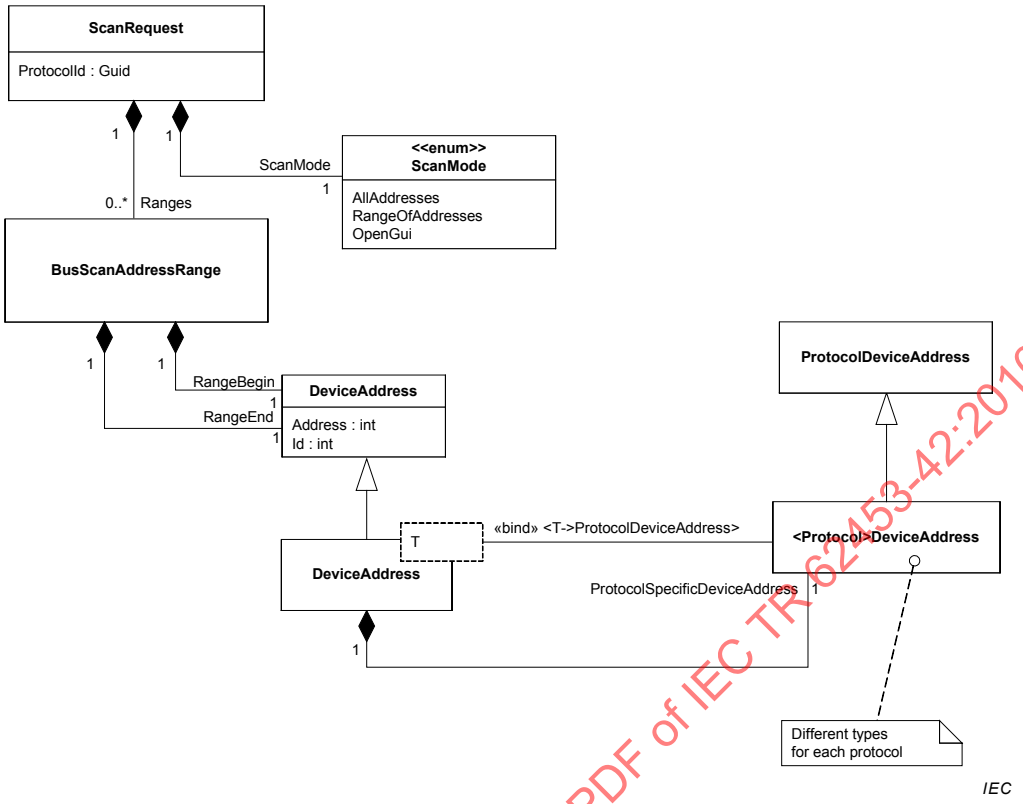
SubscribeResponse is returned in ISubscription.EndSubscriptionInitialization()

UnsubscribeRequest: ISubscription.BeginSubscriptionTermination()

UnsubscribeResponse is returned in ISubscription.EndSubscriptionTermination()

Figure 80 – Communication datatypes – Subscribe

Figure 81 shows a class diagram with datatypes used to request scanning of the sub-topology of a Communication Channel.

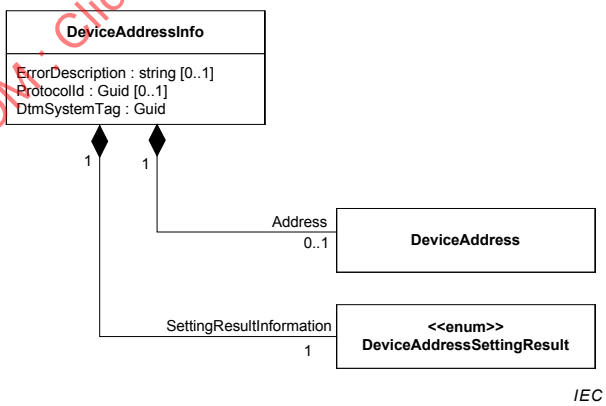


Used in:

ScanRequest: IScanning.BeginScanRequest()

Figure 81 – Communication datatypes – Scanning

Figure 82 shows a class diagram with datatypes used to request setting of device address of Child DTM of a Communication Channel.



Used in:

DeviceAddressInfo: ISubTopology.BeginSetChildrenAddresses()

DeviceAddressInfo Is returned from ISubTopology.EndSetChildrenAddresses()

Figure 82 – Communication datatypes – Address setting

Table 23 describes the communication datatypes.

Table 23 – Communication datatype description

Datatype	Description
ConnectRequest	Fieldbus protocol independent base class for information needed to establish a communication link.
ConnectResponse	Fieldbus protocol independent base class for response information about an established communication link.
TransactionRequest	Fieldbus protocol independent base class for transaction request information.
TransactionResponse	Fieldbus protocol independent base class for transaction results.
DisconnectRequest	Fieldbus protocol independent base class for disconnection information.
DisconnectResponse	Fieldbus protocol independent base class for results of disconnect operation.
AbortMessage	Information to specify an abort of a communication link.
SubscribeRequest	Fieldbus protocol independent base class with information for initialization of device initiated data transfer.
SubscribeResponse	Fieldbus protocol independent base class for information about communication data subscription.
UnsubscribeRequest	Fieldbus protocol independent base class for termination of subscription of device initiated data transfer.
UnsubscribeResponse	Fieldbus protocol independent base class for response to termination of subscription of device initiated data transfer.
ScanRequest	Information for a request to scan the sub-topology of a Communication Channel
BusScanAddressRange	Information about the address range of the requested scan
DeviceAddressInfo	Address information which is used to request the Communication Channel to set the address of its Child DTMs.
DeviceAddress	Address of the device in the network or fieldbus.

The example given in Figure 83 demonstrates how a (HART) Device DTM may connect to a device:

```
bool Connect(Guid mySystemTag, ICommunication commChannel,
             HartDeviceAddress myAddress, ref Guid communicationReference)
{
    //Create ConnectRequest
    //The required SystemTag is set by the Frame Application
    //during creation of the DTM instance
    //The Address will be set by the Communication Channel
    var request = new HartConnectRequest(mySystemTag, myAddress);

    HartConnectResponse response;

    try
    {
        //Request connection from Communication Channel
        var asyncResult =
            commChannel.BeginConnect(request, abortCallback, null, null, null);

        //Wait for finalization of the connect request
        response = commChannel.EndConnect(asyncResult) as HartConnectResponse;
    }
    catch(Exception ex)
    {
        MessageBox.Show("Connection failed\n" + "Details: " + ex.Message);
        return false;
    }

    if (response != null)
    {
        //verify response
        try
        {
            response.Verify();
        }
        catch(Exception ex)
        {
            MessageBox.Show("Connection failed\n" + "Details: " + ex.Message);
            return false;
        }

        //Connection established
        //the response contains the complete address information
        //and the communication reference of the connection
        communicationReference = response.CommunicationReference;
        MessageBox.Show("Successfully connected with device\n" +
            "Short Address: " + response.Address.ShortAddress + "\n" +
            "Short TAG: " + response.Address.ShortTag + "\n" +
            "Long TAG: " + response.Address.LongTag + "\n");
        return true;
    }

    return false;
}
```

IEC

Figure 83 – Example: Communication – Connect for HART

The example given in Figure 84 demonstrates how the HART-specific datatype assembly(Fdt.Datatypes.Hart.dll) exposes the type information over the CommunicationType attribute:

```
[assembly: CommunicationType(
                                AbortMessageType = typeof(HartAbortMessage),
                                ConnectRequestType = typeof(HartConnectRequest),
                                ConnectResponseType = typeof(HartConnectResponse),
                                DisconnectRequestType = typeof(HartDisconnectRequest),
                                DisconnectResponseType = typeof(HartDisconnectResponse),
                            ...
                                SubscribeRequestType = typeof(HartSubscribeRequest),
                                SubscribeResponseType = typeof(HartSubscribeResponse),
                                UnsubscribeRequestType = typeof(HartUnsubscribeRequest),
                                UnsubscribeResponseType = typeof(HartUnsubscribeResponse))
]
```

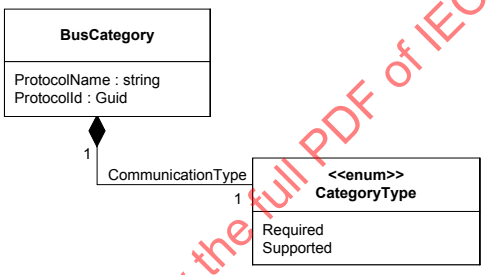
IEC

Figure 84 – Example: Communication – CommunicationType for HART

NOTE Please be aware that the above examples demonstrate how a protocol-specific datatype can be derived from the datatypes defined in this document and how such a protocol-specific datatype is intended to be used. For definition of the protocol-specific datatypes please refer to the respective specification document.

7.8 Datatypes – BusCategory

The class diagram shown in Figure 85 describes the relations of the BusCategory class.



Used in:

TypeInfo

Figure 85 – BusCategory – datatypes

Table 24 describes the datatype BusCategory and its elements

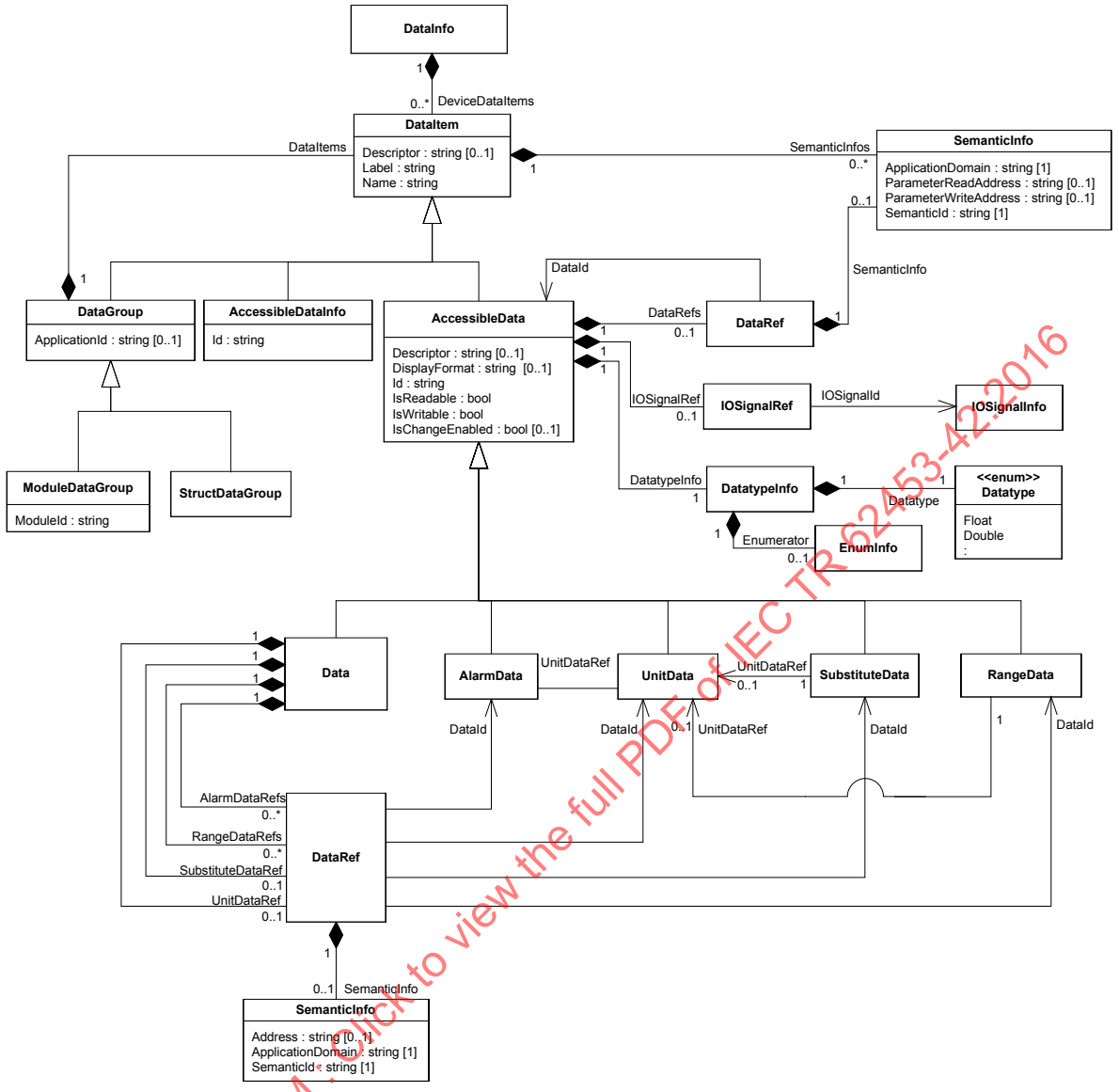
Table 24 – BusCategory datatype description

Datatype	Description
BusCategory	Bus category is a Unique Identifier for a fieldbus protocol (or a point-to-point communication).
CategoryType	Defines whether BusCategory is supported or required.

7.9 Datatypes – Device / Instance Data

7.9.1 General

The Device / Instance Data classes describe device parameters or process values that can be read from the device / instance data or written into the device / instance data. The class diagram in Figure 86 shows the classes and relations.



IEC

Used in:

returned in IDeviceData.EndGetDataInfo() / IInstanceData.EndGetDataInfo()

Figure 86 – Device / Instance data – datatypes

Table 25 describes classes related to Device / Instance Data

Table 25 – DeviceData datatype description

Datatype	Description
AccessibleData	Abstract base class for data which is readable or writable. The DTM shall provide a DisplayFormat for all AccessibleData variables with numerical values
AlarmData	Representation of an alarm parameter. An alarm shall always be of a numeric type (Float, Double, Int, Long, Uint, Ulong) or Enumerator (checked by the Verify() method).
Data	Describes a device parameter or a process value that can be read or written. The information contains descriptive attributes like name as well as information how the item is accessible.
DataGroup	Groups information about available device-specific parameters and process values.
DataInfo	Contains information about available device-specific parameters and process values.
DataItem	Abstract base class for device and instance data info classes.
DataRef	Reference to an item in DataInfo identified by its Id and optionally also information about the type (semantic) of the reference.
Datatype	List of possible datatypes.
DatatypeInfo	Information about type of data (see DataValue).
IOSignalInfo	Information about a single device IO signal.
IOSignalRef	Reference to an IO signal identified by its identifier.
ModuleDataGroup	Groups information about available module-specific parameters and process values.
RangeData	Representation of a range parameter. A range shall always be of a numeric type (Float, Double, Int, Long, Uint, Ulong) or DateTime (checked by the Verify() method). The RangeData may provide a reference to a UnitData. If no reference is provided, the same unit is applied as in the Data that references the RangeData.
SemanticInfo	This class provides semantic information for a data object.
StructDataGroup	Represents a data structure containing specific parameters and/or process values.
SubstituteData	Describes the value which shall be used as a fall back e.g. in case there is a disturbed communication. The SubstituteData may provide a reference to a UnitData. If no reference is provided, the same unit is applied as in the Data that references the SubstituteData.
UnitData	Representation of a unit parameter. Unit shall always be of type Enumerator (checked by the Verify() method).

Figure 87 shows how DataInfo may expose information on data of a HART device.

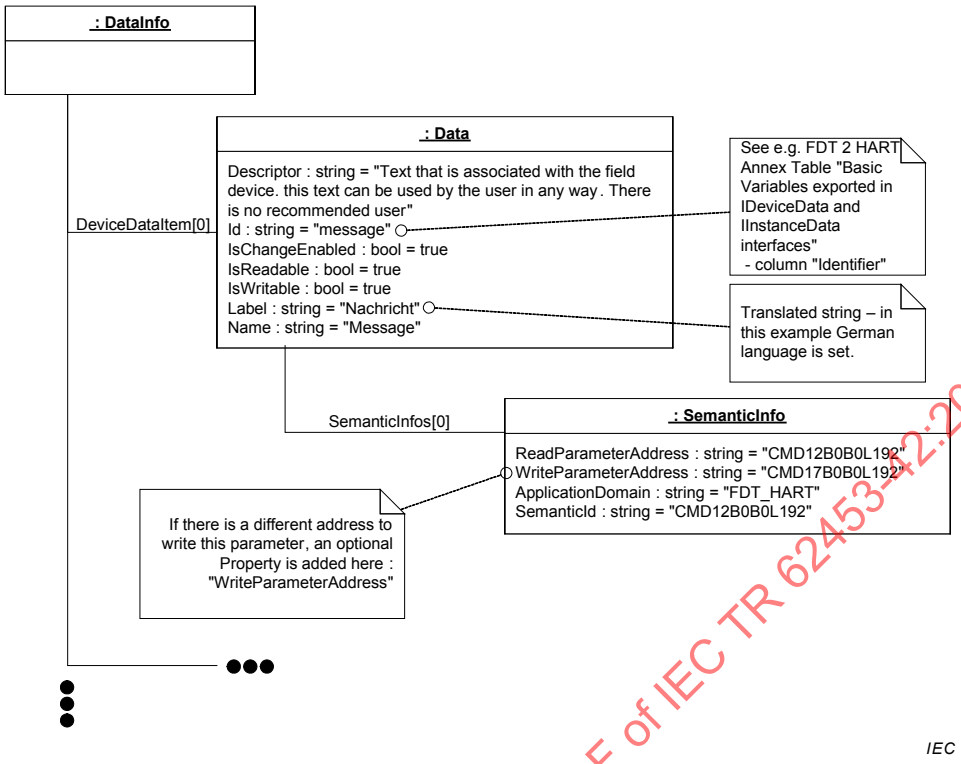


Figure 87 – Example: Providing information on data of a HART device

Figure 88 shows how DataInfo may expose information on data of a PROFIBUS device.

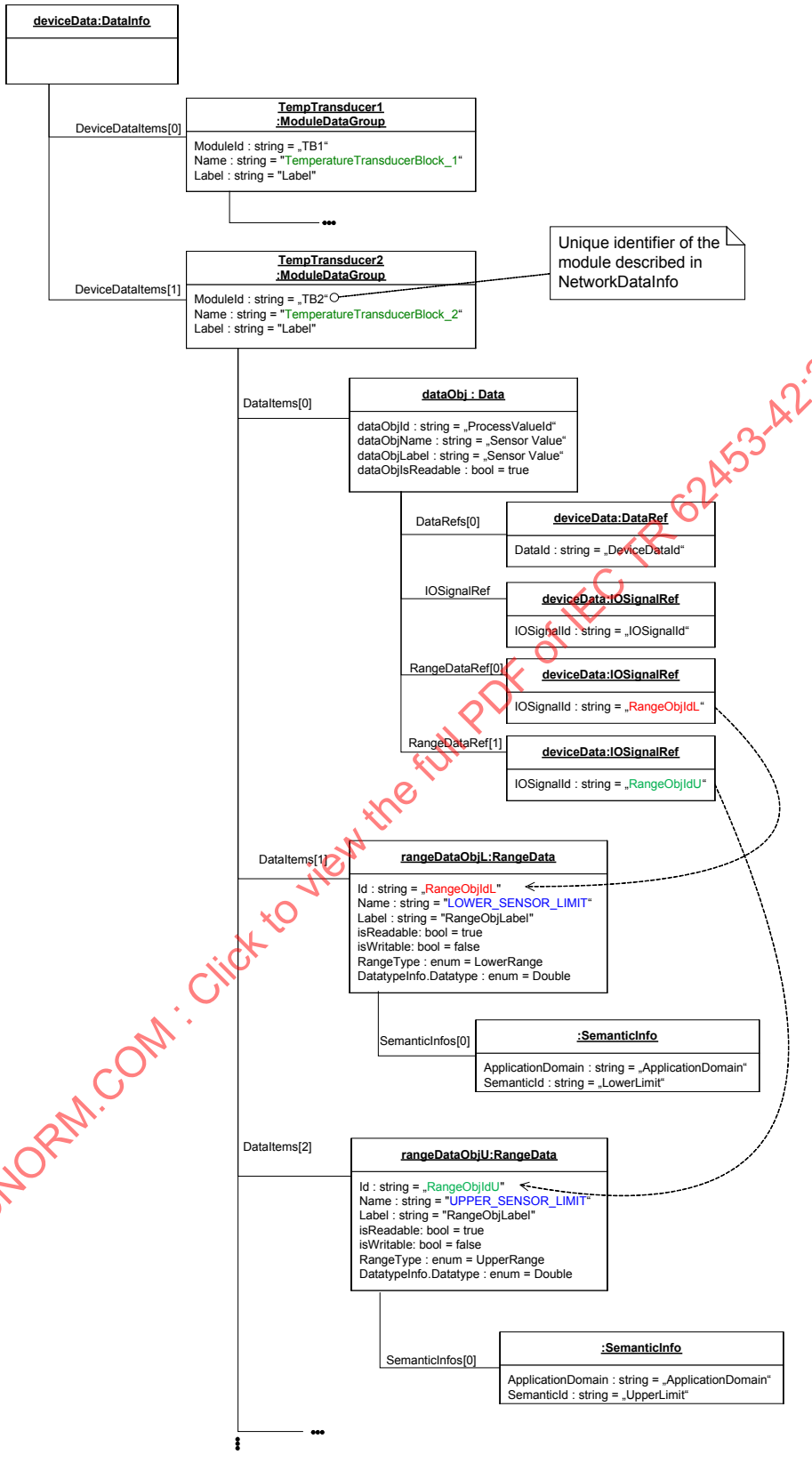


Figure 88 – Example: Providing information on module data of a PROFIBUS device

The example given in Figure 89 shows how to create DataInfo with one Data-object and a ModuleDataGroup that contains RangeData-items for lower and upper limit.

```
public DataInfo GetDataInfo()
{
    DataInfo deviceData = new DataInfo();

    // Create a Data object (inherits from AccessibleData and DataItem)
    string dataObjId = "DataObjID";
    string dataObjName = "DataObjName";
    string dataObjLabel = "DataObjLabel";
    bool dataObjIsReadable = true;
    bool dataObjIsWritable = false;
    DatatypeInfo dataObjDatatypeInfo = new DatatypeInfo(Datatype.Long);
    Data dataObj = new Data(dataObjId, dataObjName, dataObjLabel,
                            dataObjIsReadable, dataObjIsWritable,
                            dataObjDatatypeInfo);
    // Define refences to other DataItems (Optional members)
    dataObj.DataRefs = new FdtList<DataRef>() { new DataRef("DeviceDataID") };
    dataObj.IOSignalRef = new Fdt.Dtm.IO.IOSignalRef("IOSignalID");

    // Create a (Lower)RangeData object
    // (inherits from AccessibleData and DataItem)
    string rangeObjIdL = "RangeObjIdL";
    string rangeObjNameL = "LOWER_SENSOR_LIMIT";
    string rangeObjLabelL = "RangeObjLabel1";
    bool rangeObjIsReadableL = true;
    bool rangeObjIsWritableL = false;
    DatatypeInfo rangeObjDatatypeInfoL = new DatatypeInfo(Datatype.Double);
    RangeType rangeTypeL = RangeType.LowerRange;

    RangeData rangeDataObjL = new RangeData(rangeObjIdL, rangeObjNameL,
                                            rangeObjLabelL, rangeObjIsReadableL,
                                            rangeObjIsWritableL,
                                            rangeObjDatatypeInfoL, rangeTypeL);
    // Define SematicInfo-object for rangeDataObj
    rangeDataObjL.SemanticInfos = new FdtList<SemanticInfo>(
                          new SemanticInfo("ApplicationDomain", "LowerLimit"));

    // Create an (Upper)RangeData object
    // (inherits from AccessibleData and DataItem)
    string rangeObjIdU = "RangeObjIdU";
    string rangeObjNameU = "UPPER_SENSOR_LIMIT";
    string rangeObjLabelU = "RangeObjLabel2";
    bool rangeObjIsReadableU = true;
    bool rangeObjIsWritableU = false;
    DatatypeInfo rangeObjDatatypeInfoU = new DatatypeInfo(Datatype.Double);
    RangeType rangeTypeU = RangeType.UpperRange;

    RangeData rangeDataObjU = new RangeData(rangeObjIdU, rangeObjNameU,
                                            rangeObjLabelU,
                                            rangeObjIsReadableU,
                                            rangeObjIsWritableU,
                                            rangeObjDatatypeInfoU, rangeTypeU);

    // Define SematicInfo-object for rangeDataObj
    rangeDataObjU.SemanticInfos = new FdtList<SemanticInfo>(
                  new SemanticInfo("ApplicationDomain", "UpperLimit"));

    //Create a ModuleDataGroup with the two RangeData-items.
    FdtList<DataItem> dataItemsInGroup = new FdtList<DataItem>() {rangeDataObjL,
                                                    rangeDataObjU};

    ModuleDataGroup rangeDataGroupObj = new ModuleDataGroup("TB1",
                                                    "TemperatureTransducerBlock_1",
                                                    "Label",
                                                    dataItemsInGroup);
    // Put DataItem objects into the list
    deviceData.DeviceDataItems = new FdtList<DataItem>() { dataObj,
                                              rangeDataGroupObj };

    return deviceData;
}
```

IEC

Figure 89 – Example: Providing information on data

If the data is structured data, then the StructDataGroup may be used to show the structure of the data (see Figure 90).

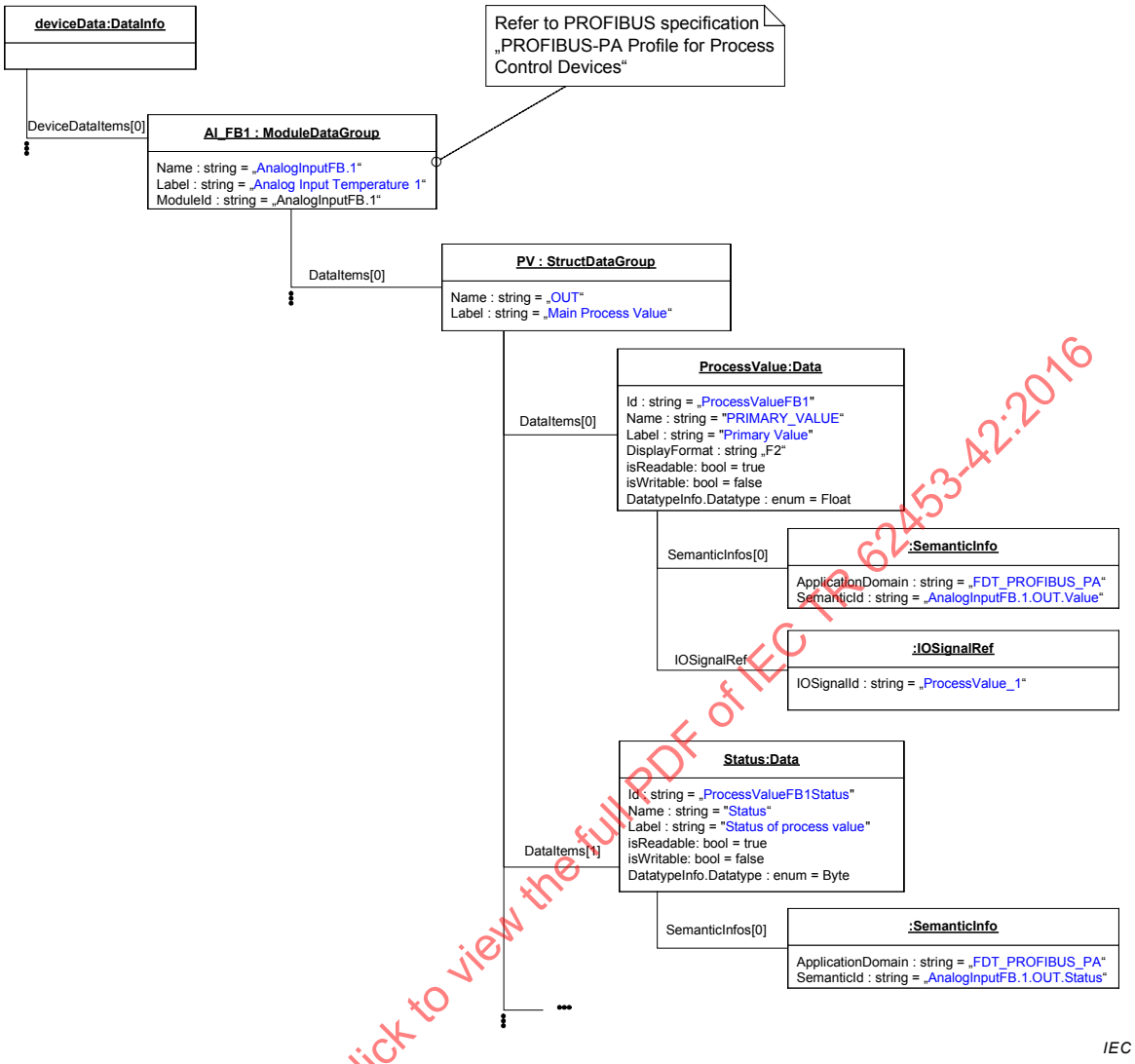
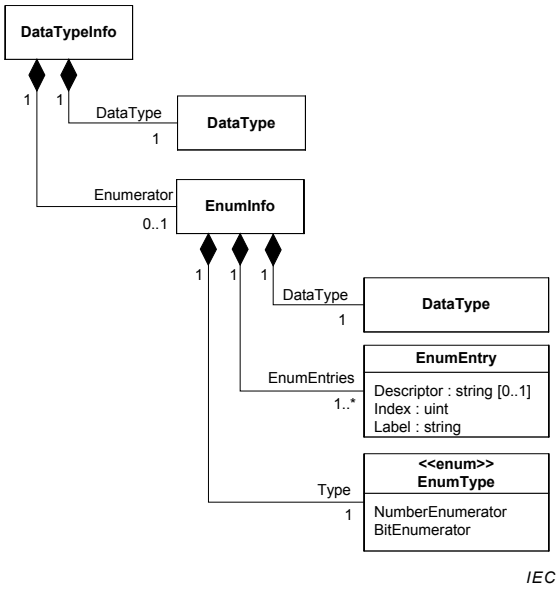


Figure 90 – Example: Providing information on structured data

If the data described in the DataInfo is provided as enumeration (DataValue = EnumValue), the EnumInfo class is used to provide the description of the value range (see Figure 91).



Used in:

IDeviceData.EndGetDataInfo() / IInstanceData.EndGetDataInfo()

Figure 91 – EnumInfo – datatype

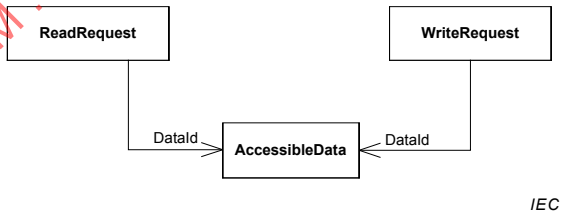
7.9.2 Datatypes used in reading and writing DeviceData

7.9.2.1 General

The IDeviceData interface provides online access to specific parameters of a device. The following chapters define datatypes used in methods for reading and writing device data.

7.9.2.2 ReadRequest and WriteRequest Datatypes

ReadRequest datatype and the WriteRequest datatype (see Figure 92 and Table 26) are used to define specific parameters which shall be read or written.



Used in:

ReadRequest: IDeviceData.BeginRead() / IInstanceData.BeginRead()

WriteRequest: IDeviceData.BeginWrite() / IInstanceData.BeginWrite()

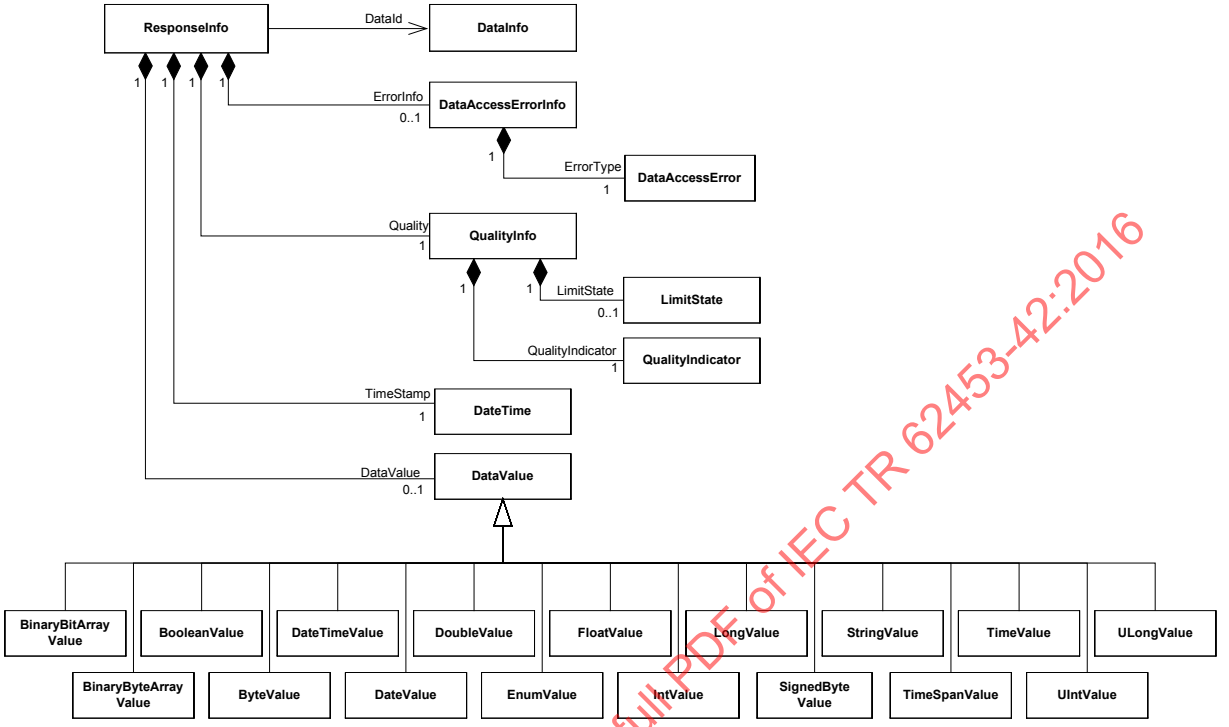
Figure 92 – Read and Write Request – datatypes

Table 26 – Reading and Writing datatype description

Datatype	Description
AccessibleData	Abstract base class for device data which is readable or writable. (See Figure 86)
ReadRequest	Read request for a single entry in DataInfo addressed by its Id. □
WriteRequest	Write request for a single entry in DataInfo addressed by its Id.□

7.9.2.3 ResponseInfo Datatype

The ResponseInfo datatype (see Figure 93 and Table 27) is used to return read or written data requested by ReadRequest or WriteRequest.



Used in:

IDeviceData.EndRead()

IInstanceData.EndRead()

IDeviceData.EndWrite()

IInstanceData.EndWrite()

Figure 93 – ResponseInfo – datatype

Table 27 – Reading and Writing datatype description

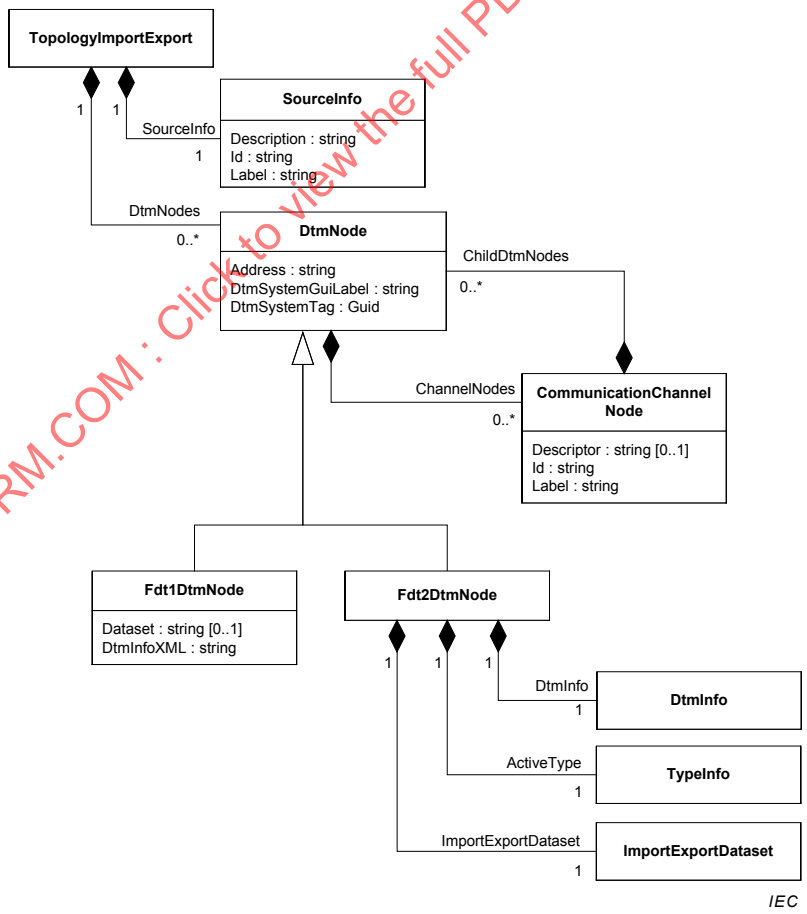
Datatype	Description
BinaryBitArrayValue	A compact array of bit values.
BinaryByteArrayValue	A compact array of byte values.
ByteValue	An 8-bit unsigned integer.
DataAccessError	Information about the type of occurred error
DataAccessErrorInfo	Information about a data access error that occurred.
DataInfo	Information about available data, e.g. parameters and process values
DataValue	Abstract base class for data values provided by a DTM
DateTime	.NET System namespace: Represents an instant in time, typically expressed as a date and time of day
DateTimeValue	DataValue with a DateTime value
DoubleValue	A double-precision (64-bit) floating-point number.
EnumValue	DataValue with an enumerator value

Datatype	Description
FloatValue	A single-precision (32-bit) floating-point number.
IntValue	A 32-bit signed integer.
LimitState	Limit status of device data.
LongValue	A 64-bit signed integer.
QualityIndicator	Quality status of device data.
QualityInfo	Description of the quality of device data
ResponseInfo	Read or write response for a single entry in DataInfo addressed by its Id
StringValue	A string of Unicode characters.
TimeSpanValue	DataValue with a TimeSpan value
UintValue	A 32-bit unsigned integer. Not CLS-compliant.
UlongValue	A 64-bit unsigned integer. Not CLS-compliant.

7.10 Datatypes for export and import

7.10.1 Datatypes – TopologyImportExport

The class TopologyImportExport (see Figure 94) can be used for the data exchange between different Frame Applications. The export contains the FDT topology structure information as well as information about contained (FDT1.2.x / FDT2.x) DTMs and their datasets.



Used in:

<product specific function of Frame Application>

Figure 94 – TopologyImportExport – datatypes

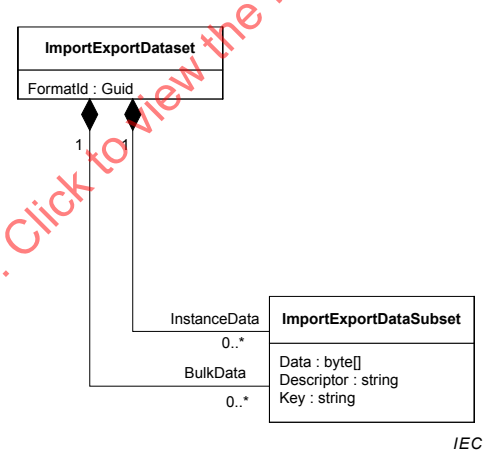
Table 28 describes the TopologyImportExport class and its related classes.

Table 28 – TopologyImportExport datatype description

Datatype	Description
CommunicationChannelNode	Represents a CommunicationChannel in the FDT Topology that is linked to further Child DTM nodes.
DtmInfo	DtmInfo contains general information about a DTM such as name, version, identifier and vendor of the software, the FDT version to which the DTM complies.
DtmNode	Abstract class for a DTM node in the FDT topology.
Fdt1DtmNode	This class represents a FDT1.2.x DTM in a topology export.
Fdt2DtmNode	This class represents a FDT2.x DTM in a topology export.
ImportExportDataset	Dataset containing the exported DTM Data Subsets.
SourceInfo	Information about the source of a topology export like unique identifier, label and description in the Frame Application that has exported the data.
TopologyImportExport	This class can be used for the data exchange between different Frame Applications. It contains the FDT topology structure information as well as information about contained (FDT 1.2.x / 2.x) DTMs and their datasets.
TypeInfo	Abstract base class used for definition of device type, block type or module type. A DTM shall contain one or more TypeInfo objects.

7.10.2 Datatypes – ImportExportDataset

The class ImportExportDataset can be used for the data exchange between different Frame Applications. The dataset contains a DTM dataset (see Figure 95).



Used in:

DataContractSerializer.WriteObject()

Figure 95 – ImportExportDataset – datatypes

Table 29 describes the ImportExportDataset class and its related classes.

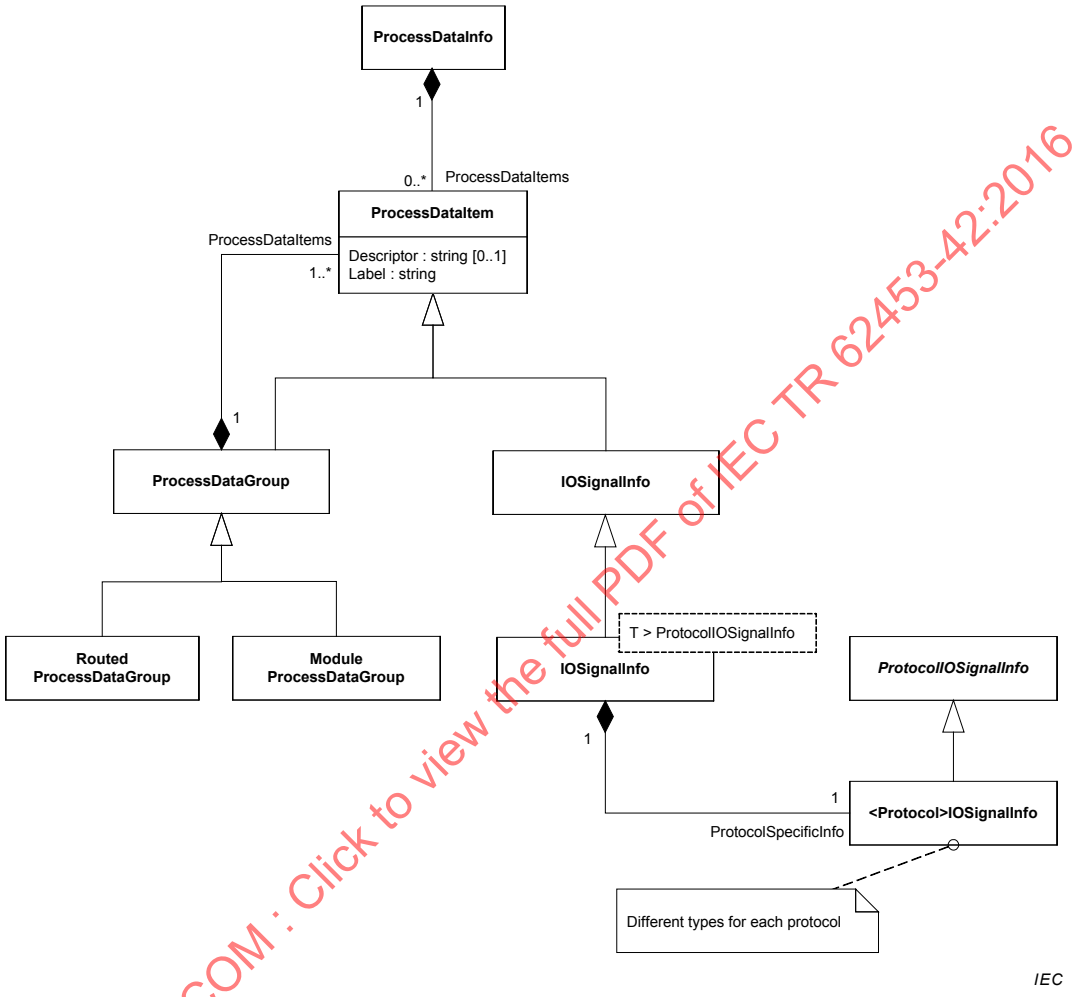
Table 29 – ImportExportDataset datatype description

Datatype	Description
ImportExportDataset	Dataset containing the exported DTM Data Subsets.
ImportExportDataSubset	The DTMDataSubsets contains the exported binary DTM data.

7.11 Datatypes for process data description

7.11.1 Datatypes – ProcessDataInfo

The ProcessDataInfo class provides IO process data related information for the integration of the device into the control system. Figure 96 shows a class diagram with related classes of ProcessDataInfo.



Used in:

Returned in IProcessData.EndGetProcessData()

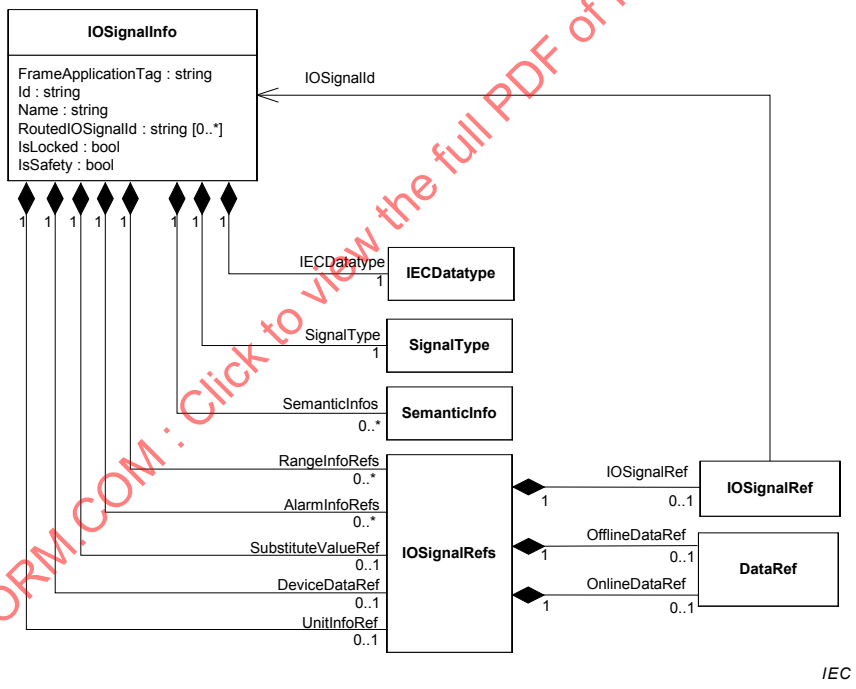
Figure 96 – ProcessDataInfo – datatypes

Table 30 describes the ProcessDataInfo class and its related classes.

Table 30 – ProcessDataInfo datatype description

Datatype	Description
ProcessDataInfo	Process data related information for the integration of the device into the control system like datatype, signal direction, engineering units, and ranges etc.
ProcessDataItem	Abstract base class for process data information.
ProcessDataGroup	Group of ProcessData.
RoutedProcessDataGroup	Information about routed IO signals which are originally provided by a sub-device (corresponding Child DTM in the FDT topology).
ModuleProcessDataGroup	Information about IO signals provided by a DTM module.
IOSignalInfo	Information about a single device IO signal.
IOSignalInfo<T>	Information about a single device IO signal where T is protocol-specific ProtocolIOSignalInfo
ProtocolIOSignalInfo	Abstract base class for protocol-specific IO signal class.
< Protocol>IOSignalInfo	Protocol-specific IO signal class.

The diagram shown in Figure 97 provides more details on IOSignalInfo, which is used not only for ProcessDataInfo but also for ProcessImage information.



Used in:

ProcessDataInfo class

ProcessImageSection class

IProcessData.SetIOSignalInfo()

Figure 97 – IOSignalInfo – datatypes

Table 31 – IOSignalInfo datatype description

Datatype	Description
DataRef	Reference to an item in DataInfo identified by its Id and optionally also information about the type (semantic) of the reference.
IECDatatype	IEC datatype of the IO signal. (automatically set by protocol-specific IO signal class).
IOSignalInfo	Information about a single device IO signal.
IOSignalInfo<T>	Information about a single device IO signal where T is protocol-specific ProtocolIOSignalInfo
IOSignalRef	Reference to an IO signal identified by its identifier.
IOSignalRefs	Reference to another IOSignalInfo, and/or DeviceDataInfo. The meaning of the references depends on the context where this class is used.
SemanticInfo	This class provides semantic information for a data object.
SignalType	Type of the IO signal.

The object diagram shown in Figure 98 shows for example a ProcessDataInfo describing analog input values provided by a HART device. Please be aware that the example shows the expected use of datatypes defined in this document, the definition of HART related datatypes may differ from this example.

NOTE Please be aware that the examples demonstrate how a protocol-specific datatype can be derived from the datatypes defined in this document and how such a protocol-specific datatype is intended to be used. For definition of the protocol-specific datatypes please refer to the respective specification document.

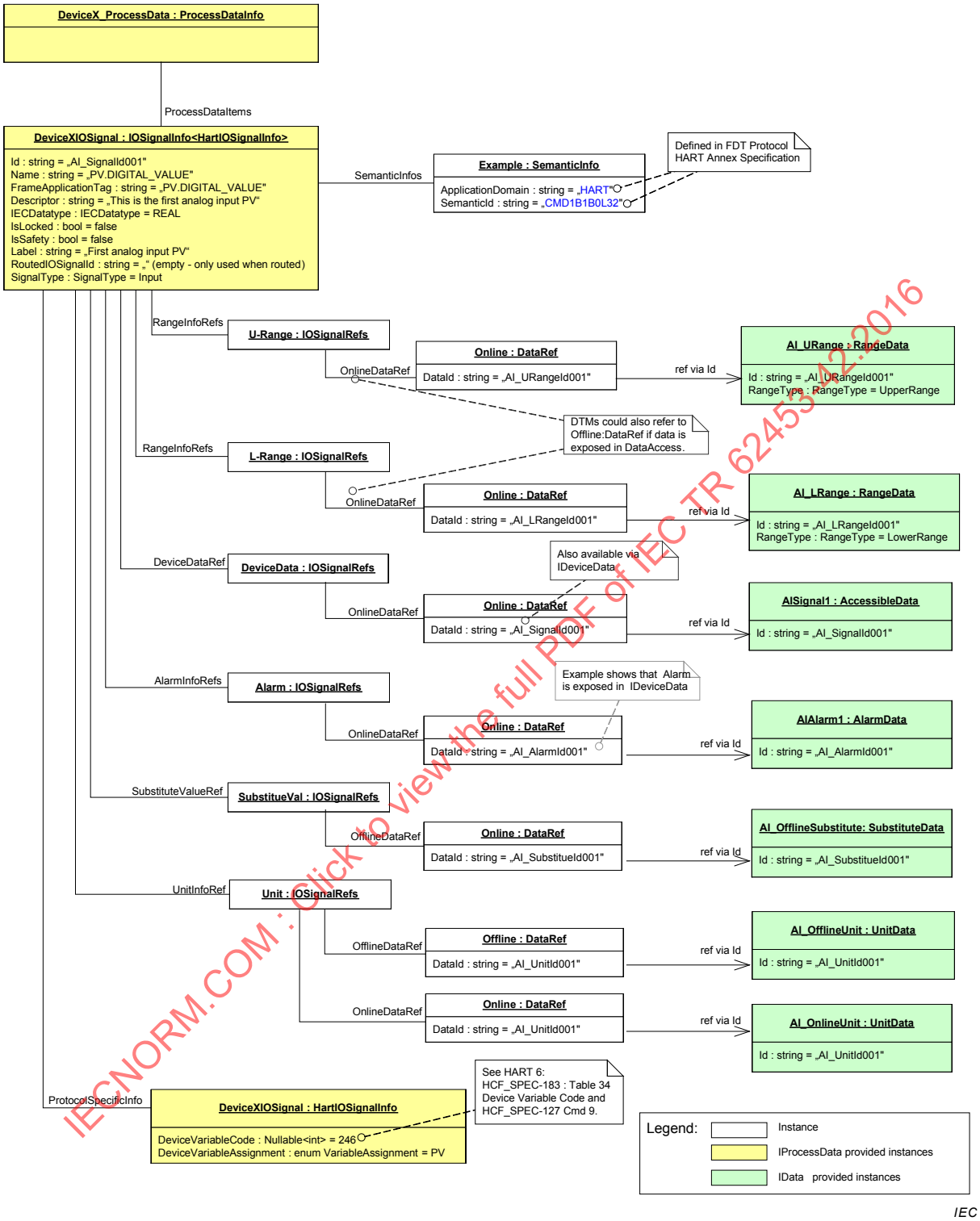


Figure 98 – Example: ProcessDataInfo for HART (UML)

The example in Figure 99 demonstrates how a (HART) Device DTM creates and returns a ProcessDataInfo instance:

```
protected ProcessDataInfo GetProcessData()
{
    // HART PV information
    HartIOSignalInfo hartPVInfo = new HartIOSignalInfo();
    //This value is the Primary Variable
    hartPVInfo.ProcessVariableAssignment = HartIOSignalInfo.VariableAssignment.PV;
    //Specify the index that is needed to read the value via command #9
    //or command #33
    hartPVInfo.Index = 0;

    // HART PV information
    IOSignalInfo<HartIOSignalInfo>  pvInfo = new IOSignalInfo<HartIOSignalInfo>();
    pvInfo.ProtocolSpecificInfo = hartPVInfo;
    pvInfo.Id = "AI_SignalId001";
    pvInfo.Name = "PV.DIGITAL_VALUE";
    pvInfo.Label = "First analog input PV";
    pvInfo.Descriptor = "This is the first analog input PV";
    pvInfo.SignalType = SignalType.Input;
    pvInfo.IECDatatype = IECDatatype.REAL;
    pvInfo.IsLocked = false;
    pvInfo.IsSafety = false;
    //Sematic info as defined in HART FDT Annex
    pvInfo.SemanticInfos = new FdtList<SemanticInfo>(new SemanticInfo("HART",
                                                     "CMD1B1B0L32"));


    // HART PV unit information
    var dataRefPvUnit = new IOSignalRefs();
    dataRefPvUnit.OfflineDataRef = new DataRef("AI_UnitId001");
    pvInfo.UnitInfoRef = dataRefPvUnit;

    // HART PV range information
    var ioURefPvUnit = new IOSignalRefs {IOSignalRef = new IOSignalRef("AI_URangeId001")};
    var ioLRefPvUnit = new IOSignalRefs {IOSignalRef = new IOSignalRef("AI_LRangeId001")};
    pvInfo.RangeInfoRefs = new FdtList<IOSignalRefs>(){ioLRefPvUnit, ioURefPvUnit};
    // other references would come here..
    // pvInfo.SubstituteValueRef = ...

    // Process Data Info
    ProcessDataInfo processData = new ProcessDataInfo();
    processData.ProcessDataItems = new FdtList<ProcessDataItem>();
    processData.ProcessDataItems.Add(pvInfo);

    // other Process Variables may follow here ....
    // ...
    // processData.ProcessDataItems.Add(svInfo);

    return processData ;
}
```

IEC

Figure 99 – Example: ProcessDataInfo creation for HART

The example in Figure 100 demonstrates how a Frame Applications requests and uses a ProcessDataInfo instance. The protocol-specific properties shown in Figure 99 are mapped automatically to the protocol-independent properties which are used in Figure 100.

```
public void ReadIoSignalInfos(IDtm dtm, Guid protocolId)
{
    IAsyncResult asyncResult =
        (dtm as IProcessData).BeginGetProcessData(protocolId, null, null);
    ProcessDataInfo processData =
        (dtm as IProcessData).EndGetProcessData(asyncResult);

    ShowIoSignalInfos(processData.ProcessDataItems);
}

public void ShowIoSignalInfos(FdtList<ProcessDataItem> processDatas)
{
    foreach (ProcessDataItem data in processDatas)
    {
        if (data is IOSignalInfo)
        {
            IOSignalInfo ioSignalInfo = (IOSignalInfo)data;
            MessageBox.Show("ID = " + ioSignalInfo.Id +
                            "Name = " + ioSignalInfo.Name +
                            "Label = " + ioSignalInfo.Label +
                            "SignalType = " + ioSignalInfo.SignalType);
        }
        else
        {
            ShowIoSignalInfos((data as ProcessDataGroup).ProcessDataItems);
        }
    }
}
```

IEC

Figure 100 – Example: Using ProcessData for HART

The example in Figure 101 demonstrates how the HART-specific datatype assembly (Fdt.Datatypes.Hart.dll) exposes the type information over the IOSignalInfoType attribute:

```
[assembly: IOSignalInfoType(
       IOSignalInfoType = typeof(IOSignalInfo<HartIOSignalInfo>),
       ProtocolIOSignalInfoType = typeof(HartIOSignalInfo))
]
```

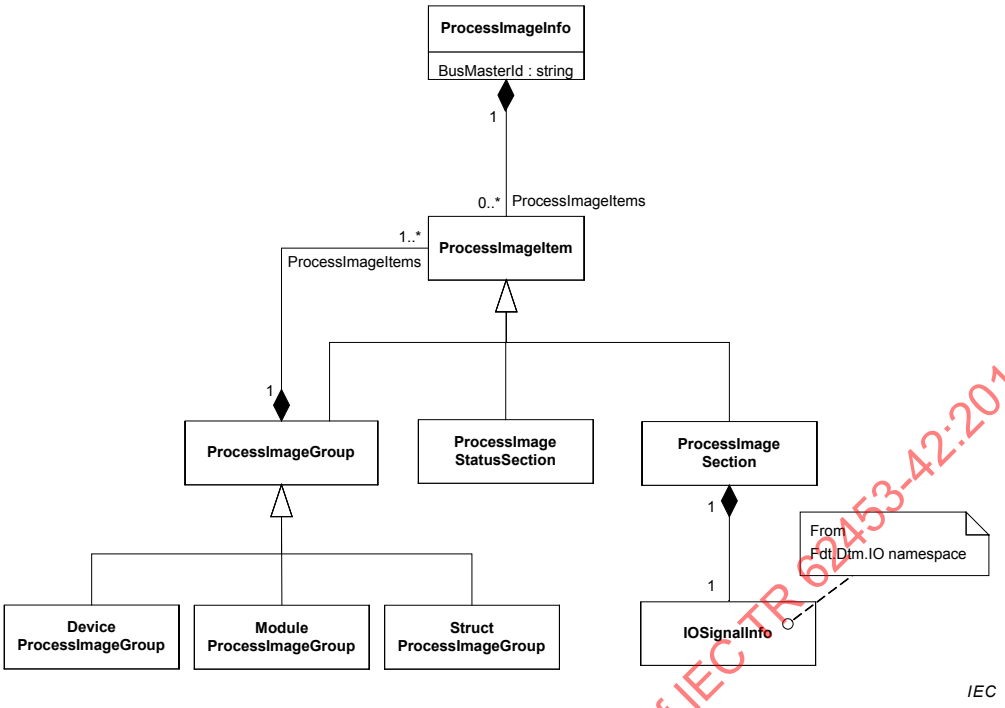
IEC

Figure 101 – Example: IOSignalInfoType attribute

NOTE Please be aware that the above examples demonstrate how a protocol-specific datatype can be derived from the datatypes defined in this document and how such a protocol-specific datatype is intended to be used. For definition of the protocol-specific datatypes please refer to the respective specification document.

7.11.2 Datatypes – Process Image

The ProcessImageInfo class provides information about the process image by the bus-master device which is represented by the DTM. Figure 102 shows a class diagram with related classes of ProcessImageInfo.



Used in:

Returned in IProcessImage.EndGetProcessImageInfo()

Figure 102 – ProcessImage – datatypes

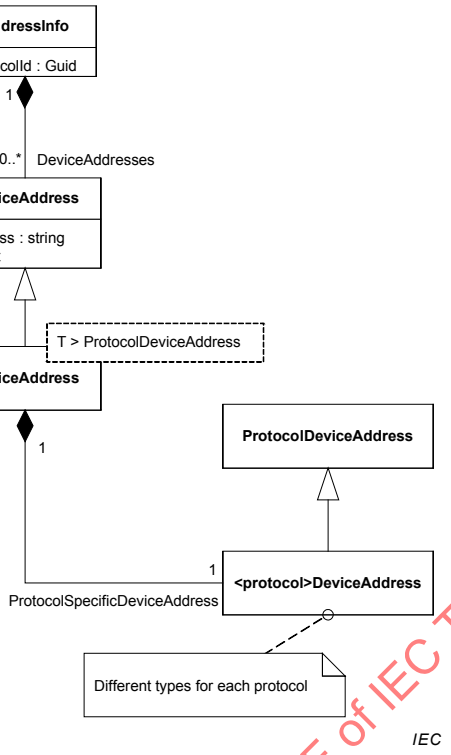
Table 32 describes ProcessImage classes.

Table 32 – ProcessImage datatype description

Datatype	Description
DeviceProcessImageGroup	Groups process image items belonging to a specific device connected to the fieldbus.
IOSignalInfo	Information about a single IO signal. NOTE The class is also used in IProcessData interface.
ModuleProcessImageGroup	Groups process image items belonging to a specific device module connected to the fieldbus.
ProcessImageInfo	Information about the fieldbus master process image, which enables for example engineering tools to map the device I/O signals to variables in an IEC program for a PLC.
ProcessImageItem	Abstract base class for process image information.
ProcessImageGroup	Groups process image information.
ProcessImageSection	Represents a single process image section in which an IO signal is mapped.
StructProcessImageGroup	Groups of process image items belonging to a structure IO signal.

7.12 Datatypes – Address information

The AddressInfo class provides information about address(es) of the device which is represented by the DTM. Figure 103 shows a class diagram with related classes of AddressInfo.



Used in:

AddressInfo is returned by INetworkData.GetAddressInfo()

Single DeviceAddresses can be set by INetworkData.SetAddressInfo()

Figure 103 – AddressInfo – datatypes

Table 33 describes AddressInfo class and its related classes.

Table 33 – AddressInfo datatype description

Datatype	Description
AddressInfo	Information about address(es) of the device which is represented by the DTM.
DeviceAddress	Address of the device in the network or fieldbus. TheDeviceAddress.Id is used to indicate the relation to the corresponding NetworkData (which has the same Id value).
DeviceAddress<T>	Address of the device in the network or fieldbus. NOTE T represents the protocol-specific class which defines the protocol-specific address properties.
ProtocolDeviceAddress	Abstract base class for protocol-specific device addresses.

The example in Figure 104 demonstrates how a (HART) Device DTM creates and returns a AddressInfo instance:

```
public AddressInfo CreateAddressInfoForHart(int shortAddress,
                                            string shortTag,
                                            string longTag,
                  HartDeviceAddress.AddressingModeSelection addressingMode,
                  HartLongAddress longAddress)
{
    var hartDeviceAddress = new HartDeviceAddress(shortAddress, shortTag,
                                                  longTag, addressingMode,
                                                  longAddress);
    DeviceAddress<HartDeviceAddress> deviceAddress =
    new DeviceAddress<HartDeviceAddress>();
    deviceAddress.ProtocolSpecificDeviceAddress = hartDeviceAddress;
    deviceAddress.Id = 1;
    AddressInfo addressInfo = new AddressInfo(Fdt.Hart.HartFskInfo.ProtocolId);
    addressInfo.DeviceAddresses = new FdtList<DeviceAddress>(deviceAddress);
    return addressInfo;
}
```

IEC

Figure 104 – Example: AddressInfo creation

The example in Figure 105 demonstrates how a Frame Application requests and uses the AddressInfo instance created in Figure 104:

```
public void ShowDeviceAddress(IDtm dtm, Guid protocolId)
{
    INetworkData networkData = dtm as INetworkData;
    if(networkData == null)
    {
        //this shall never happen because INetworkData is mandatory
        MessageBox.Show("Failure: DTM does not provide INetworkData");
        return;
    }

    AddressInfo addressInfo = networkData.GetAddressInfo(protocolId);
    if (addressInfo == null)
    {
        //this shall never happen because return value of
        //GetAddressInfo() shall never be null
        MessageBox.Show("Failure: DTM does not provide AddressInfo");
        return;
    }

    //Verify result
    try
    {
        addressInfo.Verify();
    }
    catch (Exception ex)
    {
        MessageBox.Show("Failure in verification of AddressInfo:" + ex.Message);
        return;
    }

    if (addressInfo.DeviceAddresses == null)
    {
        //This may happen if the protocol doesn't define an addressing mechanism
        MessageBox.Show("Device provides no address information.");
    }
    else
    {
        foreach (DeviceAddress deviceAddress in addressInfo.DeviceAddresses)
        {
            //verify DeviceAddress
            try
            {
                deviceAddress.Verify();
            }
            catch (Exception ex)
            {
                MessageBox.Show("Failure in verification of DeviceAddress:"
                    + ex.Message);
                return;
            }

            MessageBox.Show("Device Address = " + deviceAddress.Address);
        }
    }
}
```

IEC

Figure 105 – Example: Using AddressInfo

The example in Figure 106 demonstrates how the HART-specific datatype assembly (Fdt.Datatypes.Hart.dll) exposes the type information over the DeviceAddressInfoAttribute:

```
[assembly: DeviceAddressType(
            DeviceAddressType = typeof(DeviceAddress<HartDeviceAddress>),
            ProtocolDeviceAddressType = typeof(HartDeviceAddress))
]
```

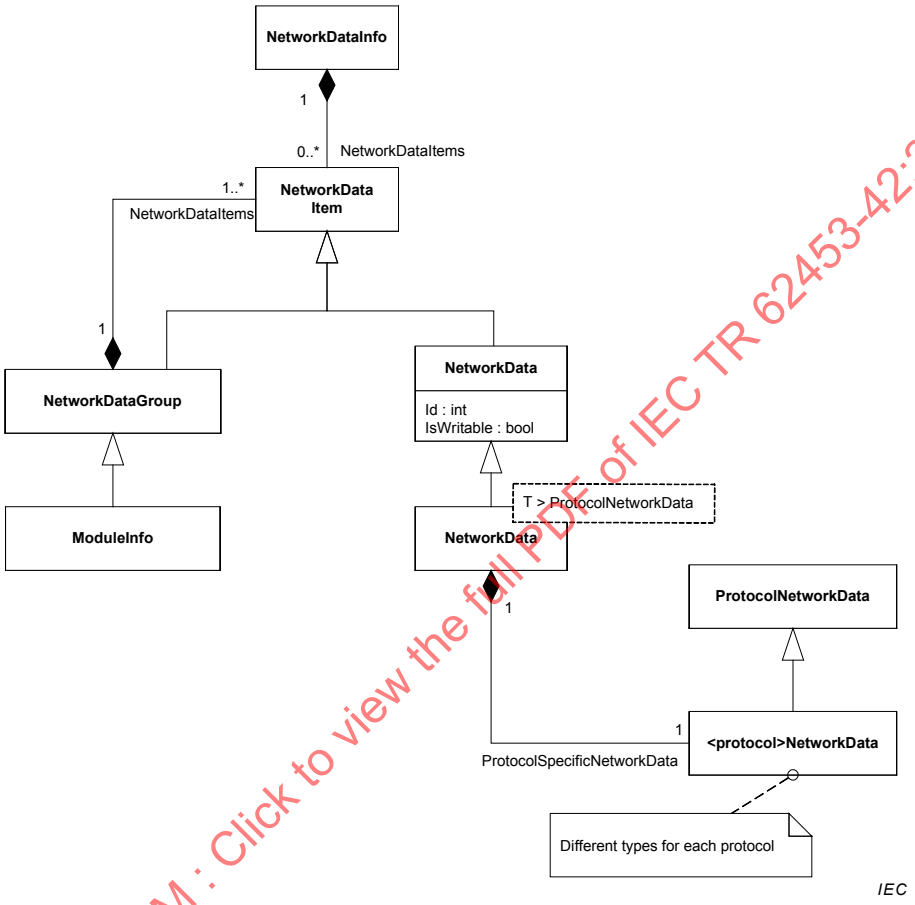
IEC

Figure 106 – Example: DeviceAddressTypeAttribute

NOTE Please be aware that the above examples demonstrate how a protocol-specific datatype can be derived from the datatypes defined in this document and how such a protocol-specific datatype is intended to be used. For definition of the protocol-specific datatypes please refer to the respective specification document.

7.13 Datatypes – NetworkDataInfo

The NetworkDataInfo class provides network management information which can for example be used for bus master configuration. Figure 107 shows a class diagram with related classes of NetworkDataInfo.



Used in:

NetworkDataInfo is returned by INetworkData.GetNetworkDataInfo()

Single NetworkData items can be set by INetworkData.SetNetworkData()

Figure 107 – NetworkDataInfo – datatypes

Table 34 describes NetworkDataInfo class and its related classes.

Table 34 – NetworkDataInfo datatype description

Datatype	Description
ModuleInfo	Represents a hardware or software module of the device. It provides general information like name, version, vendor and may also contain further NetworkDataItems providing protocol-specific information.
NetworkData	Base class for a single protocol independent network data item. The NetworkData.Id is used to indicate the relation to the corresponding DeviceAddress (which has the same Id value).
NetworkData<T>	Represents a single protocol-specific network data item. NOTE T represents a protocol-specific class which defines the protocol-specific network data properties.
NetworkDataGroup	Group of network data items provided by the DTM.
NetworkDataInfo	Contains network-specific information about the device.
NetworkDataItem	Abstract base class for network data classes.
ProtocolNetworkData	Abstract base class for protocol-specific network data information classes.

The example in Figure 108 demonstrates how a (PROFIBUS) Device DTM creates and returns a NetworkDataInfo instance:

```
public NetworkDataInfo GetNetworkDataInfo(Guid protocolId)
{
    // verify protocolId
    // ...

    // create network data for Profibus DP/V1 and set properties
    ProfibusNetworkData networkData = new ProfibusNetworkData();
    networkData.PrmDataIdentNumber = 0x1234;
    networkData.PrmDataMinTsdr = 11;
    networkData.PrmDataFreezeMode = false;
    networkData.CfgData = new byte[3];
    networkData.CfgData[0] = 0x30;
    networkData.CfgData[1] = 0x42;
    networkData.CfgData[2] = 0x27;
    // ...
    // ... (properties are device specific)
    // ...

    var nwdi = new NetworkDataInfo(protocolId, false);
    nwdi.NetworkDataItems = new Fdt.FdtList<NetworkDataItem>(
                    new NetworkData<ProfibusNetworkData>(1,networkData));
    return nwdi;
}
```

IEC

Figure 108 – Example: NetworkDataInfo creation example

The example in Figure 109 demonstrates how a Communication DTM (representing a bus-master device) requests and uses the NetworkDataInfo instance created in Figure 108:

```
public void CheckNetworkData(INetworkData networkData, Guid protocolId)
{
    var dtmNetworkData = networkData.GetNetworkDataInfo(protocolId);
    if (dtmNetworkData != null)
    {
        // network data available
        // Verify the data (if invalid data, the method exits with an exception)
        dtmNetworkData.Verify();

        foreach (NetworkDataItem item in dtmNetworkData.NetworkDataItems)
        {
            if ((item is NetworkData<ProfibusNetworkData>))
            {
                var pbDataItem = item as NetworkData<ProfibusNetworkData>;
                ProfibusNetworkData pbData = pbDataItem.ProtocolSpecificNetworkData;

                // use the network data to check master configuration
                // ...
            }
        }
    }
}
```

IEC

Figure 109 – Example: NetworkDataInfo using example

The example in Figure 110 demonstrates how the PROFIBUS-specific datatype assembly (Fdt.Datatypes.Profibus.dll) exposes the type information over the NetworkDataTypeAttribute:

```
[assembly: NetworkDataType(
            NetworkDataType = typeof(NetworkData<ProfibusNetworkData>),
            ProtocolNetworkDataType = typeof(ProfibusNetworkData))
]
```

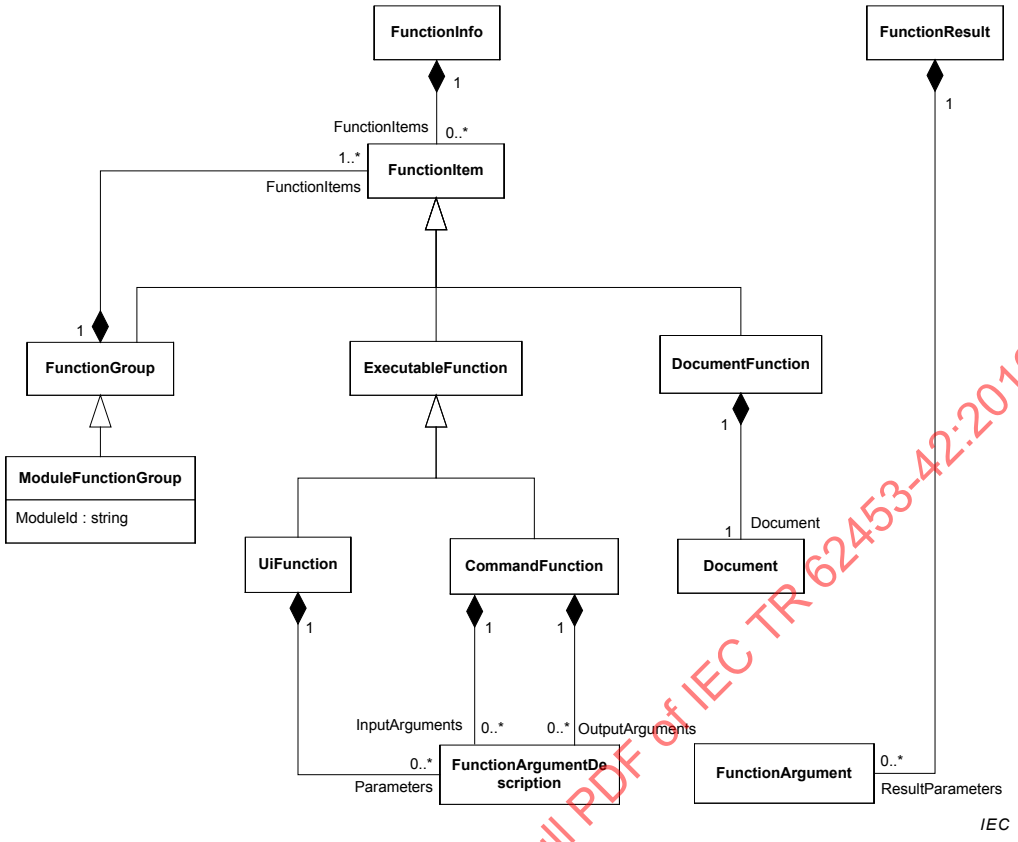
IEC

Figure 110 – Example: NetworkDataTypeAttribute example

NOTE Please be aware that the above examples demonstrate how a protocol-specific datatype can be derived from the datatypes defined in this document and how such a protocol-specific datatype is intended to be used. For definition of the protocol-specific datatypes please refer to the respective specification document.

7.14 Datatypes – DTM functions

The following class diagram (Figure 111) describes the relations of classes in the context of FunctionInfo.



Used in:

IFunction.FunctionInfo

ICommandFunction.BeginExecute()

ICommandFunction.EndExecute()

Figure 111 – DTM Function – datatypes

A DTM exposes all functions it provides in FunctionInfo, which can contain one or more functions. Each of the functions can be a function providing one or more documents, an executable function or a function group. Function groups contain one or more functions. ModuleFunctionGroup is a special FunctionGroup. Two types of executable functions are distinguished: a function which requires opening a user interface and a function, which is performed in the background without a user interface. UiFunctions provide one or more FunctionArguments, which describe function-specific information. For a list of functions with user interface see 5.9.1. CommandFunctions define InputParameters as well as ResultParameters.

A DTM should not make any assumption in regard to how a Frame Application represents the available functions of a DTM. For different use cases and on different platforms there are alternative ways of presenting this information to the user. That is why a DTM should not provide any customization(e.g. menu accelerators) for menus or for other GUI elements displaying the function list.

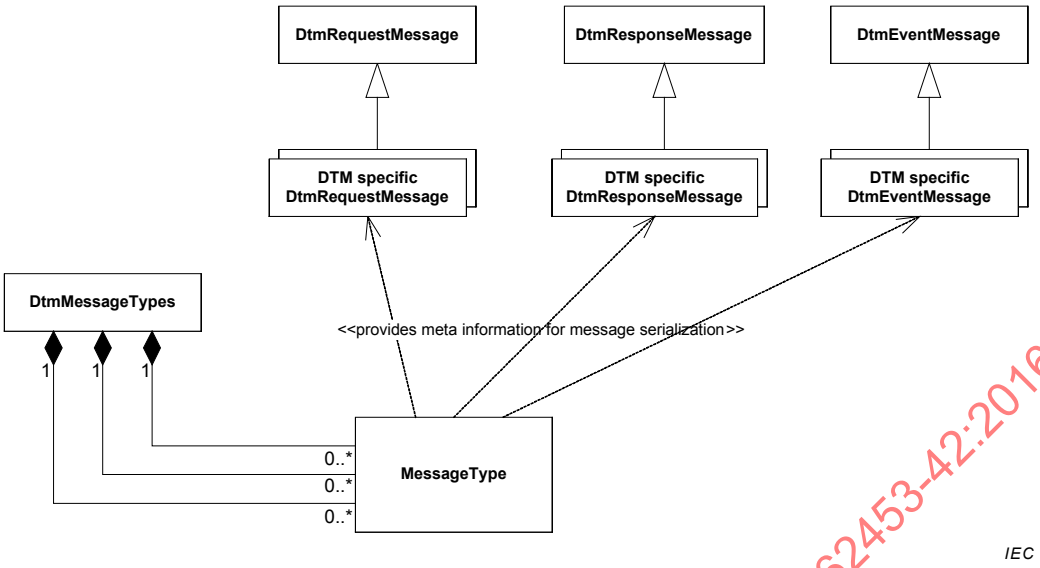
Table 35 describes datatypes in Fdt.Dtm.Functions namespace.

Table 35 – DTM Function datatype description

Datatype	Description
FunctionInfo	Returns information about functions, user interfaces and documents provided by a DTM.
FunctionItem	Abstract base class for a DTM function description class.
ExecutableFunction	Abstract base class for functions of a DTM which are “executable” by calling corresponding interface on the DTM Business Logic or creating a UI object.
DocumentFunction	Description of a document (file) provided by the DTM.
FunctionGroup	Group of DTM function descriptions.
FunctionResult	Result of a command function or a modal user interface.
UiFunction	Description of a graphical DTM User Interface.
CommandFunction	Description of a non-GUI function provided by the DTM Business Logic.
FunctionArgument	Information about a parameter of a CommandFunction or UiFunction.
Document	Information about a document on file disk or in the Web.

7.15 Datatypes – DTM messages

The class diagram shown in Figure 112 describes the relation of classes used for interaction between DTM Business Logic and DTM User Interface as well as for interaction between different instances of DTM Business Logic of two related DTMs (e.g. for a Composite DTM).



Used in:

IDtmUiMessaging.BeginSendMessages()

IDtmUiMessaging.EndSendMessages()

Event IDtmUiMessaging.DtmSpecificEventOccured()

IDtmUiMessaging.UiMessageTypes

And

IDtmMessaging.BeginSendMessages()

IDtmMessaging.EndSendMessages()

IDtmMessaging.PrivateMessageTypes

Figure 112 – DTM Messages – datatypes

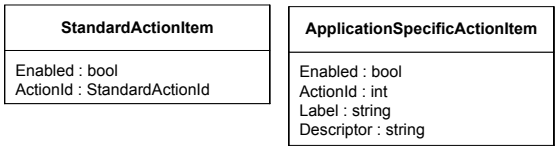
Table 34 describes datatypes related to DTM Messages.

Table 36 – DTM Messages datatype description

Datatype	Description
DtmRequestMessage	This abstract class serves as a base for interaction between the DTM User Interface and the DTM Business Logic as well as between DTMs (proprietary DTM to DTM interaction). This class encapsulates a message where a DTM UI or a DTM Business Logic requests information from a DTM Business Logic. DTMs shall derive own classes from DtmRequestMessage and use these for the interaction.
DtmResponseMessage	This abstract class serves as a base for interaction between the DTM User Interface and the DTM Business Logic as well as between DTMs (proprietary DTM to DTM interaction). This class encapsulates a message where a DTM Business Logic responses to a previous request. DTMs shall derive own classes from DtmResponseMessage and use these for the interaction.
DtmEventMessage	This abstract class serves as a base for interaction between the DTM Business Logic and DTM User Interface. DTMs shall derive own classes from DtmEventMessage and use these for the interaction.

7.16 Datatypes for delegation of DTM UI dialog actions

The ActionItem classes (see Figure 113) are used by DTMs to expose the status of their standard dialog actions and the set and status of application-specific actions.



Used in:

ApplicationSpecificActionSet delegate

Event IStandardActions.StandardActionItemSetChanged()

Event IApplicationSpecificActions.ApplicationSpecificActionItemSetChanged()

Figure 113 – ActionItem – datatypes

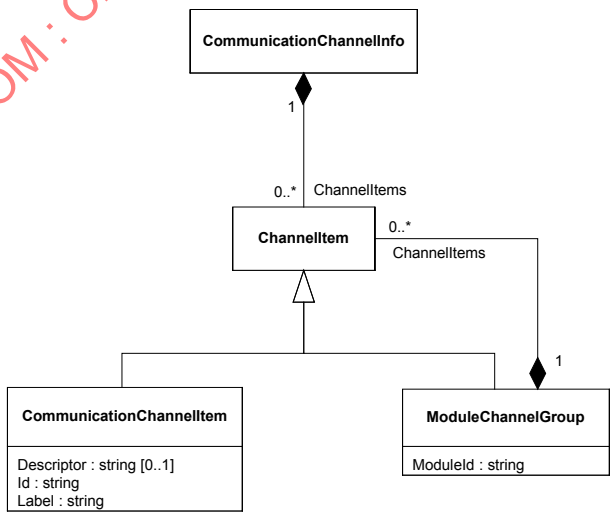
Table 37 describes datatypes related to ActionItem.

Table 37 – ActionItem datatype description

Datatype	Description
StandardActionItem	Represents standard DTM UI dialog actions with a predefined semantic meaning
ApplicationSpecificActionItem	Represents DTM UI dialog actions which do not have a predefined semantic meaning (application-specific)

7.17 Datatypes – CommunicationChannelInfo

The CommunicationChannelInfo class provides information about the modules and Communication Channels of a DTM (see Figure 114)



Used in:

IChannels.CommunicationChannelInfos

Figure 114 – CommunicationChannelInfo – datatypes

Table 38 describes CommunicationChannelInfo class and its related classes.

Table 38 – CommunicationChannelInfo datatype description

Datatype	Description
CommunicationChannelInfo	Information about Communication Channels (and modules) supported by a DTM.
ChannelItem	Abstract base class for module and Communication Channel information
ModuleChannelGroup	Information about a group of Communication Channels or underlying modules of a DTM.

The example in Figure 115 demonstrates how channel information is provided by a DTM with two modules:

```
//Member variables for channel info and channel objects
CommunicationChannelInfo _myChannelInfo;
Dictionary<string,ICommunication> _myCommChannels;

private void buildChannelInfos()
{
    //create first module info
    ModuleChannelGroup module1 = new ModuleChannelGroup("Module1");
    module1.ChannelItems = new FdtList<ChannelItem>();
    CommunicationChannelItem channelModule1 =
        new CommunicationChannelItem("Module1.Chn1", "Channel of Module1");

    module1.ChannelItems.Add(channelModule1);

    //create second module info
    ModuleChannelGroup module2 = new ModuleChannelGroup("Module2");
    module2.ChannelItems = new FdtList<ChannelItem>();
    CommunicationChannelItem channelModule2 =
        new CommunicationChannelItem("Module2.Chn1", "Channel of Module2");

    module2.ChannelItems.Add(channelModule2);

    //create info list
    _myChannelInfo = new CommunicationChannelInfo();
    _myChannelInfo.ChannelItems = new FdtList<ChannelItem>();
    _myChannelInfo.ChannelItems.Add(module1);
    _myChannelInfo.ChannelItems.Add(module2);

    //create Communication Channel objects and add them to dictionary
    _myCommChannels = new Dictionary<string, ICommunication>();

    MyCommChannelType channel1 = new MyCommChannelType();
    _myCommChannels.Add(channelModule1.Id, channel1);

    MyCommChannelType channel2 = new MyCommChannelType();
    _myCommChannels.Add(channelModule2.Id, channel2);

}

//Implementation of IChannels Members
public CommunicationChannelInfo ChannelInfos
{
    get { return _myChannelInfo; }
}

public IEnumerable<KeyValuePair<string, ICommunication>> CommunicationChannels
{
    get { return _myCommChannels; }
}
```


IEC

Figure 115 – Example: Channel information

7.18 Datatypes – HardwareIdentification and scanning

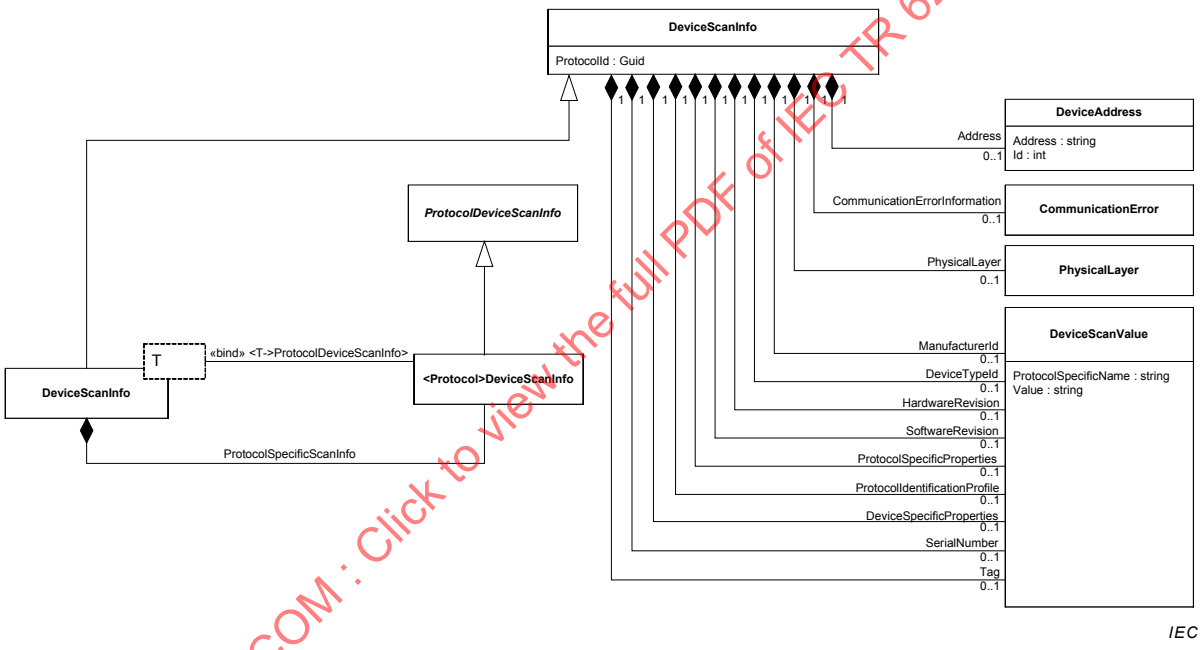
7.18.1 General

The interface IHardwareInformation is used by a Frame Application to detect if specific communication hardware is available or to request information from a field device. The method EndHardwareScan() returns a list of DeviceScanInfo. If the method is used to detect communication hardware, multiple DeviceScanInfo entries may be provided. If the method is used to request information from a field device, a single entry will be returned only.

7.18.2 Datatypes – DeviceScanInfo

The methods of IScanning are used to scan the sub-topology of a Communication Channel. The result of the scan is returned in DeviceScanInfo and contains protocol-specific identification information of found devices (see Figure 116).

NOTE DeviceScanInfo properties contain protocol independent identification information as name value pairs in string format. This allows a Frame Application to display basic identification information of a scanned device in a human readable style even if the protocol-specific types are unknown.



Used in:

IHardwareInformation.EndHardwareScan()

IScanning.EndScanRequest()

Figure 116 – DeviceScanInfo – datatypes

Table 39 describes the classes related to DeviceScanInfo

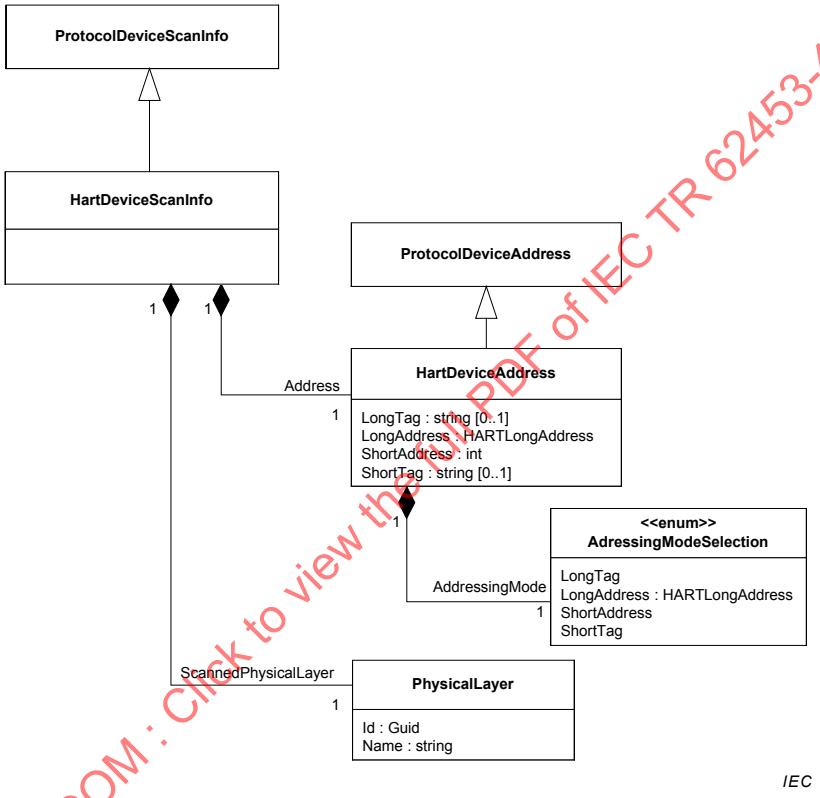
Table 39 – DeviceScanInfo datatype description

Datatype	Description
DeviceAddress	Abstract base class for protocol-specific device address. For scan result the value of DeviceAddress.Id shall be set to 0.
CommunicationError	Description of a fieldbus protocol independent error occurred during nested communication
DeviceScanInfo	This class is used to describe information from one single scanned physical device

Datatype	Description
DeviceScanValue	Represents an identification element of a scanned device. For example: Device Type Id, Manufacturer Id etc.
<Protocol>DeviceScanInfo	Is a placeholder for a DeviceScanInfo of a specific protocol. Example: HARTDeviceScanInfo
DeviceScanInfo<T>	This class is used to describe information from scanned physical devices
ProtocolDeviceScanInfo	Abstract base class for protocol-specific scan properties.

7.18.3 Example – HardwareIdentification and scanning for HART

Figure 117 shows for example the properties of the HartDeviceScanInfo Datatype.



Used in:

IHardwareInformation.EndHardwareScan()

IScanning.EndScanRequest()

Figure 117 – Example: HARTDeviceScanInfo – datatype

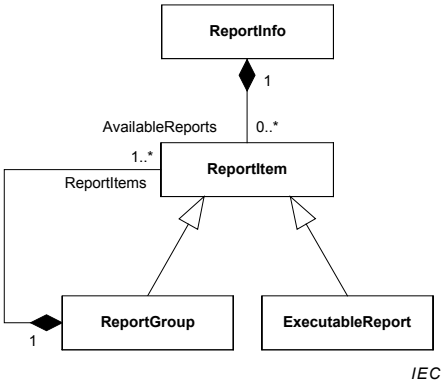
Table 40 describes classes related to HARTDeviceScanInfo

Table 40 – Example: HARTDeviceScanInfo datatype description

Datatype	Description
ProtocolDeviceScanInfo	Abstract base class for protocol-specific scan properties.
HARTDeviceScanInfo	Provides protocol-specific information returned in ScanRequest().
ProtocolDeviceAddress	Abstract base class for protocol-specific device addresses.
HartDeviceAddress	HART-specific device address.

7.19 Datatypes – DTM report types

The ReportInfo class is used by a DTM to expose information about the report types it implements. Figure 118 shows the involved classes and their relations.



Used in:

IReporting.Reports

Figure 118 – DTM Report – datatypes

A ReportInfo object comprehends the description of one or many report types. Each report type (ExecutableReport) has a unique identifier, which can be used by a Frame Application to request a specific report from a DTM. Report types may be arbitrarily grouped (ReportGroups). They may have references to an ApplicationId, that associates them with an FDT standard functionality (see definition of ApplicationID in Annex B and in Annex A), or they may have a reference to a FunctionId, that links the report type to a DTM-specific functionality.

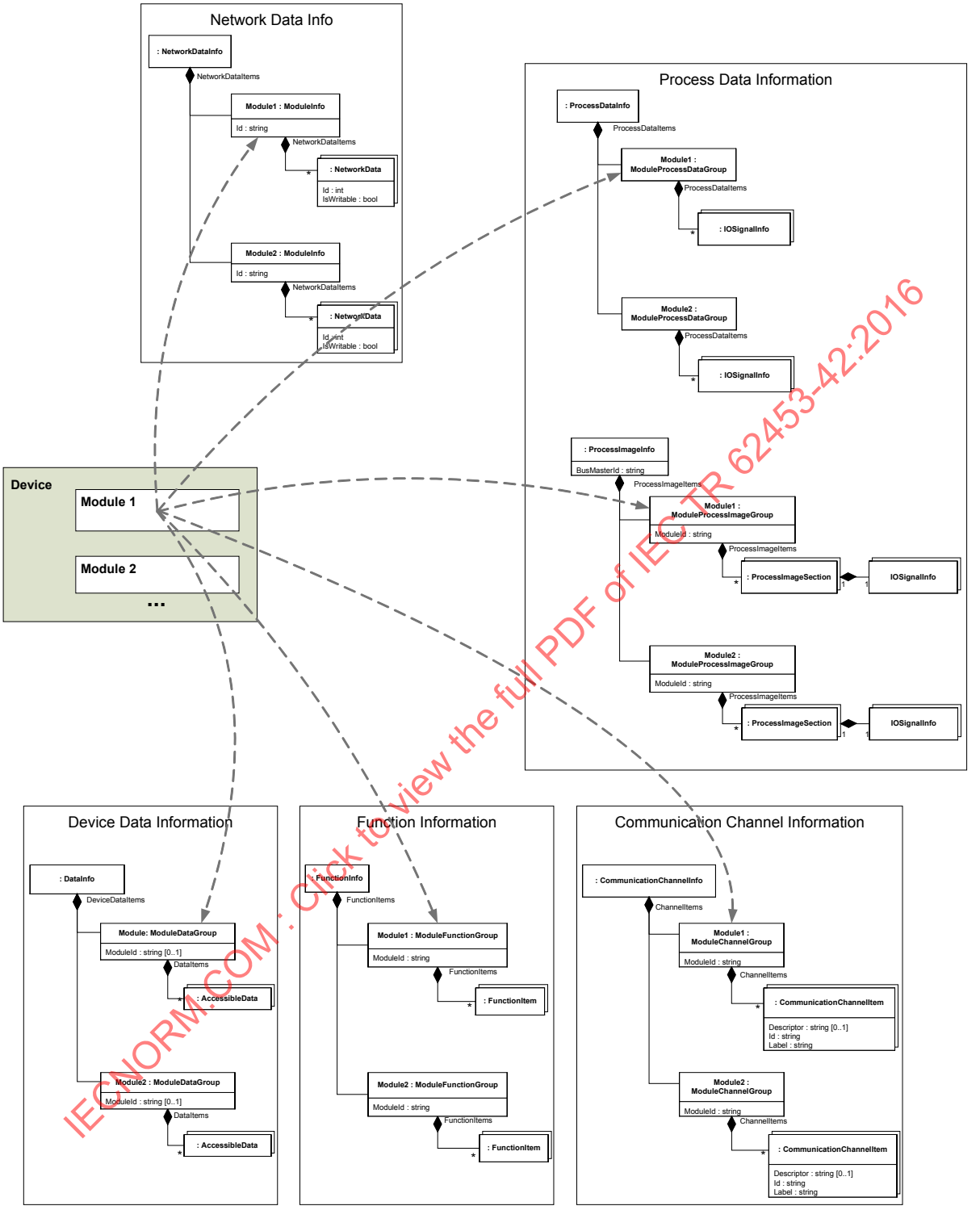
Table 41 summarizes the datatypes in the Fdt.Dtm.Reporting namespace.

Table 41 – Reporting datatype description

Datatype	Description
ExecutableReport	Information about one specific report provided by the DTM.
ReportGroup	Group of DTM report descriptions.
ReportInfo	Provides information about reports provided by a DTM.
ReportItem	Abstract base class for a report description class.

7.20 Information related to device modules in a monolithic DTM

A monolithic DTM provides information about a device with all its modules. The information regarding the modules is distributed on different datatypes. Figure 119 shows an example with involved data.



IEC

Figure 119 – Information related to device modules

Inside a monolithic DTM the modules of a device are identified by a unique ModuleId.

NOTE Examples for monolithic DTMs are DTMs for PROFIBUS PA devices, where a module would represent a function block or transducer block, or DTMs for modular devices, where a module would represent a hardware module.

The same ModuleId is used in the different datatypes (ModuleInfo, ModuleProcessDataGroup, ModuleProcessImageGroup, ModuleDataGroup, ModuleFunctionGroup, and ModuleChannelGroup) in order to show that this information describes the same module.

For a monolithic DTM it is expected that the data in IInstanceData and IDeviceData are also grouped for the modules (see Figure 88).

With ModuleFunctionGroup it is possible to provide functions specifically for modules.

If the device is a modular device and if modules provide communication, it is possible to use ModuleChannelGroup in order to associate the provided CommunicationChannels to their respective modules.

8 Workflows

8.1 General

The work flows provided in this chapter are intended to explain the expected behavior. Implemented behavior may vary, but should follow the general rules explained here and in the interface definitions.

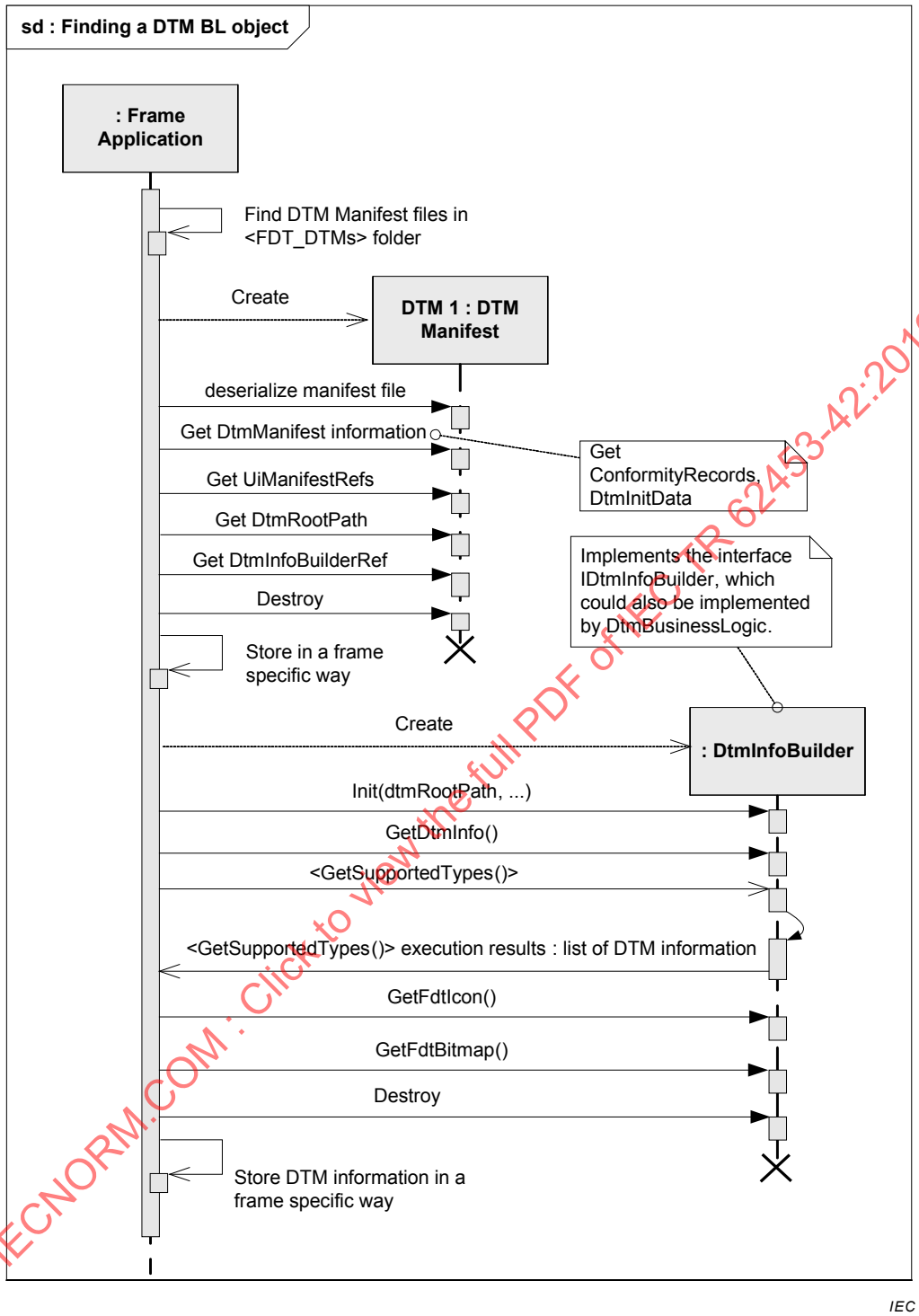
The conventions for sequence diagrams are explained in H.5.

As explained in 5.6.2 all component interactions are passed through the Frame Application or through proxy components. Since this passage shall not change interactions or inject interaction requests, it will not change the general sequence of message calls. In order to simplify the representation of sequences, the proxy objects often are omitted in the sequence diagrams in this section. If the proxy objects are important to understand the sequence of message calls, then they are shown in the sequence diagrams.

8.2 Instantiation, loading and release

8.2.1 Finding a DTM BL object

In order to execute a DTM, the Frame Application needs to find the respective DTM Business Logic object, which is located in an assembly. This section describes the sequence of finding the DTM BL object (see Figure 120).



Used methods:

IDtmInfoBuilder.Init()

IDtmInfoBuilder.GetDtmInfo()

IDtmInfoBuilder.BeginGetSupportedTypes() / IDtmInfoBuilder.EndGetSupportedTypes()

IDtmInfoBuilder.GetFdtIcon()

IDtmInfoBuilder.GetFdtBitmap()

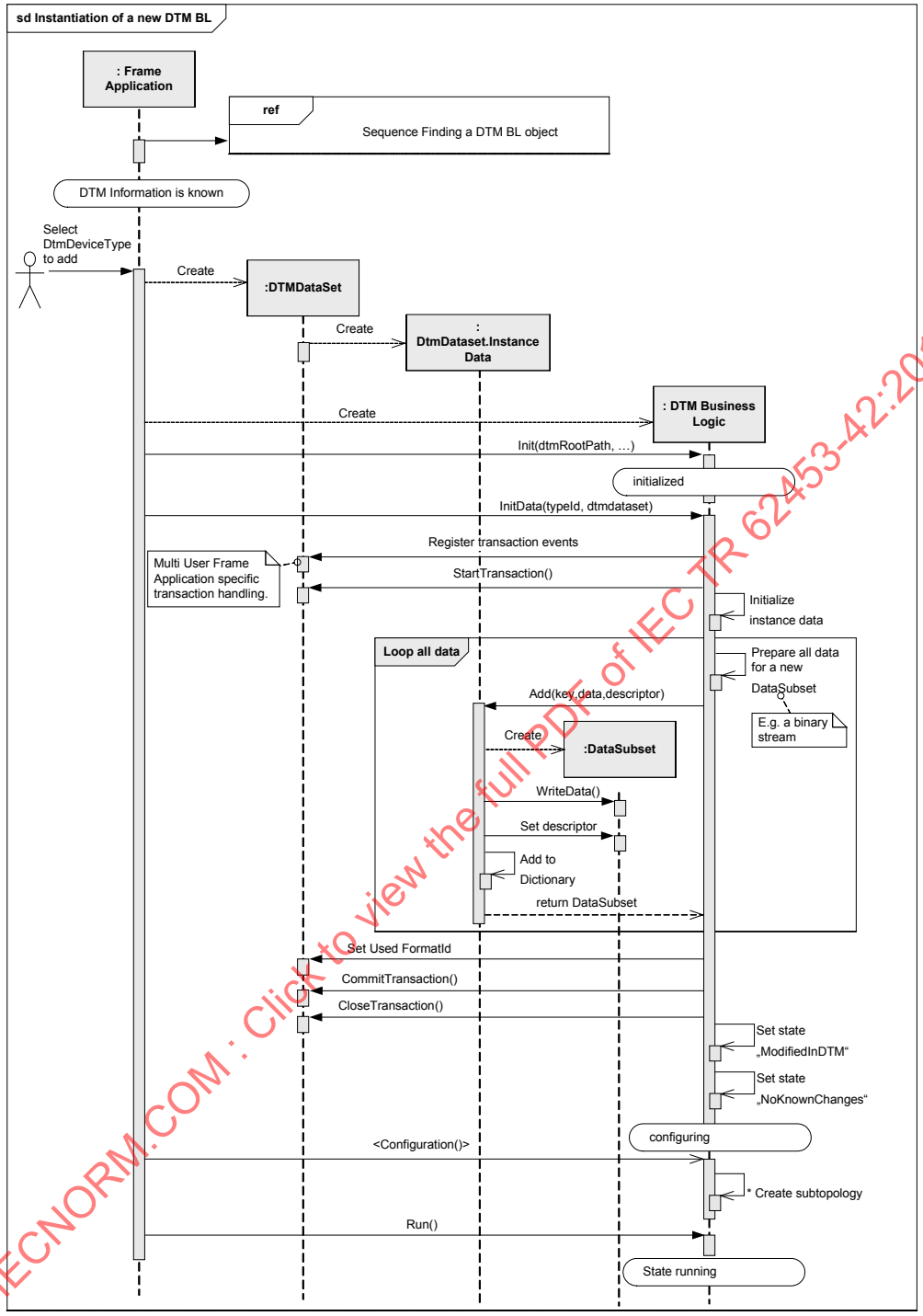
Figure 120 – Finding a DTM BL object

Typically a Frame Application stores the information about the DTM BL in a catalogue. For a more complete sequence for updating the device catalogue refer to 10.4.3.

8.2.2 Instantiation of a new DTM BL

A new DTM Business Logic is instantiated by the Frame Application with its full assembly class name. The class name can be looked up in the DtmManifest for the selected DtmDeviceType.

The Frame Application shall create a new DTMDataset object and pass a reference to IDataSet as a parameter in IDtm.InitData() to the newly created DTM Business Logic instance. Within the InitData() call, the DTM Business Logic instance adds new DTMDataSubsets to the DTMDataset and writes its instance data into the DTMDataSubsets (see Figure 121).



Used methods:

IDtm.Init()

IDtm.InitData()

IDataset.StartTransaction()

IDataset.CommitTransaction()

IDataset.CloseTransaction()

Figure 121 – Instantiation of a new DTM BL

8.2.3 Configuring access rights

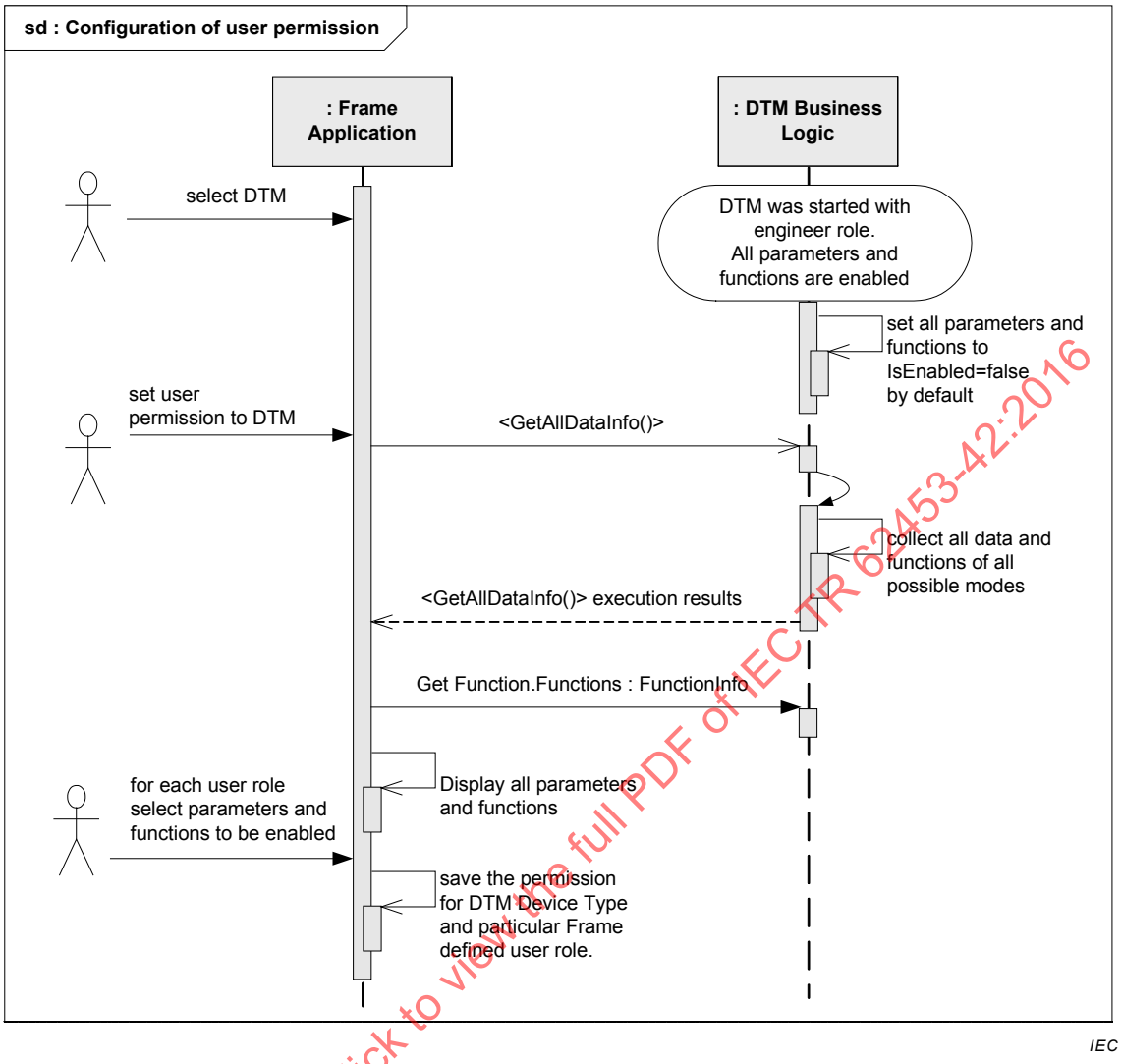
The Frame Application can provide a separate function to configure the access rights for individual users or for group of users, which will be working with the DTM. The function for Access Rights configuration is usually available only for System Administrators, which are responsible for the security of the plant.

To configure the access rights, the administrator has to instantiate the DTM with access rights set to Engineer, get the list of all Data provided by the DTM by using ICustomConfiguration::<GetAllDataInfo()>. The Administrator will get the list of all functions from IFunction.FunctionInfo property. The Administrator will use a specialized user interface, provided by the Frame Application to define the permissions for changing data and invoking functions.

This information shall be saved by the Frame Application and used when the DTM of this type is instantiated.

The user can invoke the DTM with Expert user level (see 8.2.5) and verify the correctness of the settings. The Administrator may come back to the specialized user interface, provided by the Frame Application, correct the permissions, save the data with the rest frame data and invoke the DTM again.

The following sequence diagram illustrates the configuration of the user permissions, when custom role is invoked (see Figure 122).



Used methods:

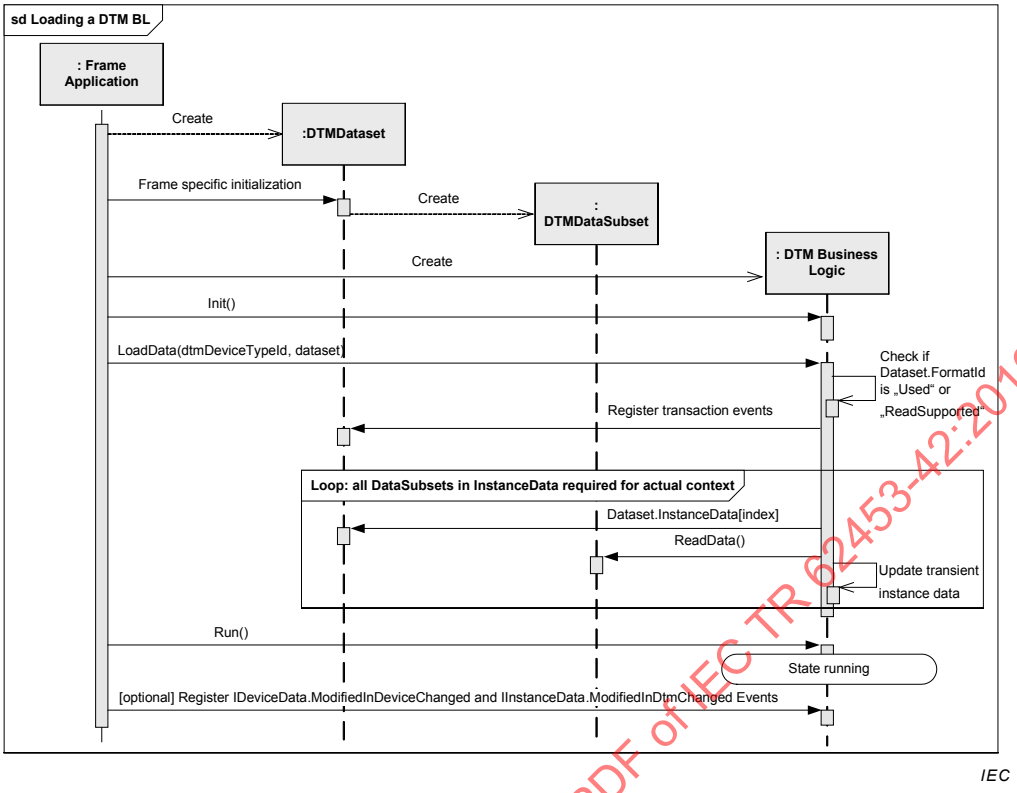
ICustomConfiguration::BeginGetAllDataInfo()

ICustomConfiguration::EndGetAllDataInfo()

Figure 122 – Configuration of user permissions

8.2.4 Loading a DTM BL

After creation of a DTM Business Logic instance for an existing DTMDataset the Frame Application shall call Init() and LoadData(), pass the identifier of the represented type (device, module, block) and the interface of the corresponding DTMDataset. The DTM checks the FormatId of the DTMDataset and reads the InstanceData from the DTMDataset to initialize its device data (see Figure 123).



Used methods:

IDtm.Init()

IDtm.LoadData()

IDataSubset.ReadData()

Event IDeviceData.ModifiedInDeviceChanged()

Event IInstanceData.ModifiedInDtmChanged()

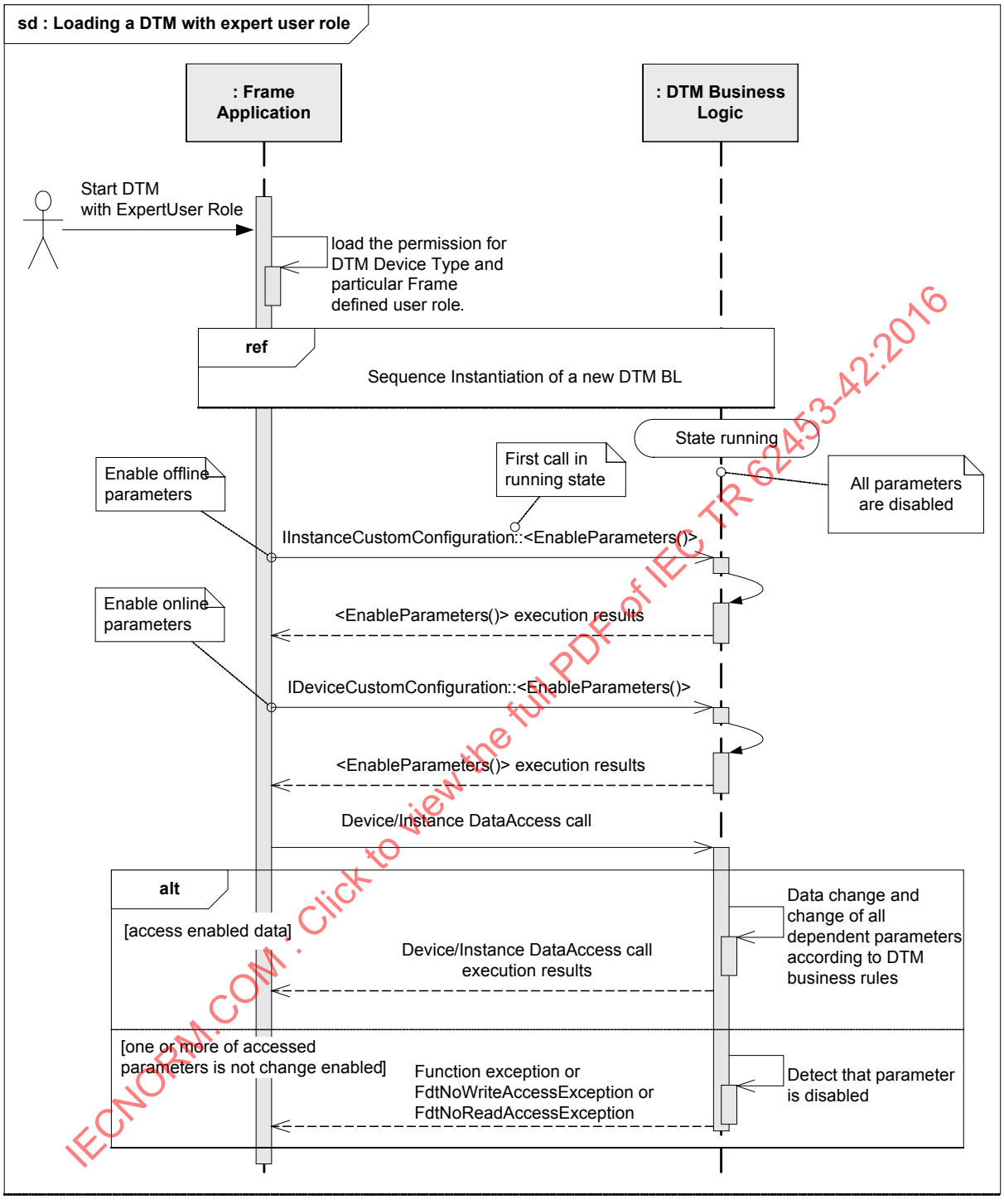
Figure 123 – Loading a DTM BL

A DTM Business Logic shall read DTMDataSubsets of Datasets InstanceData on demand when the data are required according to a business function context. When transient data are not accessed any more, a DTM shall release the transient data and reload it from the Dataset if needed (see 8.3.1).

8.2.5 Loading a DTM with Expert user level

While loading the DTM with Expert user level, the Frame Application shall set the access rights using the ICustomConfiguration::<EnableParameters()>.

If the Frame Application does not invoke the ICustomConfiguration methods to grant permissions, the user will have restricted access as if the DTM is invoked by the Observer (see Figure 124).



Used methods:

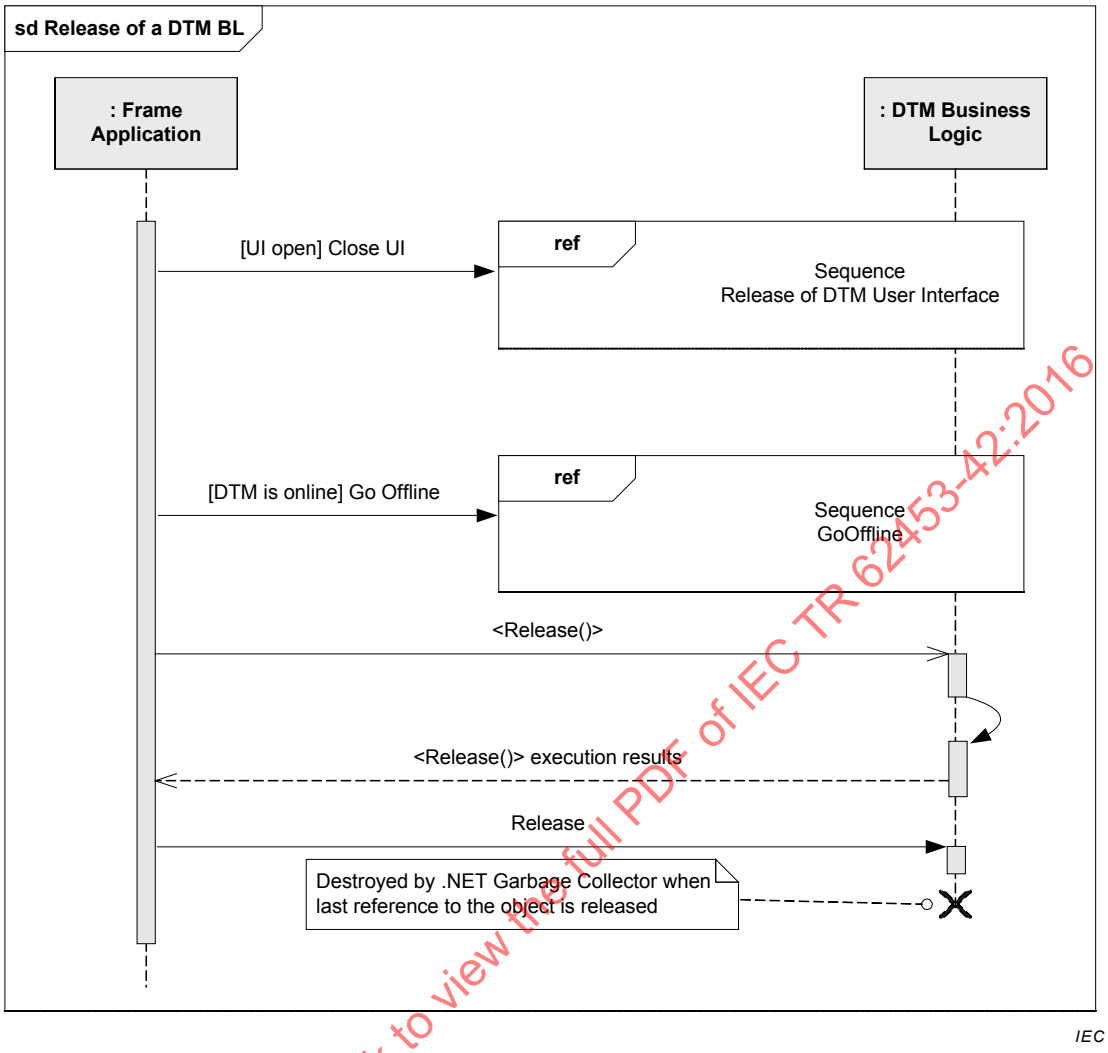
IInstanceCustomConfiguration.BeginEnableParameters() / IInstanceCustomConfiguration.EndEnableParameters()

IDeviceCustomConfiguration.BeginEnableParameters() / IDeviceCustomConfiguration.EndEnableParameters()

Figure 124 – Loading a DTM with Expert user level

8.2.6 Release of a DTM BL

In order to release a DTM BL all ongoing activities need to be terminated (see Figure 125).



Used methods:

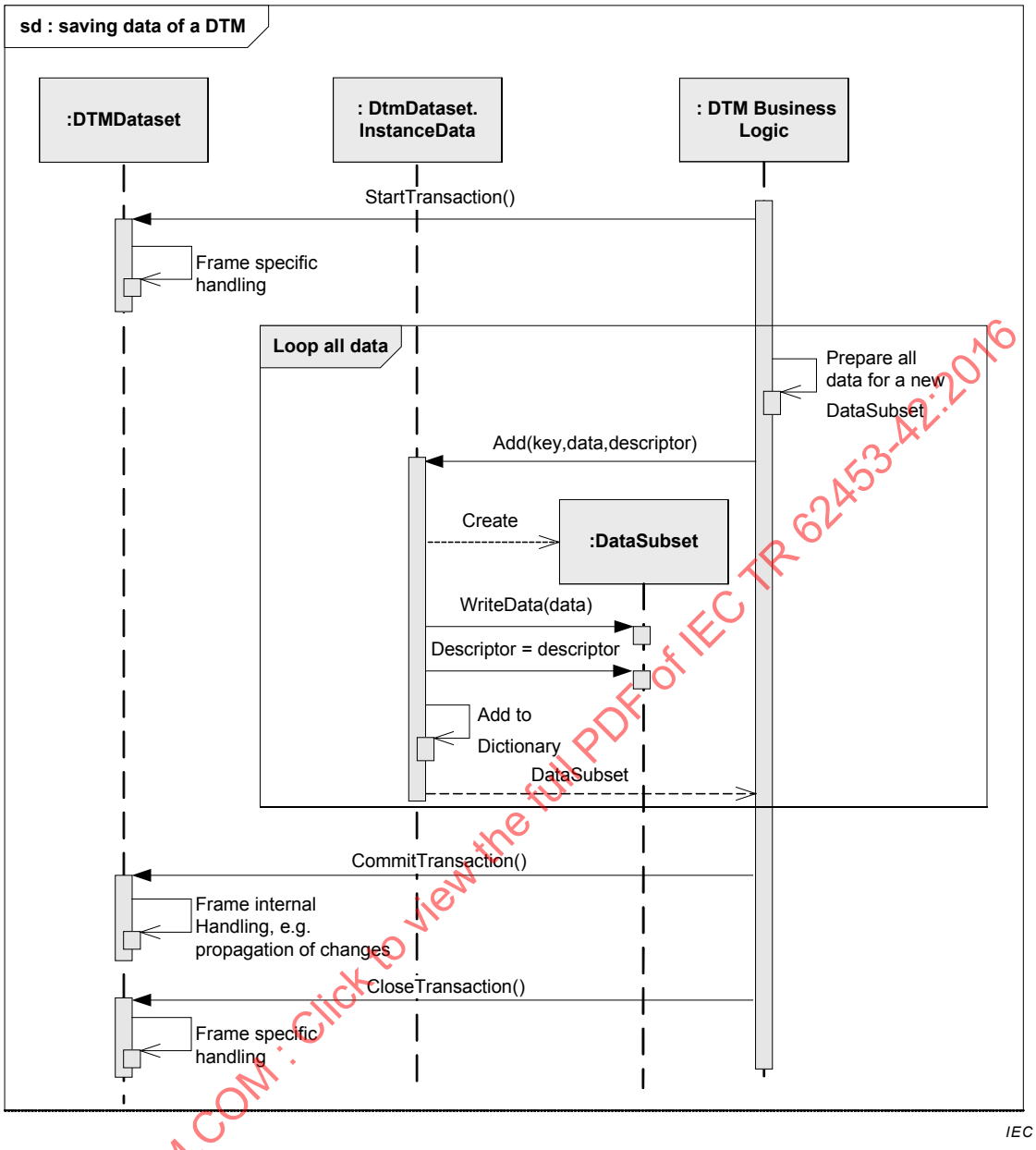
IDtm.BeginRelease() / IDtm.EndRelease()

Figure 125 – Release of a DTM BL

8.3 Persistent storage of a DTM

8.3.1 Saving instance data of a DTM

The DTM instance saves its instance data on demand in a DataTransaction. The Frame Application can release a DTM only if no DataTransaction is active. If the Frame Application performs an action which requires that all data is committed (i.e. saving of a project file) then it shall check if there are no open transactions. If there are open transactions then the Frame Application should inform the user and list the DTMs which have open transactions and thereby may have uncommitted data (see Figure 126).



Used methods:

IDataSet.StartTransaction()

IDataSet.CommitTransaction()

IDataSet.CloseTransaction()

IDataSubsetDictionary.Add()

Figure 126 – Saving data of a DTM

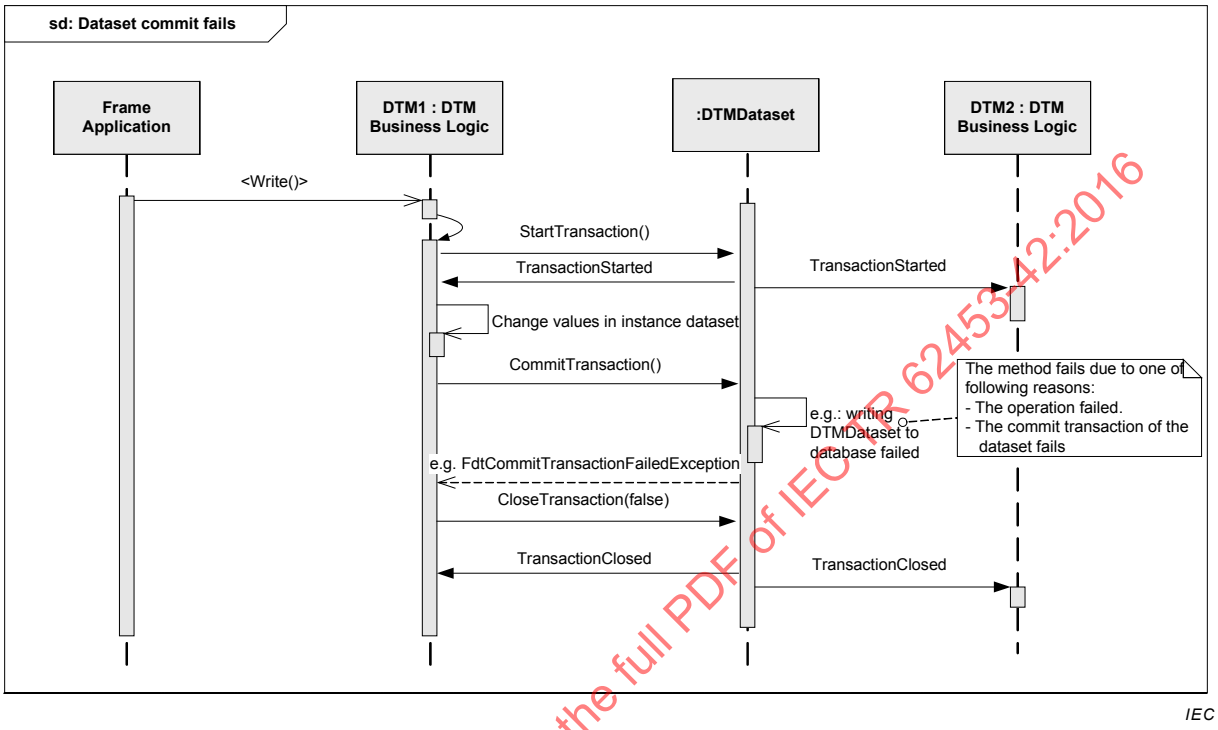
8.3.2 Copy and versioning of a DTM instance

Saved datasets can be copied by a Frame Application from one device node to a different device node. The copied Dataset is loaded with LoadData() into instances of the corresponding device node.

The Frame Application is responsible to handle the Frame Application-specific versioning aspects and to manage the different instance datasets (e.g. fieldbus address and device tag) for a device.

8.3.3 Dataset commit failed

The following workflow describes the expected behavior if committing changes in the dataset fails. This exception is usually caused by a serious problem in the Frame Application. The Frame Application shall inform the user that the latest changes could not be saved and to release the DTM (see Figure 127).



Used methods:

IDataSet.StartTransaction()

IDataSet.TransactionStarted()

IDataSet.CommitTransaction()

IDataSet.TransactionClosed()

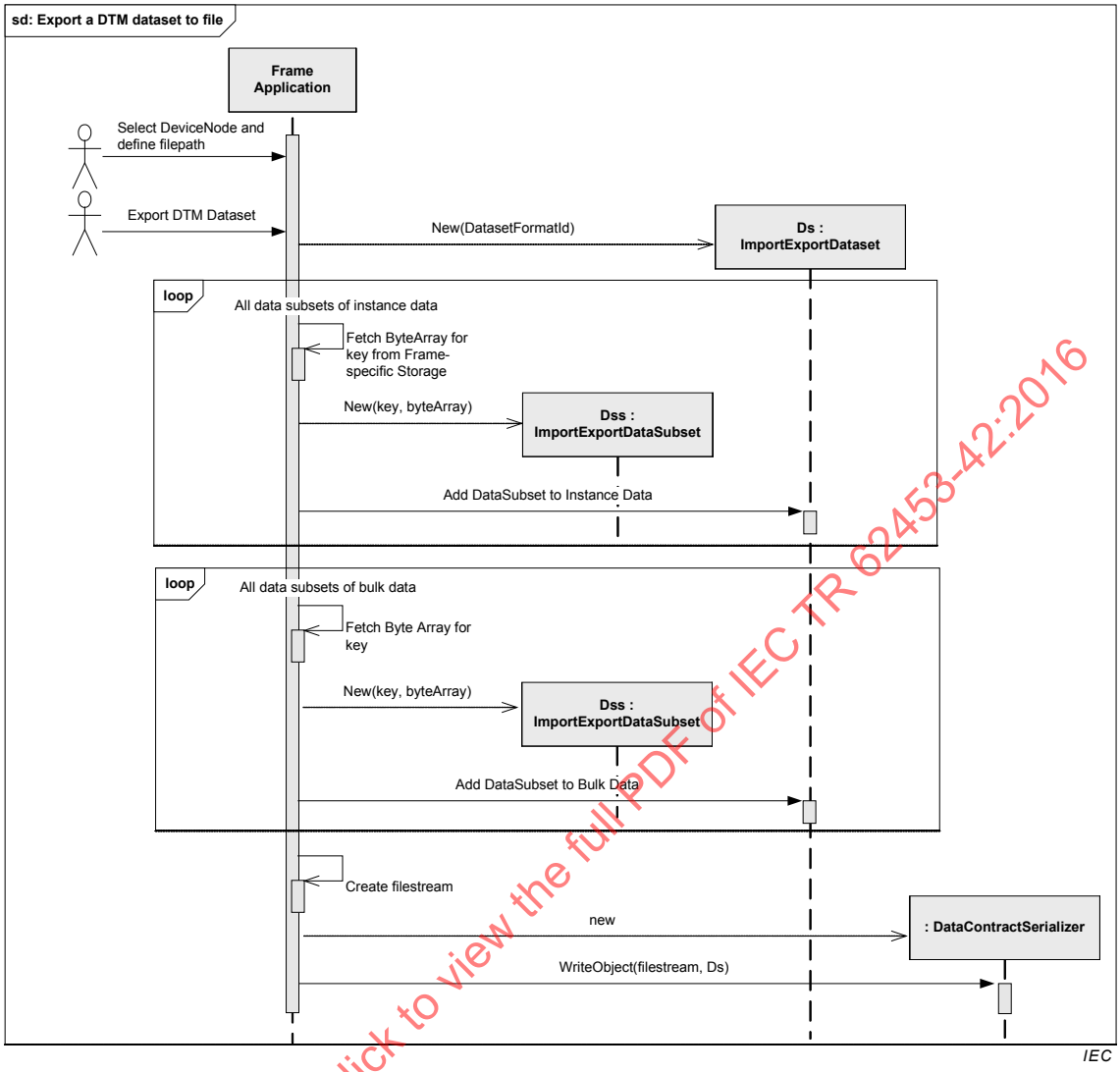
IDataSet.CloseTransaction()

IDataSubsetDictionary.Add()

Figure 127 – Dataset commit failed

8.3.4 Export a DTM dataset to file

The diagram shown in Figure 128 shows the use of datatypes for exporting the data of a DTM instance to a file.



Used methods:

Figure 128 – Export a DTM dataset to file

8.4 Locking and DataTransactions in multi-user environments

8.4.1 General

Within a multi-user environment it is common, that more than one DTM instance has access to the same dataset. To synchronize DTMs which are started by several users on different PCs FDT provides a locking mechanism. Target for this event mechanism is that only one DTM has read/write access to the instance dataset and to the device data. All other DTMs have read access only.

For this reason a DTM shall lock its dataset with StartTransaction() only if required and only during modification of the data. After the data is committed and the data is not further under modification, the DTM shall unlock its dataset with CloseTransaction() immediately to enable concurrent access to the data by other DTM instances within a multi-user environment.

- The DTM shall start a DataTransaction before an activity is started, that may change the instance data (e.g. upload, IInstanceData.<Write()>) or the data in the device (e.g. OnlineParameterize). The DTM shall close the DataTransaction after the activity is finished.

If instance data is changed, then the DTM shall save the Dataset before closing the DataTransaction. E.g.:

– While a DTM UI is opened the DTM shall try to start a DataTransaction if write access is needed. If successful, all user input fields can be enabled. If the start of the DataTransaction failed, user input fields shall be disabled. After closing all DTM GUI controls in case of a locked Dataset the DTM should write modified DTMDataSubsets and commit the Dataset and close the DataTransaction after the Frame Application has saved the Dataset.

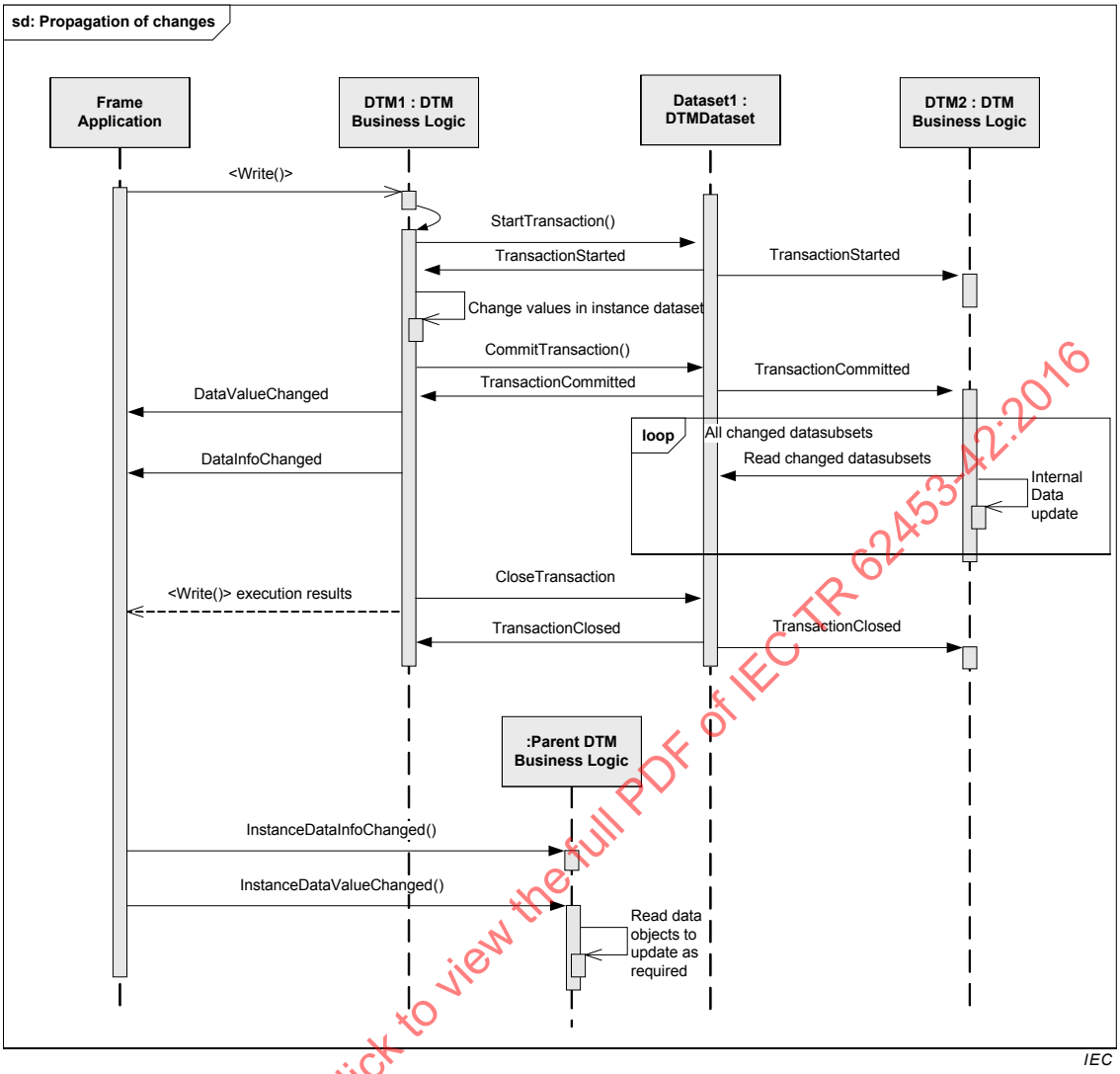
- A Frame Application shall return a negative result when a DTM calls StartTransaction while a second DTM has already an open DataTransaction. (The property LockResult.IsLocked will be set to false.)
- The Frame Application shall throw an exception if a DTM writes DTMDataSubsets while this DTM does not have an open DataTransaction.
- The DTM shall keep a DataTransaction open as short as possible. It is not allowed to set the lock for the whole time that a DTM is in states ‘running’ and ‘communicationAllowed”.
- If committing the dataset fails, then the transaction shall be closed without saving (CloseTransaction(false)) and the user shall be informed. It is recommended to stop working with the DTM.
- If closing the transaction fails, then the user shall be informed. It is recommended to stop working with the DTM.
- If a DTM receives the event TransactionCommitted(), it is mandatory to update the instance data from storage.

8.4.2 Propagation of changes

When multiple DTM instances are executed in a multi-user environment for the same device (see 4.6.2) and one DTM instance is changing the DataSet, the other DTM instance receives notifications indicating the process of change (TransactionStarted, TransactionCommitted, TransactionClosed).

Receiving the event TransactionCommitted indicates that the data in the persisted DataSet has been changed and that the DTM shall update the instance data from the storage.

The sequence diagram shown in Figure 129 shows how changes in the instance dataset of one DTM instance (DTM1) are propagated to other DTM instances (DTM2) and to the Parent DTM.



Used methods:

IDataset.StartTransaction() / IDataset.CommitTransaction() / IDataset.CloseTransaction()

IDataset.TransactionStarted / IDataset.TransactionCommitted /IDataset.TransactionClosed

IInstanceData.BeginWrite() / IInstanceData.EndWrite()

Event IInstanceData.DataValueChanged()

Event IInstanceData.DataInfoChanged()

Event IChildDtmEvents.InstanceDataValueChanged()

Event IChildDtmEvents.InstanceDataInfoChanged()

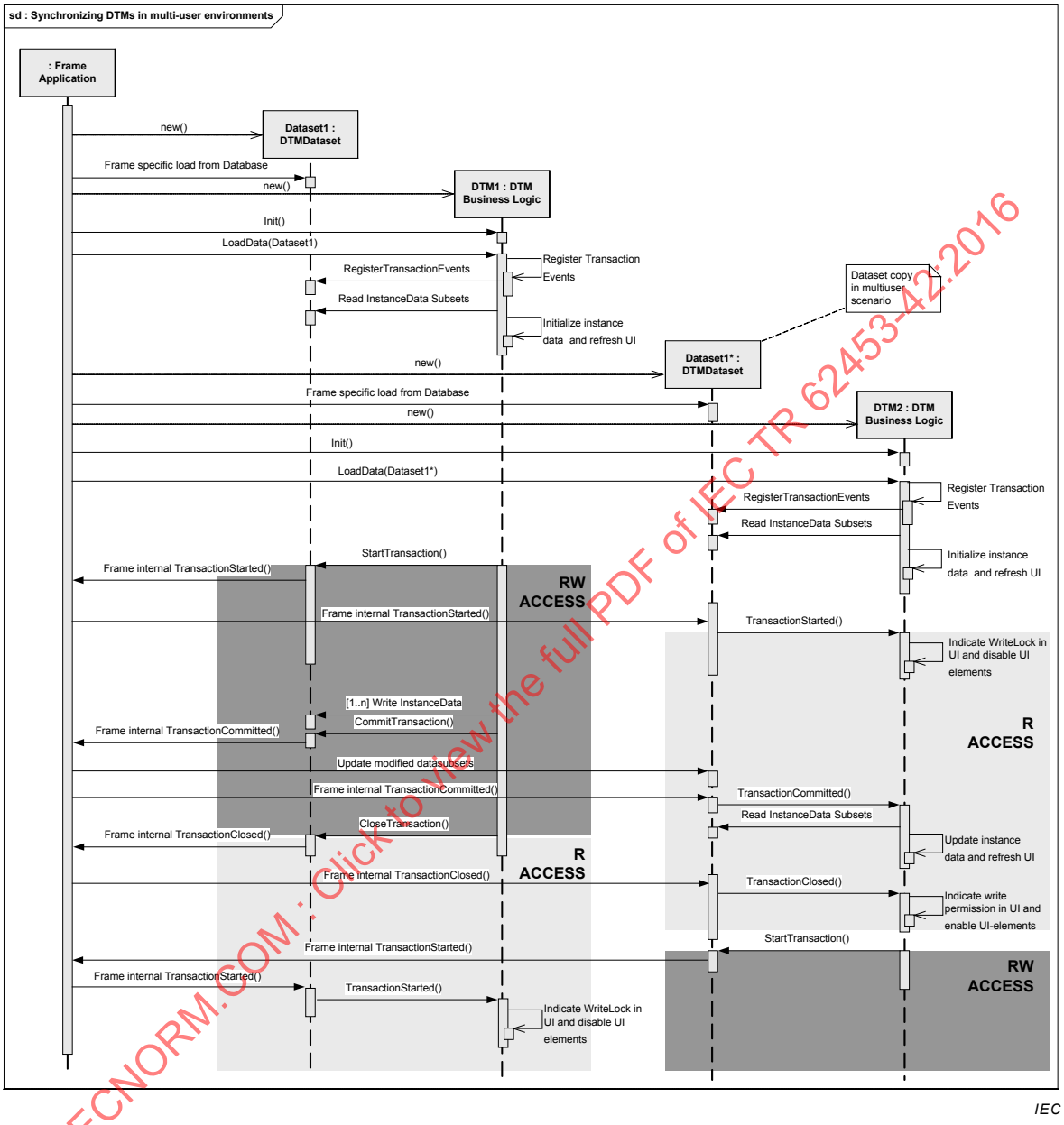
Figure 129 – Propagation of changes

The figure above shows how a DTM instance (“DTM2”) receives notifications on changes and how it updates its instance data, because the dataset was changed by a different DTM instance (“DTM1”).

NOTE For simplification, it is not shown here how “DTM2” reads already committed data while “DTM1” is still modifying data in an open transaction (refer to Figure 130, which shows this scenario).

8.4.3 Synchronizing DTMs in multi-user environments

The synchronization of DTMs is a mandatory feature to provide a better handling for the user within a multi-user environment (see Figure 130).



Used methods:

IDtm.Init()

IDtm.LoadData()

IDataset.CloseTransaction()

IDataset.CommitTransaction()

IDataset.StartTransaction()

IDataset.TransactionClosed()

IDataset.TransactionCommitted()

IDataset.TransactionStarted()

Figure 130 – Synchronizing DTMs in multi-user environments

The sequence diagram in Figure 130 describes an implementation example where a Frame Application provides a copy of the last committed DTMDataset (Dataset1*) for concurrently accessing DTM instances in multi-user scenarios. These instances cannot change the dataset at the same time (FA rejects StartTransaction()), but can read from the dataset last committed. How the Frame Application synchronizes the two instances of DtmDataset, is not in scope of FDT but specific to the Frame Application (shown as Frame Internal methods).

8.5 Execution of DTM Functions

8.5.1 General

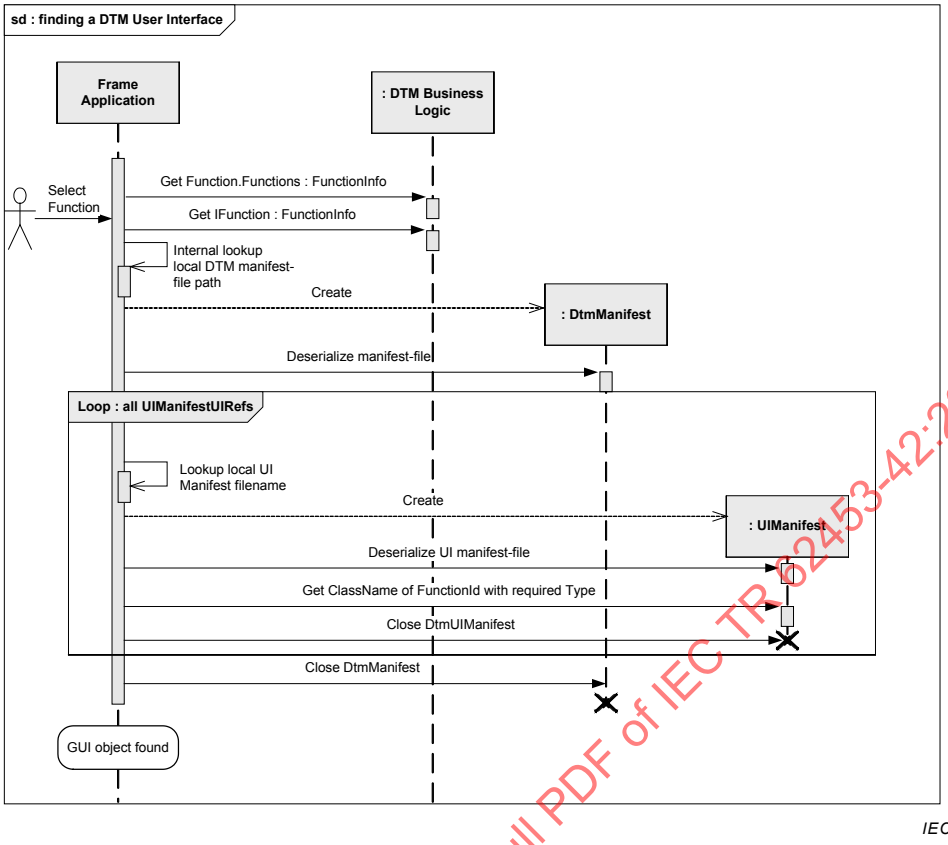
This specification defines different types of DTM User Interfaces:

- WinForms controls or WPF controls that can be embedded into the Frame Application user interface
- Applications which can be started by a DTM User Interface class
- Command functions which are provided by a DTM BL or a DTM User Interface class

The sequence diagrams in this subclause show the different handling of these user interface types.

8.5.2 Finding a DTM User Interface object

The FunctionInfo property of IFunction interface provides access to user interfaces provided by a DTM. If a DTM provides user interfaces the FunctionInfo property contains a list of UiFunction objects. A UiFunction object represents a DTM User Interface function. The actual information about the object which implements this function is provided in a manifest file (see DtmManifest.UiManifestRefs description). The Frame Application shall use the property UiFunction.FunctionId to find the information in the manifest (see UiFunction description) (see Figure 131).



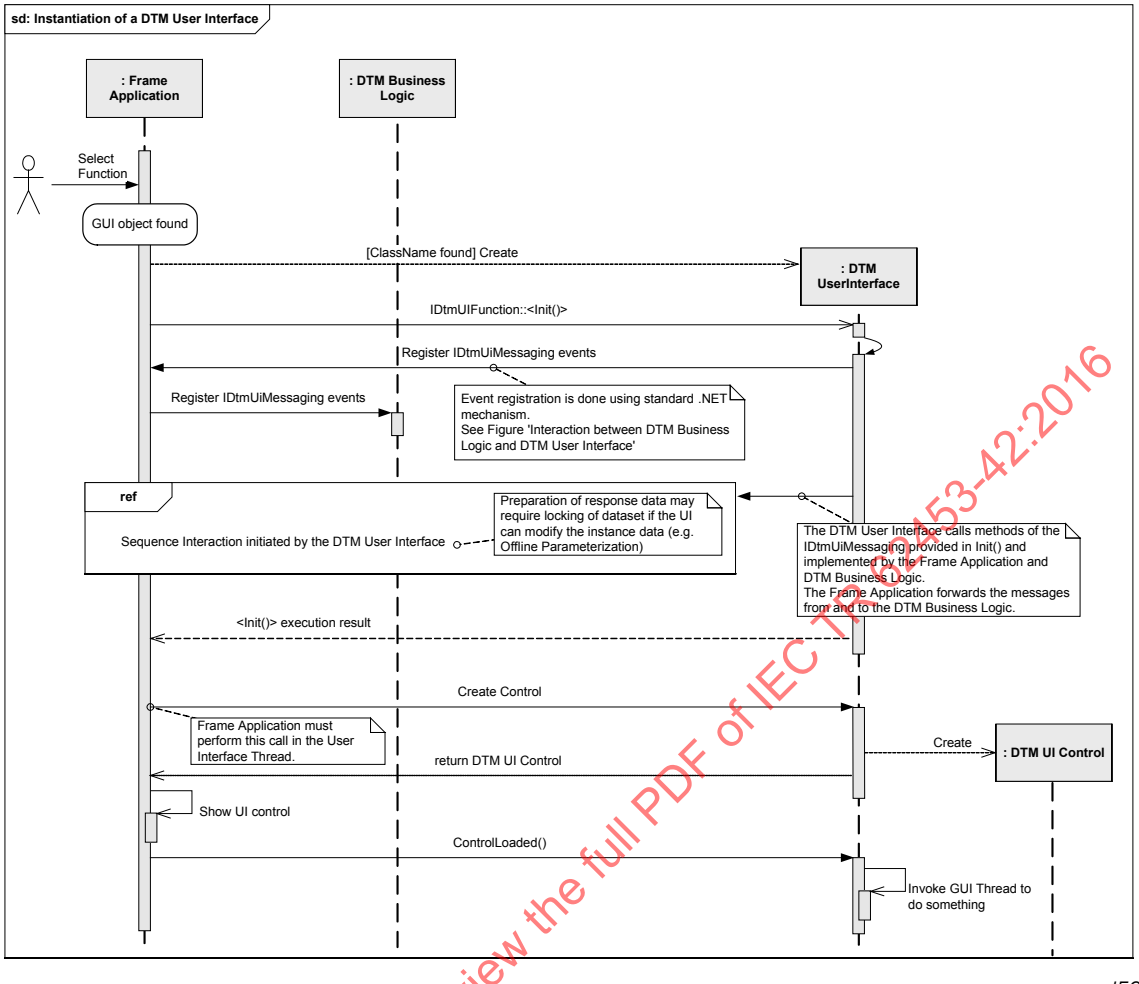
Used methods:

IFunction.Functioninfo

Figure 131 – Finding a DTM User Interface

8.5.3 Instantiation of an integrated DTM graphical user interface

This sequence diagram outlines the opening of a DTM User Interface for a DTM function selected by the user (e.g. in a DTM-specific context menu) (see Figure 132). The sequence may also be started by a different trigger (e.g. by a Frame Application function).



Used methods:

IFunction.FunctionInfo

IDtmUiFunction.BeginInit() / IDtmUiFunction.EndInit()

IDtmUiControlFunction.CreateControl()

IDtmUiControlFunction.ControlLoaded()

IDtmUiMessaging.BeginSendMessages()

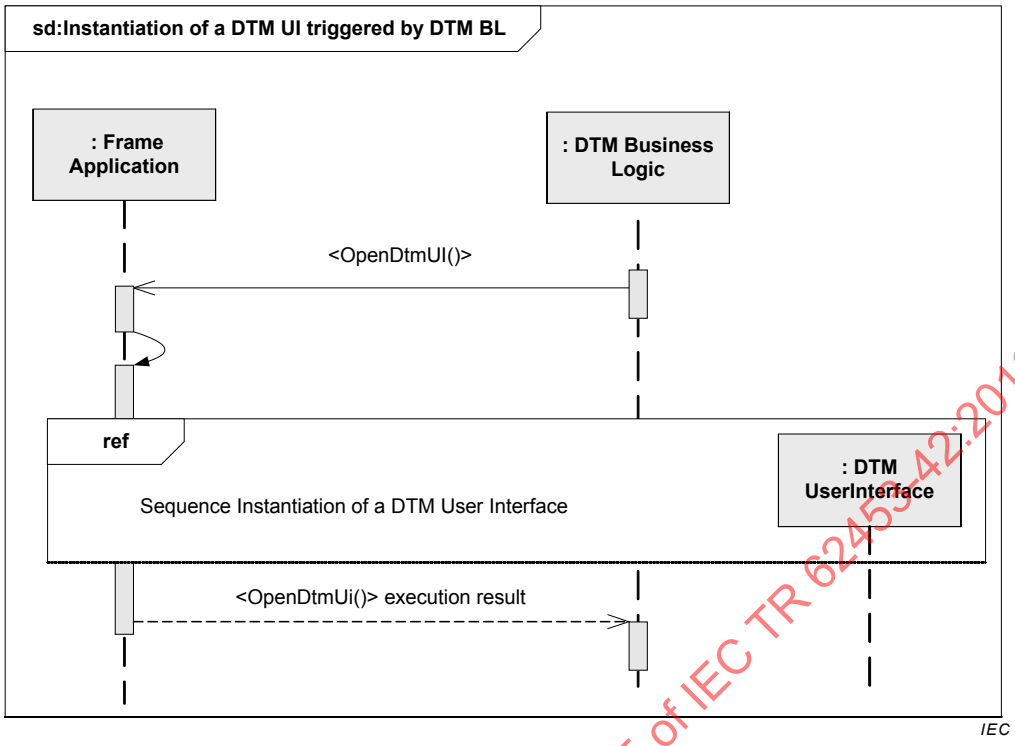
IDtmUiMessaging.EndSendMessages()

Figure 132 – Instantiation of a DTM User Interface

8.5.4 Instantiation of a DTM UI triggered by the DTM BL

In this scenario the DTM Business Logic requests to open one of its user interfaces (see Figure 133).

If the user interface is successfully instantiated and initialized, the Frame Application returns the InvokeId of the new user interface with the EndOpenDtmUi method.



Used methods:

IFrameUi.BeginOpenDtmUi()

IFrameUi.EndOpenDtmUi()

Figure 133 – Instantiation of a DTM UI triggered by DTM BL

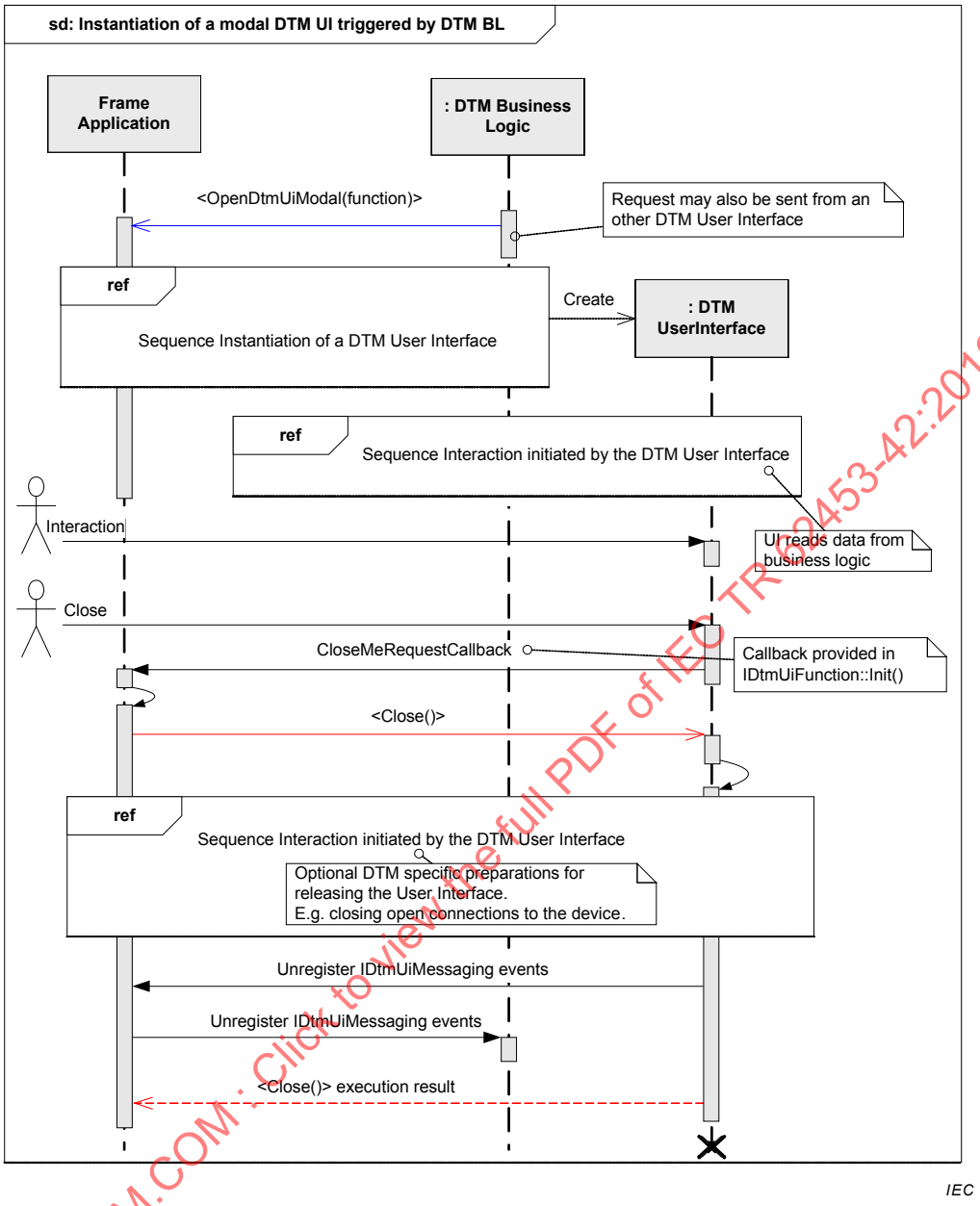
8.5.5 Instantiation of a modal DTM UI triggered by DTM BL

In this scenario the DTM Business Logic requests to open one of its user interfaces modally (see Figure 134).

IFrameUi.OpenDtmUiModal() behaves always modal. The Frame Application has to ensure that at least all user interface controls of the calling DTM are disabled; no further user input shall be possible (DTM instance modal).

The opened user interface shall call the delegate CloseMeRequestHandler() if it needs to be closed. The Frame Application then closes the DTM User Interface and receives the result when calling the method IDtmUiFunctionModal. EndClose().

IFrameUi.<OpenDtmUiModal()> shall be called in a way that the caller is blocked until the user interface is closed.



Used methods:

IFrameUi.BeginOpenDtmUiModal() / IFrameUi.EndOpenDtmUiModal()

CloseMeRequestCallback()

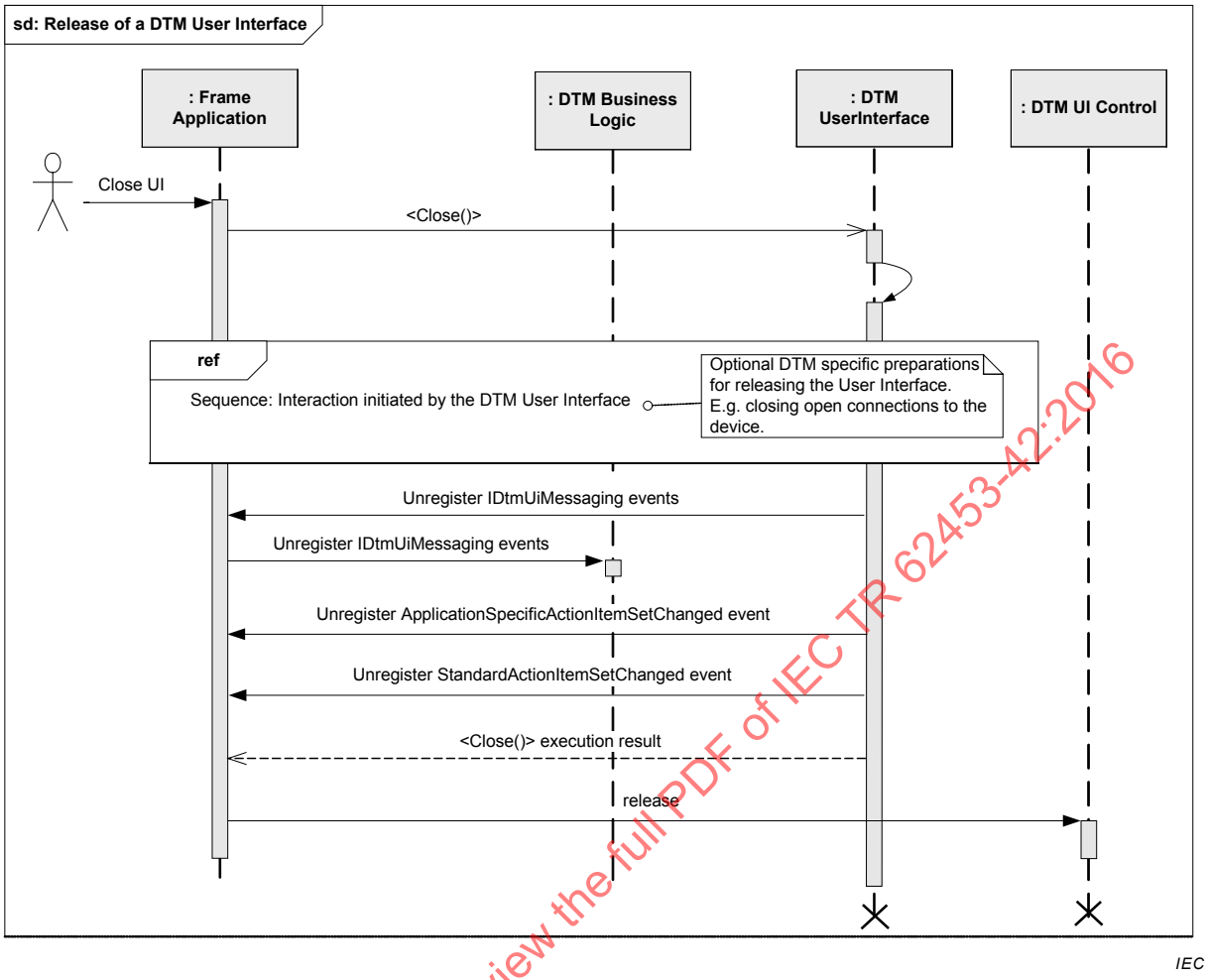
IDtmUiFunction.BeginClose()

IDtmUiFunction.EndClose()

Figure 134 – Instantiation of a modal DTM UI triggered by DTM BL

8.5.6 Release of a DTM User Interface

This sequence diagram outlines the closing of a DTM User Interface for a DTM function as a result of a request to the Frame Application (e.g. windows system menu – close). If the Frame Application releases a user interface of a DTM, it has to prepare the release by sending a notification to the presentation object first (see Figure 135). After receiving the call to IDtmUiFunction.BeginClose() the user interface shall release its references to other components and can call DTM-specific releasing methods.



Used methods:

IDtmUiFunction.BeginClose()

IDtmUiFunction.EndClose()

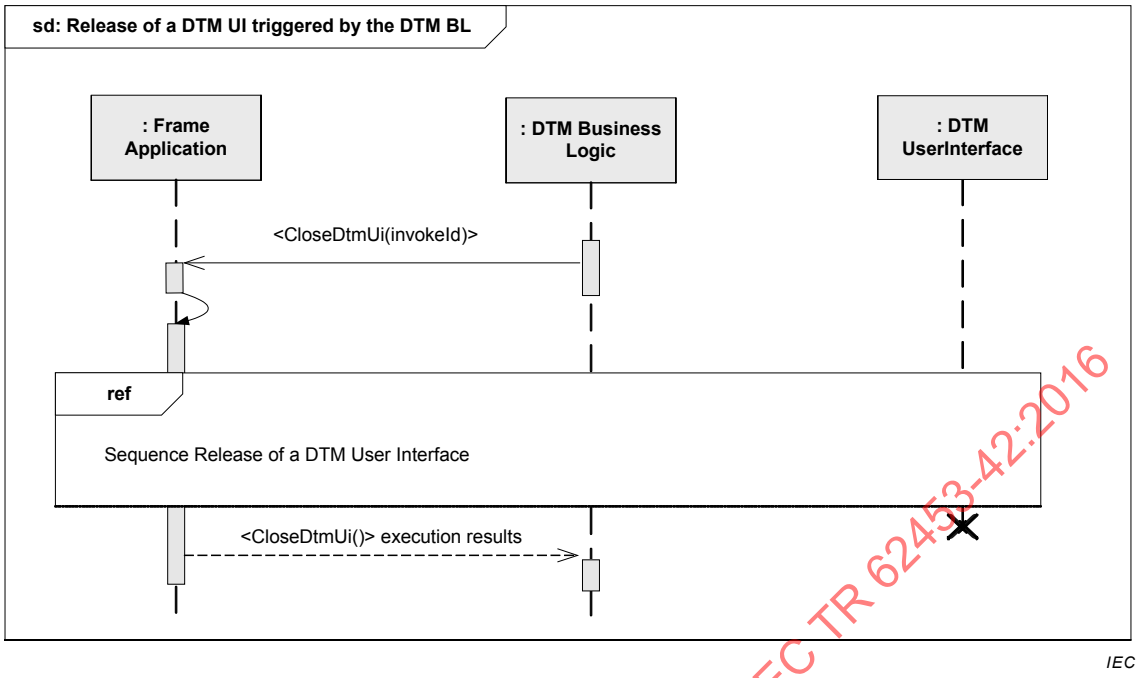
IDtmUiMessaging.BeginSendMessages()

IDtmUiMessaging.EndSendMessages()

Figure 135 – Release of a DTM User Interface

8.5.7 Release of a DTM UI triggered by the DTM BL

This sequence diagram outlines the closing of a DTM User Interface for a DTM function as a result of a request by the corresponding DTM BL (see Figure 136).



Used methods:

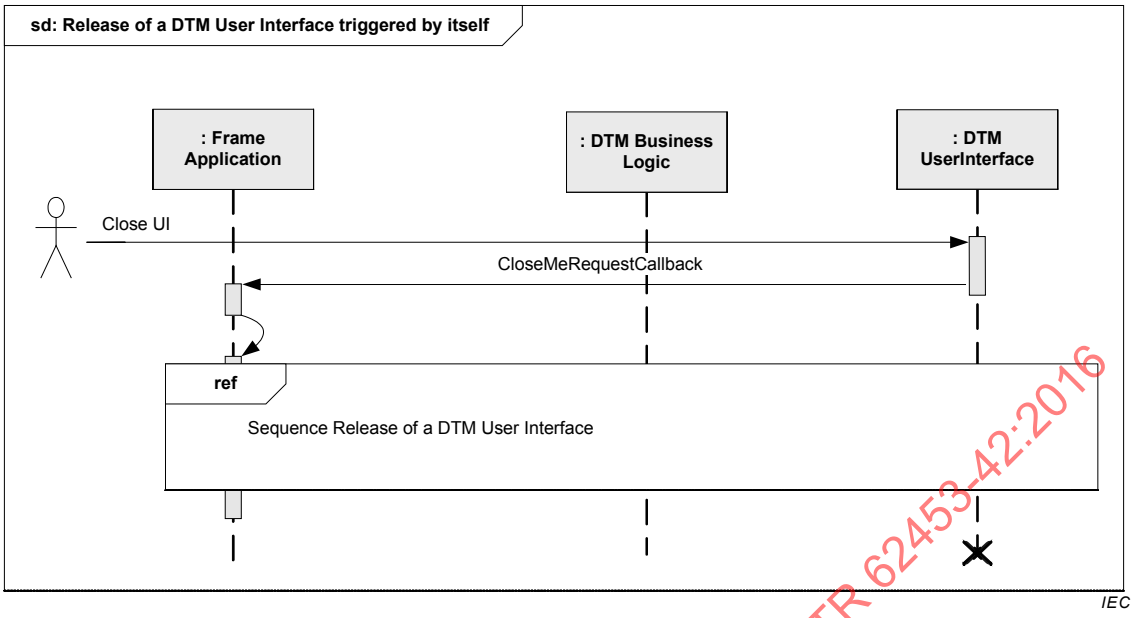
IFrameUI.BeginCloseDtmUi()

IFrameUI.EndCloseDtmUi()

Figure 136 – Release of a DTM UI triggered by the DTM BL

8.5.8 Release of a DTM User Interface triggered by itself

In this scenario the DTM User Interface requests to close itself (e.g. after the user presses the 'Close' button on the DTM User Interface). The user interface shall call the delegate CloseMeRequestHandler(), provided in the IDtmUiFunction.<Init()> method, if it needs to be closed (see Figure 137). The Frame Application then closes the DTM User Interface.



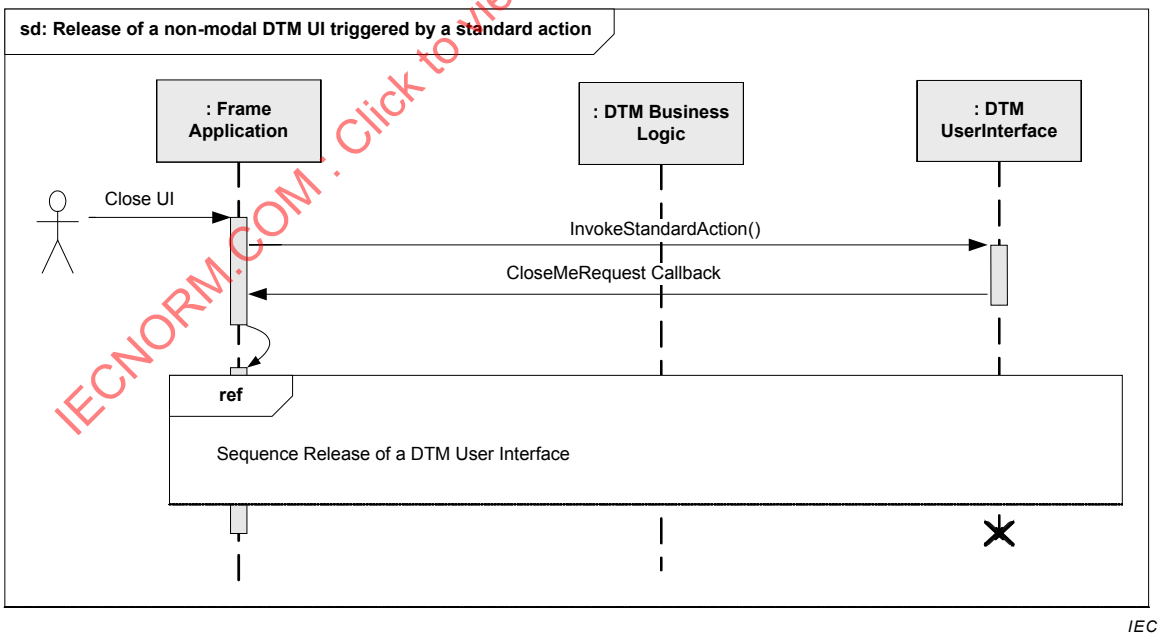
Used methods:

CloseMeRequestCallback()

Figure 137 – Release of a DTM User Interface triggered by itself

8.5.9 Release of a non-modal DTM User Interface triggered by a standard action

For a modeless DTM UI ,which supports the interface IDtmUiFunctionNonModal, the trigger for closing always comes from the Frame Application. Figure 138 shows the sequence for closing such a DTM User Interface.



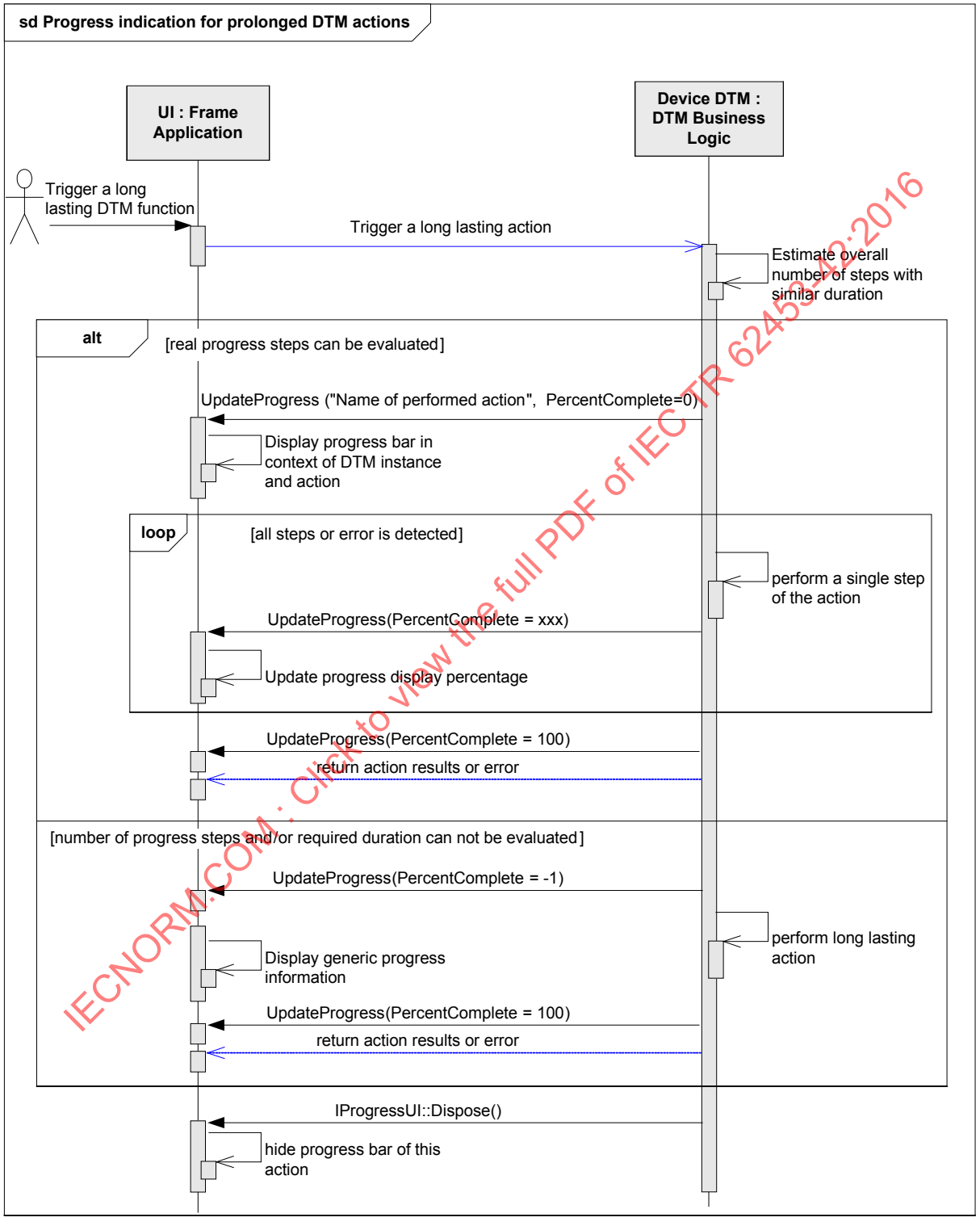
Used methods:

CloseMeRequestCallback()

Figure 138 – Release of a non-modal DTM UI triggered by a standard action

8.5.10 Progress indication for prolonged DTM actions

Figure 139 shows how a Frame Application informs the user in the user interface about the progress of prolonged DTM actions.



Used methods:

IProgressUI.UpdateProgress()

IProgressUI.Dispose()

Figure 139 – Progress indication for prolonged DTM actions

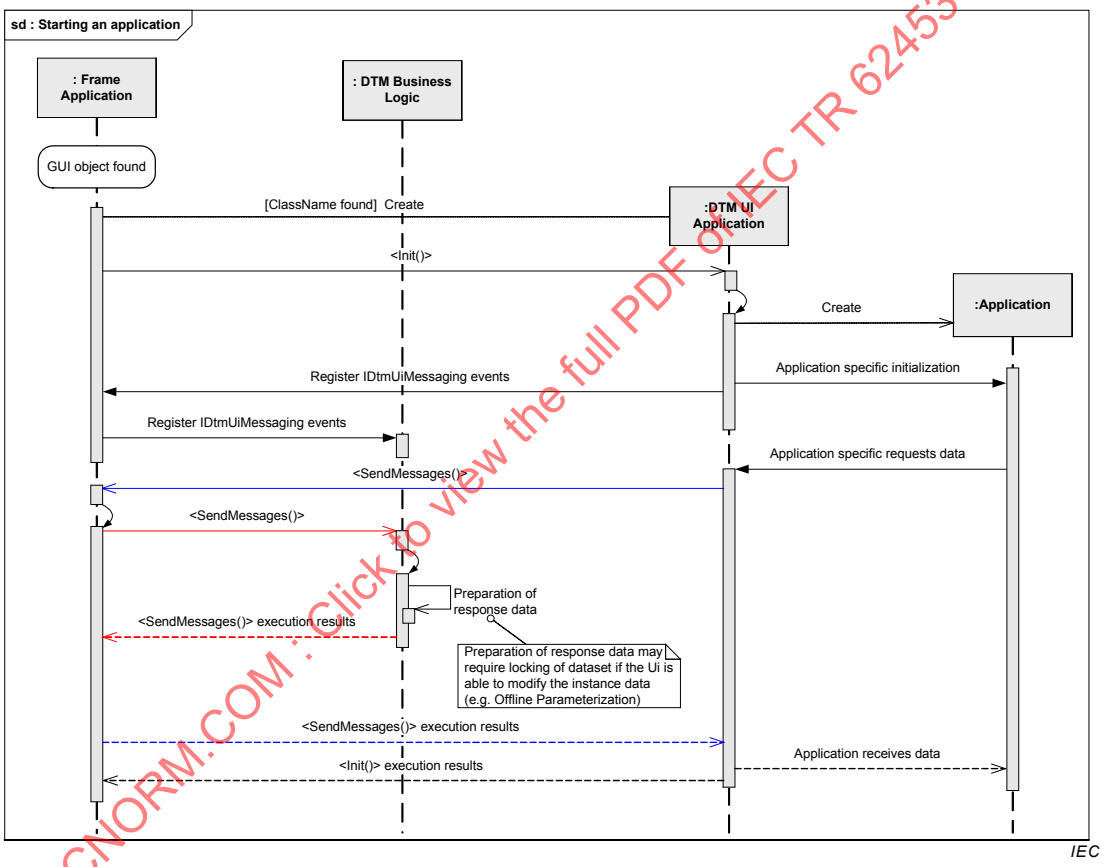
DTMs shall use UpdateProgress() only for prolonged actions, which are not defined as "Progress pattern" (see 5.6.7.3).

A Frame Application shall be prepared to handle several UpdateProgress() calls from different DTM instances and for different actions in parallel.

8.5.11 Starting an application

In general the Frame Application uses the same mechanism to start an application and to open an embedded DTM GUI. In order to start an application, a DTM has to provide a DTM UI Application object, which may be used to start the application and to interact with the application.

The sequence diagram in Figure 140 shows how the Frame Application starts an application and how the application interacts with the DTM.



Used methods:

IFunction.FunctionInfo

IDtmUiFunction.BeginInit() / IDtmUiFunction.EndInit()

IDtmUiMessaging.BeginSendMessages()

IDtmUiMessaging.EndSendMessages()

Figure 140 – Starting an application

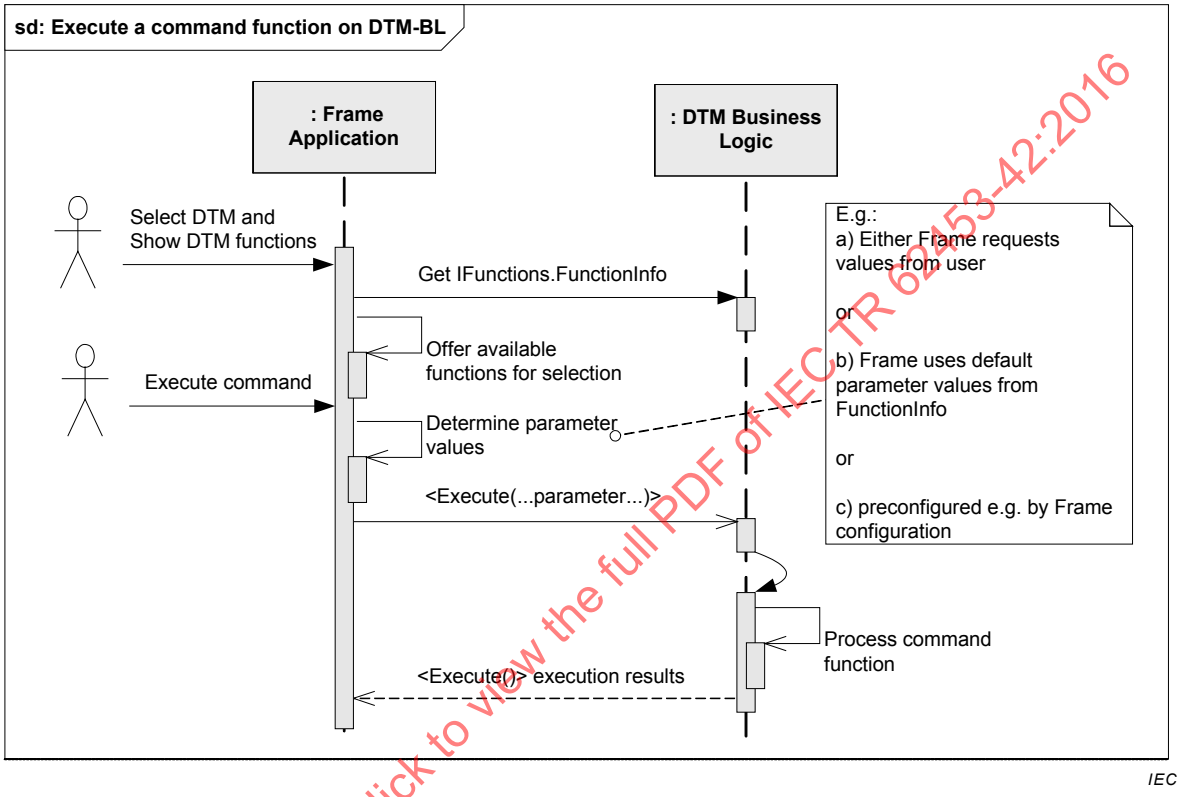
The DTM UI Application object acts as an adapter to the external application and implements the FDT interfaces so that the Frame Application may interact with the application. The interactions between the DTM UI Application object and the application is not in scope of the FDT specification and may be implemented with private interfaces.

8.5.12 Terminating applications

An application may be terminated similar to an embedded DTM UI. See 8.5.6, 8.5.7 and 8.5.8.

8.5.13 Execution of command functions

The execution of a command function on the DTM BL is started via the ICommandFunction interface. The execution of the command is triggered by BeginExecute() and EndExecute() (see Figure 141).



Used methods:

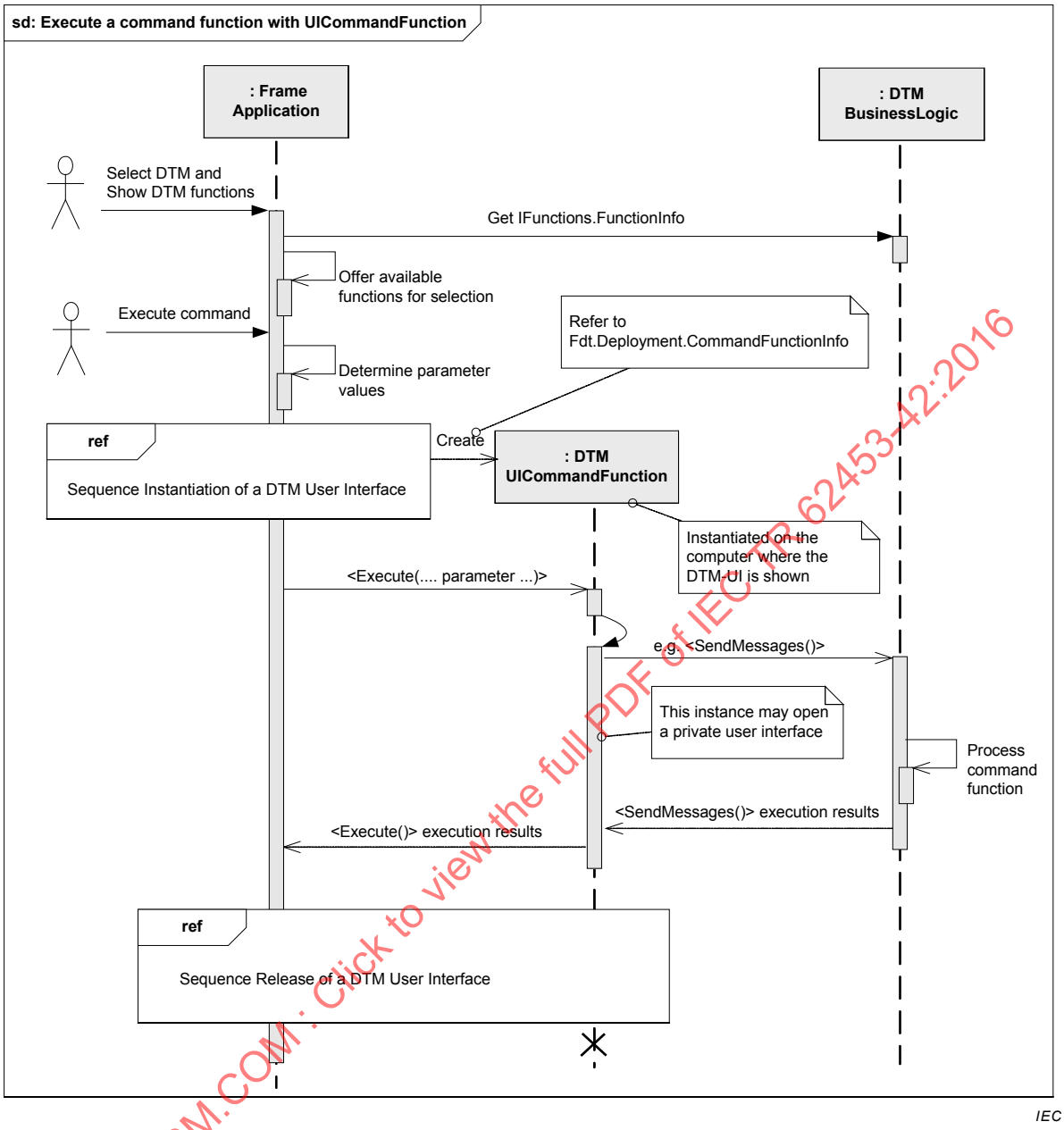
IFunction.FunctionInfo

ICommandFunction.BeginExecute() / ICommandFunction.EndExecute()

Figure 141 – Execute a command function

8.5.14 Execution of a command function with user interface

Some Command Functions may need to open user interfaces and therefore may require knowing where DTM user interfaces are opened. First the UiCommandFunction is initialized and then the execution of the command function is triggered. The following workflow describes how a Frame Application starts the execution and passes command-specific parameters (see Figure 142).



Used methods:

IFunction.FunctionInfo

ICommandFunction.BeginExecute() / ICommandFunction.EndExecute()

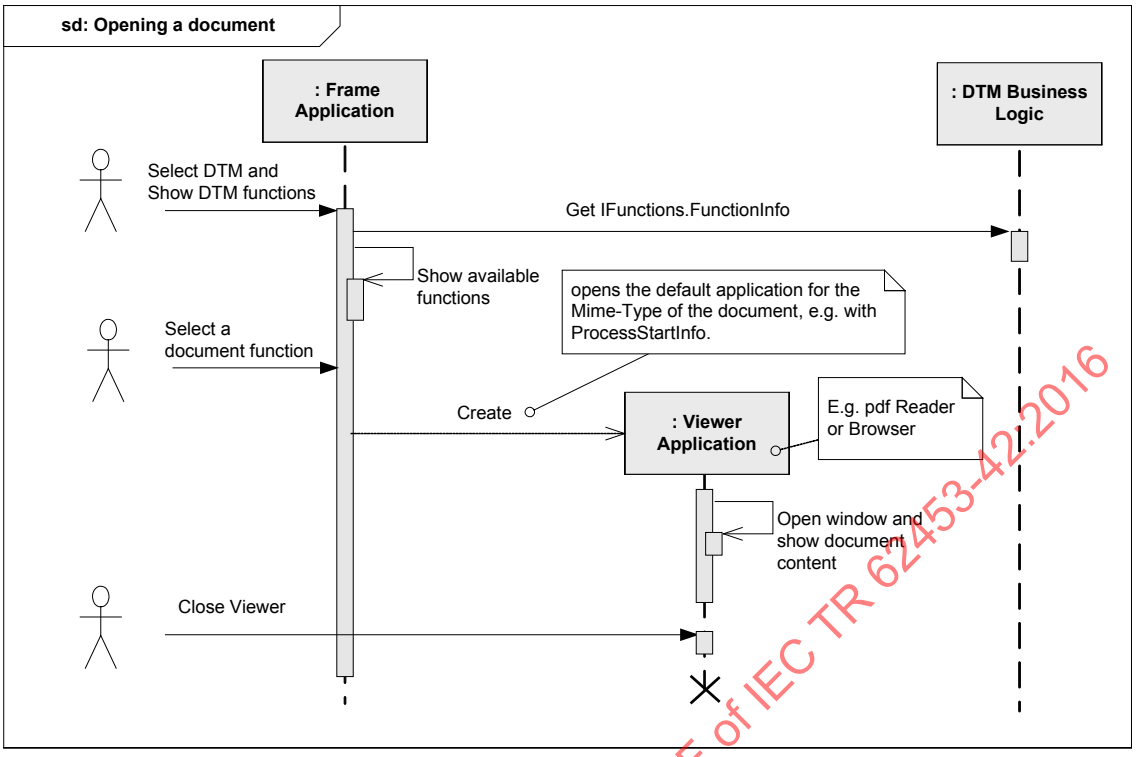
IDtmUiMessaging.BeginSendMessages()

IDtmUiMessaging.EndSendMessages()

Figure 142 – Execute a command function with user interface

8.5.15 Opening of documents

In order to open a document which is provided by a DTM, the Frame Application opens the default application for the Mime-Type of the document, for instance by calling the method ProcessStartInfo() (provided by the operating system) (see Figure 143).



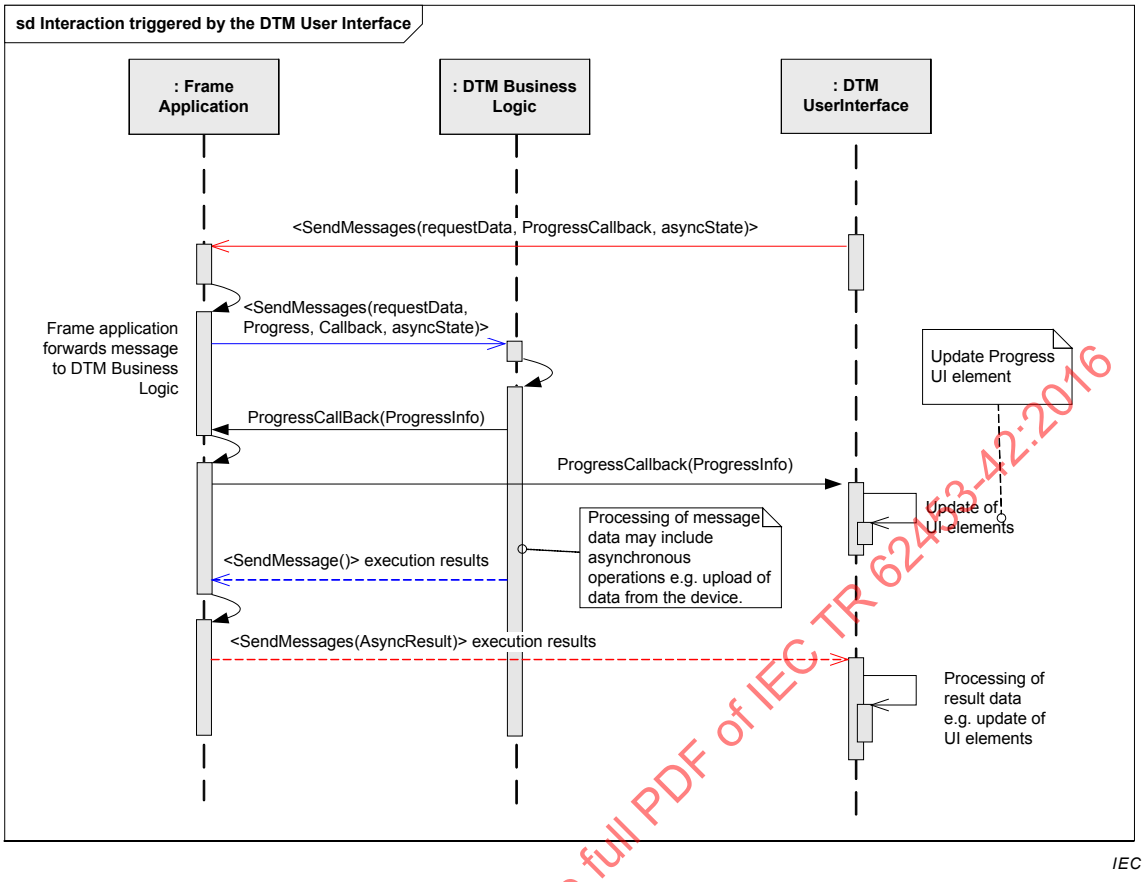
Used methods:

IFunction.FunctionInfo

Figure 143 – Opening a document

8.5.16 Interaction between DTM User Interface and DTM Business Logic

This sequence diagram outlines the interaction of a DTM User Interface with its Business Logic over the messaging interface provided by the Frame Application (see Figure 144).



Used methods:

IDtmUiMessaging.BeginSendMessages()

IDtmUiMessaging.EndSendMessages()

ProgressCallback()

Figure 144 – Interaction triggered by the DTM User Interface

NOTE The callback ProgressCallback and the callback SendMessageCompleted are provided as a parameter of the BeginSendMessage

In this scenario the DTM User Interface requests data from the DTM Business Logic (e.g. to read measured values from the device) by sending a DTM-specific request message(s) derived from the abstract DtmRequestMessage class.

The IDtmUiMessaging interface is implemented by the DTM Business Logic and the Frame Application. The reference to the Frame Application implemented interface shall be passed to a DTM User Interface with the IDtmUiFunction.<Init()> call. The Frame Application shall forward the messages between the DTM User Interface and the DTM Business Logic.

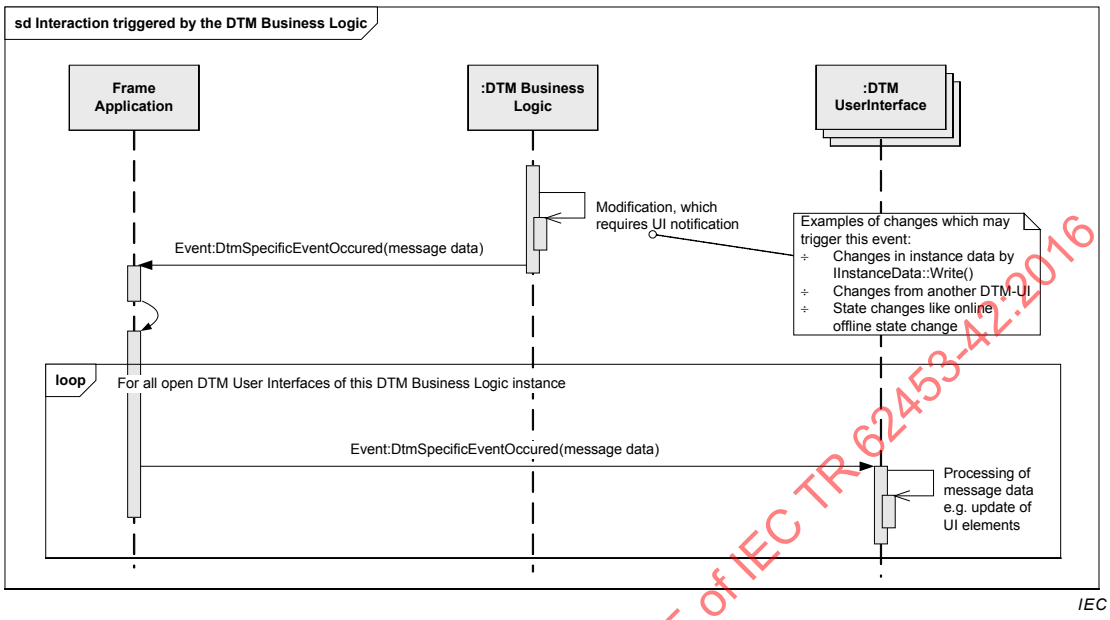
The DTM Business Logic evaluates the requests and creates corresponding response message(s) derived from the abstract DtmResponseMessage class. The response messages contain the requested data and are sent back by calling the Progress and Callback methods.

More detailed information can be found in descriptions of:

– IDtmUiMessaging
– DtmRequestMessage
– DtmResponseMessage

8.5.17 Interaction between DTM Business Logic and DTM User Interface

This sequence diagram outlines the sending of events from a DTM Business Logic to all its opened DTM User Interfaces (see Figure 145).



Used methods:

Event IDtmUiMessaging.DtmSpecificEventOccured()

Figure 145 – Interaction triggered by the DTM Business Logic

In this scenario the DTM Business Logic sends data to the DTM User Interface (e.g. in case of a broken connection to the device).

The DTM User Interface(s) shall register to IDtmUiMessaging events during initialization of the DTM User Interface in order to receive the events.

The Frame Application shall forward the events from the DTM Business Logic to all DTM User Interfaces opened for this instance.

Following standard events are defined in IDtmUiMessaging:

– DtmSpecificEventOccured
– TransactionStarted
– TransactionCommitted
– TransactionClosed

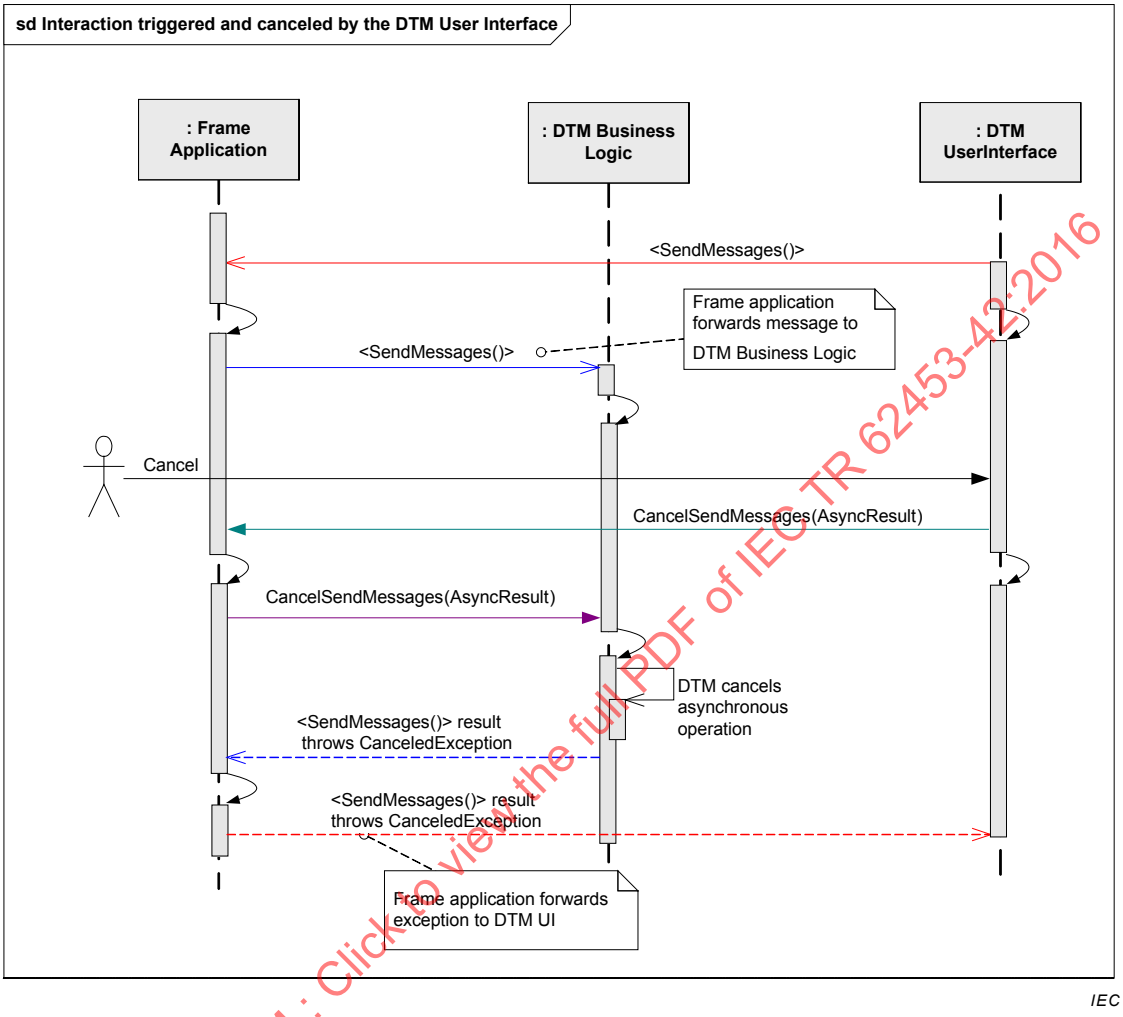
The event DtmSpecificEventOccured can be used for DTM-specific notifications. The DTM Business Logic creates corresponding event message(s) derived from the abstract DtmEventMessage class and passes it to the event handler.

More detailed information can be found in descriptions of:

– IDtmUiMessaging
– DtmEventMessage

8.5.18 Interaction between DTM User Interface and DTM Business Logic with Cancel

This sequence diagram outlines the canceling of a pending user interface message on request of the user (see Figure 146).



Used methods:

IDtmUiMessaging.BeginSendMessages()

IDtmUiMessaging.EndSendMessages()

IDtmUiMessaging.CancelSendMessages()

Used exceptions:

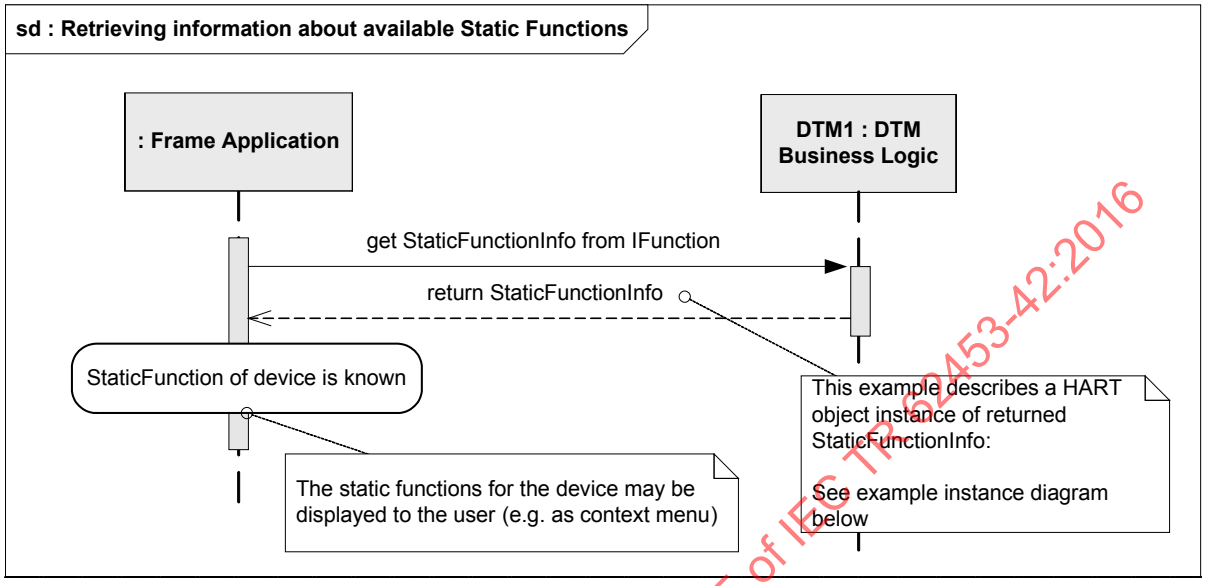
Fdt.FdtOperationCancelledException

Figure 146 – Interaction triggered and canceled by the DTM User Interface

In this scenario the DTM User Interface requests execution of an asynchronous operation from the DTM Business Logic. During execution, the DTM User Interface sends a cancel request. The Frame Application shall forward the CancelSendMessages() request to the DTM Business Logic. The DTM Business Logic shall stop execution and throw an exception in the EndSendMessages() method.

8.5.19 Retrieving information about available Static Functions

In order to use a Static Function for a specific device, the Frame Application retrieves the information about available static functions from the corresponding DTM instance (see Figure 147).



Used methods:

Event IFunction.StaticFunctionsChanged()

IFunction.StaticFunctions()

Figure 147 – Retrieving information about available Static Functions

Figure 148 shows the example for StaticFunctionInfo data, which was retrieved from a DTM.

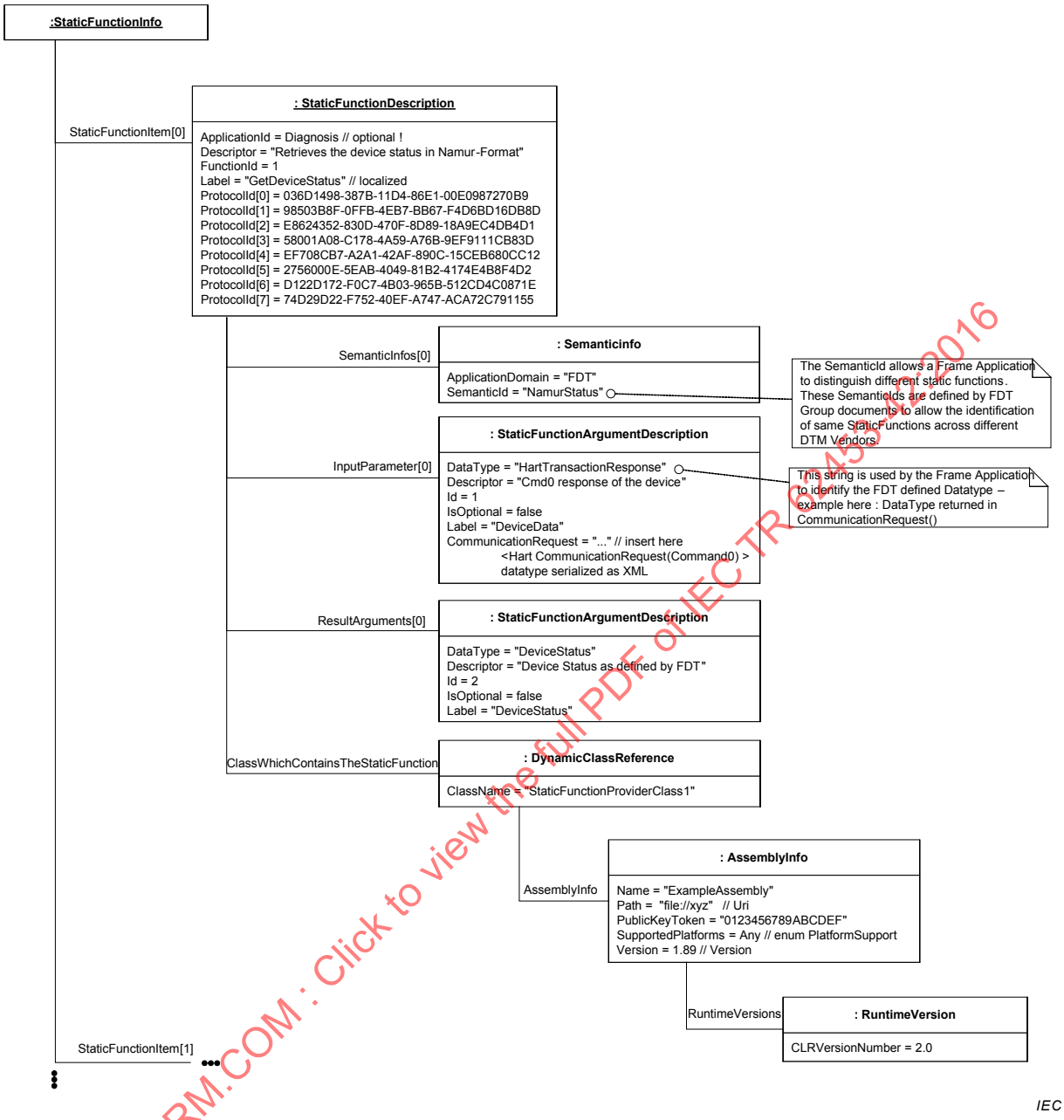
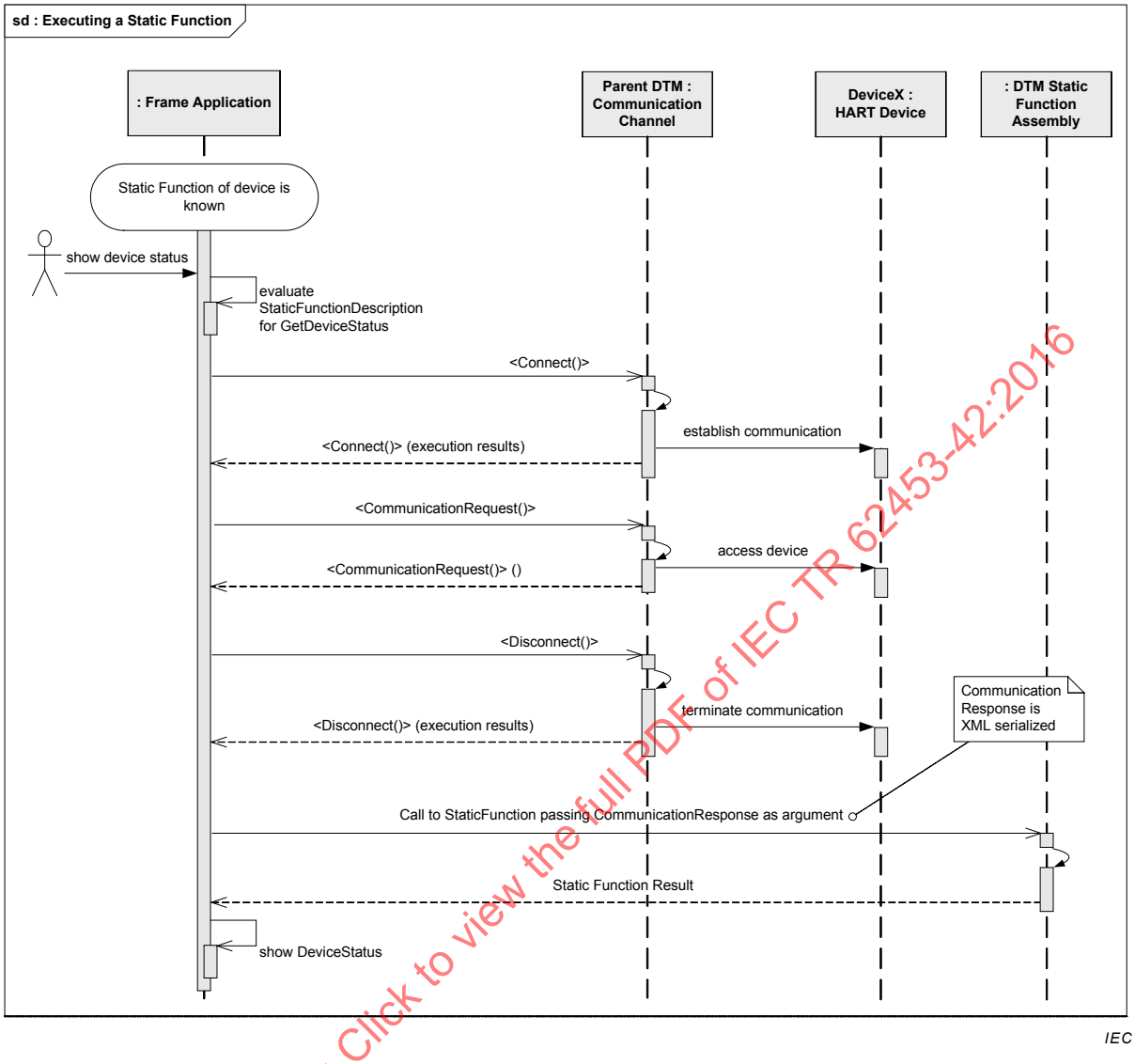


Figure 148 – Example: Information about available Static Functions

8.5.20 Executing a Static Function

After retrieving the information regarding the available static functions, the Frame Application may provide triggers for execution of the Static Functions to the user (e.g. in a menu) or may use internal triggers to execute a Static Function (see Figure 149).



Used methods:

ICommunication.BeginConnect()

ICommunication.EndConnect()

ICommunication.BeginCommunicationRequest()

ICommunication.EndCommunicationRequest()

ICommunication.BeginDisconnect()

ICommunication.EndDisconnect()

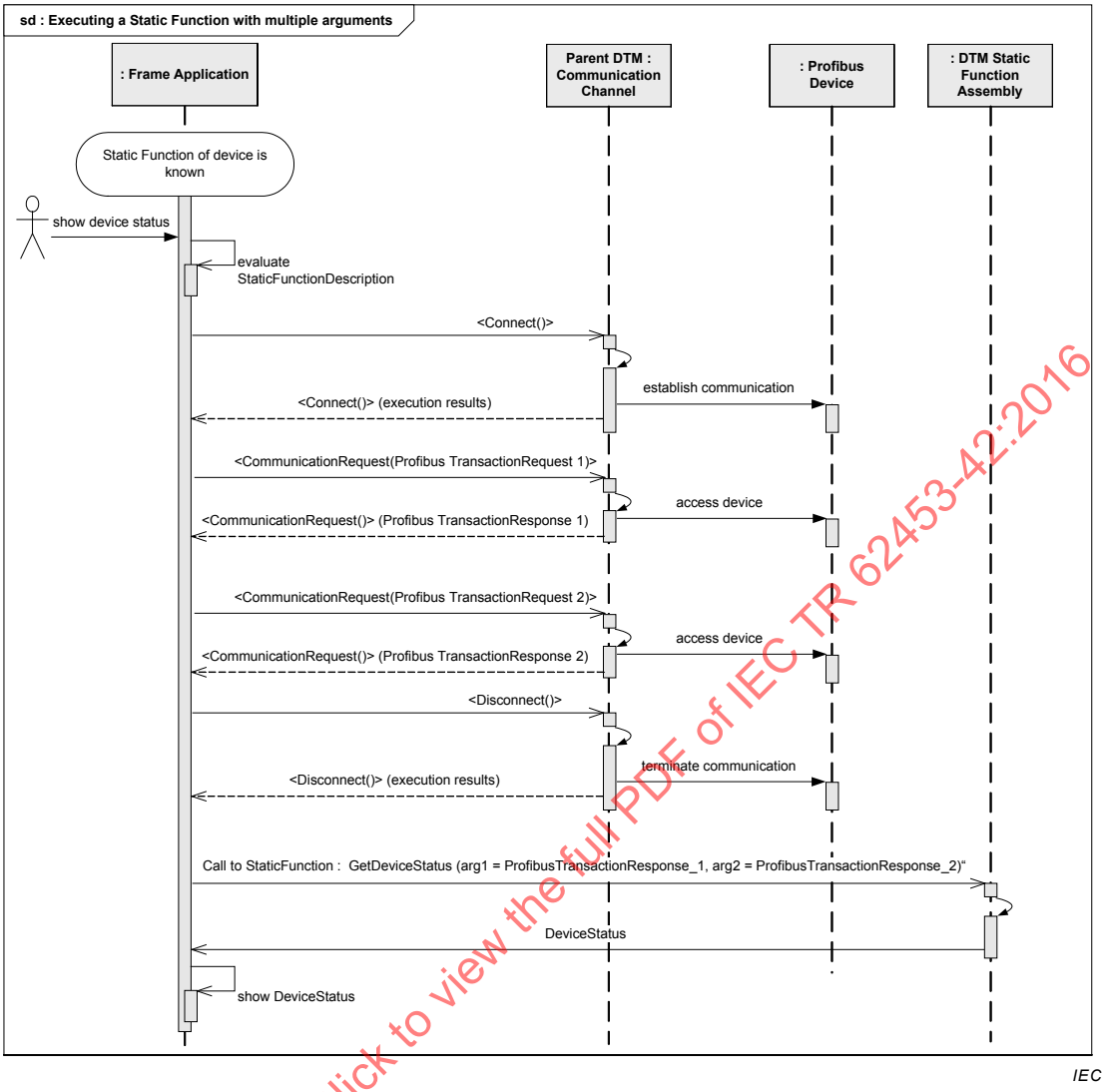
IStaticFunction.BeginExecute()

IStaticFunction.EndExecute()

Figure 149 – Executing a Static Function

8.5.21 Executing a Static Function with multiple arguments

If a Static Function is using multiple input arguments, that are CommunicationResponses, then the Frame Application shall retrieve the CommunicationResponses in the same order that is used to list the InputArgumentDescriptions (see Figure 150).



Used methods:

ICommunication.BeginConnect()

ICommunication.EndConnect()

ICommunication.BeginCommunicationRequest()

ICommunication.EndCommunicationRequest()

ICommunication.BeginDisconnect()

ICommunication.EndDisconnect()

IStaticFunction.BeginExecute()

IStaticFunction.EndExecute()

Figure 150 – Executing a Static Function with multiple Arguments

8.6 DTM communication

8.6.1 General

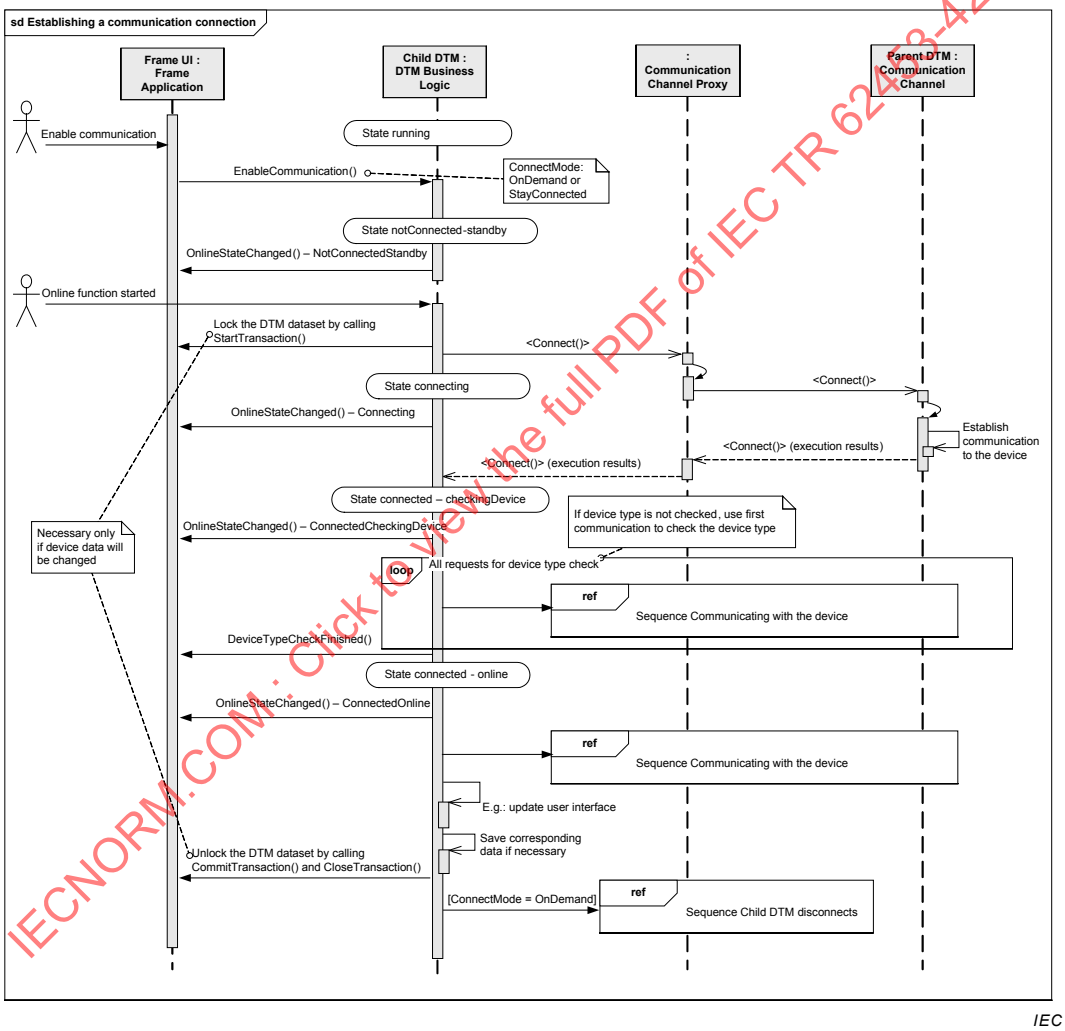
Each communication connection for a DTM is established as a point-to-point connection. This subclause describes the field communication related workflows. Communication Channels implement the interface ICommunicationChannel. The interface ICommunication can be

accessed by the ICommunicationChannel property “Communication” and provides services for fieldbus connection and communication requests.

In order to ensure that only the Frame Application can modify the sub-topology of a Communication Channel, DTMs cannot directly access the ICommunicationChannel interface of the parent channel. Instead the Frame Application provides a proxy for the channel implementing the ICommunicationChannelProxy interface. This proxy provides access to all Communication Channel interfaces except the interface for sub-topology management. The proxy redirects all method calls to the Communication Channel.

8.6.2 Establishing a communication connection

The following sequence diagram describes the calling sequence of a DTM when connecting to the field device (see Figure 151).



Used methods:

IDtm.EnableCommunication()

ICommunication.BeginConnect()

ICommunication.EndConnect()

Event IDtm.OnlineStateChanged()

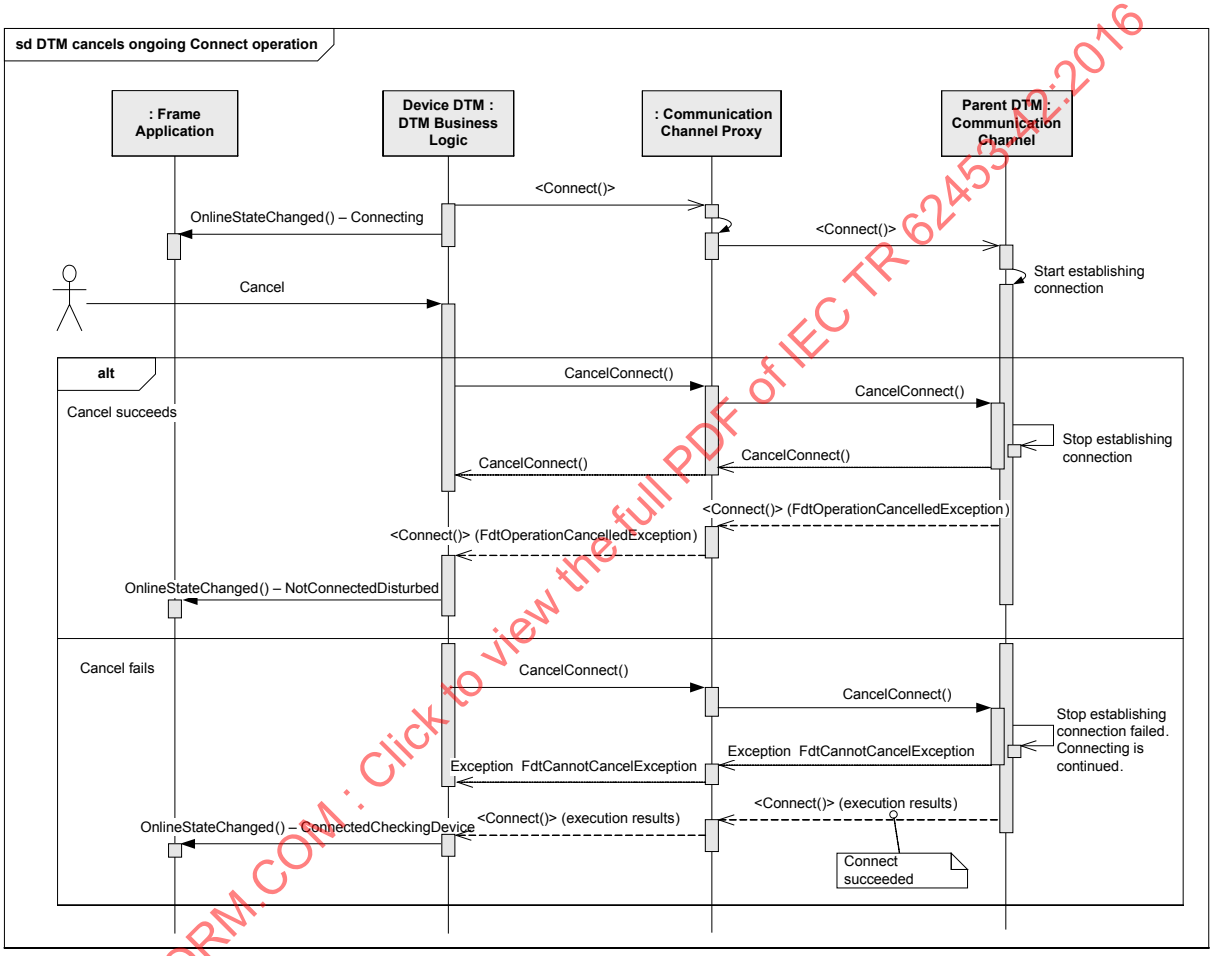
Figure 151 – Establishing a communication connection

Online functions which affect the device data or the instance data require a locked DtmDataset. Prolonged locks shall be avoided to support multi-user Frame Applications. Thus, it is a DTM-specific decision to balance between the granularity of online operations and the drawback of prolonged locks.

8.6.3 Cancel establishment of communication connection

This workflow describes how an ongoing connect request is canceled (see Figure 152).

If the connect action cannot be canceled, the call of the method CancelConnect() throws an exception.



Used methods:

IDtm.EnableCommunication()

ICommunication.BeginConnect()

ICommunication.CancelConnect()

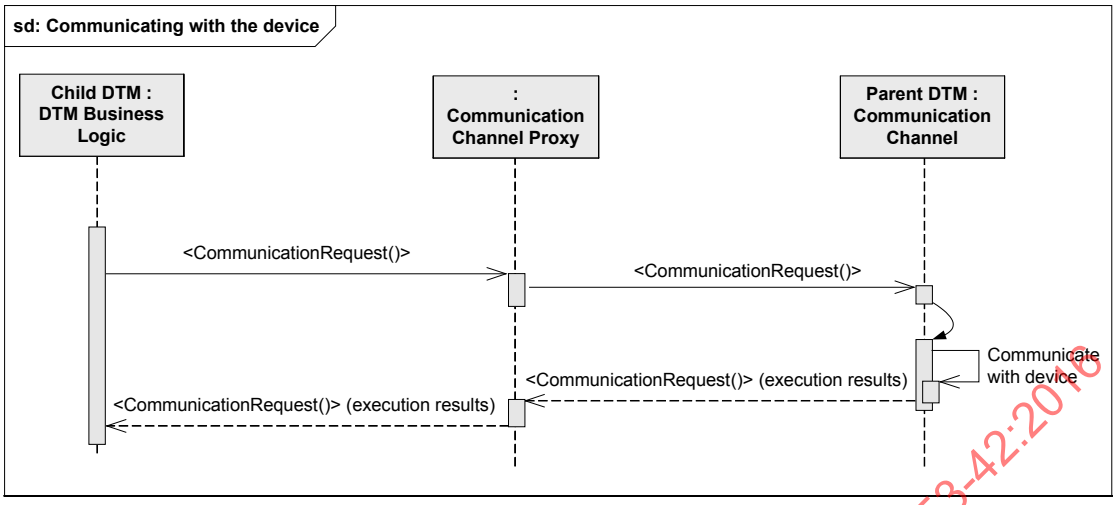
ICommunication.EndConnect()

Event IDtm.OnlineStateChanged()

Figure 152 – DTM cancels ongoing Connect operation

8.6.4 Communicating with the device

The following sequence diagram explains the Device DTM communication with the device using a Communication Channel (see Figure 153).



Used methods:

ICommunication.BeginCommunicationRequest()

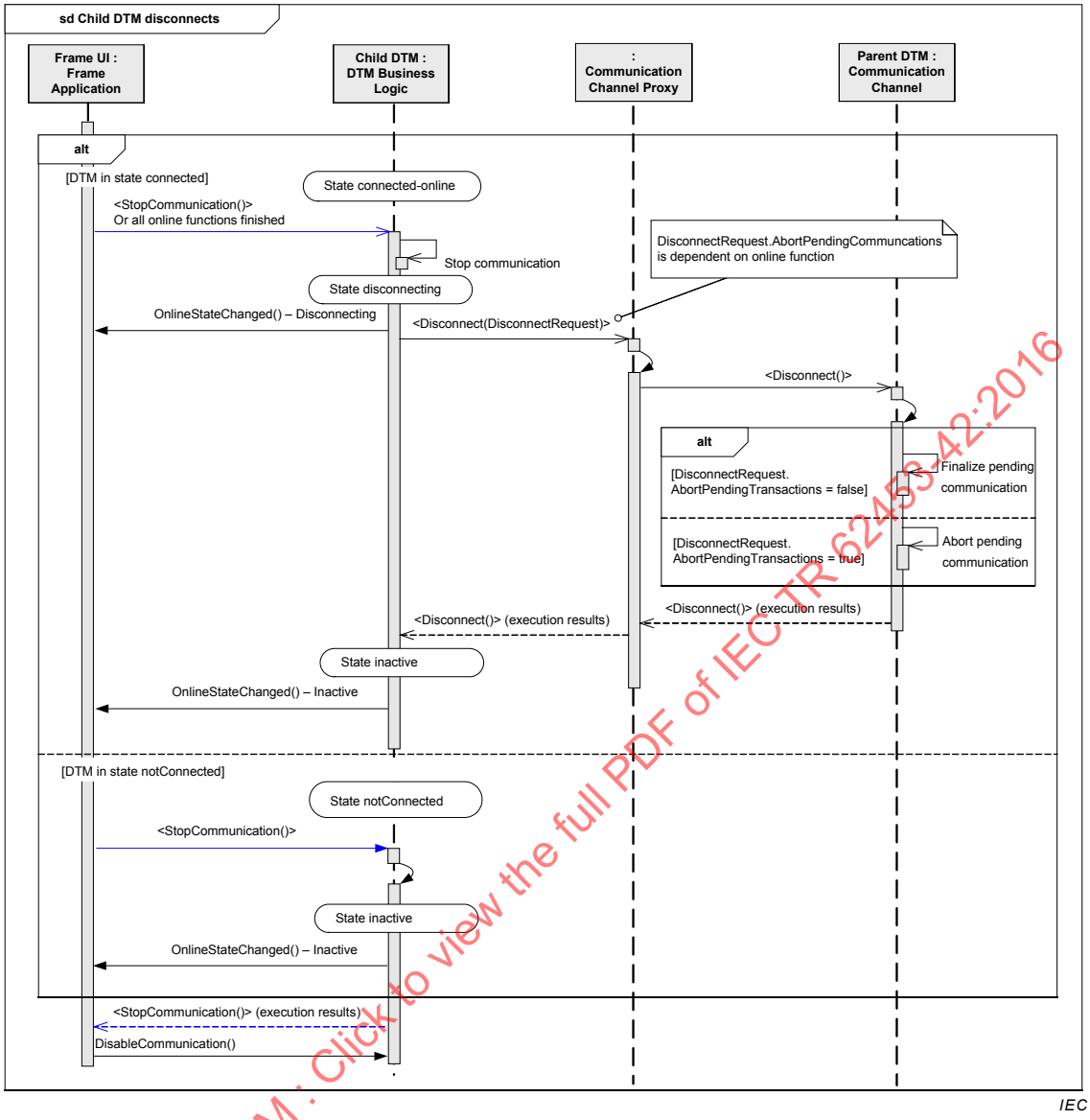
ICommunication.EndCommunicationRequest()

Figure 153 – Communicating with the device

8.6.5 Frame Application or Child DTM disconnect a device

Figure 154 shows the flow of messages, when a Frame Application sets a DTM offline.

It depends on the Child DTM, whether pending communication requests are finalized or aborted.



Used methods:

IDtm.BeginStopCommunication()

IDtm.EndStopCommunication()

Event IDtm.OnlineStateChanged()

ICommunication.BeginDisconnect()

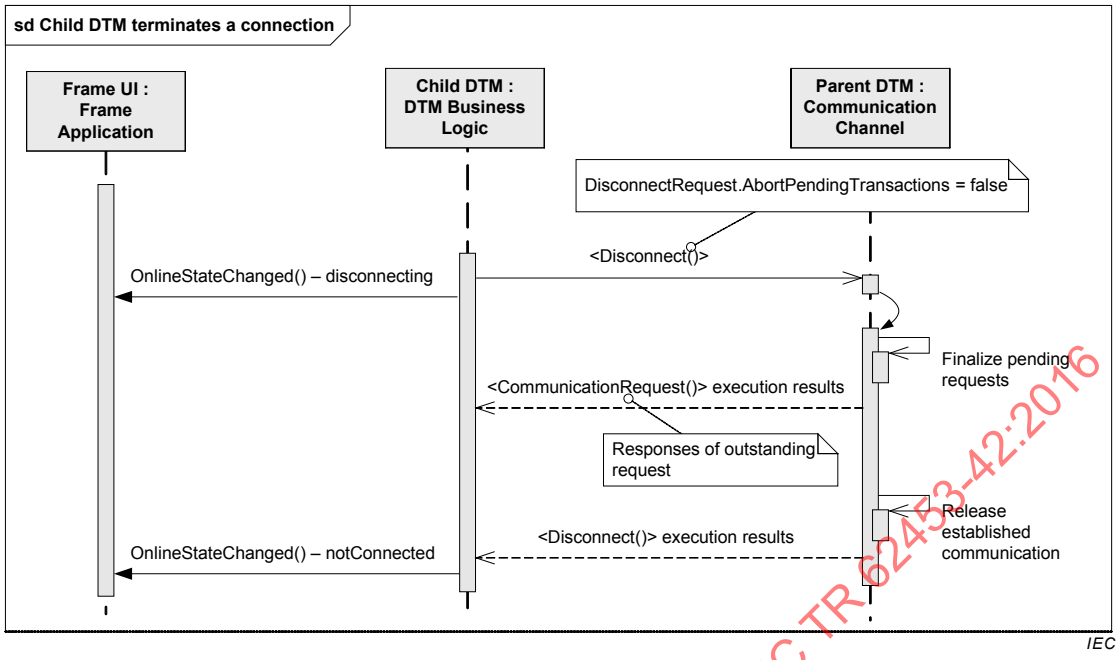
ICommunication.EndDisconnect()

IDtm.DisableCommunication()

Figure 154 – Child DTM disconnects

8.6.6 Terminating a communication connection

The sequence diagram shown in see Figure 155 shows how a communication connection is terminated by a Child DTM.



Used methods:

ICommunication.BeginDisconnect()

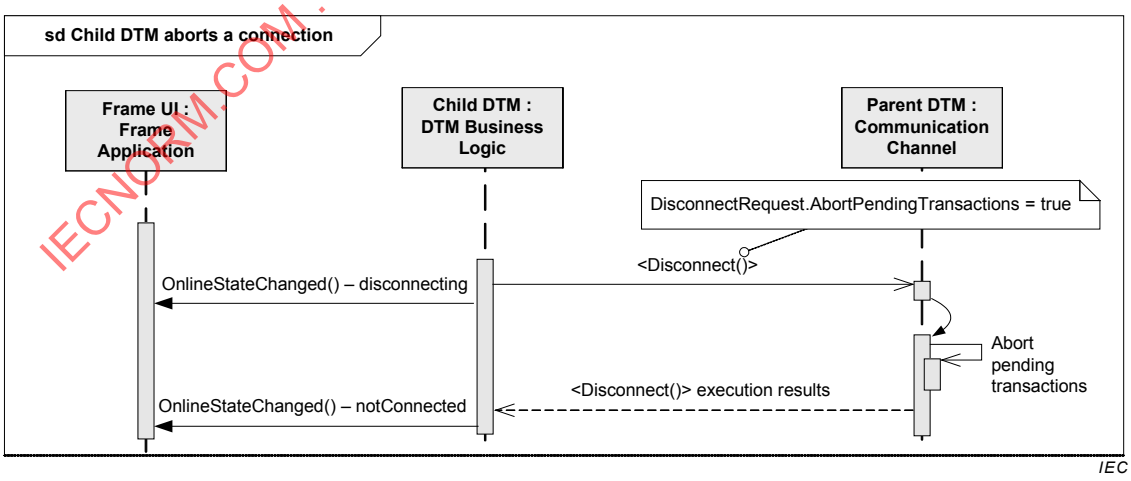
ICommunication.EndDisconnect()

Figure 155 – Child DTM terminates a connection

In case of a <Disconnect()> with argument AbortPendingTransactions set to 'false', the Communication Channel executes all outstanding communication requests. The Child DTM will receive responses with the respective communication data.

8.6.7 DTM aborts communication connection

This sequence (Figure 156) describes the abort of a communication link to a device without expecting any further communication response.



Used methods:

ICommunication.BeginDisconnect()

ICommunication.EndDisconnect()

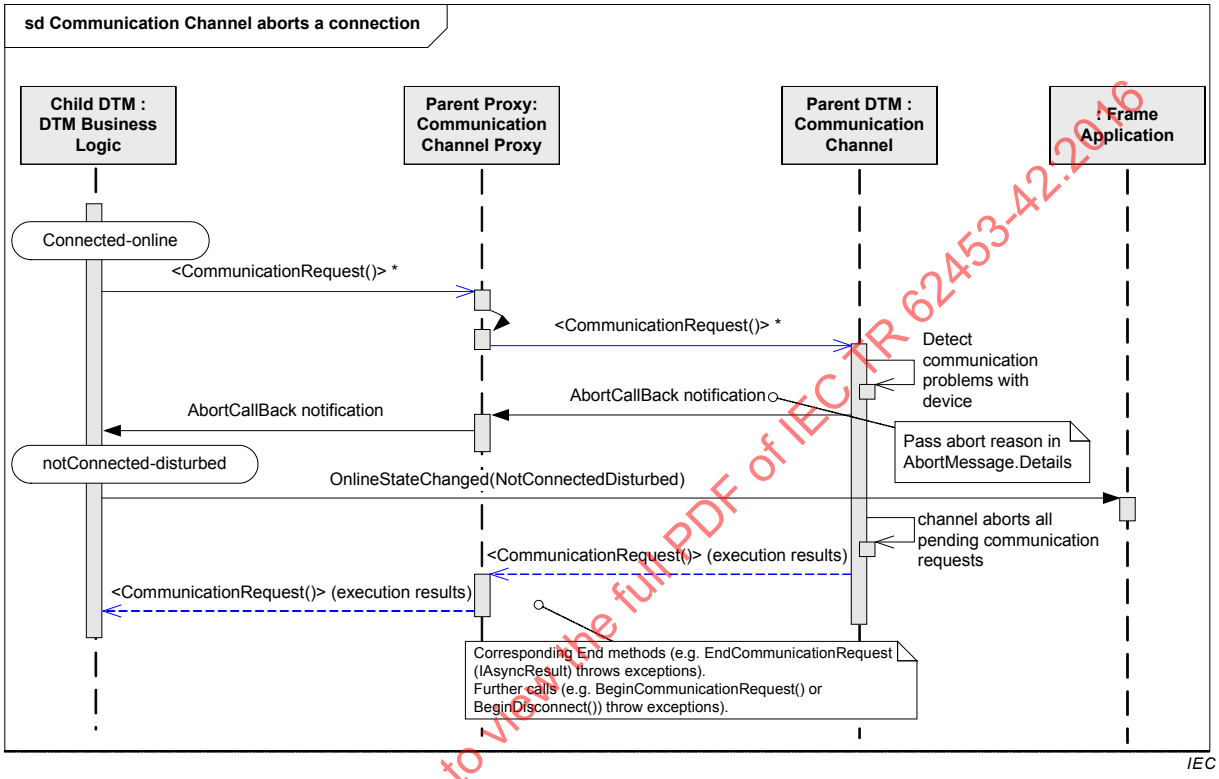
Event OnlineStateChanged()

Figure 156 – Child DTM aborts a connection

In case of a <Disconnect ()> with argument AbortPendingTransactions set to 'true', the Communication Channel cancels all outstanding communication requests. The Child DTM will receive responses with the information that the communication request was not executed.

8.6.8 Communication Channel aborts communication connection

This sequence (Figure 157) describes how a Communication Channel aborts an active communication connection to a device.



Used methods:

ICommunication.BeginCommunicationRequest()

ICommunication.EndCommunicationRequest()

Used events:

AbortCallBack delegate

Event OnlineStateChanged()

Figure 157 – Communication Channel aborts a connection

8.7 Nested communication

8.7.1 General

This subclause describes communication related to devices with gateway functionality like remote I/Os. Nested communication is used to establish the connection to a device on a sub-system.

The example in Figure 158 shows how a Device DTM communicates to a field device which is connected to a Communication Channel of a Gateway DTM, which in turn is connected to a Communication Channel of a Communication DTM. Since the Device DTM represents a HART field device, it is communicating based on HART protocol. The Gateway DTM represents a PROFIBUS/HART gateway (e.g. a Remote IO), that is why the Gateway DTM is

communicating to the gateway based on PROFIBUS protocol. The Communication DTM represents the fieldbus interface, the DTM accesses the driver of the fieldbus interface. (This example will also be used in other subclauses of 8.7).

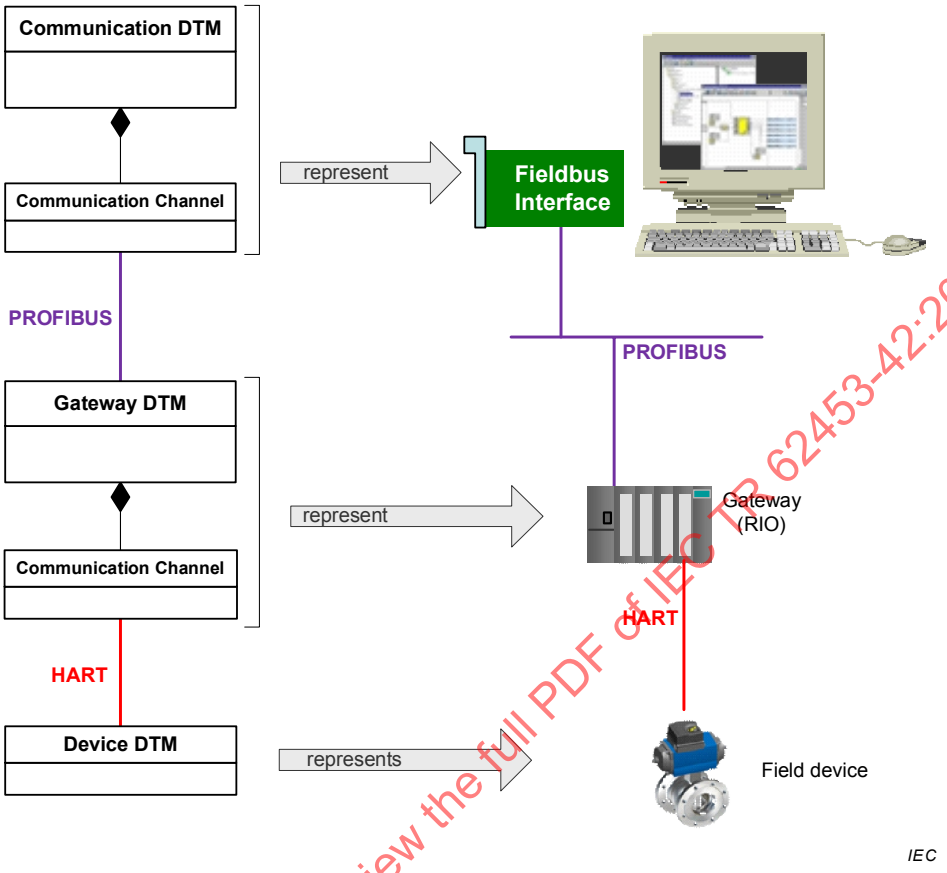


Figure 158 – Example: Nested communication behavior

Gateway DTMs (e.g. for a remote I/O) have to provide one or more Communication Channels that are used by other DTMs.

The requirement is that a DTM shall not need to know anything about the communication hierarchy. Nevertheless, the structure of the sub-system is well known to Frame Application and by the Gateway DTM.

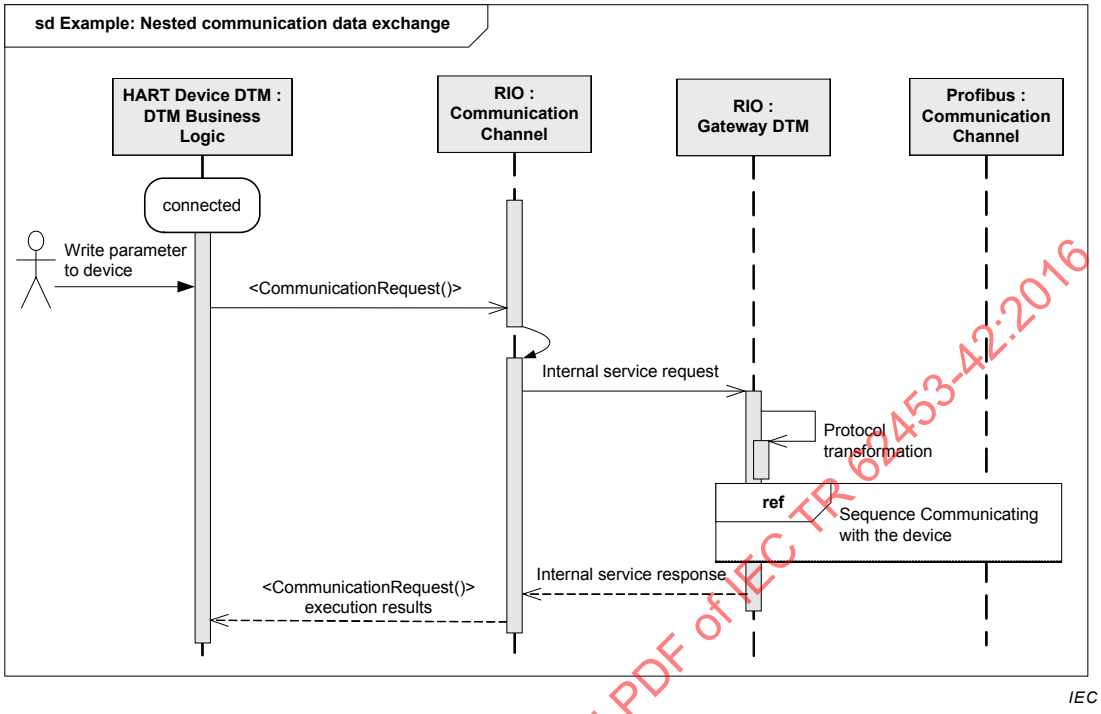
The functionality for address management is always provided by the Frame Application or by the Parent DTM. Therefore each DTM has to allow setting the network parameters like 'tag' and 'BusInformation' according to the communication protocol (see also: INetworkData, NetworkDataInfo, AddressInfo).

8.7.2 Communication request for a nested connection

The sequence in Figure 159 shows an example how the HART Device DTM from Figure 158 communicates to its field device. The internal communication of the Gateway DTM and the communication to the PROFIBUS Communication Channel are transparent to the Device DTM.

To write a parameter to the device, the HART Child DTM calls BeginCommunicationRequest() at the Communication Channel. The HART request is wrapped in the remote I/O channel to a PROFIBUS communication message sent to the parent PROFIBUS Communication DTM.

The corresponding response is provided by the PROFIBUS Parent Communication DTM. After extracting the HART response, the remote I/O Gateway DTM sends the response via the Communication Channel to the HART Device DTM.



Used methods:

ICommunication.BeginCommunicationRequest()

ICommunication.EndCommunicationRequest()

Figure 159 – Example: Nested communication data exchange

8.7.3 Propagation of errors for a nested connection

In a nested communication hierarchy there may be several sources for communication errors. If we consider the example from Figure 158, possible sources are:

- Field device responds to communication requests with errors (e.g. wire break)
- Gateway device (RIO) has communication problems (e.g. field device does not communicate) and responds with errors
- Gateway device (RIO) has internal problems (e.g. module failure) and responds with errors
- Fieldbus interface has communication problems (e.g. gateway device does not communicate) and responds with errors
- Fieldbus interface has internal problems (e.g. not configured) and responds with errors

If errors occur during execution of communication requests, the error have to be propagated back to the origin of the communication request (see 4.9.2).

In order to support fixing the problem, the DTM representing the component, where the error occurred shall inform the user about the source of error within the CommunicationError. This helps to avoid a situation, where the user receives several error reports (e.g. if gateway device detects, that the field device does not respond, the Gateway DTM will produce a user message and the Device DTM will produce a user message).

If an intermediate component receives such a communication error, it shall in turn generate a communication error, provide own additional information and shall pack the received communication error as inner communication error into the generated communication error (similar to exceptions/inner exceptions).

If the origin of communication request receives such a communication error, it shall inform the user with a user message that includes the information from the inner communication errors.

8.8 Topology planning

8.8.1 General

The Frame Application is responsible to generate and manage the topology.

The requirement is that a DTM shall not need to know anything about the communication hierarchy. Nevertheless, the structure of the whole topology is well known to a Frame Application.

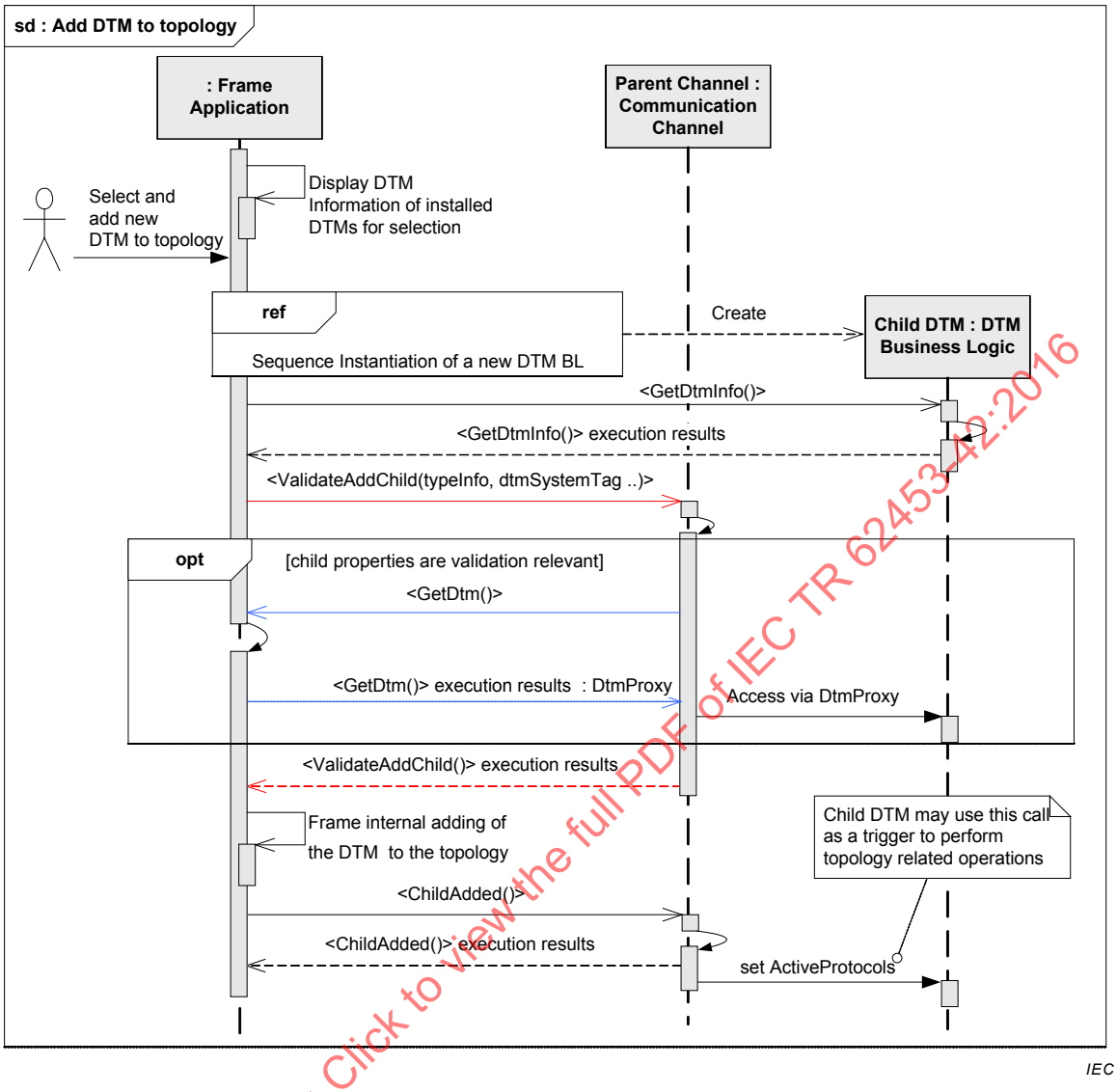
Subclause 8.8.2 describes how a Frame Application creates a topology. The example in 8.8.5 shows how a Gateway DTM generates a sub-topology.

8.8.2 Adding a DTM to the topology

If a DTM is added new to the topology, a validation is executed whether the DTM fits into the topology.

This validation is executed by the Communication Channel to which the new Child DTM is added. During the validation the Communication Channel may access the Child DTM.

Since the Child DTM at this point is not yet part of the topology, the Child DTM does not yet have a Parent DTM and may not access the Parent DTM (see Figure 160).



Used methods:

IDtmInformation.BeginGetInfo()

IDtmInformation.EndGetInfo()

ISubTopology.BeginValidateAddChild() / ISubTopology.EndValidateAddChild()

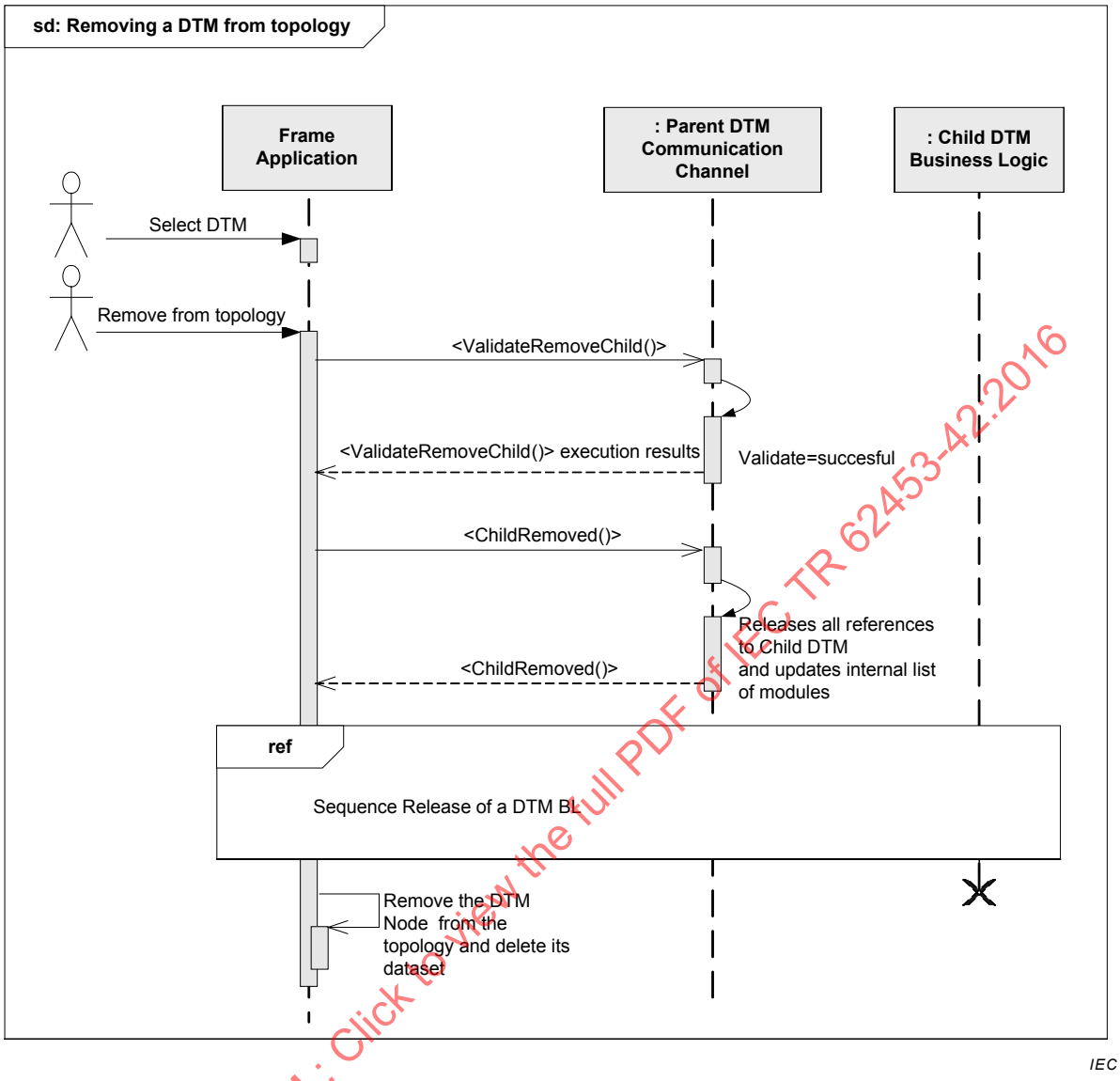
ISubTopology.BeginChildAdded() / ISubTopology.EndChildAdded()

INetworkData.ActiveProtocols

Figure 160 – Add DTM to topology

8.8.3 Removing a DTM from topology

Figure 161 shows how a DTM is removed from a topology. Before the Frame Application removes the device node and its dataset from the topology, the Parent DTM shall validate the removal, release all references to the Child DTM and update the internal list of modules.



Used methods:

ISubTopology.BeginValidateRemoveChild()

ISubTopology.EndValidateRemoveChild()

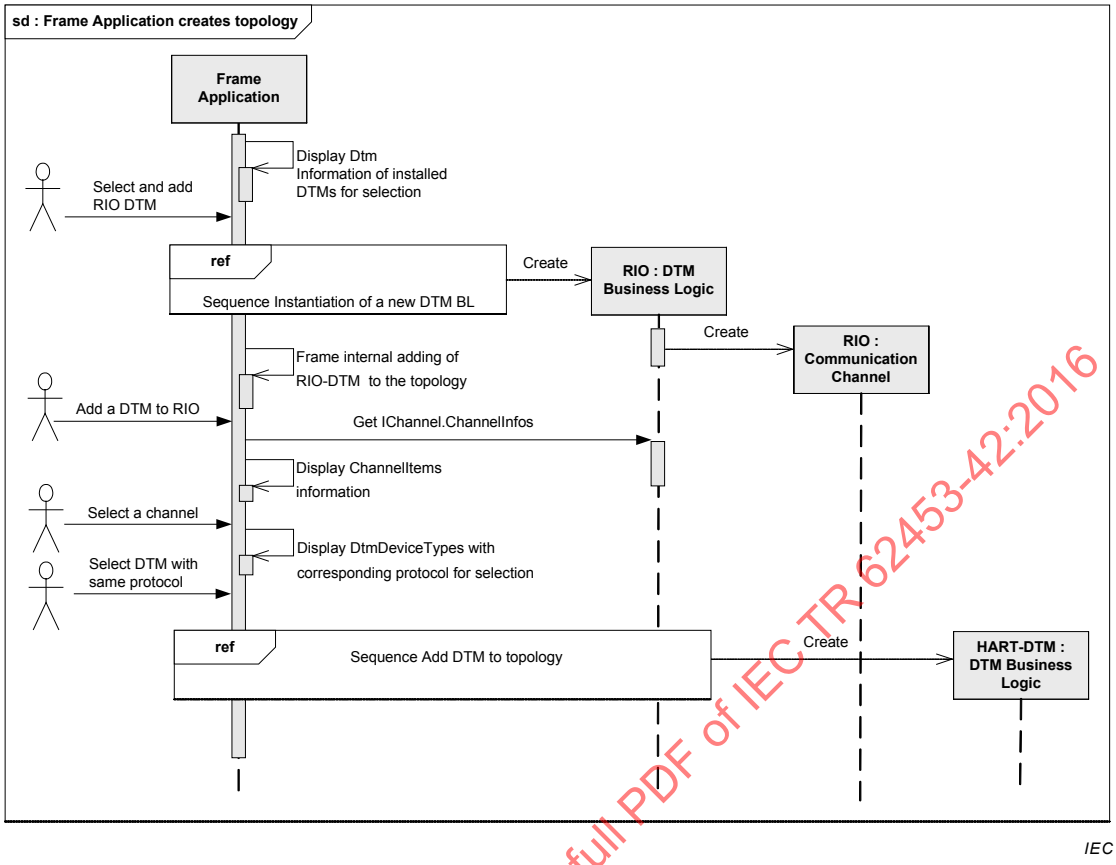
ISubTopology.BeginChildRemoved()

ISubTopology.EndChildRemoved()

Figure 161 – Removing a DTM from topology

8.8.4 Frame Application creates topology

The following sequence diagram (Figure 162) shows an example workflow how a Frame Application first adds a Gateway DTM (for a remote IO) to the topology and afterwards adds a Device DTM (for a HART device).



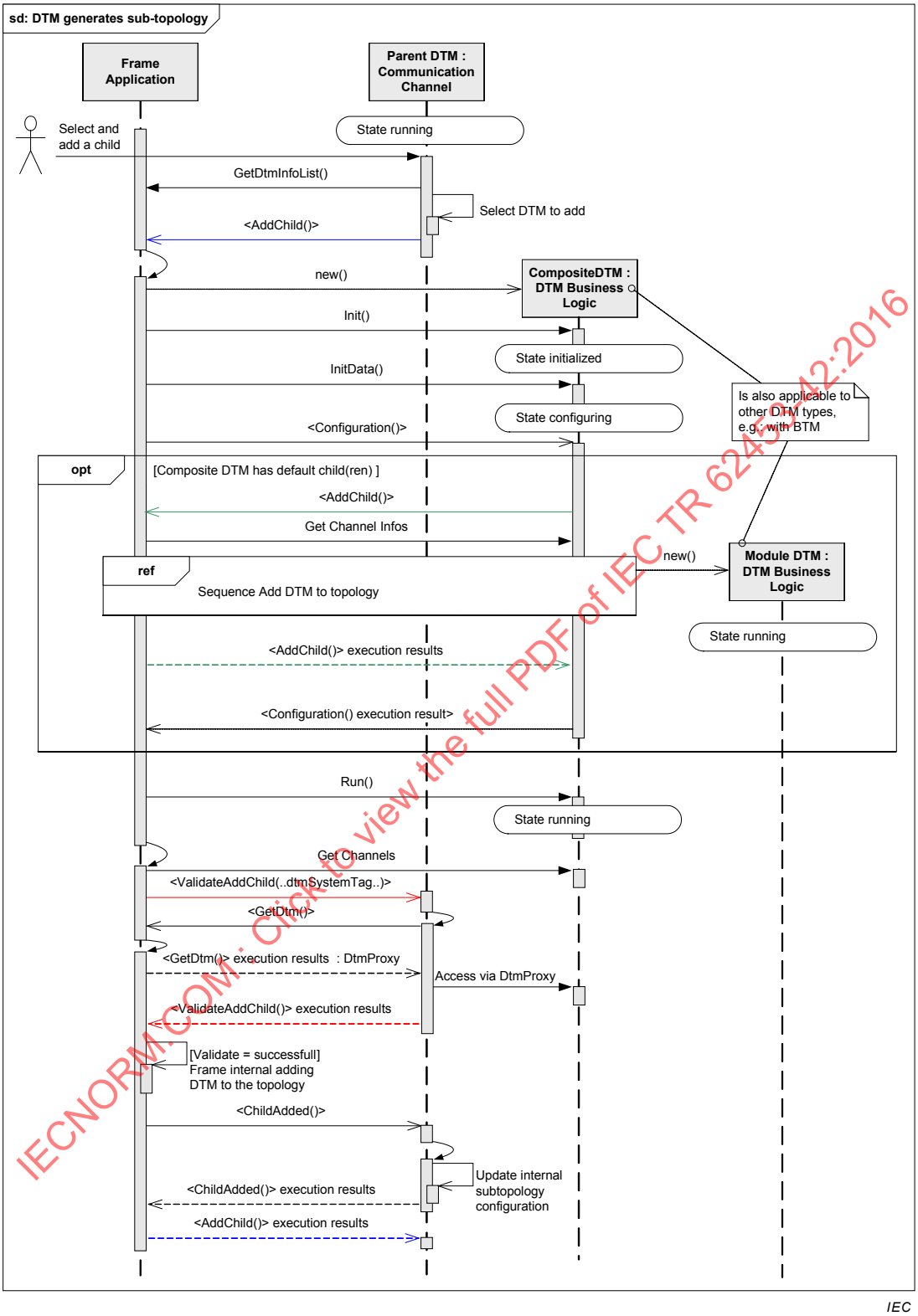
Used methods:

IChannels.ChannelInfos

Figure 162 – Frame Application creates topology

8.8.5 DTM generates sub-topology

This sequence diagram shows the generation of the sub-topology triggered by a DTM (see Figure 163).



Used methods:

ITopology.GetDtmInfoList()

ITopology.BeginAddChild() / ITopology.EndAddChild()

ISubTopology.BeginValidateAddChild() / ISubTopology.EndValidateAddChild()

ISubTopology.BeginChildAdded() / ISubTopology.EndChildAdded()

Figure 163 – DTM generates sub-topology

The same sequence can be used for adding Module DTMs to a Composite Device DTM and for adding BTMs to Device DTMs.

8.8.6 Physical Layer and DataLinkLayer

IEC 61158-2 defines a wide range of possible physical media that is used by different fieldbus protocols. Many fieldbus protocols support different physical media. For example HART supports wired (4-20 mA) and wireless connections, while PROFIBUS supports RS485, manchester-coded bus powered (MBP) and optical media. Even if a field device supports the same fieldbus protocol as a communication component (e.g. fieldbus interface or gateway), communication may be impossible, because device and communication component support different physical media. In such cases the use of media converters or gateway devices is required.

In order to avoid such incompatibility during offline planning of a physical topology, a Frame Application should use the physical layer information, which is exposed in the property Port.PhysicalLayers.

On the other hand, different protocols may share the same physical layer (e.g. Ethernet based protocols). If a physical layer is shared between protocols, it depends additionally on the IEC 61158-2 Data Link Layer, whether a physical connection is feasible or not.

In order to facilitate such checks, a Frame Application should use the data link layer information, which is exposed by the property Port.DataLinkLayers.

For comparison of the supported physical layer and data link layer, the properties PhysicalLayer and DataLinkLayer are used.

The following rules apply for Frame Applications managing the physical topology:

– If PhysicalLayer values do not match and DataLinkLayer values do not match, the Frame Application shall reject the new connection.
– If PhysicalLayer values match but DataLinkLayer values do not match, the Frame Application may reject the new connection.
– If PhysicalLayer values do not match but DataLinkLayer values do match, the Frame Application may issue a warning and accept the new connection, since the planned physical topology might contain transparent media converters, which are not part of the physical topology in the Frame Application.
– If both layers match, the Frame Application shall accept the new connection.

See Annex I for examples of PhysicalLayer values.

8.9 Instantiation, configuration, move and release of Child DTMs

8.9.1 General

The following workflows describe interactions between Parent DTM and Child DTMs. Such interactions may occur for instance between:

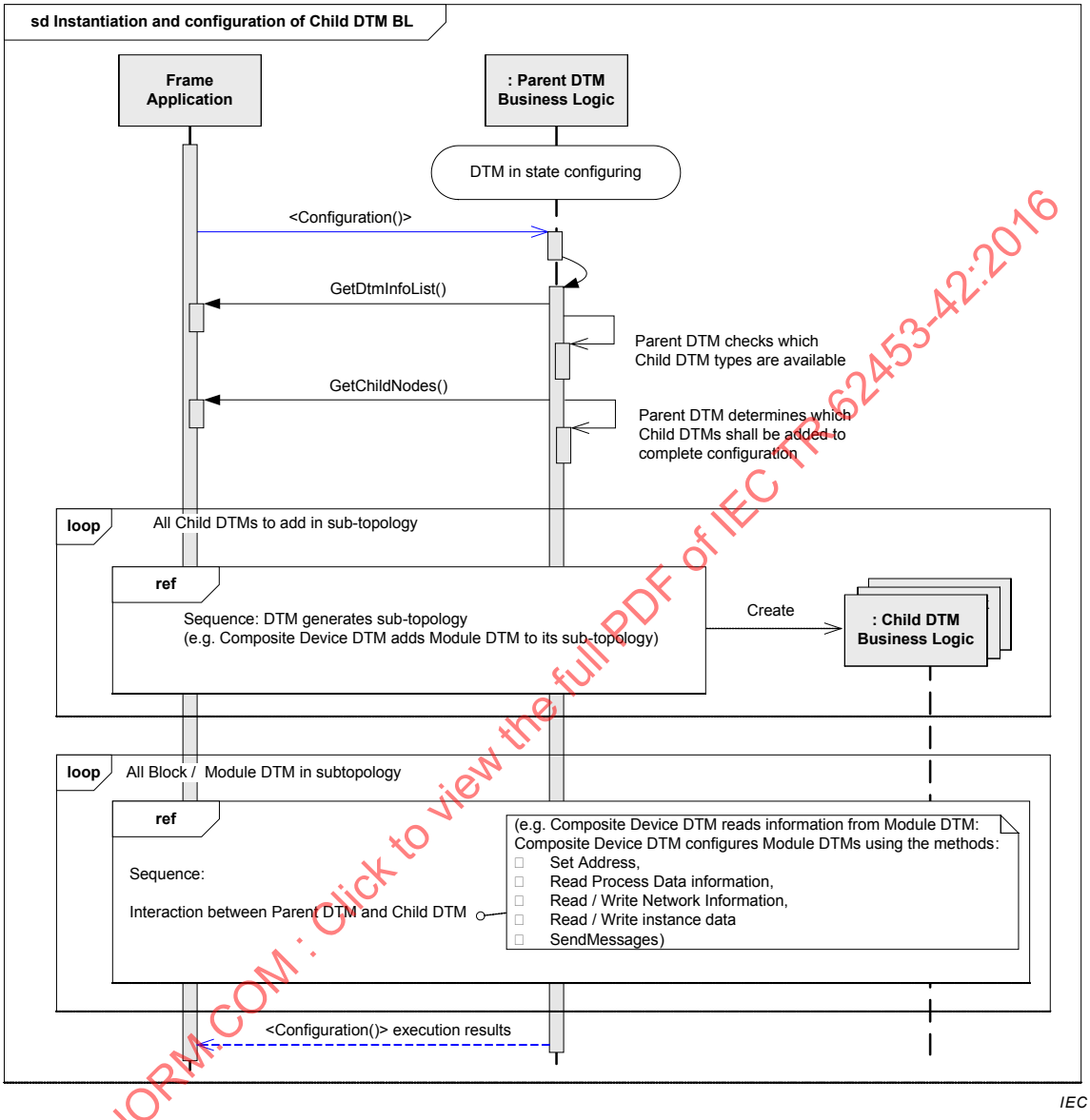
– Composite Device DTM and related Module DTMs
– Device DTM and related Block DTMs
– Gateway DTM and related Device DTMs

8.9.2 Instantiation and configuration of Child DTM BL

The Diagram in Figure 164 shows how a Parent DTM can create and configure its sub-topology. In order to enable configuration of a sub-topology, the Parent DTM has to implement

the <Configuration()> method. The Frame Application shall call <Configuration()> when the DTM is in state 'configuring'.

Be aware that a Parent DTM shall add Child DTMs only to its own channels.



Used methods:

IDtm.BeginConfiguration()

IDtm.EndConfiguration()

ITopology.GetDtmInfoList()

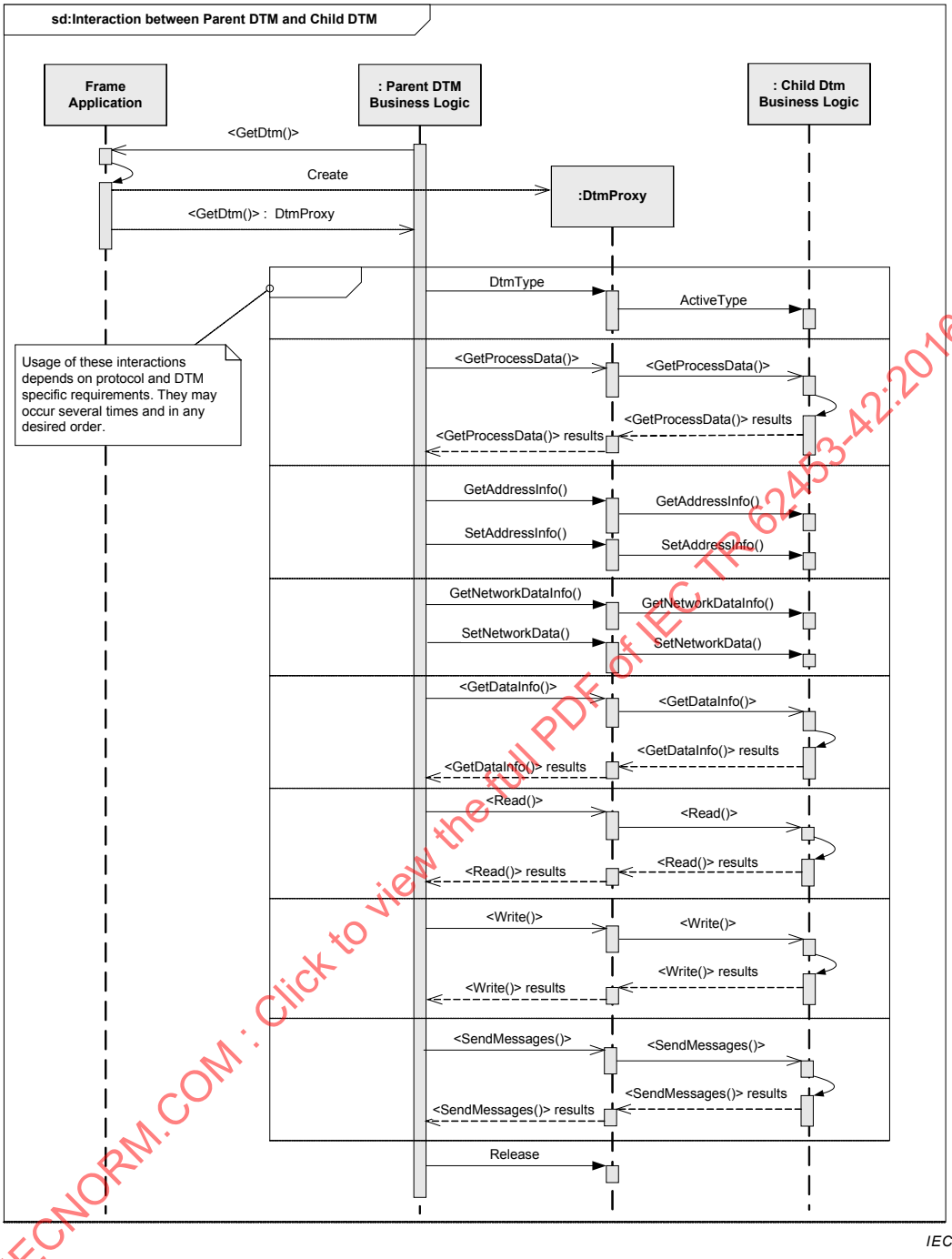
ITopology.GetChildNodes()

Figure 164 – Instantiation and configuration of Child DTM BL

8.9.3 Interaction between Parent DTM and Child DTM

Figure 165 shows how a Parent DTM can exchange data with its Child DTM.

Be aware that for interaction between DTMs only the interfaces shall be used which are provided by IDtmProxy.



Used methods:

ITopology.BeginGetDtm() / ITopology.EndGetDtm()

IDtmProxy.Dispose()

IDtmProxy.DtmType

IDtm.ActiveType

IProcessData.BeginGetProcessData() / IProcessData.EndGetProcessData()

INetworkData.GetAddressInfo() / INetworkData.SetAddressInfo()

INetworkData.GetNetworkDataInfo() / INetworkData.SetNetworkData()

IInstanceData.BeginGetDataInfo() / IInstanceData.EndGetDataInfo()

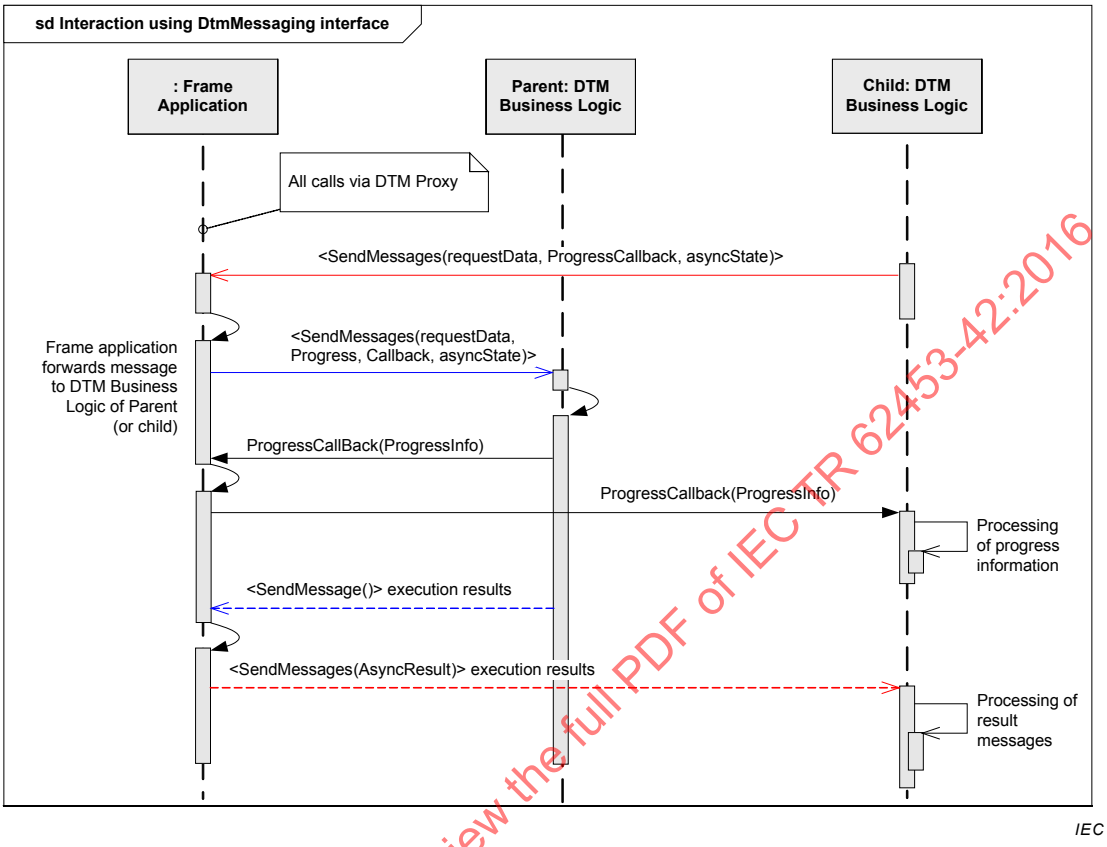
IInstanceData.BeginRead() / IInstanceData.EndRead()

IInstanceData.BeginWrite() / IInstanceData.EndWrite()

Figure 165 – Interaction between Parent DTM and Child DTM

8.9.4 Interaction between Parent DTM and Child DTM using IDtmMessaging

This sequence diagram outlines the interaction between two DTMs using the IDtmMessaging interface.



Used methods:

IDtmMessaging.BeginSendMessages()

IDtmMessaging.EndSendMessages()

Figure 166 – Interaction using IDtmMessaging

In this scenario the DTM Business Logic of a Child DTM sends a list of proprietary messages to its Parent DTM. The Frame Application provides access to the IDtmMessaging by means of the IDtmProxy. It shall forward the messages to the corresponding DTM.

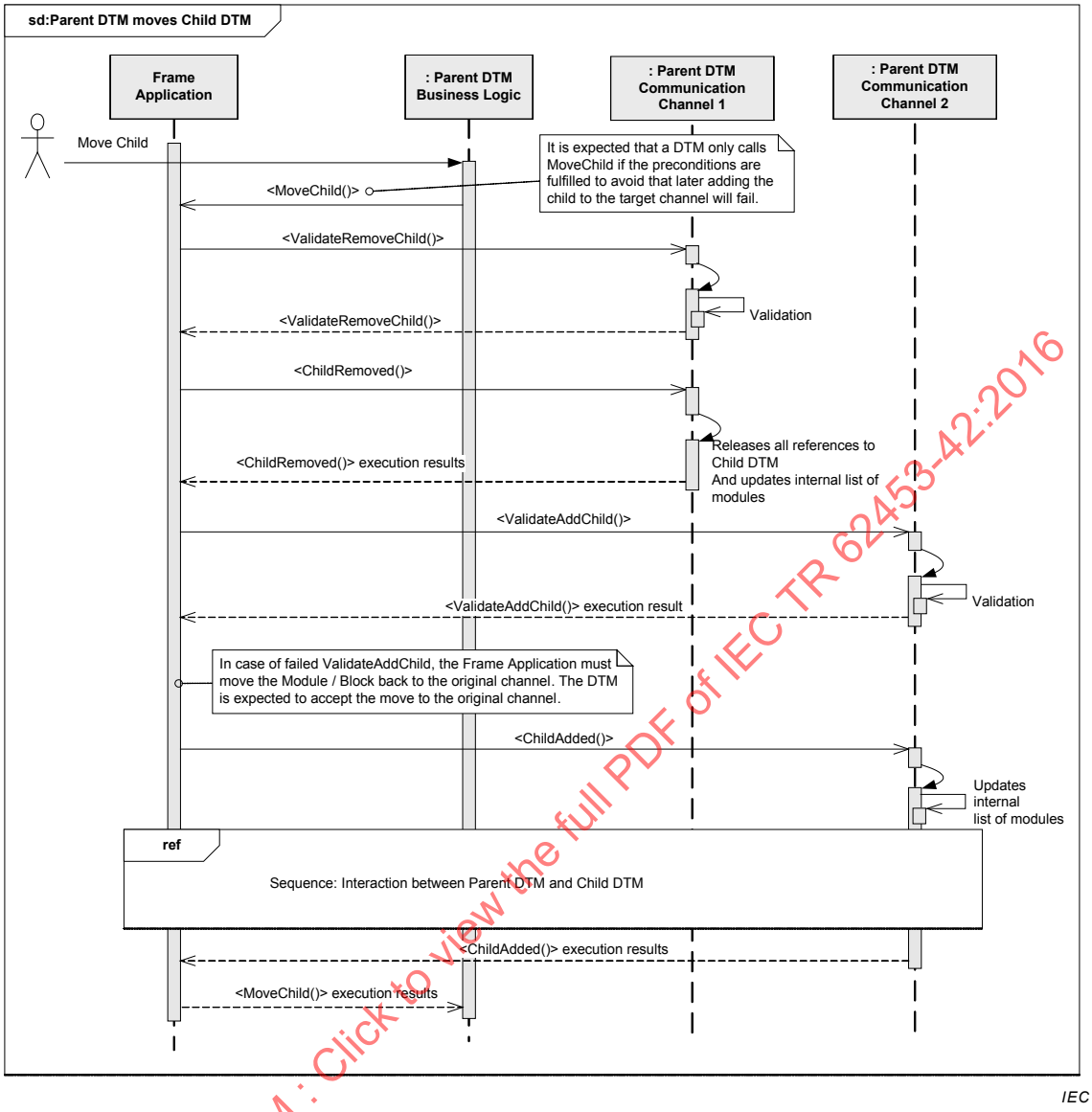
More detailed information can be found in descriptions of:

– IDtmMessaging
– DtmRequestMessage
– DtmResponseMessage

8.9.5 Parent DTM moves a Child DTM

Figure 167 shows how a Parent DTM can move one of its Child DTMs from one channel to another channel.

Be aware that a Parent DTM shall move Child DTMs only between its own channels.



Used methods:

ITopology.BeginMoveChild() / ITopology.EndMoveChild()

ISubTopology.BeginValidateAddChild() / ISubTopology.EndValidateAddChild()

ISubTopology.BeginChildAdded() / ISubTopology.EndChildAdded()

ISubTopology.BeginValidateRemoveChild() / ISubTopology.EndValidateRemoveChild()

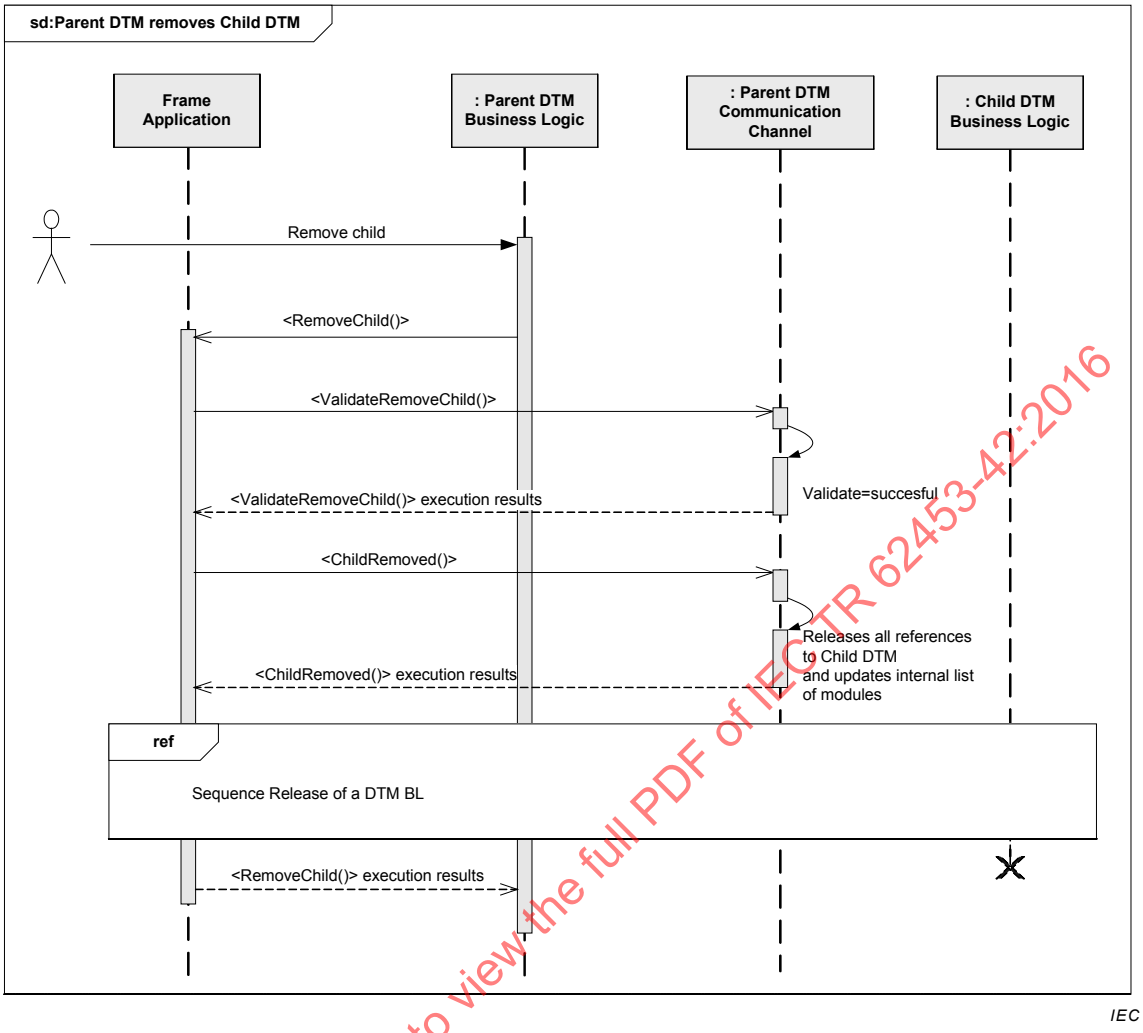
ISubTopology.BeginChildRemoved() / ISubTopology.EndChildRemoved()

Figure 167 – Parent DTM moves a Child DTM

8.9.6 Parent DTM removes Child DTM

Figure 168 shows how a Parent DTM can remove one of its Child DTM

Be aware that a Parent DTM can remove only its own Child DTMs.



Used methods:

ITopology.BeginRemoveChild() / ITopology.EndRemoveChild()

ISubTopology.BeginValidateRemoveChild() / ISubTopology.EndValidateRemoveChild()

ISubTopology.BeginChildRemoved() / ISubTopology.EndChildRemoved()

Figure 168 – Parent DTM removes Child DTM

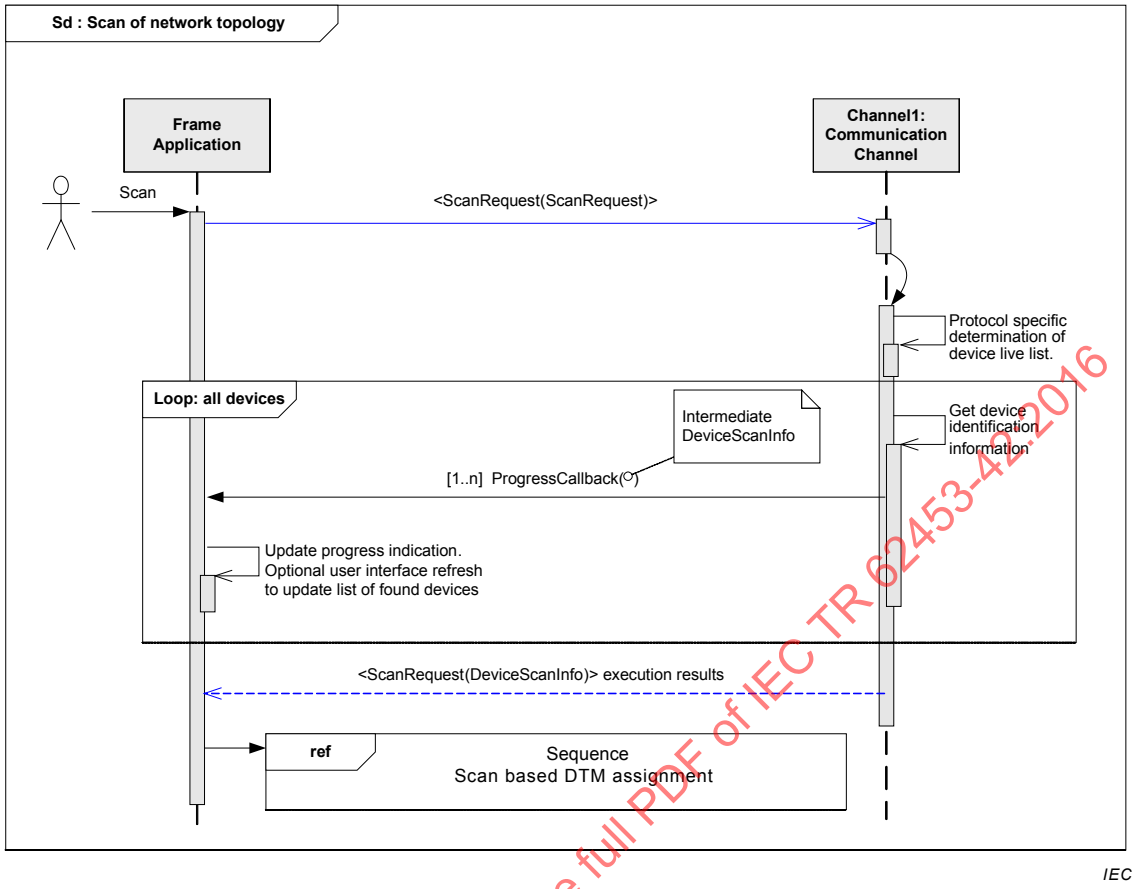
8.10 Topology scan

8.10.1 General

For a description of the general mechanism see IEC TR 624532:−, 6.2.

8.10.2 Scan of network topology

The following workflow describes, how a Frame Application can request a list of connected devices and their protocol-specific device identification information from a Communication Channel (see Figure 169).



Used methods:

IScanning.BeginScanRequest()

IScanning.EndScanRequest()

ProgressCallback

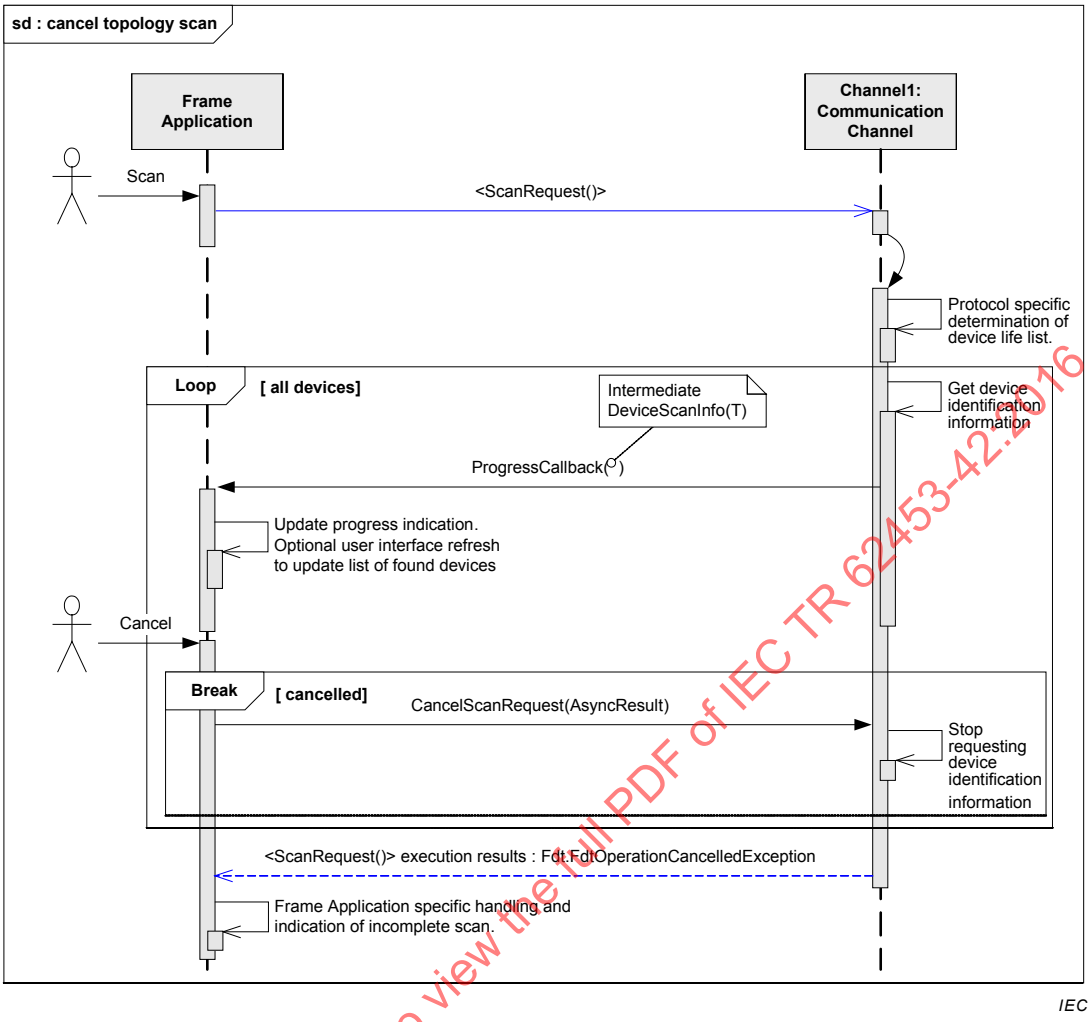
Figure 169 – Scan of network topology

The final result data for the scan received with the IScanning.EndScanRequest() contains a list of DeviceScanInfo objects where each object contains information about a single device found on the bus. If the order of devices is relevant for the protocol of the Communication Channel, the order of objects in the final result list shall match the order of the devices on the bus. Contrary to the final result, the order of devices in the intermediate results may depend from the scanning algorithm and may differ from the final result.

For information on how a protocol-specific DeviceScanInfo(T) can be transformed into a protocol-independent DeviceScanInfo, please refer to the datatype definition (see Annex B).

8.10.3 Cancel topology scan

Scanning a sub-topology may take some time. The FDT methods are designed to be called asynchronously. If a Frame Application calls the scan methods asynchronously, CancelScanRequest() may be called to cancel an ongoing scanning operation in the Communication Channel. The following sequence shows the related flow of events (see Figure 170).



Used methods:

IScanning.BeginScanRequest()

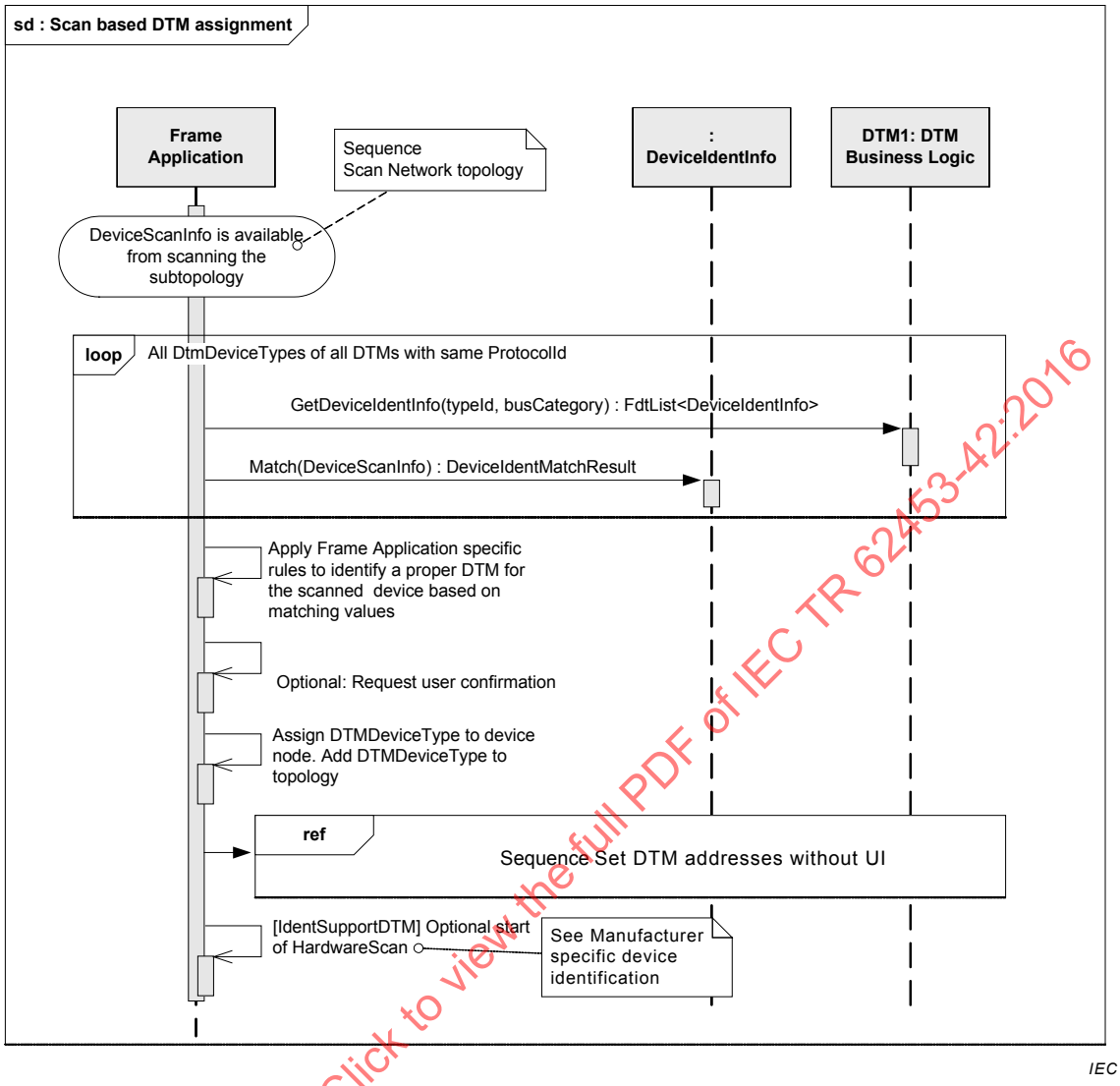
IScanning.CancelScanRequest()

Callback ScanProgress

Figure 170 – Cancel topology scan

8.10.4 Scan based DTM assignment

A Frame Application may use the scanned life list to find appropriate DTMDeviceTypes and create a sub-topology accordingly. The following sequence chart describes the related flow of events (see Figure 171).



Used methods:

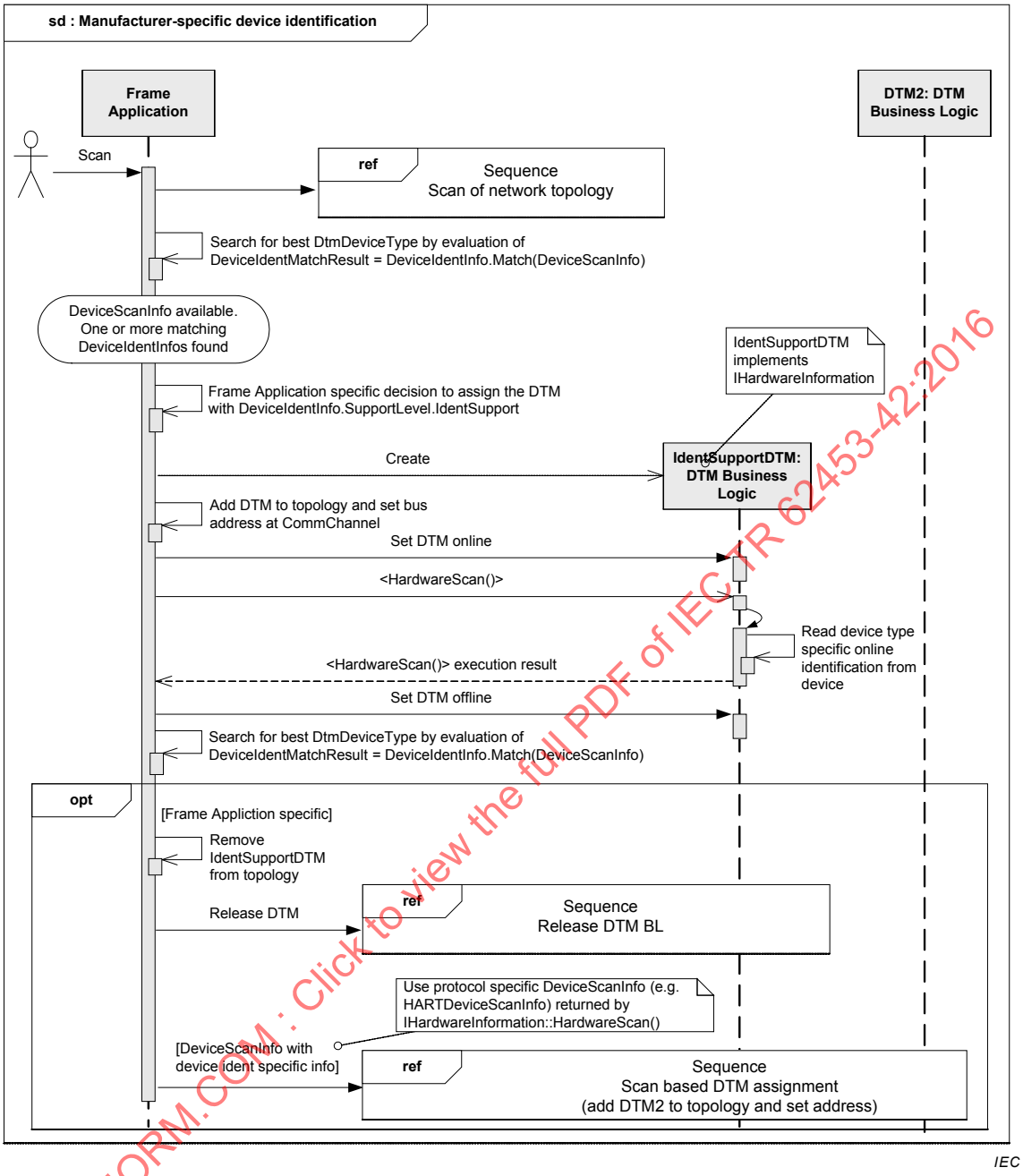
IDtmInformation.GetDeviceIdentInfo()

DeviceIdentValue<T>.Match()

Figure 171 – Scan based DTM assignment

8.10.5 Manufacturer-specific device identification

In this scenario a Frame Application scans an existing fieldbus network and uses DTM implementing IHardwareInformation interface to identify devices for which manufacturer-specific operation shall be performed (see Figure 172).



Used methods:

IHardwareInformation.BeginHardwareScan()

IHardwareInformation.CancelHardwareScan()

IHardwareInformation.EndHardwareScan()

DeviceScanInfo

Figure 172 – Manufacturer-specific device identification

For more information on manufacturer-specific device identification refer to IEC TR 62453-2:−, 6.2.4.

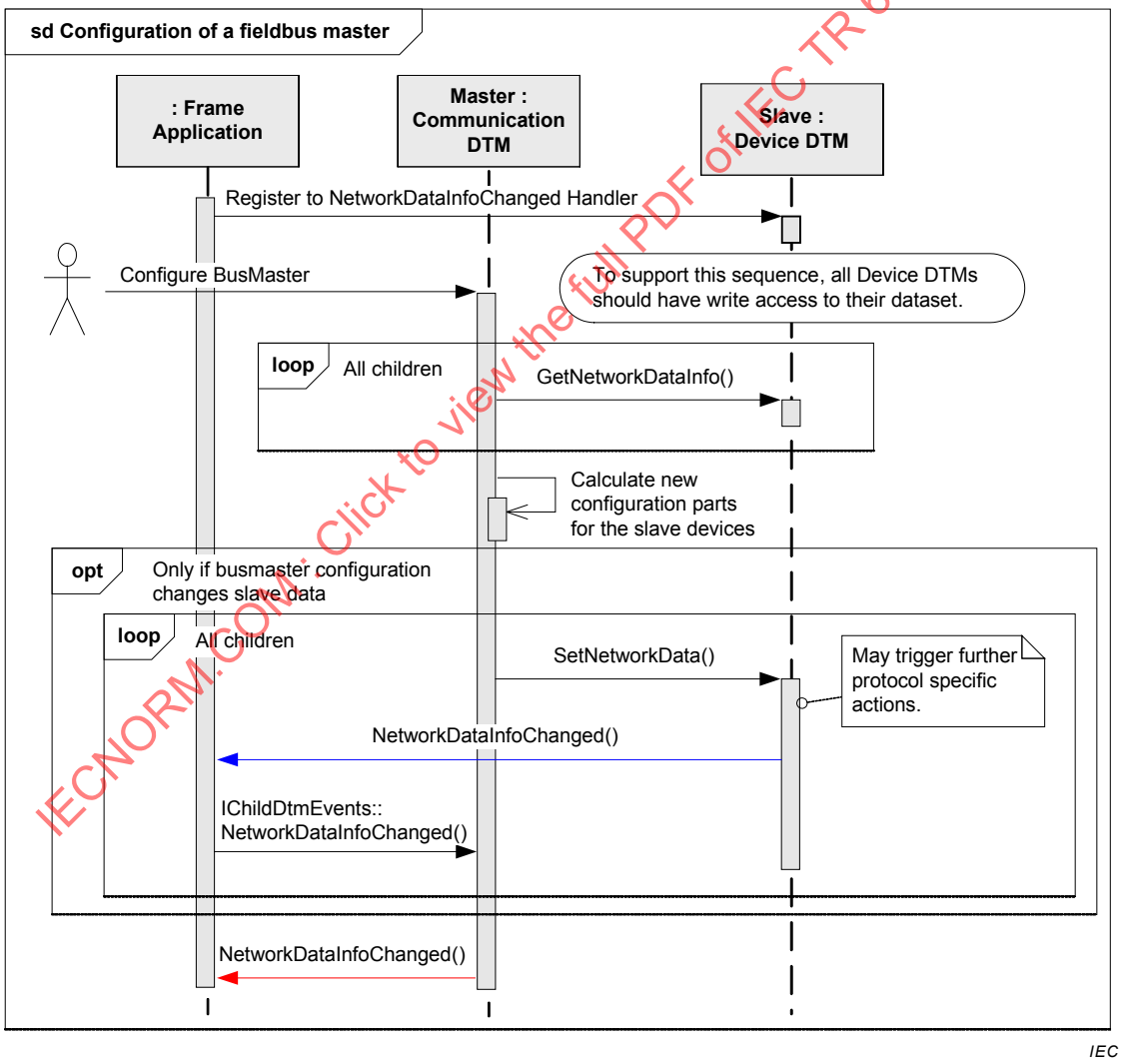
8.11 Configuration of communication networks

8.11.1 Configuration of a fieldbus master

Device-specific bus parameters are needed to configure the fieldbus master or communication scheduler. To retrieve these parameters an interaction between DTMs and a master configuration tool (e.g. provided by Master Communication DTM or by Frame Application) is required. Bus-specific data information is provided by Device DTMs in NetworkDataInfo and contains the device specific bus information according to the fieldbus-protocol-specification (see FDT Protocol Annex specifications for protocol-specific definitions).

When NetworkDataInfo is available from all Slave DTMs, the master configuration tool can commission the fieldbus (see Figure 173). For that purpose, it uses protocol-specific master configuration information from each network participant and calculates the bus parameters of the corresponding master device.

The master configuration can be provided by the DTM (Figure 173) representing the bus master hardware or by a bus master-specific Frame Application.



Used methods:

INetworkData.GetNetworkDataInfo()

INetworkData.SetNetworkData()

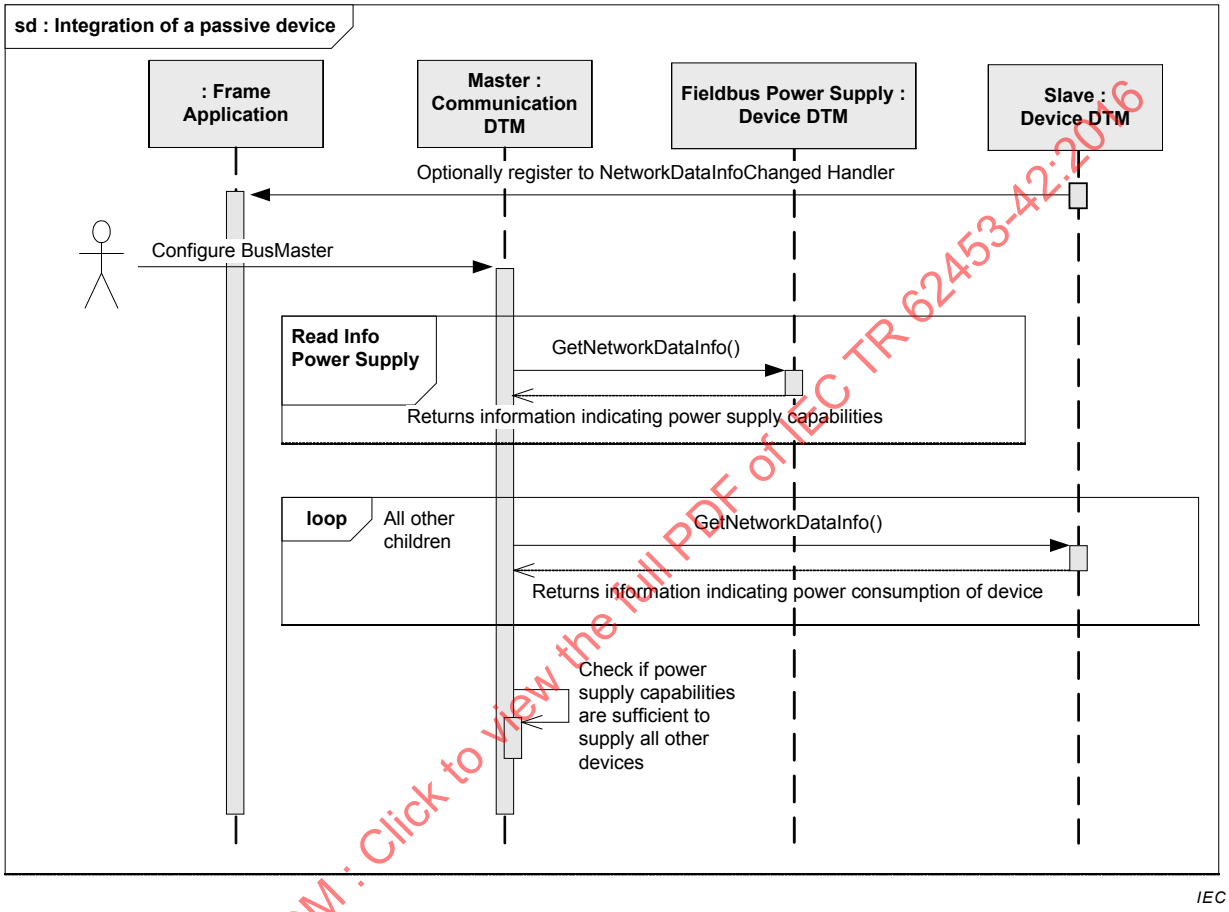
Event IChildDtmEvents.NetworkDataInfoChanged()

Figure 173 – Configuration of a fieldbus master

The transfer of the network information to the network (master device and/or field devices) is protocol-specific or product-specific. For description of protocol-specific rules please refer to the respective protocol annex.

8.11.2 Integration of a passive device

This section shows the sequence when integrating information for a passive device as part of network configuration (see Figure 174).



Used methods:

INetworkData.GetNetworkDataInfo()

Figure 174 – Integration of a passive device

After retrieving the NetworkDataInfo from the Device DTM for the fieldbus power supply and for the field devices, it is possible to compare the power consumption of the field devices with the power provided by the fieldbus power supply. If the consumption exceeds the provided power, the user should be informed.

8.12 Using IO information

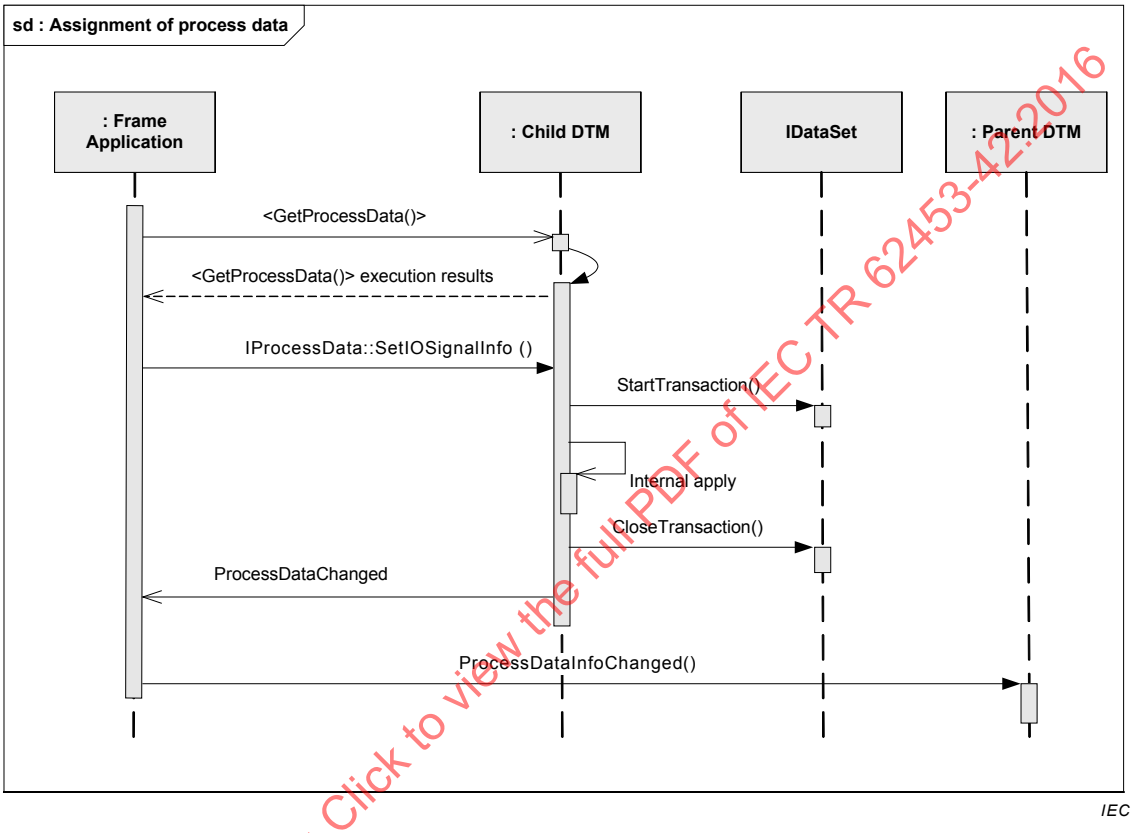
8.12.1 Assignment of symbolic name to process data

Figure 175 shows an example workflow of how a PLC Tool Frame Application assigns an IO Signal defined by IProcessData to a variable used for PLC programming.

NOTE The same mechanism is used for assignment of variables in DCS tools. This process may be referred to as “DCS channel assignment”.

The Frame Application first fetches a list of available process data (IO signals) from the DTM. It can then offer the user to assign a symbolic name to each of the IO signals contained in the list of process data. (See IEC 61131-3:2003, 2.4.3.1, Type assignment)

The symbolic variable name defined in the PLC program is stored in the property "FrameApplicationTag" of IOSignalInfo. If an IO Signal is used by the Frame Application (in a PLC program or otherwise), then this shall also be indicated by the property "IsLocked" of IOSignalInfo. Setting of the FrameApplicationTag and IsLocked is done using the method SetIOSignalInfo().



Used methods:

IProcessData.BeginGetProcessData()

IProcessData.EndGetProcessData()

IProcessData.SetIOSignalInfo()

IDataset.StartTransaction()

IDataset.CloseTransaction()

Event IProcessData.ProcessDataChanged()

Event IChildDtmEvents.ProcessDataInfoChanged()

Figure 175 – Assignment of process data

A Frame Application shall set only the FrameApplicationTag for IO signals provided by a DTM directly using IProcessData. If a DTM provides IOSignals for Child DTMs (see 4.4.4) then the Frame Application shall set the respective properties at the Child DTMs, but not at the Parent DTMs.

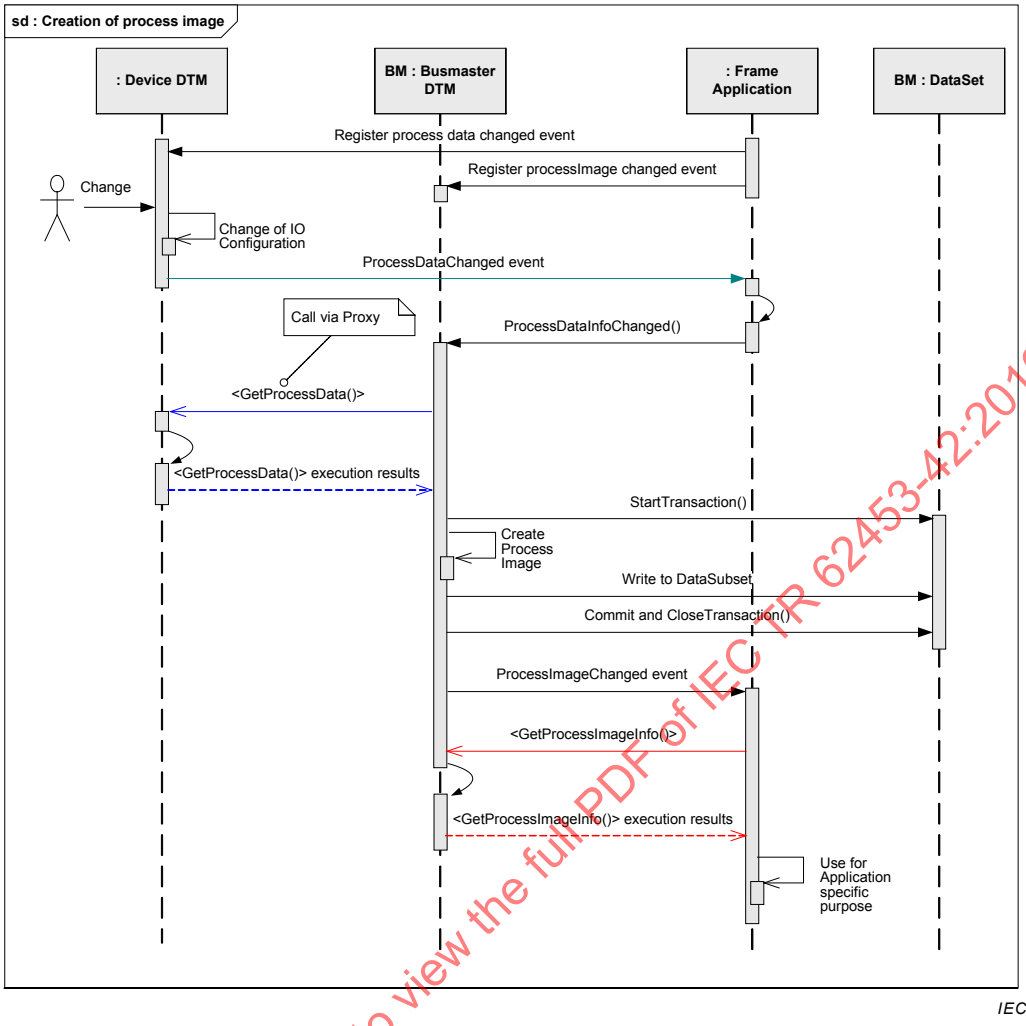
Alternatively the interface IProcessImage can be used if it is provided by the corresponding Parent DTM (see 8.12.4) to change IOSignals properties for Child DTMs.

Note that assignment of process data to a PLC variable using the interface IProcessData is protocol-specific. Protocol independent assignment can be done using interface IProcessImage.

8.12.2 Creation of Process Image

This sequence shows the creation and publishing of the process image by a DTM representing a busmaster (see Figure 176). Note that this sequence diagram shows no validation of changes. Validation is described in 8.12.3.

If the user changes the IO Configuration e.g. on a DTM-specific interface of a Device DTM the Frame Application receives a notification about this change. The notification is then forwarded to the Busmaster DTM. Since the notification contains the IDs of accessible data which is changed, the Busmaster DTM can examine the changes. Depending on the kind of changes the Busmaster DTM might fetch the process data of the Device DTM and cache this information.



Used methods:

IProcessData.BeginGetProcessData()

IProcessData.EndGetProcessData()

Event IProcessData.ProcessDataChanged()

IProcessImage.BeginGetProcessImageInfo()

IProcessImage.EndGetProcessImageInfo()

IProcessImage.EndGetProcessImageInfo()

Event IChildDtmEvents.ProcessDataInfoChanged()

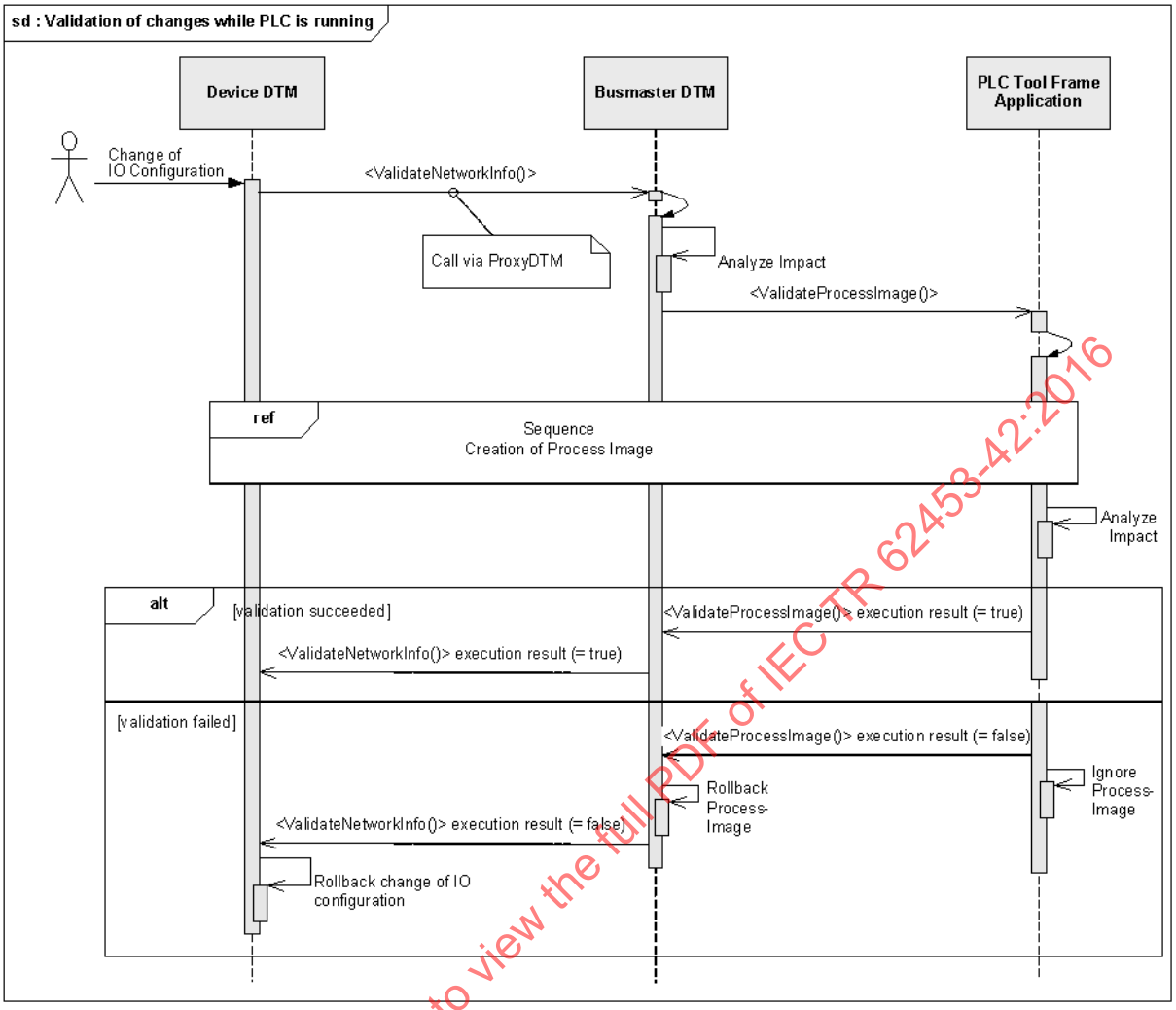
IDataset.StartTransaction()

IDataset.CloseTransaction()

Figure 176 – Creation of process image

8.12.3 Validation of changes in process image while PLC is running

The sequence diagram shown in Figure 177 shows the validations which can be done in case a PLC tool Frame Application supports changes of the configuration while the PLC is running.



IEC

Used methods:

INetworkInfoValidation.BeginValidateNetworkInfo()

INetworkInfoValidation.EndValidateNetworkInfo()

IProcessImageValidation.BeginValidateProcessImage()

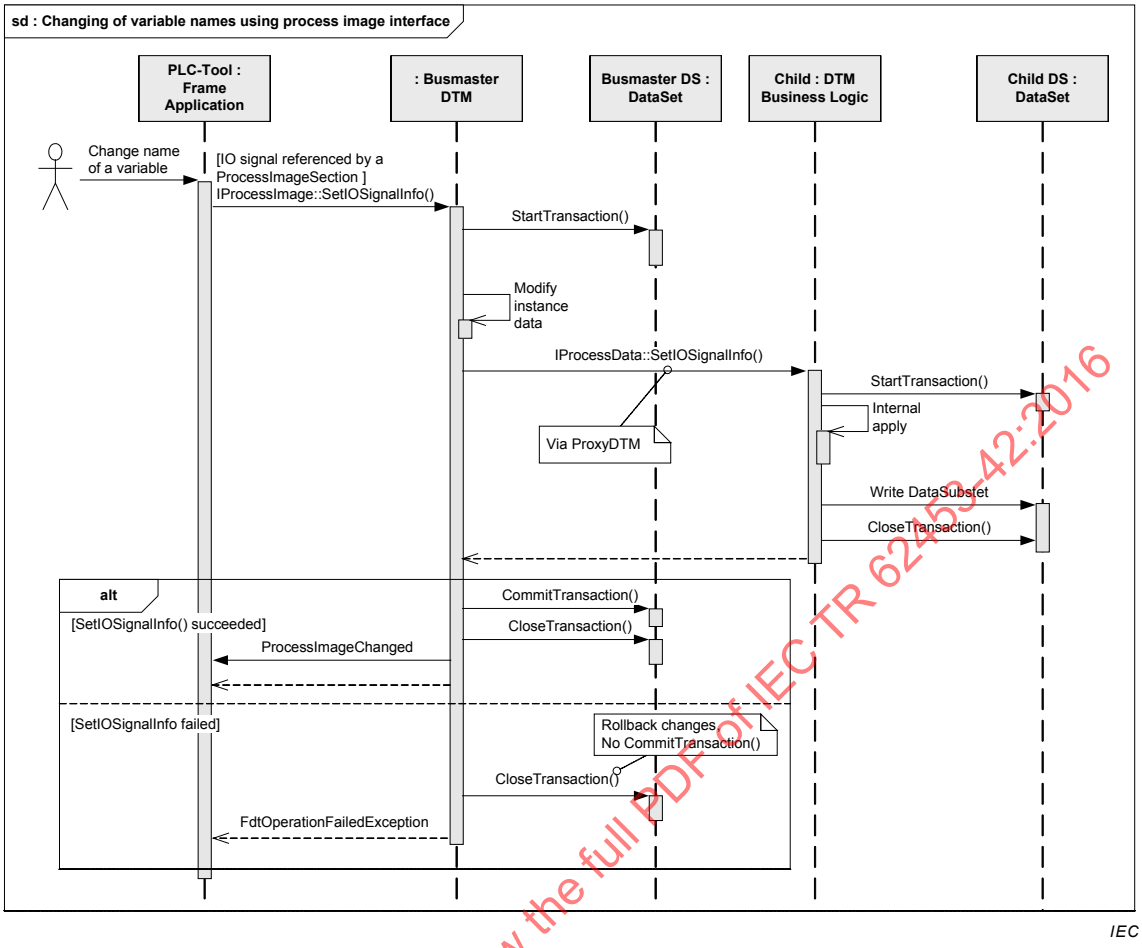
IProcessImageValidation.EndValidateProcessImage()

Figure 177 – Validation of changes while PLC is running

8.12.4 Changing of variable names using process image interface

Figure 178 shows how a PLC Tool Frame Application can change the names of variables using the Process Image interface.

The DTM shall also forward the call to corresponding Child DTMs by calling SetIOSignalInfo on the Process Data interface of the Child DTM.



Used methods:

IProcessImage.SetIOSignalInfo()

IProcessData.SetIOSignalInfo()

IDataset.StartTransaction()

IDataset.CommitTransaction()

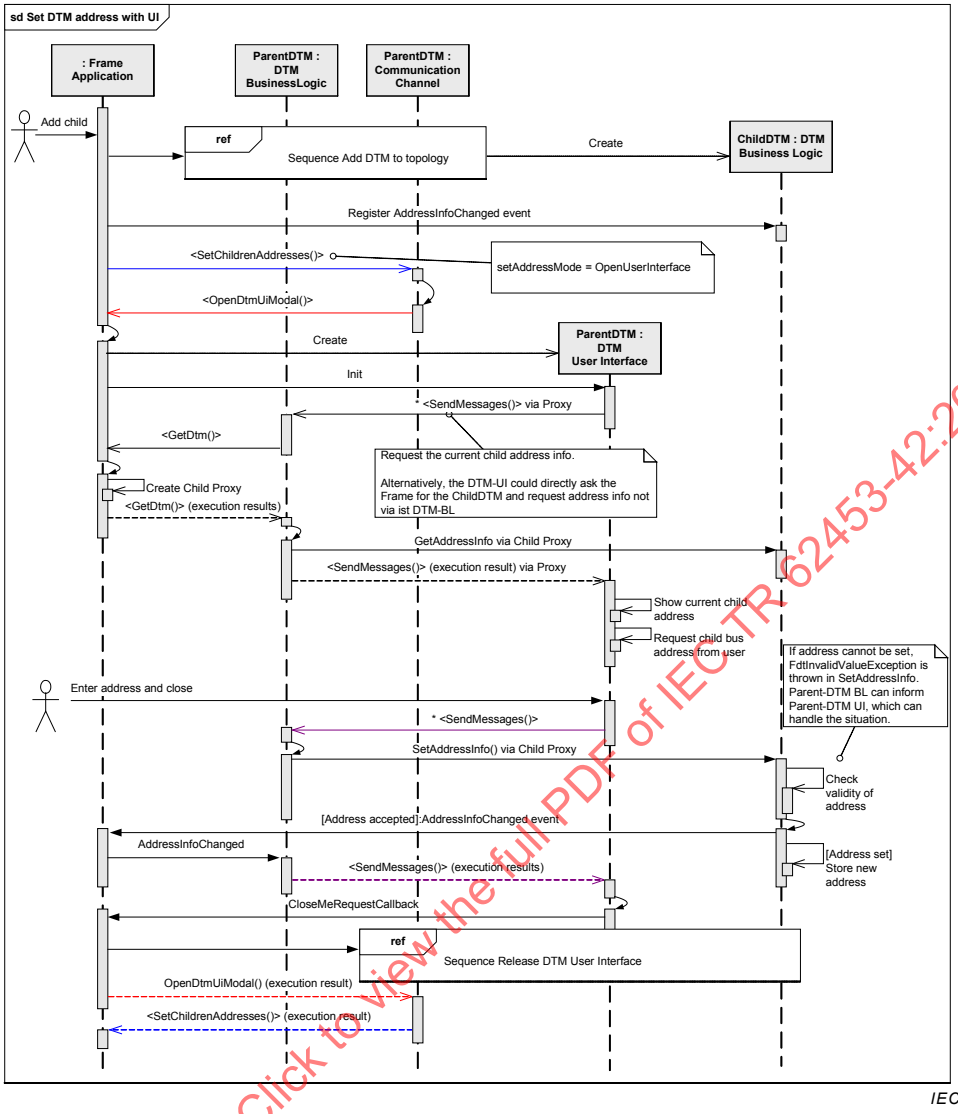
IDataset.CloseTransaction()

Figure 178 – Changing of variable names using process image interface

8.13 Managing addresses

8.13.1 Set DTM address with user interface

In this scenario the Frame Application requests setting child device addresses at the parent Communication Channel (e.g.: bus master DTMs). This sequence is started (see Figure 179) for example when a new DTM is added to the topology. A similar sequence can be applied if a Frame Application offers changing the address of a DTM manually.



Used methods:

ISubTopology.BeginSetChildrenAddresses() / ISubTopology.EndSetChildrenAddresses()

IFrameUi.BeginOpenDtmUiModal() / IFrameUi.EndOpenDtmUiModal()

IDtmUiFunction.BeginInit() / IDtmUiFunction.EndInit()

IDtmUiMessaging.BeginSendMessages() / IDtmUiMessaging.EndSendMessages

CloseMeRequestHandler

ITopology.BeginGetDtm() / ITopology.EndGetDtm()

INetworkData.GetAddressInfo() / INetworkData.SetAddressInfo()

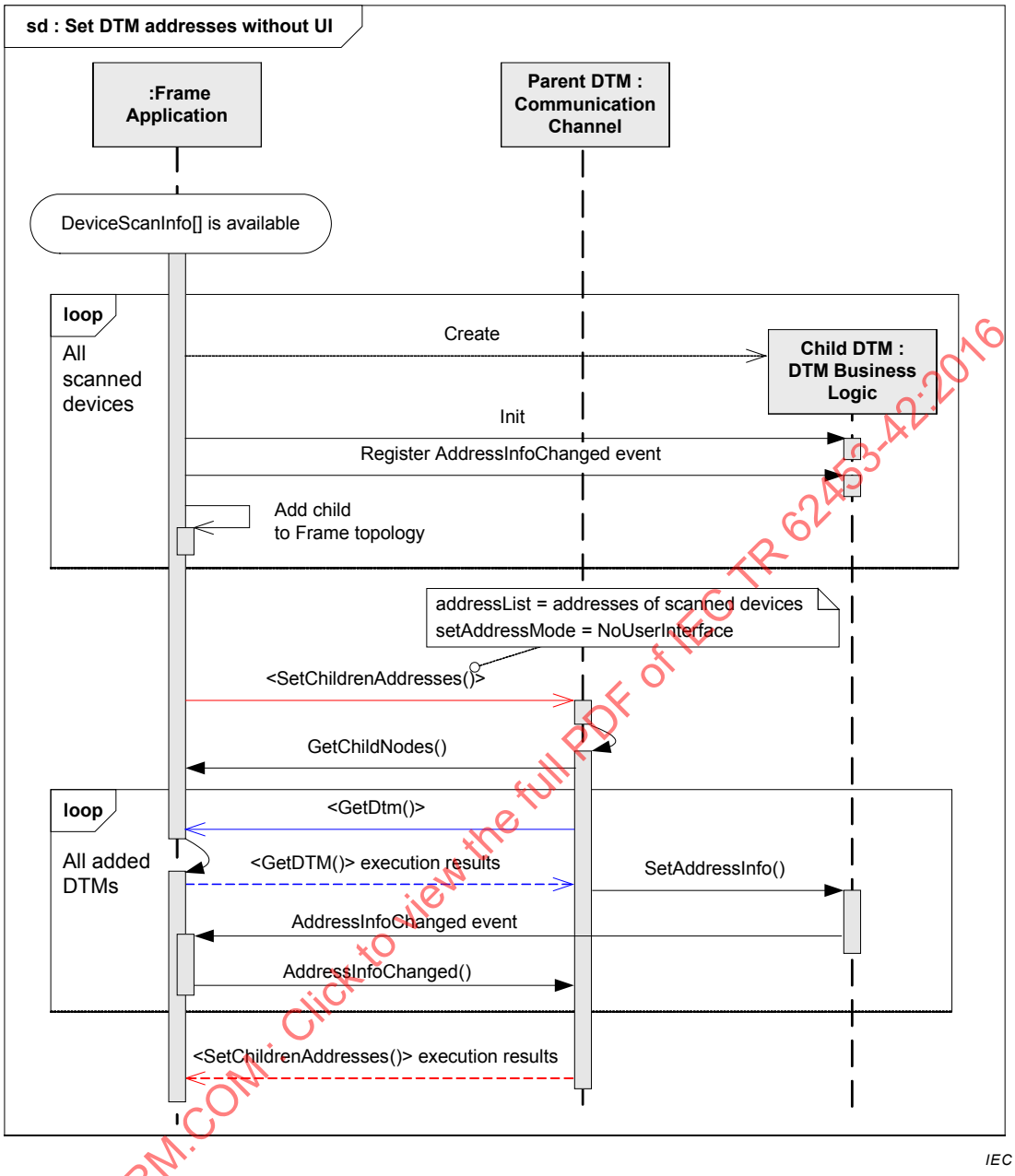
Event INetworkData.AddressInfoChanged()

Event IChildDtmEvents.AddressInfoChanged()

Figure 179 – Set DTM address with UI

8.13.2 Set DTM addresses without user interface

The following example shows the sequence of setting Child DTM addresses after scanning and DTM assignment. The Frame Application requests at a Communication Channel to set a number of known device addresses at Child DTMs (see Figure 180).



Used methods:

ISubTopology.BeginSetChildrenAddresses() / ISubTopology.EndSetChildrenAddresses()

ITopology.GetChildNodes()

ITopology.BeginGetDtm() / ITopology.EndGetDtm()

INetworkData.SetAddressInfo()

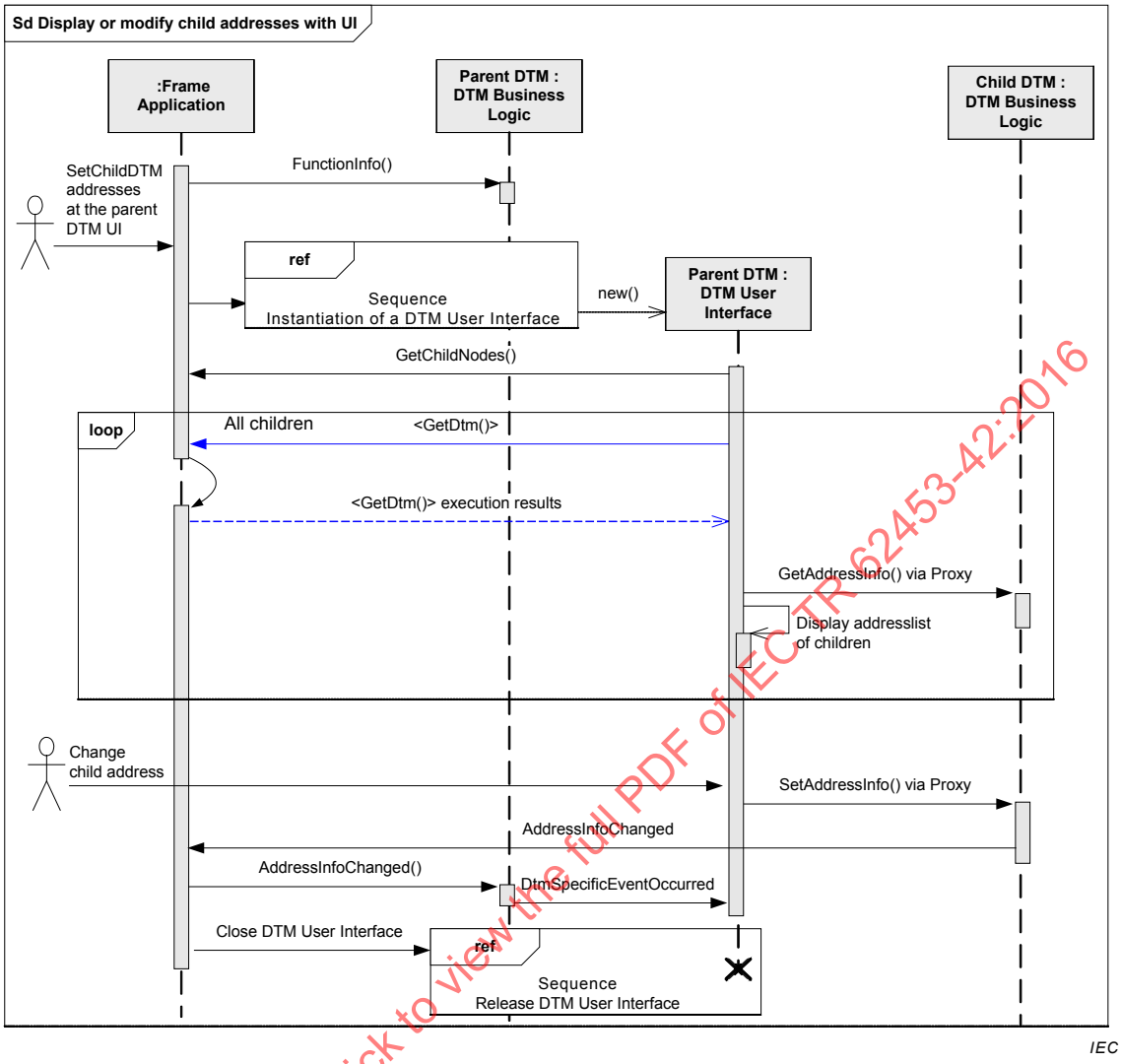
Event INetworkData.AddressInfoChanged()

Event IChildDtmEvents.AddressInfoChanged()

Figure 180 – Set DTM addresses without UI

8.13.3 Display or modify addresses of all Child DTMs with user interface

In this scenario Frame Application requests to display or modify all Child DTM addresses at a Parent DTM. This sequence (see Figure 181) for example is started when a user selects the corresponding menu entry in context of a Communication DTM or a Gateway DTM.



Used methods:

IFunction.Functioninfo

ITopology.GetChildNodes()

ITopology.BeginGetDtm() / ITopology.EndGetDtm()

INetworkData.GetAddressInfo()

INetworkData.SetAddressInfo()

Event INetworkData.AddressInfoChanged()

Event IChildDtmEvents.AddressInfoChanged()

Event IDtmUiMessaging.DtmSpecificEventOccured()

Figure 181 – Display or modify child addresses with UI

8.14 Device-initiated data transfer

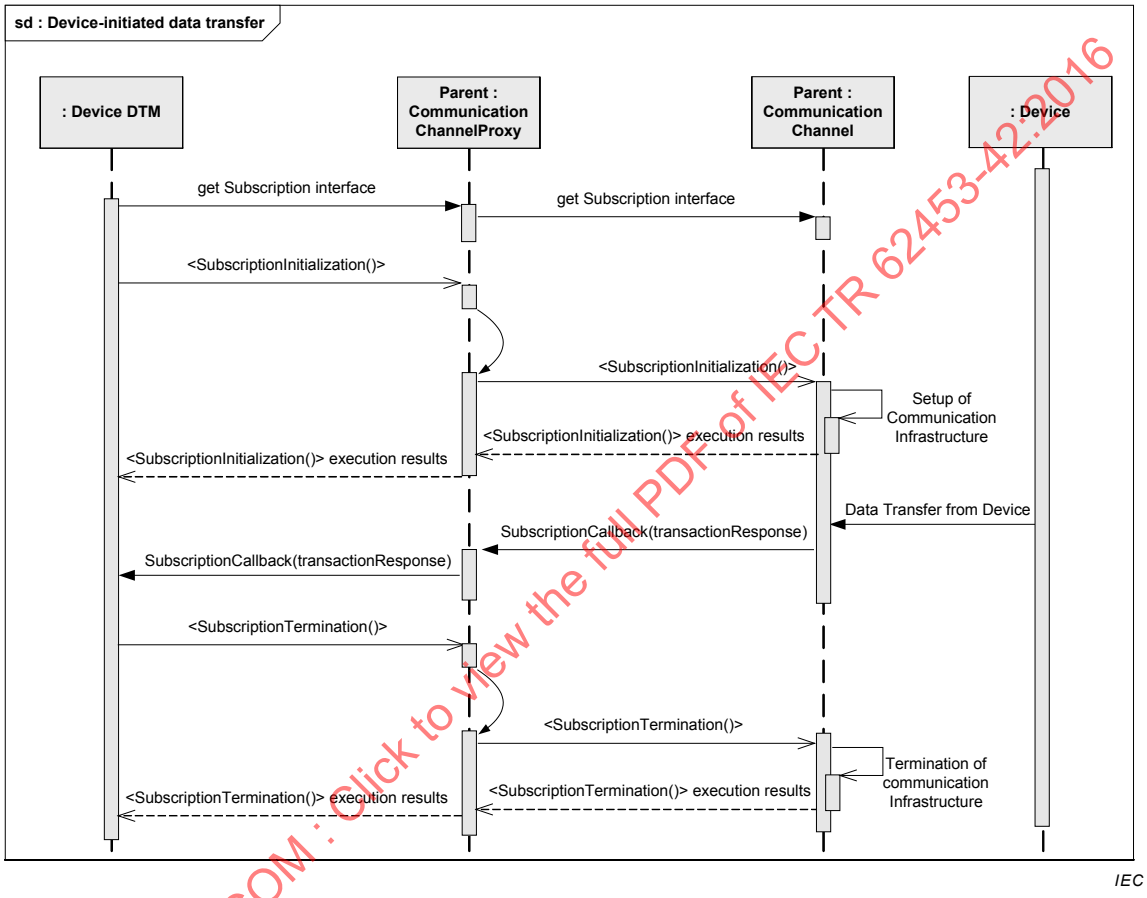
Some protocols support data transfer services which are initiated by the device and not by the DTM. A Communication Channel supports this by providing the ISubscription interface. For an example of device initiated data transfer see Figure 182.

A Child DTM requests the ISubscription instance from the Communication Channel of the Parent DTM to access the subscription services.

The infrastructure (e.g. filter, service queue) for such services is initiated by a protocol-specific request of the DTM to initialize the subscription.

The device initiated data transfer is transported by multiple invocations of the SubscriptionCallback() of the DTM with protocol-specific communication responses as arguments.

The infrastructure for these services is terminated by a protocol-specific request of the DTM to terminate the subscription.



Used methods:

ICommunicationChannelProxy.Subscription()

ICommunicationChannel.Subscription()

ISubscription.BeginSubscriptionInitialization() / ISubscription.EndSubscriptionInitialization()

Fdt.Communication.SubscriptionCallback()

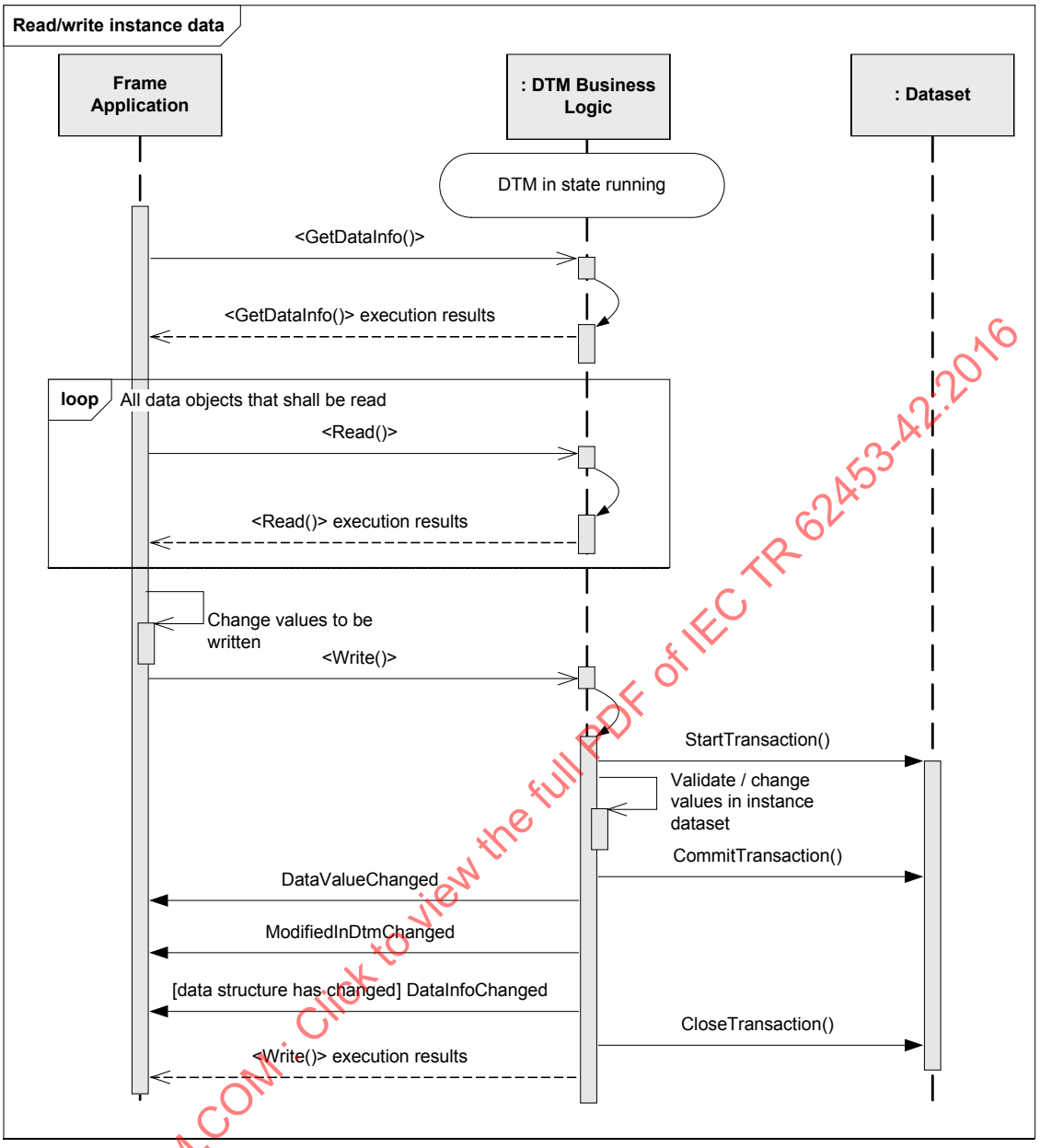
ISubscription.BeginSubscriptionTermination() / ISubscription.EndSubscriptionTermination()

Figure 182 – Device-initiated data transfer

8.15 Reading and writing data

8.15.1 Read/write instance data

The following sequence diagram (Figure 183) shows how instance data is read from / written to the instance dataset using IInstanceData interface.



Used methods:

IInstanceData.GetDataInfo()

IDataset.StartTransaction() / IDataset.CommitTransaction() / IDataset.CloseTransaction()

IDataset.TransactionStarted / IDataset.TransactionCommitted /IDataset.TransactionClosed

IInstanceData.BeginRead() / IInstanceData.EndRead()

IInstanceData.BeginWrite() / IInstanceData.EndWrite()

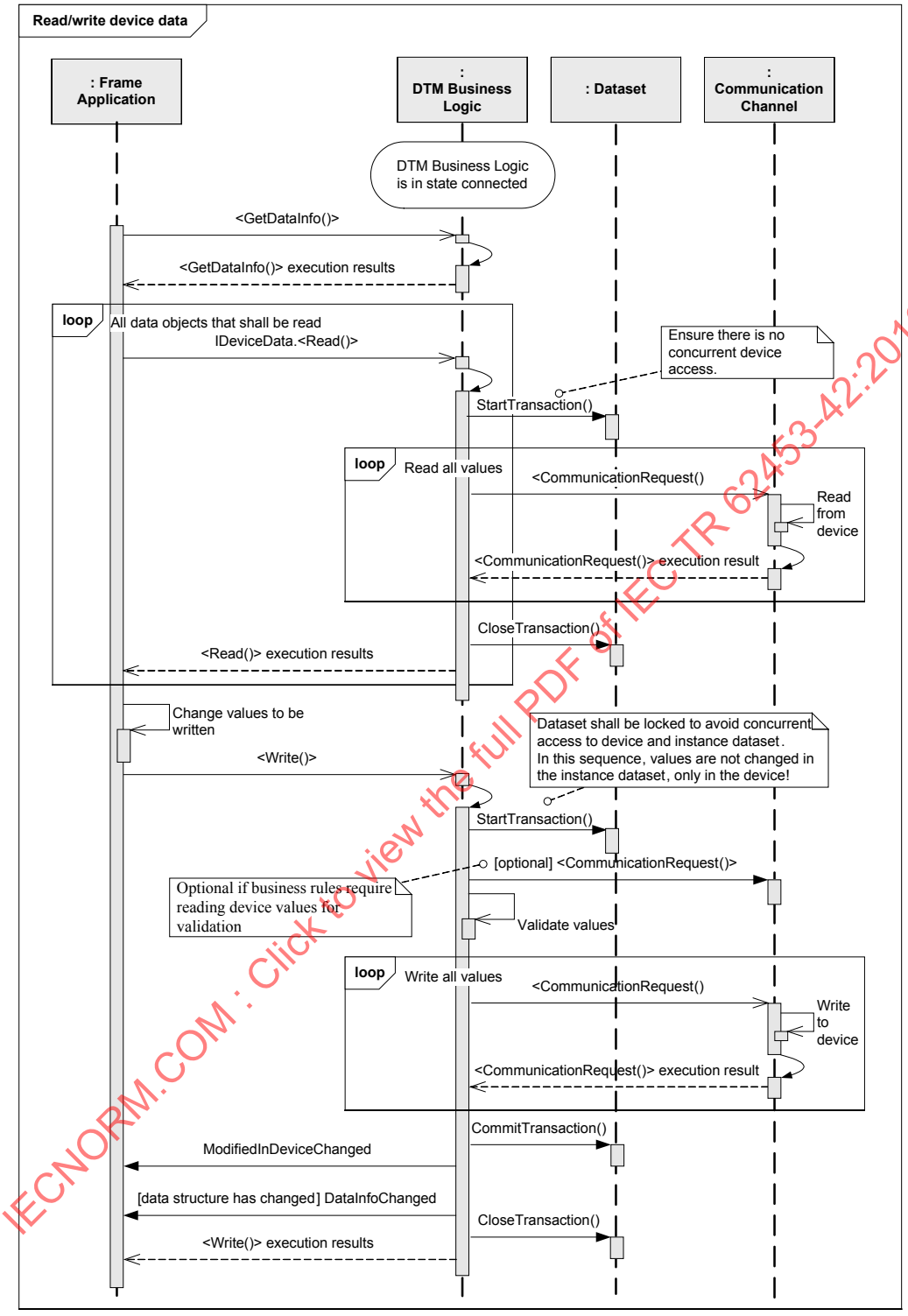
Event IInstanceData.DataValueChanged()

Event IInstanceData.DataInfoChanged()

Figure 183 – Read/write instance data

8.15.2 Read/write device data

The following sequence diagram (Figure 184) shows how device data is read from / written to the device using IDeviceData interface.



Used methods:

IInstanceData.GetDataInfo()

IDataset.StartTransaction() / IDataset.CommitTransaction() / IDataset.CloseTransaction()

IDataset.TransactionStarted / IDataset.TransactionCommitted /IDataset.TransactionClosed

IDeviceData.BeginRead() / IDeviceData.EndRead()

IDeviceData.BeginWrite() / IDeviceData.EndWrite()

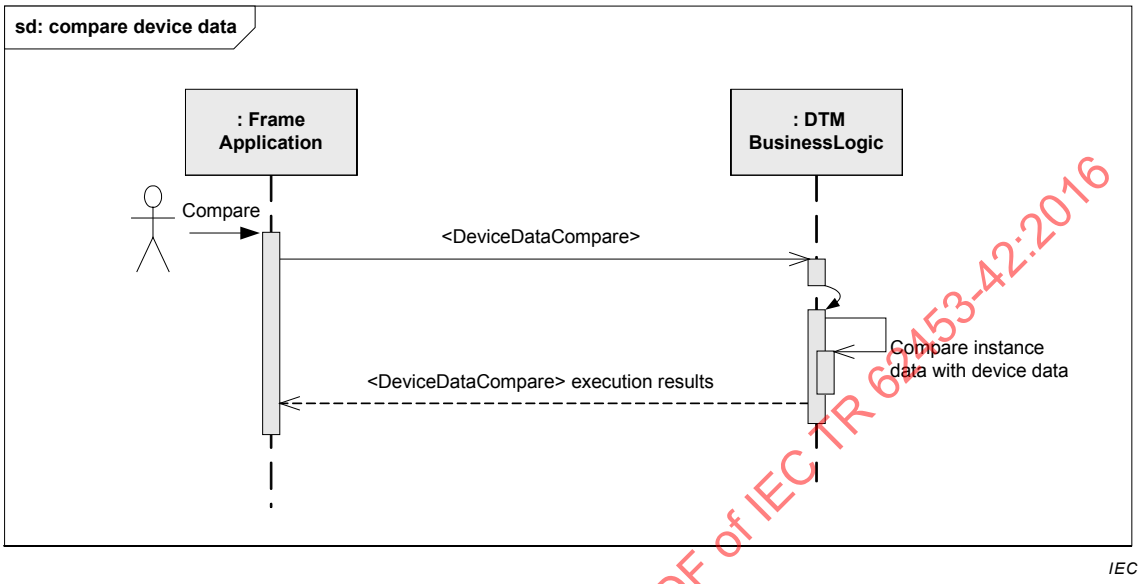
Event IDeviceData.ModifiedInDeviceChanged()

Figure 184 – Read/write device data

8.16 Comparing data

8.16.1 Comparing device dataset and instance dataset

In order to compare the data of a DTM instance with the data of the respective device, the action <DeviceDataCompare()> (defined in 5.13.2) is executed (see Figure 185).



Used methods:

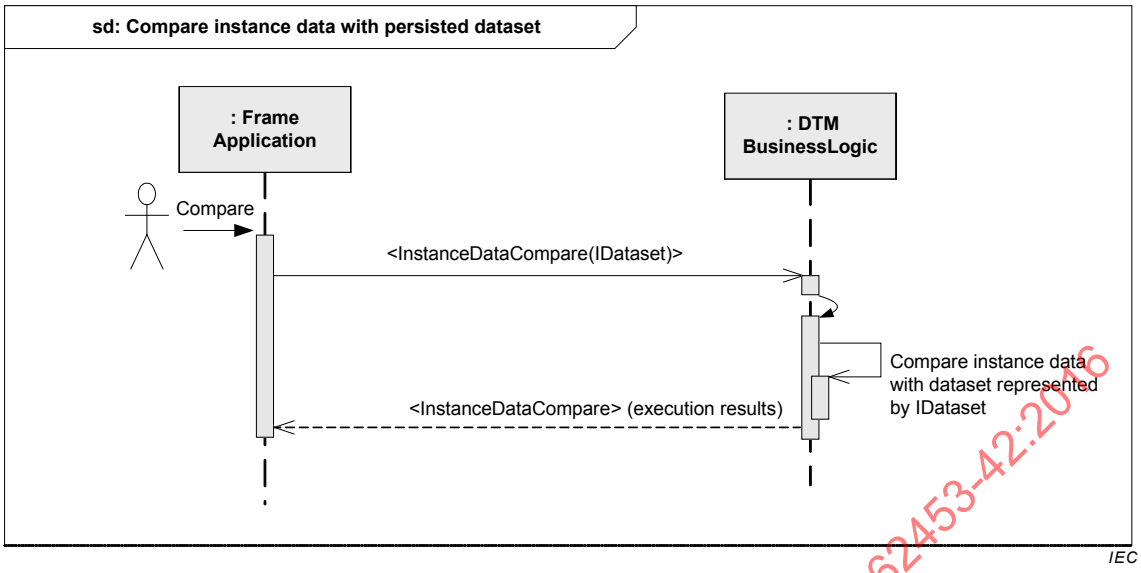
IComparison.BeginDeviceDataCompare() / IComparison.EndDeviceDataCompare()

Figure 185 – Comparing device dataset and instance dataset

The comparison is executed for the data in the DTM dataset and the data that can be uploaded from the device. The comparison should include all identification, configuration, and parameterization data. Dynamic data and status data should not be included in the comparison.

8.16.2 Comparing different instance datasets

In order to compare the data of one DTM instance with the data of a different DTM instance, the action <InstanceDataCompare()> is executed (see Figure 186).



Used methods:

IComparison.BeginInstanceDataCompare() / IComparison.EndInstanceDataCompare()

Figure 186 – Compare instance data with persisted dataset

8.17 Reassigning a different DtmDeviceType at a device node

8.17.1 General

Over the lifetime of the FDT Frame Application project it can be necessary to reassign the DtmDeviceType of a device node to a different DtmDeviceType (see Figure 189). Reasons for the reassignment may be:

1) An engineer reassigns a DtmDeviceType during offline planning of the FDT topology.
2) A DTM is available which supports the same device type better than the currently assigned DTM (for instance instead of a Generic DTM, a specific DTM can be assigned).

NOTE The DTM of the DtmDeviceType may be updated or upgraded. If the device of the device node is unchanged, a reassignment is not required due to FDT rules regarding DTM replacing installations for DTM Updates and DTM Upgrades (See chapter 10).

3) A physical device was or is going to be exchanged. This means, the device which is logically connected to a device node in the FDT topology will be replaced. The replacement may require a reassignment of the DtmDeviceType if the DtmDeviceType, which is currently in use, does not support the new device type or the version of the new device.

NOTE Relevant is the identification of the device firmware. A device replacement as well as a firmware update can be incompatible in respect to the DtmDeviceType.

In regard to cases 1 and 2: Do not consider scanned information from a connected device. Usually, an existing dataset cannot be migrated in these cases.

Subclause 8.17.2 describes the scenario, where a DTM detects that the device type of the connected device can be better supported by a different DtmDeviceType.

Subclauses 8.17.3 and 8.17.4 show sequence diagrams explaining the steps in relation to use case 3(device exchange).

Whenever a DtmDeviceType is reassigned, two post conditions need to be considered:

- Device support:

The new DtmDeviceType shall be able to operate the device connected to the device node (refer to lifecycle concept regarding evaluation of device support in advance).

- Dataset support:

  Dependent on the dataset format support of old and new DtmDeviceType, the existing dataset could be migrated to the new DtmDeviceType. The dataset migration is not possible in all cases. If a migration is not possible, the existing dataset cannot be used by the new DtmDeviceType. Frame Applications are responsible to inform the user about this and propose following action: An upload should be performed with the new DtmDeviceType in order to synchronize and store the device data with the project data.

NOTE In general all descriptions in this chapter do not only apply to DtmDeviceTypes, but also apply to the two other DtmTypes: DtmModuleTypes and DtmBlockTypes.

8.17.2 DTM detects a change in connected device type

This subclause describes the scenario where a DTM detects that the device type of the connected device can be better supported by a different DtmDeviceType.

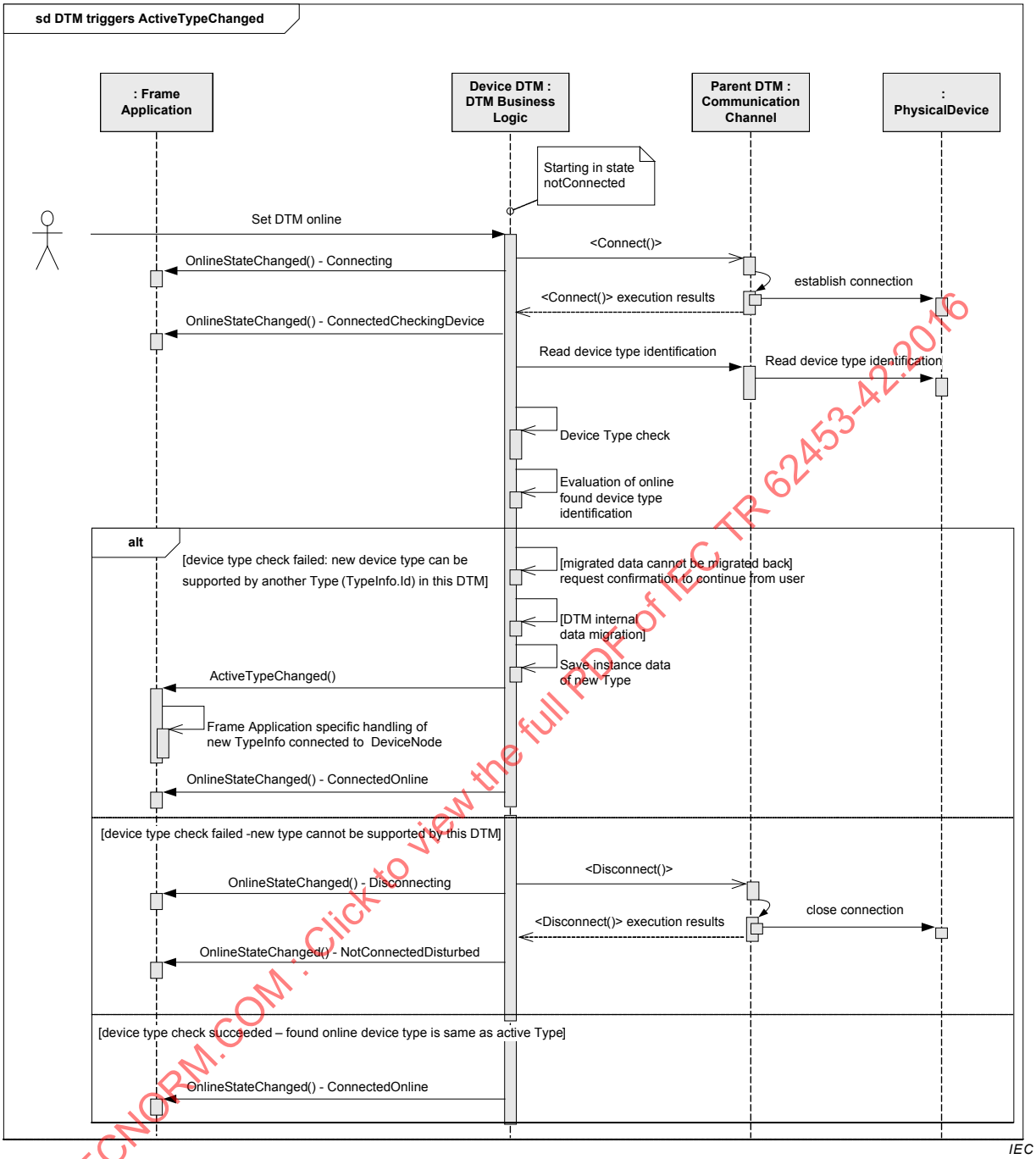
3 possible scenarios are shown in the sequence diagram in Figure 187:

a) The connected device type can be better supported by a different DTM Type. In this case the DTM internally activates another DTM Type and informs the Frame Application with ActiveTypeChanged about the change.

b) The connected device type cannot be supported by the DTM

c) The unchanged connected device type: DtmDeviceType is not changed.

Scenario a) may occur if a DTM connects to the device again, after the device has been replaced by a compatible device type. Also Scenario a) may occur when the DTM was assigned with a generic DtmDeviceType to the device (e.g. during offline engineering) and detects that it can provide better support for the connected device with a different DtmDeviceType.



Used methods:

Event IDtm.OnlineStateChanged()

ICommunication.BeginConnect() / ICommunication.EndConnect()

ICommunication.BeginCommunicationRequest()

ICommunication.EndCommunicationRequest()

ICommunication.BeginDisconnect() / ICommunication.EndDisconnect()

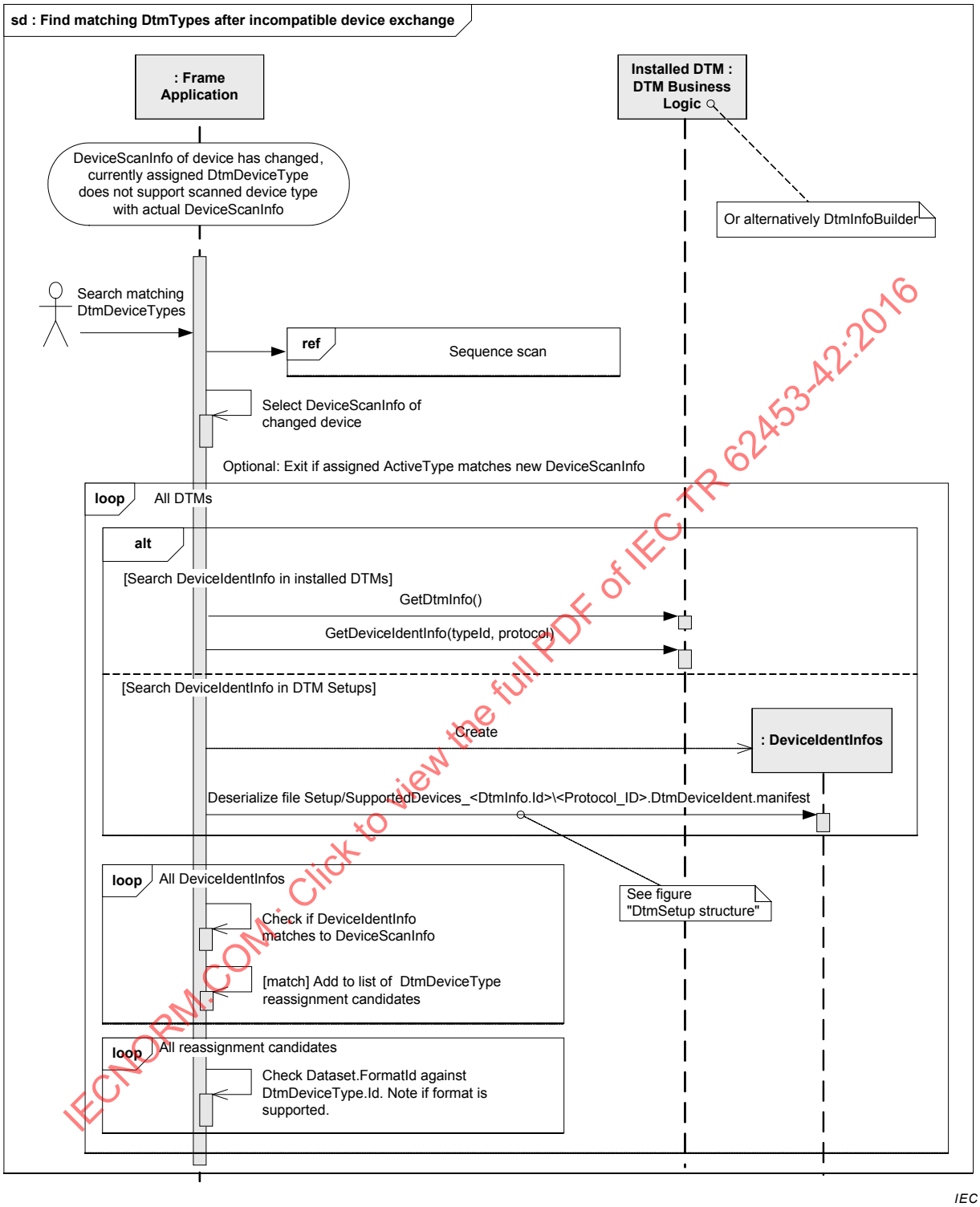
Event IDtm.ActiveTypeChanged()

Figure 187 – DTM triggers ActiveTypeChanged event

8.17.3 Search matching DtmDeviceTypes after incompatible device exchange

After a device exchange, a Frame Application should support the verification of the DtmDeviceType currently assigned to a device node in the FDT topology. In addition to the identification of the device types supported by installed DTMs, FDT provides a concept to explore DTM setups and the included DTMs before the DTMs are installed (see chapter 10). This can be used to find out if there are DTMs available (uninstalled DTMs), which include DtmDeviceTypes to support a scanned device.

The sequence diagram shown in Figure 188 shows how a list of matching DtmDeviceTypes in installed DTMs and DTM setups can be determined by a Frame Application.



Used methods:

IDtmInformation.GetDtmInfo()

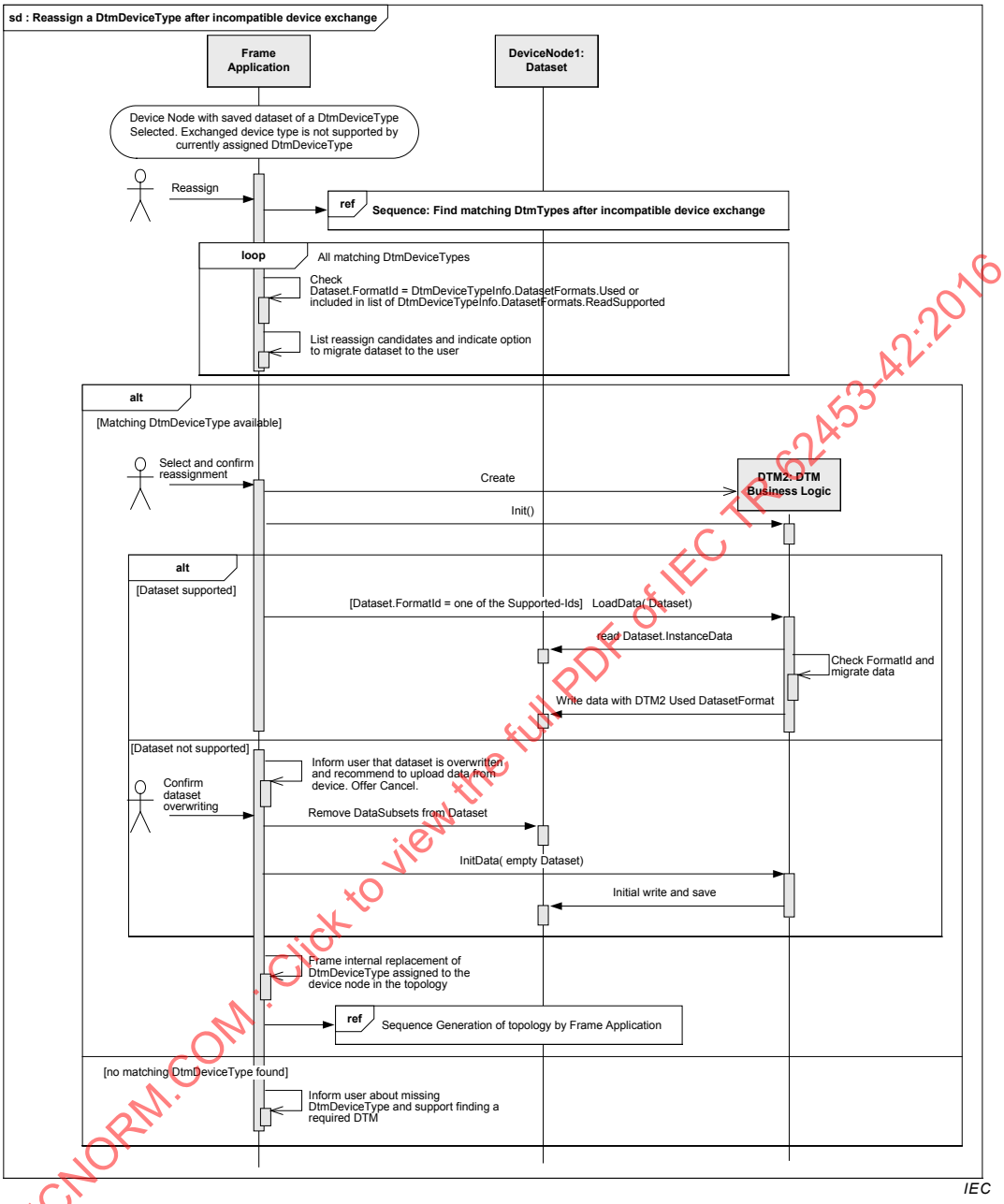
IDtmInformation.GetDeviceIdentInfo()

Figure 188 – Find matching DtmDeviceTypes after incompatible device exchange

8.17.4 Reassign DtmDeviceType after incompatible device exchange

The sequence diagram shown in Figure 189 shows how a Frame Application verifies the validity of a currently assigned DtmDeviceType after a device change. The sequence diagram

describes the DtmDeviceType reassignment if a better matching or newer DtmDeviceType is found.



Used methods:

IDtm.Init()

IDtm.InitData()

IDtm.LoadData()

IDataSubset.ReadData()

IDataSubset.WriteData()

IDtmInformation.GetDtmInfo()

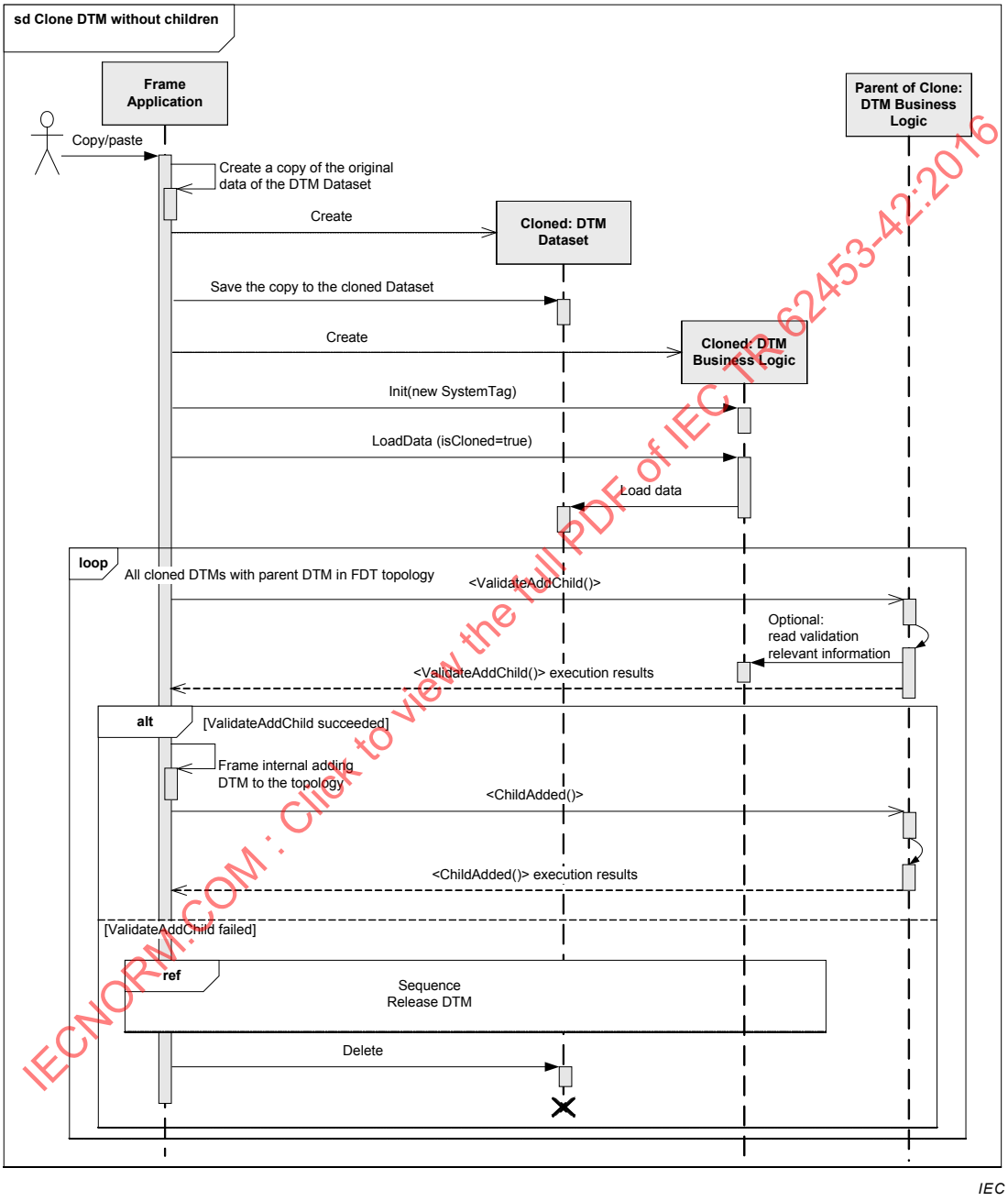
IDtmInformation.GetDeviceIdentInfo()

Figure 189 – Reassign a DtmDeviceType after incompatible device exchange

8.18 Copying part of FDT Topology

8.18.1 Cloning of a single DTM without Children

A Frame Application might provide functionality to copy and paste a DTM which has no children to the same parent or to another one. Figure 190 shows the workflow for this functionality.



Used methods:

IDtm.LoadData()

IDtm.Init()

ISubTopology.BeginValidateAddChild()

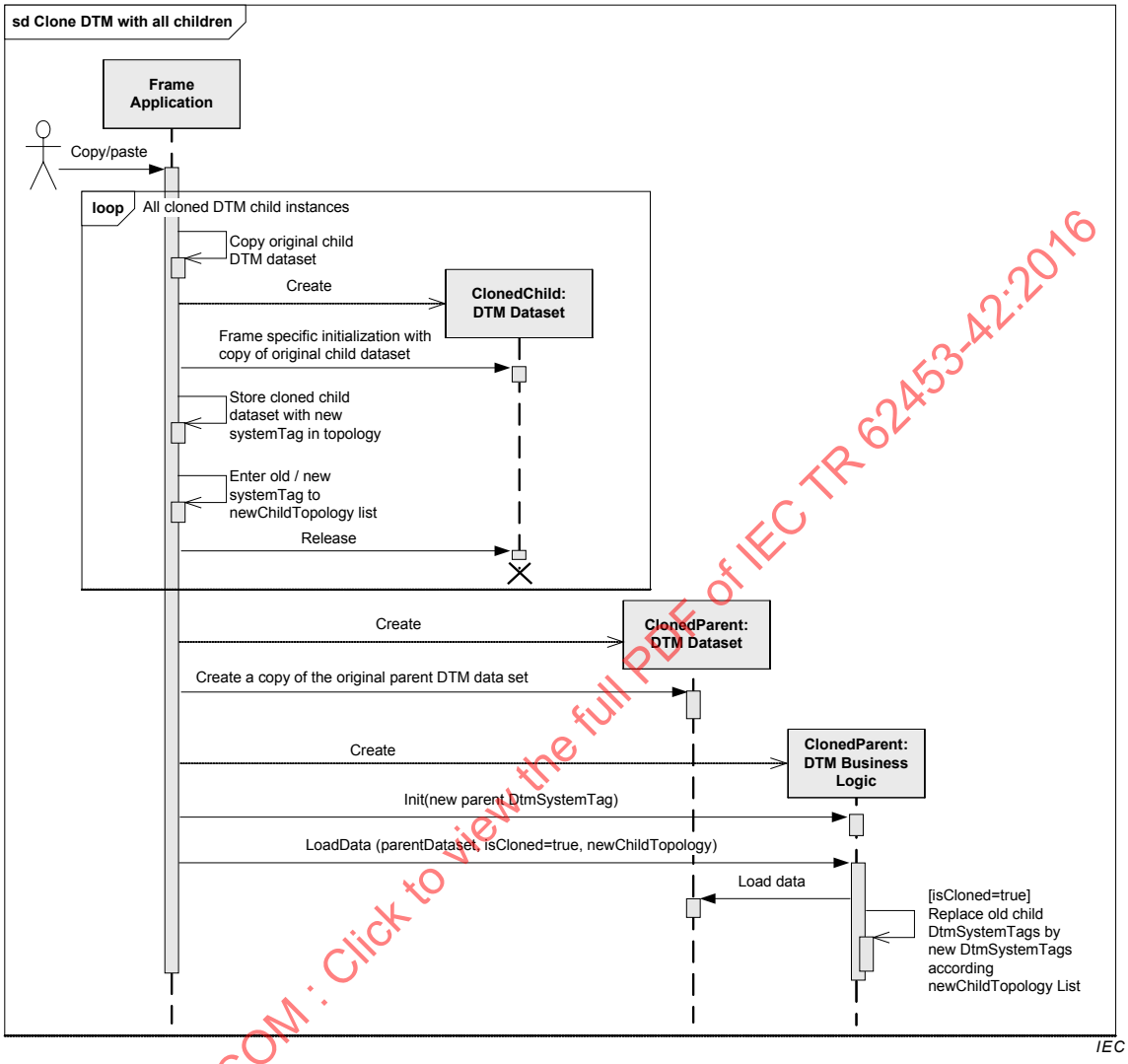
ISubTopology.EndValidateAddChild()

BeginChildAdded() / EndChildAdded()

Figure 190 – Clone DTM without children

8.18.2 Cloning of a DTM with all its Children

A Frame Application might provide functionality to copy and paste a DTM with all its children to the same parent or to another parent. Figure 191 shows the workflow of this functionality.



Used methods:

IDtm.LoadData()

IDtm.Init()

Figure 191 – Clone DTM with all children

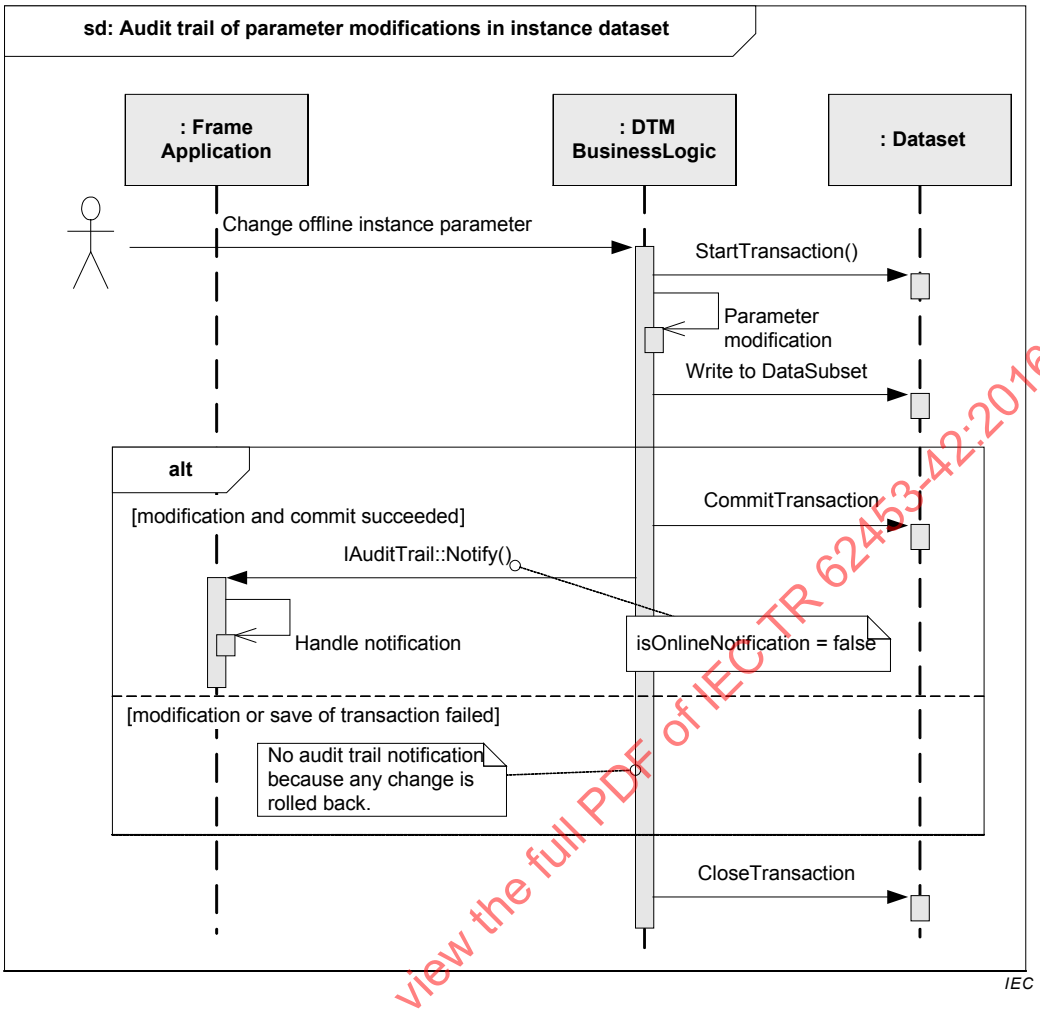
8.19 Sequences for audit trail

8.19.1 General

This section shows how the audit trail concept (described in 4.15) is implemented.

8.19.2 Audit trail of parameter modifications in instance dataset

Figure 192 shows how changes in the instance dataset are traced.



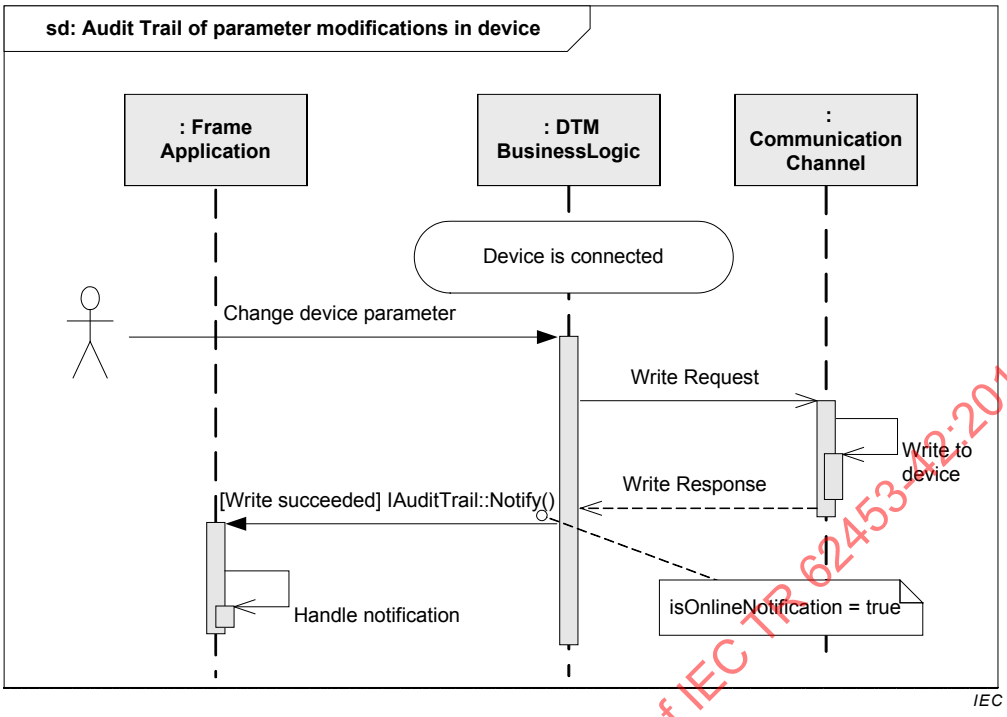
Used methods:

IAuditTrail.Notify()

Figure 192 – Audit trail of parameter modifications in instance dataset

8.19.3 Audit trail of parameter modifications in device dataset

Figure 193 shows how changes in the device data are tracked.



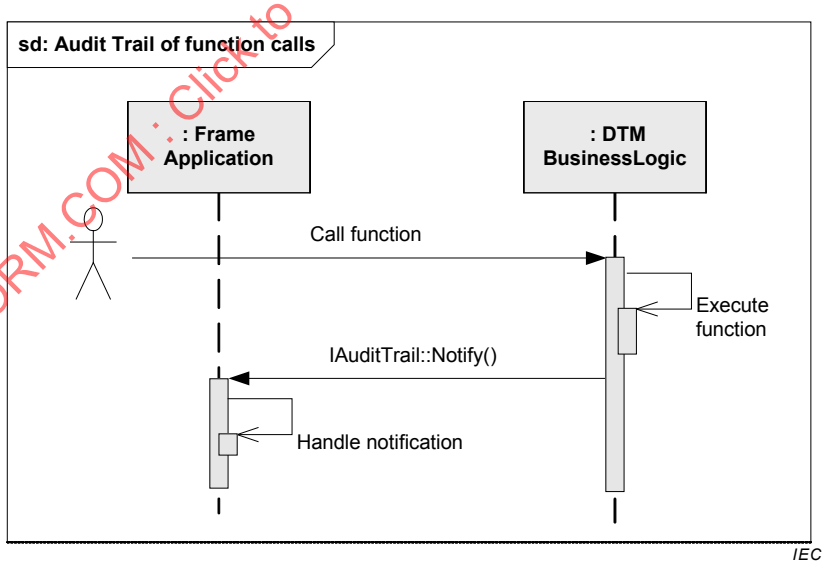
Used methods:

IAuditTrail.Notify()

Figure 193 – Audit trail of parameter modifications in device

8.19.4 Audit trail of function calls

Figure 194 shows how function calls are tracked.



Used methods:

IAuditTrail.Notify()

Figure 194 – Audit trail of function calls

8.19.5 Audit trail of general notification

This shall only be used in case it is not a Function Notification or a Parameter Change Notification. General notifications are used by the DTM to provide audit trail notifications in the scenarios like device state information updates.

9 Installation

9.1 General

This chapter describes the installation of FDT core assemblies, FDT communication protocols and DTMs as well as the structure and rules for creating DTM setups.

9.2 Common rules

9.2.1 Predefined installation paths

This chapter defines the common installation paths where FDT core assemblies, FDT protocol assemblies and DTMs are installed and registered (see Table 42, Figure 195 and Figure 196).

Table 42 – Predefined FDT installation paths

Path Name	Value	Description
<GAC>	(OS version specific)	This is the Windows Global Assembly Cache.
<FDT_Registry>	"<CommonApplicationData>\FDT\"	Root folder for the registration of IEC 62453-42 DTMs and protocols.
<FDT_DTMs>	"<FDT_Registry>\DTMs"	Folder contains the vendor-specific subfolders with DTM manifest files that are used for DTM registration. Frame Applications search this folder for installed DTMs in order to create a device catalog.
<FDT_Protocols>	"<FDT_Registry>\Protocols"	Folder contains the communication protocol manifest files. Frame Applications search this folder for installed communication protocols.
<FDT_GUIs>	"<FDT_DTMS>\<Vendor Name>\ <DTM Name>\User Interfaces"	Folder contains the user interface manifest files. Frame Applications search this folder for installed user interfaces.
<FDT_X86>	"<CommonProgramFilesX86>\FDT"	Folder for 32-bit / Any CPU FDT components.
<FDT>	"<CommonProgramFilesX64>\FDT"	Folder for 64-bit FDT components.
<DTM_X86>	"<FDT_X86>\DTMs\"	Folder contains all 32-bit / Any CPU DTMs.
<DTM>	"<FDT>\DTMs\"	Folder contains all 64-bit CPU DTMs.
<DTMs>	32-bit / Any CPU DTM: "<DTM_X86>"	The folder name is used as short statement for "<DTM_X86> or <DTM>". The actual meaning depends on the environment supported by the DTM.
	64-bit DTM: "<DTM>	
<DTM_root>	32-bit / Any CPU DTM: "<DTM_X86>\<Vendor Name>\<DTM Name>"	The folder contains all DTM binaries (assemblies) and data files. This includes main DTM BL assembly, DTMInfoBuilder assembly, DTM UI assemblies, resource assemblies and other files. NOTE <Vendor Name> is the name of the DTM vendor.
	64-bit DTM: "<DTM>\<Vendor Name>\<DTM Name>"	

NOTE:

<CommonApplicationData> is the folder returned by the System.Environment.GetFolderPath() method for the special folder ID CommonApplicationData

<CommonProgramFilesX86> is the folder returned by the System.Environment.GetFolderPath() method for the special folder ID CommonProgramFiles (32-bit application) or CommonProgramFilesX86 (64-bit application)

<CommonProgramFilesX64> is the folder returned to a 64-bit application by the System.Environment.GetFolderPath() method for the special folder ID CommonProgramFiles. This folder is accessible from 64-bit applications / on 64-bit OS versions only (see [32])

NOTE It is a product-specific decision whether files are shared between different DTMs. The handling and version management of these files are out of scope of FDT. One possible approach could be to store these files under the folder "<DTMs>\<Vendor Name>\" according to vendor-specific needs and use relative paths to reference these files.

The FDT core assemblies (interfaces, datatypes, and exceptions) and communication protocol assemblies are installed in the Global Assembly Cache. Shared .NET assemblies may also be installed in the Global Assembly Cache. Please refer to 5.4.4 when using shared .NET assemblies.

All DTMs are registered in the predefined path
"...\<CommonApplicationData>\FDT\DTMs".

All communication protocols are registered in the predefined path
"...\<CommonApplicationData >\FDT\Protocols".

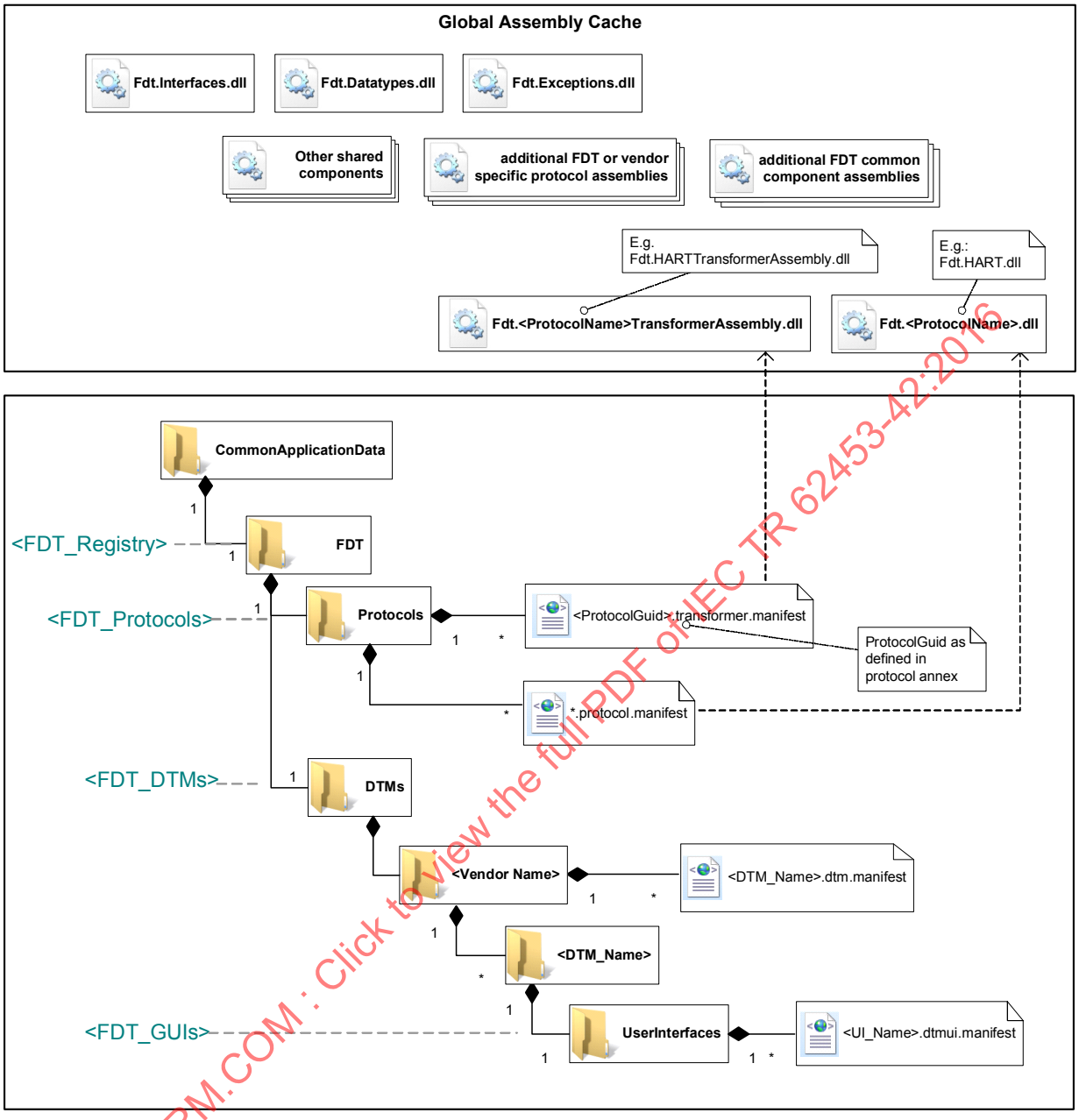


Figure 195 – GAC and FDT_Registry

On 32-bit operating systems all DTMs are installed in the predefined path "<CommonProgramFilesX86>\FDT\DTMs".

On 64-bit operating systems all DTMs built for 64-bit are installed in the predefined path "<CommonProgramFilesX64>\FDT\DTMs"

and all DTMs built for 32-bit or for 'any'-target are installed in the predefined path ""<CommonProgramFilesX86>\FDT\DTMs". (see Figure 196)

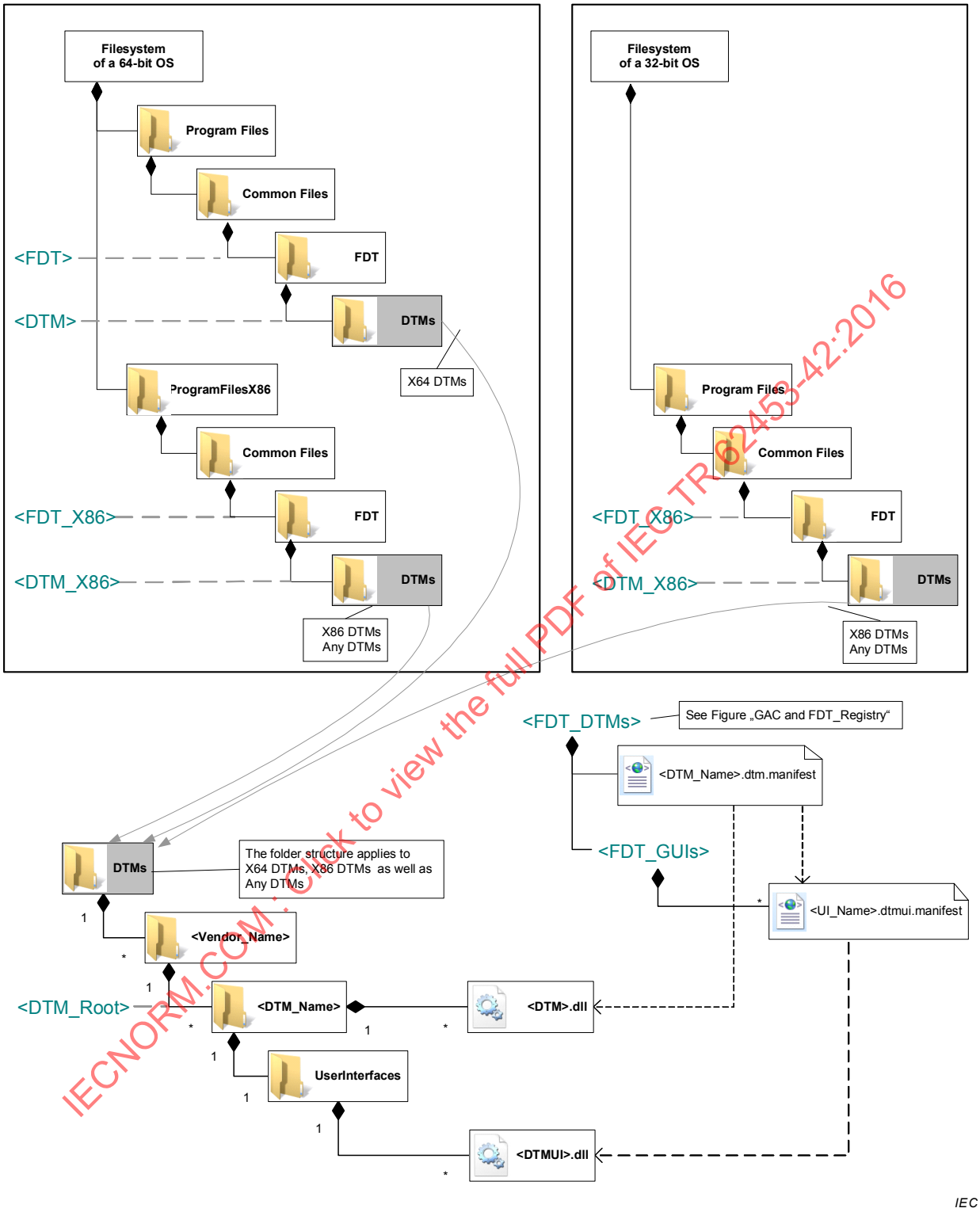


Figure 196 – Installation paths (with example DTM)

9.2.2 Manifest files

All components related to FDT provide manifest files in order to register the component in the FDT system (except for the FDT core assemblies) or to provide pre-installation information. Manifest files are XML-files, which follow a defined format. The format corresponds to .NET datatypes, which are part of the FDT core specification.

9.2.3 Paths in manifest files

All manifest files may include paths to assembly or resource files.

If some parameter in any manifest file represents a path to an assembly, icon, bitmap, documentation or data file, then it shall be relative to the component root path.

NOTE For example, if a PDF document is provided for a DTM, which is installed in "<DTMs>\Vendor1\MyDtm1": The document is located in "<DTMs>\Vendor1\MyDtm1\Documentation\help.pdf", the reference to the document is "Documentation\help.pdf". The component is the DTM BL. The component root path is "<DTMs>\Vendor1\MyDtm1".

In order to access the referenced files, the Frame Application shall add the component root path at the beginning of relative paths.

9.2.4 Common command line arguments

DTM setups and its components shall have a unified way for installing and uninstalling by using the standard Windows Installer command line. This includes support for predefined setup properties and setup command line parameters.

The following predefined setup properties shall be available (Table 43):

Table 43 – Predefined setup properties

Setup property	Value range	Default value	Description
FDT20_DTMBO	"True", "False".	True	Installs only DTM Business Logic components for selected device types.
FDT20_DTMUI	"True", "False"	True	Installs only DTM User Interface components for selected device types.
FDT20_LICENSEKEY	String license key		Transfers optional setup license key in case if product should be licensed.

Command line parameters shall be supported as defined in Table 44.

Table 44 – Setup command line parameters

Command line parameter	Description	Example
/q /i "{msi-file}"	Install setup in silent mode.	msiexec.exe /q /i "*.msi"
/q /i "{ProductCode}" REMOVE="ALL"	Uninstall setup in silent mode.	msiexec.exe /q /i "{ProductCode}" REMOVE="ALL"
ADDLOCAL="{feature-list}"	Install setup with selected features (DTMs/DeviceTypes).	msiexec.exe /q /i "*.msi" ADDLOCAL="FEATURE1;FEATURE2"
REMOVE="{feature-list}"	Uninstall selected features(DTMs/DeviceTypes).	msiexec.exe /q /i "{ProductCode}" REMOVE="FEATURE1;FEATURE2"
/lv "{log-file}"	Install setup with log.	msiexec.exe /q /lv "..\log.txt" /i "*.msi"
FDT20_DTMUI="False"	Install DTM Business Logic components only	msiexec.exe /q /i "*.msi" FDT20_DTMUI="False"
FDT20_DTMBO="False"	Install DTM User Interface components only	msiexec.exe /q /i "*.msi" FDT20_DTMBO="False"
FDT20_LICENSEKEY="ABC"	Install DTM using specified license key.	msiexec.exe /q /i "*.msi" FDT20_LICENSEKEY="ABC"

If only some selected DTMs should be installed, then Frame Application should read setup installation groups (features) for these DTMs from setup manifest file (see 9.6.2 for details). If this information is not available, then setup does not support this functionality.

NOTE During development of a setup it is necessary to observe the general limitation of operating systems in regard to length of command line and length of pathname. For example see [34] and [35].

9.2.5 Digital signatures of setup components

The Windows Installer has an embedded mechanism for checking setup components correctness based on digital signatures. The Windows Installer performs automatic signatures verification for signed external cabinet files. If a cabinet file is corrupted during the download, this will be detected by Windows Installer during the installation process. Additionally the Windows Installer database (MSI file) can be also protected with a digital signature. Authors of Windows Installer installations shall adhere to the following to ensure that all parts of the installation are covered by a digital signature:

- Only signed external cabinet files shall be provided.

  (This means that the MsiDigitalSignature table and MsiDigitalCertificate table need to be authored correctly).
- Custom actions stored within the package or installed with the package shall be used.
- The installation package shall be signed.

NOTE The tool SignTool from CryptoAPI Tools can be used for signing of cabinet files. If the bootstrapper application is used, then it should check MSI package signature itself using the Crypto API. For more details refer to [18][19].

Installation files provided by FDT Group (e.g. for core assemblies and protocol-specific files) will be signed accordingly.

9.3 Installation of FDT core assemblies

FDT core assemblies (see 5.1) shall be installed by each Frame Application and by each DTM using the standard FDT Group merge modules. The FDT core assemblies are installed in the Global Assembly Cache and are not registered otherwise.

9.4 Installation of communication protocols

9.4.1 General

Each Communication-/ Gateway-DTM shall install the supported communication protocols. The protocol assemblies (see 5.5.11) shall be installed in the Global Assembly Cache.

Protocols defined by an FDT Protocol Annex are provided as merge modules by the FDT Group. The merge modules of supported communication protocols shall be integrated in the Communication-/ Gateway-DTM setups. Vendor-specific communication protocol assemblies are installed with vendor-specific Communication-/ Gateway-DTMs.

9.4.2 Registration

Communication protocols are registered by protocol manifest files that are installed in the <FDT_Protocols> path. A protocol manifest file describes a communication protocol with its ID and assembly reference.

9.4.3 Protocol manifest

A protocol manifest is used to register additional communication protocol assemblies in the system in order to enable Frame Applications and DTMs to find it. Protocol manifest files shall be installed in the <Protocols> path (see 9.2.1). The file name shall be composed of the unique communication protocol ID and the suffix ".protocol.manifest".

A protocol manifest xml file contains following information:

- AssemblyInfo: Information about the protocol assembly that contains the communication protocol classes and data structures.

- ProtocolId: Unique identifier of the protocol (as UUID).
- ProtocolName: Human readable name of the protocol

Figure 197 shows an example for a protocol manifest.

```
<?xml version="1.0" encoding="utf-16"?>
<ProtocolManifest xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://schemas.datacontract.org/2004/07/Fdt.Deployment">
  <ProtocolId>b803f1b4-d992-44bc-a62d-08ec71b0b4cd</ProtocolId>
  <ProtocolName>XyzBus</ProtocolName>
  <AssemblyInfo>
    <Name>Fdt.XyzBus</Name>
    <Version xmlns:d3p1="http://schemas.datacontract.org/2004/07/System">
      <d3p1:_Build>0</d3p1:_Build><d3p1:_Major>1</d3p1:_Major>
      <d3p1:_Minor>0</d3p1:_Minor><d3p1:_Revision>0</d3p1:_Revision>
    </Version>
    <PublicKeyToken>1234567890123456</PublicKeyToken>
    <RuntimeVersions>
      <RuntimeVersion>
        <CLRVersionNumber xmlns:d5p1="http://schemas.datacontract.org/2004/07/System">
          <d5p1:_Build>-1</d5p1:_Build><d5p1:_Major>2</d5p1:_Major>
          <d5p1:_Minor>0</d5p1:_Minor><d5p1:_Revision>-1</d5p1:_Revision>
        </CLRVersionNumber>
      </RuntimeVersion>
      <RuntimeVersion>
        <CLRVersionNumber xmlns:d5p1="http://schemas.datacontract.org/2004/07/System">
          <d5p1:_Build>-1</d5p1:_Build><d5p1:_Major>4</d5p1:_Major>
          <d5p1:_Minor>0</d5p1:_Minor><d5p1:_Revision>-1</d5p1:_Revision>
        </CLRVersionNumber>
      </RuntimeVersion>
    </RuntimeVersions>
    <SupportedPlatforms>Any</SupportedPlatforms>
    <Path i:nil="true" />
  </AssemblyInfo>
</ProtocolManifest>
```

IEC

Figure 197 – Example: Protocol manifest

9.5 Installation of DTMs

9.5.1 General

Prior to installation of a DTM, it is possible to retrieve information about a DTM from the respective DTM Setup Manifest (see 9.6.2).

After installation the respective information can be retrieved within the FDT system.

All DTMs shall be installed in the predefined <DTMs> path (see 9.2.1). Each DTM vendor creates a subfolder <Vendor Name>. Each DTM is placed in a subfolder <DTM Name>. The path to this folder is the <DTM_root> path. All DTM assemblies and data files for a single DTM installation shall be installed in this path. The <DTM_root> folder may have a free substructure according to product requirements. Following components of a DTM are installed here:

- DTM BL assembly (see DTM component in Figure 195). It implements the main DTM Business Logic. This assembly can use or reference some other dependent assemblies. There are no limitations on file names.
- DTM User Interfaces assemblies (see DtmUI component in Figure 195). This component implements one or more DTM User Interfaces. There are no limitations on file names.
- DTM Information Builder assembly (see DtmInfoBuilder concept in 4.4.2). This component implements support for getting dynamic DTM information (e.g. for DD-Interpreter DTMs). There are no limitations in regard to file name. The DtmInfoBuilder can be integrated with the main DTM Business Logic assembly.

If a DTM uses some third-party components or re-uses some existing components from other products (especially COM / ActiveX) that cannot be installed into the path <DTM_root>, then they may be placed outside of this folder. In this case it shall be guaranteed that such components do not corrupt other running products or setups (e.g. by updating a shared component with a new revision).

NOTE Components that are used in multiple DTMs of a vendor could be installed in a common folder (e.g. "<DTMs>\<vendor name>\Common").

9.5.2 Registration

The Frame Application shall be able to retrieve information about installed DTMs and supported device types. The registration of DTMs is performed using manifest files that are installed in the predefined installation paths (see 9.2.1).

The <FDT_Registry> folder and its subfolders contain all manifest files that describe the installed DTMs and their DTM User Interfaces. Following manifest files are defined:

– *.dtm.manifest.
This xml file describes an installed DTM (see chapter 9.5.3).

– *.dtmui.manifest.
This xml file describes an installed DTM User Interface (see chapter 9.5.4).

A Frame Application searches for information about installed DTMs in three steps (see Figure 198):

a) The Frame Application reads all DTM manifest files in <DTMs > path.

b) For each DTM, the Frame Application loads the referenced assembly and starts the DtmInfoBuilder. Further information about device-, block-, module types, supported protocols etc. is retrieved from the DtmInfoBuilder.

c) For each DTM, the Frame Application reads the DTM User Interface manifest files (referenced in *.dtm.manifest) and stores user interface assembly and function information (e.g. in a device catalogue).

d) After a DTM is found, the DTM Business Logic and the respective DTM User Interface may be executed.

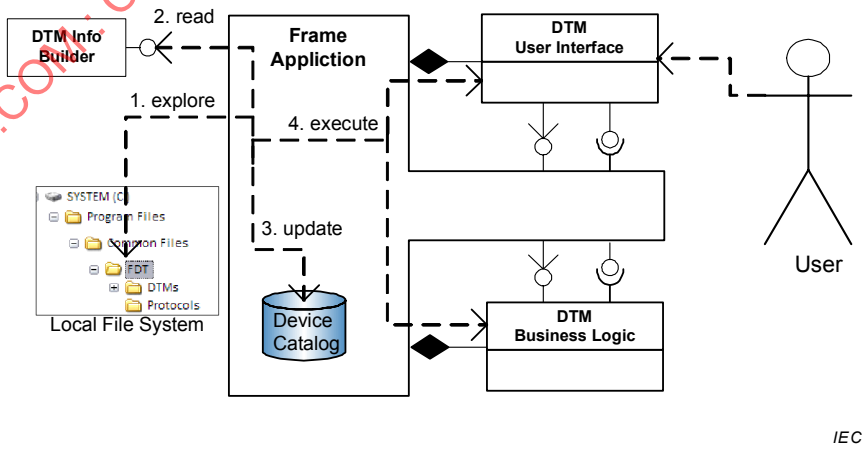


Figure 198 – Search for installed DTMs

See the descriptions of DTM manifest and DTM UI manifest datatypes for details about different manifest files that are used for DTM registration.

A Frame Application can check if new DTMs are registered by comparing the date of the DTM Manifest file with the last checking date.

9.5.3 DTM manifest

A DTM manifest file is used to register a DTM in the system in order to enable Frame Applications to find it. Therefore it contains references to the main DTM BL assembly and to the class that implements the DtmInfoBuilder for the DTM. DTM manifest files shall be copied to the vendor-specific subfolder of the <FDT_DTMs> path by the DTM setup during installation or during update of the DTM. The file name is composed by a unique DTM name and the suffix ".dtm.manifest". A DTM vendor is responsible for the uniqueness of his DTMs and DTM names within the vendor-specific name space.

A DTM manifest file (DtmManifest) contains following information:

– DynamicClassReference: Information about DtmInfoBuilder class, which shall be used to request TypeInfos and corresponding device identification information supported by the DTM.

– DtmRootPath: Root installation path of the DTM (relative path from common FDT installation path). The DTM main assembly and DtmInfoBuilder assembly (if provided) is located in this path.

– DtmInitData: [Optional] DTM initialization information. This string is passed to the DTM in the IDtm.Init call.

– ConformityRecords: [Optional] if the DTM has been certified, then this entry references a conformity record which tells details about the FDT compliance certification of the DTM.

NOTE Information about DTM device types is not included in the DTM manifest file.

DTM manifest files shall be created by using the DtmManifest datatype. See the description of DtmManifest datatype in 7.6.2 for further information about DTM manifest files.

Figure 199 shows an example for a DtmManifest.

```
<?xml version="1.0" encoding="utf-16"?>
<DtmManifest xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://schemas.datacontract.org/2004/07/Fdt.Deployment">
  <DtmRootPath>..\VendorX\DtmY\</DtmRootPath>
  <DtmInfoBuilderRef xmlns:d2p1="http://schemas.datacontract.org/2004/07/Fdt.Dtm">
    <d2p1:AssemblyInfo>
      <Name>Fdt.VendorX.DtmY</Name>
      <Version xmlns:d4p1="http://schemas.datacontract.org/2004/07/System">
        <d4p1:_Build>0</d4p1:_Build><d4p1:_Major>2</d4p1:_Major>
        <d4p1:_Minor>3</d4p1:_Minor><d4p1:_Revision>0</d4p1:_Revision>
      </Version>
      <PublicKeyToken>1234567890123456</PublicKeyToken>
      <RuntimeVersions>
        <RuntimeVersion>
          <CLRVersionNumber xmlns:d6p1="http://schemas.datacontract.org/2004/07/System">
            <d6p1:_Build>1</d6p1:_Build><d6p1:_Major>2</d6p1:_Major>
            <d6p1:_Minor>0</d6p1:_Minor><d6p1:_Revision>23456</d6p1:_Revision>
          </CLRVersionNumber>
        </RuntimeVersion>
      </RuntimeVersions>
      <SupportedPlatforms>Any</SupportedPlatforms>
      <Path>..%5CVendorX%5CDtmY%5C</Path>
    </d2p1:AssemblyInfo>
    <d2p1:ClassName>Fdt.VendorX.DtmY.DtmMainInfoBuilder</d2p1:ClassName>
  </DtmInfoBuilderRef>
  <DtmCategory>DeviceDTM</DtmCategory>
  <UiManifestRefs>
    <UiManifestRef>
      <ManifestType>DtmY.Ui.Type</ManifestType>
      <FileName>DtmY.Ui</FileName>
    </UiManifestRef>
  </UiManifestRefs>
  <ConformityRecords i:nil="true" />
  <DtmInitData>My Initialization Data</DtmInitData>
</DtmManifest>
```

IEC

Figure 199 – Example: DtmManifest

9.5.4 DTM User Interface manifest

A DTM User Interface manifest file is used to register a DTM User Interface in the system in order to enable Frame Applications to find it. These files shall be copied to the <FDT_GUIs> path by the DTM setup during installation or update of the DTM. The file name is composed of the DTM User Interface name (unique for the DTM) and the suffix ".dtmui.manifest". Each DTM User Interface manifest file shall be referenced and declared in the DTM manifest file. The DTM User Interface manifest contains following information:

– AssemblyInfo: Information about the DTM User Interface assembly.

– UIFunctionInfos: Information about the DTM User Interface functions included in the assembly described by AssemblyInfo.

– Information about type of user interface.

See the description of DtmUiManifest datatype in 7.6.3 for the syntax of DTM User Interface manifest files.

Figure 200 shows an example of a DtmUiManifest.

```
<?xml version="1.0" encoding="utf-16"?>
<DtmUiManifest xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://schemas.datacontract.org/2004/07/Fdt.Deployment">
  <AssemblyInfo>
    <Name>Fdt.VendorX.DtmY.UI</Name>
    <Version xmlns:d3p1="http://schemas.datacontract.org/2004/07/System">
      <d3p1:_Build>0</d3p1:_Build>
      <d3p1:_Major>2</d3p1:_Major>
      <d3p1:_Minor>3</d3p1:_Minor>
      <d3p1:_Revision>0</d3p1:_Revision>
    </Version>
    <PublicKeyToken>1234567890123456</PublicKeyToken>
    <RuntimeVersions>
      <RuntimeVersion>
        <CLRVersionNumber xmlns:d5p1="http://schemas.datacontract.org/2004/07/System">
          <d5p1:_Build>1</d5p1:_Build>
          <d5p1:_Major>2</d5p1:_Major>
          <d5p1:_Minor>0</d5p1:_Minor>
          <d5p1:_Revision>23456</d5p1:_Revision>
        </CLRVersionNumber>
      </RuntimeVersion>
    </RuntimeVersions>
    <SupportedPlatforms>Any</SupportedPlatforms>
    <Path>..%5CVendorX%5CDtmY%5CUserInterfaces%5C</Path>
  </AssemblyInfo>
  <UiFunctionInfos>
    <UiFunctionInfo i:type="UiControlFunctionInfo">
      <FunctionId>1</FunctionId>
      <ClassName>Fdt.VendorX.DtmY.UI01</ClassName>
      <Type>WinForm</Type>
    </UiFunctionInfo>
    <UiFunctionInfo i:type="UiControlFunctionInfo">
      <FunctionId>2</FunctionId>
      <ClassName>Fdt.VendorX.DtmY.UI02</ClassName>
      <Type>WinForm</Type>
    </UiFunctionInfo>
  </UiFunctionInfos>
</DtmUiManifest>
```

IEC

Figure 200 – Example: DtmUiManifest

9.6 DTM setup

9.6.1 Structure

The DTM setup structure defines the contents and structure of a DTM setup (see Figure 201). Each DTM setup consists of following mandatory and optional parts:

– MSI file [mandatory]. This is main Windows Installer database file. It contains complete installation logic and setup user interface. This file is always located in the setup root folder and is used to start the installation.

– Setup manifest file [mandatory]. This file describes the setup itself and contains information about included DTMs. This file is always located in the setup root folder.

– Setup Bootstrapper [optional]. This is a simple application (e.g. setup.exe) that can perform some additional actions before the MSI file is started (e.g. setup localization). In order to provide better integration, it is not recommended to use a bootstrapper. This file is always located in setup root folder.

– Cabinet files [optional]. These files contain compressed installation files of the DTM. They are always located in the Files subfolder.

– Device identification files [optional]. These files describe identification information for device types that are supported by DTMs in this setup. These protocol-specific files are always located in SupportedDevices_<DtmInfo.Id> folder. The files can be deserialized to initialize the class DtmDeviceIdentManifest.

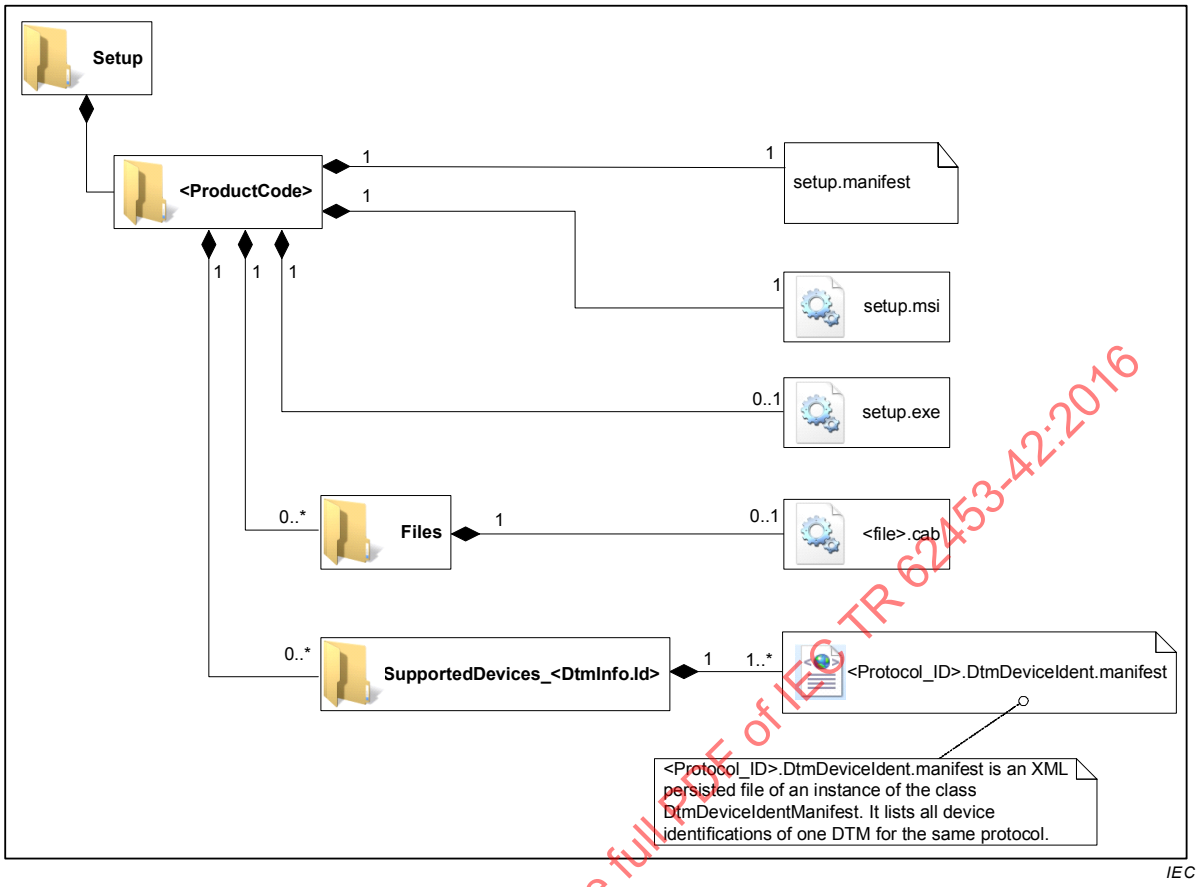


Figure 201 – DTM setup structure

9.6.2 DTM setup manifest

A setup manifest describes a DTM setup and shall be provided together with a DTM setup. Using this information the Frame Applications can check whether a DTM update is required or it can install the DTM automatically. The file name shall be composed of the SetupName and the suffix “setup.manifest”.

A DTM Setup manifest.xml file contains following information:

– DtmInfos: Information about DTMs which are included in the setup
– ProductCode: Unique identifier of the product (Windows Installer ProductCode)
– SetupName: Name of the DTM product setup.
– SetupVersion: Version of the product setup.
– PublisherName: Name of the company that provides the DTM
– ProductFeatures: List of setup features that can be installed individually. This may be used for DTM device types or additional features.
– SetupUrl: Reference to the setup msi file
– SupportedWindowsVersions: Lists the versions (including service pack level) of the operating system for which the DTMs contained in the Setup manifest are explicitly tested.
– VendorName: Name of the company which provides the DTM.
– MinimumInstallerVersion: Required version of Windows Installer (minimum version). If the required minimum version is not present on the system, the Frame Application shall use the setup.exe (bootstrapper) to start the installation instead of the msi file.

See 7.6.1 for further information about setup manifest files.

Figure 202 shows an example of a SetupManifest.

```
<?xml version="1.0" encoding="utf-16"?>
<SetupManifest xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://schemas.datacontract.org/2004/07/Fdt.Deployment">
  <VendorName>VendorX</VendorName>
  <SetupName>DtmY</SetupName>
  <ProductCode>6b6719d5-12c0-488f-897c-af440e6c5a36</ProductCode>
  <SetupVersion xmlns:d2p1="http://schemas.datacontract.org/2004/07/System">
    <d2p1:_Build>0</d2p1:_Build><d2p1:_Major>2</d2p1:_Major>
    <d2p1:_Minor>3</d2p1:_Minor><d2p1:_Revision>-1</d2p1:_Revision>
  </SetupVersion>
  <SetupUrl>../setup.msi</SetupUrl>
  <DtmInfos xmlns:d2p1="http://schemas.datacontract.org/2004/07/Fdt.Dtm">
    <d2p1:DtmInfo>
      <d2p1:DtmRef>
        <d2p1:AssemblyInfo>
          <Name>Fdt.VendorX.DtmY</Name>
          <Version xmlns:d6p1="http://schemas.datacontract.org/2004/07/System">
            <d6p1:_Build>0</d6p1:_Build><d6p1:_Major>2</d6p1:_Major>
            <d6p1:_Minor>3</d6p1:_Minor><d6p1:_Revision>0</d6p1:_Revision>
          </Version>
          <PublicKeyToken>1234567890123456</PublicKeyToken>
          <RuntimeVersions>
            <RuntimeVersion>
              <CLRVersionNumber
xmlns:d8p1="http://schemas.datacontract.org/2004/07/System">
                <d8p1:_Build>1</d8p1:_Build><d8p1:_Major>2</d8p1:_Major>
                <d8p1:_Minor>0</d8p1:_Minor><d8p1:_Revision>23456</d8p1:_Revision>
              </CLRVersionNumber>
            </RuntimeVersion>
          </RuntimeVersions>
          <SupportedPlatforms>Any</SupportedPlatforms>
          <Path>..%5CVendorX%5CDtmY%5C</Path>
        </d2p1:AssemblyInfo>
        <d2p1:ClassName>Fdt.VendorX.DtmY.DtmMain</d2p1:ClassName>
      </d2p1:DtmRef>
      <d2p1:Name>DtmY</d2p1:Name>
      <d2p1:Vendor>VendorX</d2p1:Vendor>
      <d2p1:Id>69cde55a-5bf8-45a4-90a3-7bf20184d61f</d2p1:Id>
      <d2p1:Version xmlns:d4p1="http://schemas.datacontract.org/2004/07/System">
        <d4p1:_Build>0</d4p1:_Build><d4p1:_Major>4</d4p1:_Major>
        <d4p1:_Minor>2</d4p1:_Minor><d4p1:_Revision>-1</d4p1:_Revision>
      </d2p1:Version>
      <d2p1:FdtVersion xmlns:d4p1="http://schemas.datacontract.org/2004/07/System">
        <d4p1:_Build>-1</d4p1:_Build><d4p1:_Major>2</d4p1:_Major>
        <d4p1:_Minor>0</d4p1:_Minor><d4p1:_Revision>-1</d4p1:_Revision>
      </d2p1:FdtVersion>
    </d2p1:DtmInfo>
  </DtmInfos>
  <MinimumInstallerVersion xmlns:d2p1="http://schemas.datacontract.org/2004/07/System">
    <d2p1:_Build>0</d2p1:_Build><d2p1:_Major>4</d2p1:_Major>
    <d2p1:_Minor>5</d2p1:_Minor><d2p1:_Revision>0</d2p1:_Revision>
  </MinimumInstallerVersion>
  <SupportedWindowsVersions>
    <OSVersion>
      <OSVersionNumber xmlns:d4p1="http://schemas.datacontract.org/2004/07/System">
        <d4p1:_Build>2600</d4p1:_Build><d4p1:_Major>5</d4p1:_Major>
        <d4p1:_Minor>1</d4p1:_Minor><d4p1:_Revision>196608</d4p1:_Revision>
      </OSVersionNumber>
      <ServicePack>Service Pack 3, v.5657</ServicePack>
    </OSVersion>
  </SupportedWindowsVersions>
  <ProductFeatures i:nil="true" />
</SetupManifest>
```


IEC

Figure 202 – Example: DtmSetupManifest

9.6.3 DTM device identification manifest

The device identification manifest file describes additional physical device parameters that are required for device identification. These files are dependent on the respective

communication protocol. Each file has to be placed into special subfolder that has the same name as used communication protocol identifier. Device identification manifest files are used only during setup. They are not required for installed DTMs because the DtmInfoBuilder deliver the same information.

The file name is composed of a unique name identifier and the fixed suffix ".deviceident.manifest".

A device type manifest xml file contains following information:

– DeviceIdentInfo: This information is used to describe physical device types which are supported by a DTM Device Type. It contains identification elements of a physical device type or device type group.
– ProductFeatureRefs: [Optional] list of references to product feature(s) listed in the setup manifest. These features (e.g. device type) represent the required components that shall be installed to support the detected device type.

Figure 203 shows an example of a DeviceIndentManifest.

```
<?xml version="1.0" encoding="utf-16"?>
<DtmDeviceIdentManifest xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://schemas.datacontract.org/2004/07/Fdt.Deployment">
  <BusCategory xmlns:d2p1="http://schemas.datacontract.org/2004/07/Fdt">
    <d2p1:CommunicationType>Required</d2p1:CommunicationType>
    <d2p1:PhysicalLayers>
      <d2p1:PhysicalLayer>
        <d2p1:Id>bab2091a-c0a7-4614-b9de-fcc2709dcf5d</d2p1:Id>
        <d2p1:Name>HART FSK Physical Layer</d2p1:Name>
      </d2p1:PhysicalLayer>
    </d2p1:PhysicalLayers>
    <d2p1:ProtocolId>036d1498-387b-11d4-86e1-00e0987270b9</d2p1:ProtocolId>
    <d2p1:ProtocolName>HART</d2p1:ProtocolName>
  </BusCategory>
  <DeviceIdentInfos xmlns:d2p1="http://schemas.datacontract.org/2004/07/Fdt.Dtm">
    <d2p1:DeviceIdentInfo i:type="d2p1:DeviceIdentInfo_HartDeviceIdentInfo">
      <d2p1:DeviceSpecificProperties i:nil="true" />
      <d2p1:SupportLevel>SpecificSupport</d2p1:SupportLevel>
      <d2p1:ProtocolSpecificIdentInfo
xmlns:d4p1="http://schemas.datacontract.org/2004/07/Fdt.Dtm.Hart">
        <d4p1:BusProtocolVersion>
          <d2p1:ProtocolSpecificName i:nil="true" /> <d2p1:RegularExpressions i:nil="true"
/><d2p1:Value>5</d2p1:Value>
        </d4p1:BusProtocolVersion>
        <d4p1:DeviceCommandRevisionLevel>
          <d2p1:ProtocolSpecificName i:nil="true" /> <d2p1:RegularExpressions i:nil="true" />
<d2p1:Value>4</d2p1:Value>
        </d4p1:DeviceCommandRevisionLevel>
        <d4p1:DeviceFlags>
          <d2p1:ProtocolSpecificName i:nil="true" /> <d2p1:RegularExpressions i:nil="true" />
<d2p1:Value>3</d2p1:Value>
        </d4p1:DeviceFlags>
        <d4p1:DeviceProfile>
          <d2p1:ProtocolSpecificName i:nil="true" /> <d2p1:RegularExpressions i:nil="true" />
<d2p1:Value>0</d2p1:Value>
        </d4p1:DeviceProfile>
        <d4p1:DeviceTypeCode>
          <d2p1:ProtocolSpecificName i:nil="true" /><d2p1:RegularExpressions i:nil="true" />
<d2p1:Value>123</d2p1:Value>
        </d4p1:DeviceTypeCode>
        <d4p1:HardwareRevisionLevel>
          <d2p1:ProtocolSpecificName i:nil="true" /> <d2p1:RegularExpressions i:nil="true"
/><d2p1:Value>1</d2p1:Value>
        </d4p1:HardwareRevisionLevel>
        <d4p1:ManufacturerId>
          <d2p1:ProtocolSpecificName i:nil="true" /> <d2p1:RegularExpressions i:nil="true" />
<d2p1:Value>98</d2p1:Value>
        </d4p1:ManufacturerId>
        <d4p1:PhysicalSignalingCode>
          <d2p1:ProtocolSpecificName i:nil="true" /> <d2p1:RegularExpressions i:nil="true" />
<d2p1:Value>0</d2p1:Value>
        </d4p1:PhysicalSignalingCode>
        <d4p1:SoftwareRevision>
          <d2p1:ProtocolSpecificName i:nil="true" /> <d2p1:RegularExpressions i:nil="true" />
<d2p1:Value>3</d2p1:Value>
        </d4p1:SoftwareRevision>
      </d2p1:ProtocolSpecificIdentInfo>
    </d2p1:DeviceIdentInfo>
  </DeviceIdentInfos>
  <ProductFeatures i:nil="true" />
</DtmDeviceIdentManifest>
```

IEC

Figure 203 – Example: DeviceIdentManifest

9.6.4 Setup creation rules

This chapter describes the rules and recommendations that each DTM setup shall follow in order to achieve a reliable and standardized behavior of setups. This is required in order to enable a Frame Application to retrieve information about a DTM setup, automatically install or remove it, perform updates, etc.

The (mandatory) rules are:

a) **Use Windows Installer as base installation technology**. The required version of Windows Installer is indicated in the setup manifest for each DTM setup. The minimum required version is 4.5. This version shall be provided by the Frame Application on each target system.

b) **All Windows Installer rules should be followed.** In order to achieve correct Windows Installer setups all Windows Installer rules should be followed (see [15], [16], [17]). It is strongly recommended for a setup, that it should not require a restart of the system. If the setup may require a restart of the system, the documentation for the setup and the setup manifest shall indicate this situation.

NOTE Especially for a DCS the reboot of a system is not a normal operation.

c) **DTM setup shall be provided as standalone MSI package**. The Windows Installer setup file format is ".msi". This file can be used for standalone installation as well as in context of other installations. Output (progress, running actions, log and error messages) can be integrated into external user interface (e.g. into Frame Application). Additionally, it automatically supports full Windows Installer command line syntax. Setup bootstrapper (e.g. setup.exe) may be used in addition if required.

d) **MSI packages shall be always executable in silent mode** (when user interface tables are not processed). If custom actions are used, they should be started independent from the setup user interface.

e) **Always sign external CAB files and MSI file with a digital signature**. This activates automatic check of setup consistency. If downloaded files are broken, then digital signature differs from the signature stored in the MSI file.

f) **DTM setup features shall be self-consistent**. Each feature in setup components tree that represents one DTM should be independent from other features. This means if such feature is selected for installation, the DTM will be fully installed and work. Uninstall of such feature removes the DTM completely. All dependent components, shared components etc. are installed or removed automatically.

g) **All DTM components (assemblies) shall have strong name**. This avoids DLL version conflicts if some shared components are used between different DTMs or between different versions of the same DTM.

h) **All FDT components shall be installed using official FDT merge modules**. FDT binary files cannot be used directly in setups (e.g. as automatic dependencies). All central FDT settings (e.g. Registry entries) shall be entered by FDT merge modules only. FDT components that are installed in Global Assembly Cache shall be marked as shared.

i) **An installation package shall be uniquely identifiable within the operating system dialog 'Add / Remove Programs'**. That means support information (version number and build index) shall be available to identify the version of the installation package. The visible entry shown in 'Add / Remove Programs' shall start with the name of the DTM vendor.

The recommendations are:

j) **DeviceTypes should be used as setup-features**. Other features are also possible. All Setup-features shall be listed in setup manifest. Categories for setup features shall be defined. This also allows identifying the device types that are provided with a DTM setup. In order to install only BL or UI components a frame will use command line options (realized as properties of the setup)

k) **DTM setups should use features of Windows Installer 4.5**. Newer versions of Windows Installer engine (> 4.5) have many advanced features. On the other hand, OS version limitations exist (e.g. Windows Vista is required). In this case also the additional Windows Installer runtime distribution is required what is not convenient for the end user and makes setup much larger (Windows Installer runtime shall be delivered too).

9.7 DTM deployment

A DTM setup shall support the features listed below in order to enable Frame Applications to perform automatic DTM deployment.

- A DtmSetup manifest file containing basic DTM information (also used for DTM registration, 9.6.2) is provided together with the setup
- The DTM setup can be executed from command line without a user interface (silent setup)