DRAFT INTERNATIONAL STANDARD ISO/DIS 22900-3

ISO/TC **22**/SC **3**

Secretariat: **DIN**

Voting begins on
2011-10-28

Voting terminates on
2012-03-28

Road vehicles — Modular vehicle communication interface (MVCI) —

Part 3: Diagnostic server application programming interface (D-Server API)

Véhicules routiers — Interface de communication modulaire du véhicule (MVCI) —

Partie 3: Interface pour la programmation des applications du serveur de diagnostic (D-Server API)

[Revision of first edition (ISO 22900-3:2009)]

ICS 43.040.15

In accordance with the provisions of Council Resolution 15/1993 this document is circulated in the English language only.

Conformément aux dispositions de la Résolution du Conseil 15/1993, ce document est distribué en version anglaise seulement.

To expedite distribution, this document is circulated as received from the committee secretariat. ISO Central Secretariat work of editing and text composition will be undertaken at publication stage.

Pour accélérer la distribution, le présent document est distribué tel qu'il est parvenu du secrétariat du comité. Le travail de rédaction et de composition de texte sera effectué au Secrétariat central de l'ISO au stade de publication.

Contents

Page

Foreword

ISO (the International Organization for Standardization) is a worldwide federation of national standards bodies (ISO member bodies). The work of preparing International Standards is normally carried out through ISO technical committees. Each member body interested in a subject for which a technical committee has been established has the right to be represented on that committee. International organizations, governmental and non-governmental, in liaison with ISO, also take part in the work. ISO collaborates closely with the International Electrotechnical Commission (IEC) on all matters of electrotechnical standardization.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of technical committees is to prepare International Standards. Draft International Standards adopted by the technical committees are circulated to the member bodies for voting. Publication as an International Standard requires approval by at least 75 % of the member bodies casting a vote.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO shall not be held responsible for identifying any or all such patent rights.

ISO 22900-3 was prepared by Technical Committee ISO/TC 22, *Road vehicles*, Subcommittee SC 3, *Electrical and electronic equipment*.

This second edition cancels and replaces the first edition ISO 22900-3:2009 which has been technically revised.

ISO 22900 consists of the following parts, under the general title *Road vehicles — Modular vehicle communication interface (MVCI)*:

— *Part 1: Hardware design requirements*

— *Part 2: Diagnostic protocol data unit application programming interface (D-PDU API)*

— *Part 3: Diagnostic server application programming interface (D-Server API)*

Introduction

0.1 Overview

This International Standard has been established in order to define a universal application programmer interface of a vehicle communication server application. Today's situation in the automotive market requires different vehicle communication interfaces for different vehicle OEMs supporting multiple communication protocols. However, until today, many vehicle communication interfaces are incompatible with regard to interoperability with multiple communication applications and vehicle communication protocols.

Implementation of the MVCI diagnostic server concept supports overall cost reduction to the end user because e.g. a single diagnostic or programming application will support many vehicle communication interfaces supporting different communication protocols and different vehicle communication modules of different vendors at one time.

A vehicle communication application, compliant with this document supports a protocol independent D-PDU API (Protocol Data Unit Application Programming Interface) as specified in ISO 22900-2. The server application will need to be configured with vehicle and ECU specific information. This is accomplished by supporting the ODX data format (Open Diagnostic Exchange format) as specified in ISO 22901-1.

0.2 ASAM e.V. implementation reference documents

The ISO 22900-3 references several ASAM e.V. documents which contain the Technology Reference Mapping Rules for COM-IDL, C++ and Java.

The following ASAM documents are relevant for the implementation of this part of ISO 22900:

— ASAM Technology Reference COM-IDL, *COM-IDL Technology Reference Mapping Rules [8]*:
this document describes the platform, programming language and linking mechanisms for the implementation of the generic object model in COM-IDL.

— ASAM Technology Reference C++, *C++ Technology Reference Mapping Rules [9]*:
this document describes the platform, programming language and linking mechanisms for the implementation of the generic object model in C++.

— ASAM Technology Reference Java, *Java Technology Reference Mapping Rules [10]*:
this document describes the platform, programming language and linking mechanisms for the implementation of the generic object model in Java.

Road vehicles — Modular vehicle communication interface (MVCI) —

Part 3:
Diagnostic server application programming interface (D-Server API)

1   Scope

This document focuses on the description of an object oriented programming interface. The objective is the ability to implement server applications, used during the design, production and maintenance phase of a vehicle communication system, compatible to each other and thus exchangeable. From a user's perspective, access and integration of on-board control units is provided by a corresponding application, the communication server and a VCI module for diagnostics. The user is granted access for the handling of control units (ECUs) in vehicles for the diagnostic services.

A server compliant with this document support the function block Diagnostics (D). A compliant server also supports Job-Language (Java) and may support optional features like ECU (re)- programming. The defined object oriented API provides for a simple, time saving and efficient interchangeability of different servers.

The client application and the communication server do not necessarily need to run on the same computer. A remote use via an interface may also be envisaged and is supported by the design of the server API. This interface is provided for ASAM GDI, COM/DCOM [8] [Technology Reference COM-IDL], for C++ [9] [Technology Reference C++] and for Java [10][Technology Reference Java].

2 Normative references

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

ISO 7637-2:2004, *Road Vehicles — Electrical disturbance from conduction and coupling — Part 2: Electrical transient conduction along supply lines only*

ISO 8601:2004, *Data elements and interchange formats — Information interchange — Representation of dates and times*

ISO/IEC 8859-1:1998, *Information technology — 8-bit single-byte coded graphic character sets — Part 1: Latin alphabet No. 1*

ISO/IEC 8859-2:1999, *Information technology -- 8-bit single-byte coded graphic character sets -- Part 2: Latin alphabet No. 2*

ISO 9141-2:1994, *Road vehicles — Diagnostic systems — Part 2: CARB requirements for interchange of digital information*

ISO 9141-2:1994 Amd.1:1996, *Road vehicles — Diagnostic systems — Part 2: CARB requirements for interchange of digital information Amendment 1*

ISO 14229 (all parts), *Road vehicles — Unified diagnostic services*

ISO 14230 (all parts), *Road vehicles — Diagnostic systems — Keyword protocol 2000*

ISO 15031-3, *Road vehicles — Emission-related diagnostics — Communication between vehicle and external equipment — Part 3: Diagnostic connector and related electrical circuits: specification and use*

ISO 15031-4, *Road vehicles — Communication between vehicle and external equipment for emissions-related diagnostics — Part 4: External test equipment*

ISO 15031-5, *Road vehicles — Communication between vehicle and external equipment for emissions-related diagnostics — Part 5: Emissions-related diagnostic services*

ISO 15765 (all parts), *Road vehicles — Diagnostic communication over Controller Area Network (DoCAN)*

ISO 16750-2, *Road vehicles — Environmental conditions and testing for electrical and electronical equipment — Part 2: Electrical load*

ISO 22901-1, *Road vehicles — Open diagnostic data exchange — Part 1: Data model specification*

ISO 22900-2, *Road vehicles — Modular vehicle communication interface (MVCI) —Part 2: Diagnostic protocol data unit application programmer interface (D-PDU API)*

ISO 27145 (all parts), *Road vehicles — Implementation of WWH-OBD communication requirements*

3   Terms, definitions, symbols and abbreviated terms

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

3.1 ˙Terms and definitions

3.1.1
AccessKey
a path identifier through the inheritance hierarchy as defined in ISO 22901-1 ODX to a diagnostic data element.

3.1.2
ancestor object
located above in the object hierarchy with respect to a given object. It may also be termed a parent object.

3.1.3
descendant object
object, located below in the object hierarchy with respect to a given object. It may also be termed a child object.

3.1.4
FlashJob
new class derived from MCDJob which is used to start FlashSessions within the MVCI diagnostic server. This information is provided by the data bases. At the runtime object it is possible to set the FlashSession that has to be flashed by this service. Only one session can be set for one job. The application can access the priority defined in the database for every FlashSession and sort the sessions according to this priority.

The job interface of flash jobs (MCDFlashJob) extends the job interface of normal diagnostic jobs (MCDSingleECUJob) by a session object, i.e. its method prototype shall be extended as follows:

```
JobName(...,MCDDbFlashSession session)
```

3.1.5
FlashKey
unique identification for a flash session. This FlashKey has to be absolutely unique within the whole MVCI diagnostic server. This identification can be used for production purposes to add a company specific unique Id to the ECU MEMs delivered by the ECU-supplier.

3.1.6
FlashSessionClass
user defined collection of FlashSessions and can be used to separate FlashSessions for different tasks (e.g. sessions for data, sessions for boot, or sessions for code and data).

3.1.7
FlashSession
smallest unit that can be flashed separately by the MVCI diagnostic server. Every FlashSession may consist of several data blocks.

3.1.8
functional class
set of diagnostic services.

3.1.9
function dictionary
definition of a hierarchical function catalog to organize external test equipment user interfaces (available at MCDDbProject):

—   References to one or several ECUs and their diagnostic data content relevant for that function.

—   References to services/jobs to make functions "executable".

⸺ Definition of function input and output parameters with optional references to parameters of related services.

3.1.10
interface connector
connector at the vehicle's end of the interface cable between the vehicle and the communication device.

3.1.11
job
sequence of diagnostic services and other jobs with a control flow.

3.1.12
location
set of diagnostic data valid on a given hierarchical level of inheritance as defined in ISO 22901-1 ODX. The following locations exist:

⸺ Multiple ECU Job,

⸺ Protocol,

⸺ Functional Group,

⸺ ECU Base Variant,

⸺ ECU Variant.

3.1.13
Logical Link
set of data, identifying the physical line, the interface and protocol used for an ECU.

3.1.14
physical interface link
physical connection between the VCI connector of a VCI and the interface connector.

3.1.15
physical link
physical vehicle link connected to a physical interface link, so it is the connection from the interface of the diagnostic server to the ECU in the vehicle.

3.1.16
physical vehicle link
unique bus system in a vehicle, so it is the connection between the vehicle connector and the ECU.

3.1.17
priority
used by test systems to decide in which order the sessions have to be flashed.

3.1.18
project
pool of diagnostic data. References between such data shall all be resolvable inside this same project.

3.1.19
sub component
description of ECU sub functionality- or components e.g. LIN-slaves (available at MCDDbLocation).

3.1.20
vehicle connector
connector on a vehicle providing access to the bus systems in the vehicle.

3.2 Symbols

Figure 1 depicts the legend of hierarchical models.

	color print	black/white print
● Interface not derived from MCDObject	blue	black
○ Interface not directly used	white	white
Ⓓ D data base interface	yellow	grey
Ⓓ D run time interface	green	dark grey
MCD MCD data base interface	yellow	grey
MCD MCD run time interface	green	dark grey

Figure 1 — Legend of hierarchical models

3.3 Abbreviated terms

API	Application Programmers Interface
ASAM	Association for Standardisation of Automation and Measuring Systems
ASCII	American Standard for Character Information Interchange
AUSY	AUtomation SYstem
CAN	Controller Area Network
COM/DCOM	Distributed Component Object Model
CORBA	Common Object Request Broker Architecture
CRC	Cyclic Redundancy Check
D	Diagnostics
Diag	Diagnostic
DLL	Dynamic Link Library
DoCAN	Diagnostic communication over CAN
DOP	diagnostic Data Object Property
DoIP	Diagnostic Over Internet Protocol
DTC	Diagnostic Trouble Code
DTD	Document Type Definition
DynID	Dynamically Defined Identifiers
ECU	Electronic Control Unit
ECU MEM	Electronic Control Unit MEMory
ERD	Entity Relationship Diagram
IDL	Interface Description Language
JAVA RMI	JAVA Remote Method Invocation
KWP	KeyWord Protocol
LIN	Local Interconnect Network

MCD	Measurement Calibration Diagnostic
MDF	Module Description File
MVCI	Modular Vehicle Communication Interface
ODX	Open Diagnostic data eXchange
OEM	Original Equipment Manufacturer
PC	Personal Computer
PDU	Protocol Data Unit
SDG	Special Data Groups
SI	Systéme International d'unités
UDS	Unified Diagnostic Services
UTC	Coordinated Universal Time
VI	Variant Identification
VIS	Variant Identification and Selection
VIT	VehicleInformationTable
XML	eXtended Markup Language

4   Conventions

4.1   General

ISO/DIS 22900-3 is based on the conventions discussed in the OSI Service Conventions (ISO/IEC 10731:1994) as they apply for diagnostic services.

4.2   Typographical conventions and mnemonics

Normal text of the specification is presented like this.

Source code and technical artifacts within the text are presented like `this`.

Diagrams that denote interaction sequences, relationships or dependencies between interfaces are presented using the Unified Modeling Language's (UML) convention.

The name of each interface and each class defined by this standard shall use the prefix of the stereotype, e.g. "D".

The leading letter of each method and each parameter is small.

The leading word of each method shall be a verb.

The letter "_" is not allowed in interface names, method names and parameter names, but it is allowed for constants.

The leading letter of each constant is "e" and behind this the name is written in capital letters.

ODX element names are written in upper cases, e.g. SHORT-NAME. MVCI diagnostic server Names are written in mixed fixed, e.g. MCDDbProject.

4.3 Sequence diagrams

With the help of Sequence Diagrams the interactive use of the API and the sequences for certain general cases are presented in chronological order.

The sequence diagrams are oriented according to the presentation in UML and are structured as follows. The chronological sequence arises while reading from top downwards. At the left margin the commentary column is placed, in which single activities are commented. Within the sequence diagram the Client application is shown left, if necessary for the respective case, the EventHandler is shown there as well. Right of the Client (with or without EventHandler) the API objects necessary for the respective case are located. At the right outside the MVCI diagnostic server is presented if necessary.

Not all API objects possible for the respective instant of time are shown, but only those of relevance for the respective case. The thin line leading down vertically from the objects represents the life line, the wider sections on it represent activities of the object.

The black horizontal arrows between the single objects, Client and MVCI diagnostic server represent the actions necessary for the respective case. The object to which the arrow points at will execute the action. The grey horizontal arrows represent the return of objects.

4.4 Stereotypes

Stereotypes are abbreviation characters which are used in MVCI diagnostic server to mark the affiliation of statements, interfaces and methods to one of the possible parts.

Table 1 defines the stereotypes which are used in MVCI diagnostic server.

Table 1 — Stereotypes

Stereotype	Usage of method and class is in following Function Blocks allowed
<<MCD>>	Measurement, Calibration and Diagnostic
<<D>>	Diagnosis
<<JD>>	Methods with this stereotype can only be used inside of Diagnostic Job. These methods are not available for use at the API.

5 Specification release version information

The specification release version of this part of ISO 22900 is: 2.9.5.

6 Structure of a MVCI diagnostic server

Each server is divided into the functional block "D" (Diagnostic) and the data base.

Figure 2 depicts the architecture of a MVCI diagnostic server.
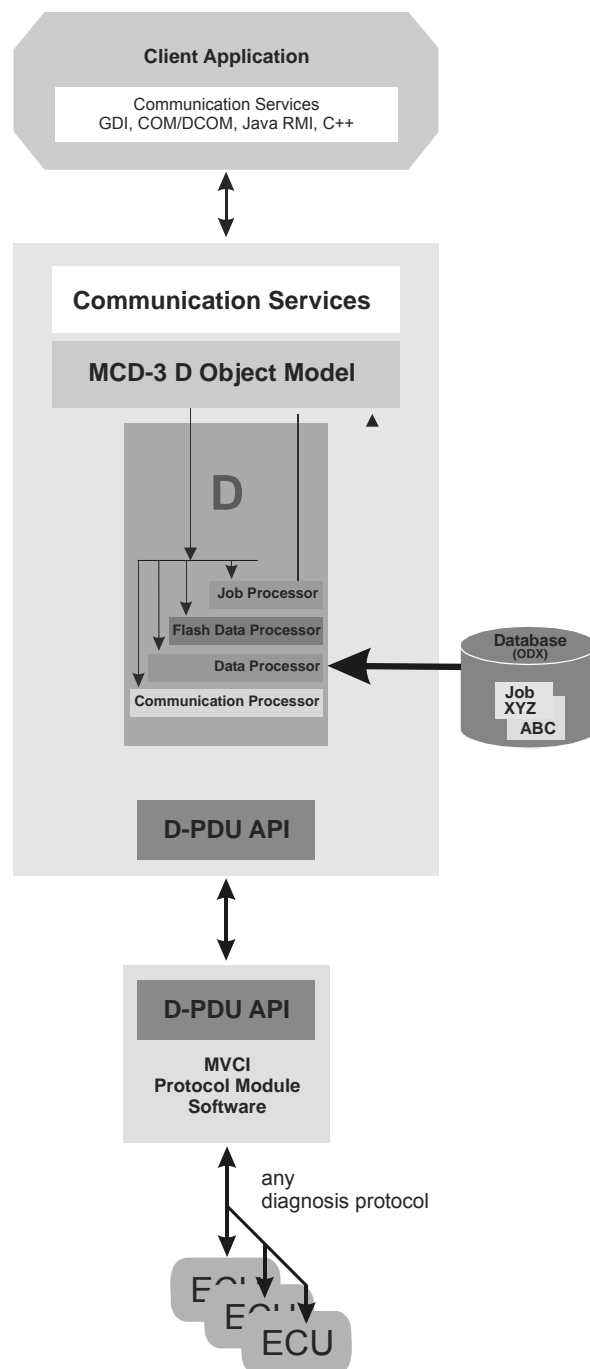
Figure 2 — Architecture of a MVCI diagnostic server

With the help of server the control units are optimally adapted to the relevant requirements for their use in vehicles. This procedure is often referred to as "Applying".

The following features (interfaces and methods) are optional:

— MCDDbProjectConfiguration,

— ECU Configuration,

— ECU Reprogramming (Flash),

— DynID,

— Monitoring,

— System properties,

— Function dictionary,

— SubComponents,

— Audiences,

— ECU state,

— Multiple ECU Jobs,

— PDU Time stamps,

— Library concept,

— DoIP,

— PDU API support,

— the concept of System generated Vehicle Information table.

Optional means that the runtime- as well as the database part of the model do not have to be implemented by a diagnostic server that is omitting the feature in question. In case a client application calls a method that is part of an optional feature, the diagnostic server should return an empty collection in case the return type of the method is inheriting from MCDCollection. Otherwise, such a method call should throw an `MCDSystemException` of type `eSYSTEM_METHOD_NOT_SUPPORTED`. In case of support of optional features these have to be implemented completely.

The number of control units applied in vehicles is continuously increasing. The capabilities of the single control unit concerning diagnosis become available for the server by means of control unit description files (Data Description Interface). The control unit description files represent a manufacturer independent data exchange format, that means any server may handle the data out of a control unit description file. All configuration data of the diagnostic server, the internal data of ECUs or ECU nets and the communication methods for the ECU access are stored in the ODX data base. This database is server and operating system independent and therefore allows data to be exchanged between vehicle manufacturers and ECU suppliers.

An application can read out all data from the database that is necessary to drive the MVCI diagnostic server, that means only the MVCI diagnostic server can access the information of the separate control unit description files comprised within one data base. With this, at the same time the consistency of the information between AUSY and MVCI diagnostic server is guaranteed.

Also, a decoupling from the used data description exchange format (XML) takes place.

The MVCI diagnostic server has to manage the database and to provide the required and necessary information for the single MVCI diagnostic server objects. The database does not belong to one specific MVCI diagnostic server object, but is available within the whole diagnostic server. The organization of the object or reference allocation is solved implementation specifically by the diagnostic server.

The object model supports Single Client Systems, to provide for a simple use for this most typical and most frequently occurring application case. That means, that no client references are included within the single objects. The administration of the client references is done by the diagnostic server and has to be solved implementation specifically.

The object model has been designed in a technology and programming language independent way. It may be used remotely as well as locally.

The object model of MVCI diagnostic server enables the linking of MVCI diagnostic servers to automation systems. Objective of this linking is the remote control of the MVCI diagnostic server in test stands. By means of the object model the functionality, that means the interfaces with accompanying methods, are standardized. The communication has to be realized via the particular implementation of the object model for the used platform, programming language and linking mechanisms.

Among others are realized:

— ASAM GDI,

— JAVA,

— COM / DCOM and

— C++.

The necessary specifications for this will be described and published in separate documents. For this process design patterns and mapping rules are defined and published.

All other specifications will be set up and realized implementation specifically by the respective system provider.

Within the function block diagnostic a breaking down into characteristic sub tasks will take place, which are shown in the following:

The Communication Processor is responsible for generating and analysing of request and response telegrams to ECUs. This processor handles all protocol specific tasks like timings, creation of protocol headers and checksums, etc. Diagnostic protocol specification is (at the moment) not a task of ASAM, because this is covered by ISO activities like ISO 14230 KWP 2000, ISO 14229 UDS and ISO 15765 DoCAN. Nevertheless, the communication processor shall be parameterised via Communication Parameters. The Communication Processor is an interface (the only one) to the ECU .

The Data Processor is responsible for the supply of parameters and results on physical level. By means of the Data Processor all necessary information is fetched from the data base. Additionally, the Data Processor converts ECU answers from hexadecimal representation into a physical or any text representation and vice versa. The Data Processor is an interface (the only one) to the ODX database and offers an ODX library to the Job Processor. The Data processor also handles Jobs, as they are stored in the ODX data base.

The Flash Data Processor is responsible for the loading of programs and data in ECUs. The flash data is part of the data base. The Flash Data Processor provides access to the ASAM MCD2-ECU-MEM which contains all information about physical / logical data- / code-layout and possible combinations of data and code segments and more. The Flash Data Processor is an interface (the only one) to the flash data and offers a flash library (flash object) to the Job Processor.

The Job Processor is responsible for the execution of service sequences and only uses objects of the ASAM MVCI diagnostic server API. Via the Job Processor all processors may be accessed. The Jobs are part of the data base. The Job Processor is based on the ODX format. The Job Processor provides several libraries for standardized access to the Communication Processor, Data Processor, Flash Data Processor and to the Job Processor itself. The Job Processor uses the same objects to interact with the different Processors like the ASAM MVCI diagnostic server API to insure consistency between ASAM MVCI diagnostic server API and Job Processor. The Job Processor gets its code to be executed from the Data Processor. The Data Processor itself reads the ODX database file.

7   Diagnostic server

7.1   MCD system object

The server interface is a client's first access point to the MVCI diagnostic server. From this every interface is reachable. Each client gets its own `MCDSystem` object (implements the `MCDSystem` interface) from the MVCI diagnostic server. But all clients work on the same project and data base. The project has the connection to a special part of the whole database and this part will be made available after selecting the project. The selection of another project at the same time is not allowed and will throw an exception.

Figure 3 depicts the system scheduling.
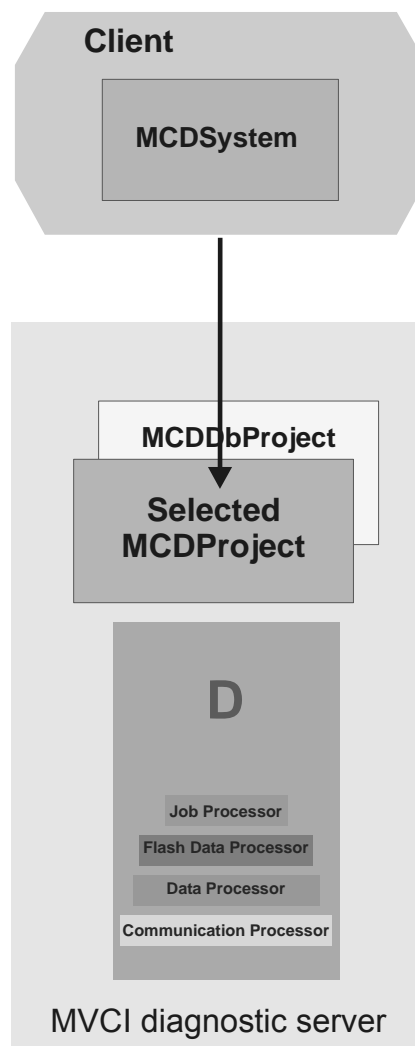
Figure 3 — System scheduling

The diagrams specified in this part of ISO 22900 always refer to the representation within the client and are not designed for MultiClient scenarios.

7.2 Description of terms

7.2.1 General

This section describes the most important terms in more detail. A brief definition is included in 3.1. For each term which is directly related to an ISO 22901-1 ODX element, the corresponding ODX element name is given in parentheses.

7.2.2 Access key (AccessKey)

By means of the AccessKey, the access position within the inheritance hierarchy of the ODX diag layers is identified.

One AccessKey element is composed of the type information [ElementIdentifier] embedded in square brackets followed by the short name of the element instance. That is, the AccessKey is a sequence of tuples of ElementIdentifier and short name. The allowed combinations of ElementIdentifiers are defined by the Locations. AccessKeys are unique within one data base.

Table 2 defines the ElementIdentifiers.

Table 2 — ElementIdentifiers

ElementIdentifiers
[MultipleEcuJob]
[Protocol]
[FunctionalGroup]
[EcuBaseVariant]
[EcuVariant]

For every element accessed via an AccessKey there is a LongName and a Description. LongName and Description shall be in UNICODE.

7.2.3 Functional Class (FUNCTIONAL-CLASS)

Functional classes are groups of services (freely definable). A service can be part of multiple functional classes but can have only one semantics. A functional class is an arbitrary, user definable group of services.

7.2.4 Job (SINGLE-ECU-JOB, MULTIPLE-ECU-JOB)

Sequence of diagnostic services and other jobs with control flow inside job, based on received results. Use cases for jobs are ECU (re-) programming, Encryption of seed key algorithm and gateway tests.

7.2.5 Location

A Location represents a hierarchical level for diagnostic Services.

The following locations are permitted:

— [D] Multiple ECU Job,

— [D] Protocol,

— [D] Functional Group,

 ⏤ [D] ECU Base Variant,

 ⏤ [D] ECU Variant.

The is entry point for available Services, DiagComPrimitives, CompuMethods and so on.

Figure 4 depicts the location hierarchy of ASAM MCD data base.
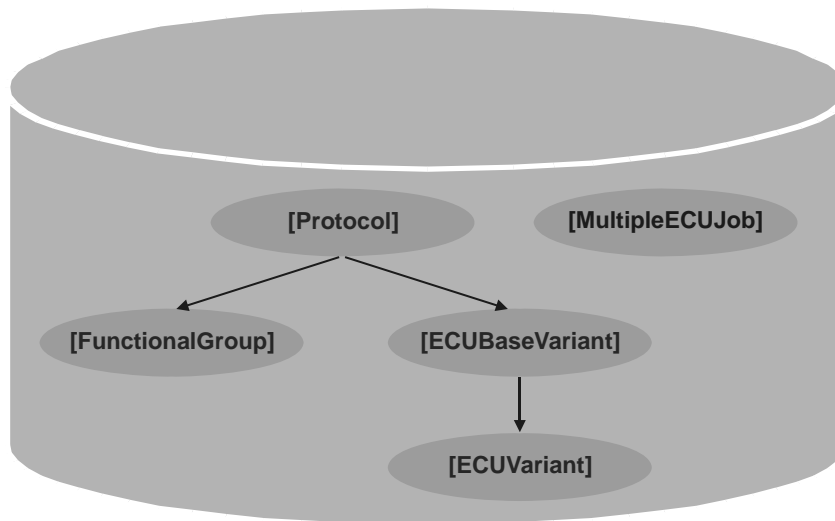
Figure 4 — Location hierarchy of ASAM MCD data base

Each Location is addressed by means of an AccesKey. The following AccessKeys of possible Locations in the hierarchical system are allowed:

 ⏤ [Protocol]Instancename

 ⏤ [Protocol]Instancename.[FunctionalGroup]Instancename

 ⏤ [Protocol]Instancename.[EcuBaseVariant]Instancename

 ⏤ [Protocol]Instancename.[EcuBaseVariant]Instancename.[EcuVariant]Instancename

 ⏤ [MultipleEcuJob]MultipleECUJob

Figure 5 depicts the AccessKey example.
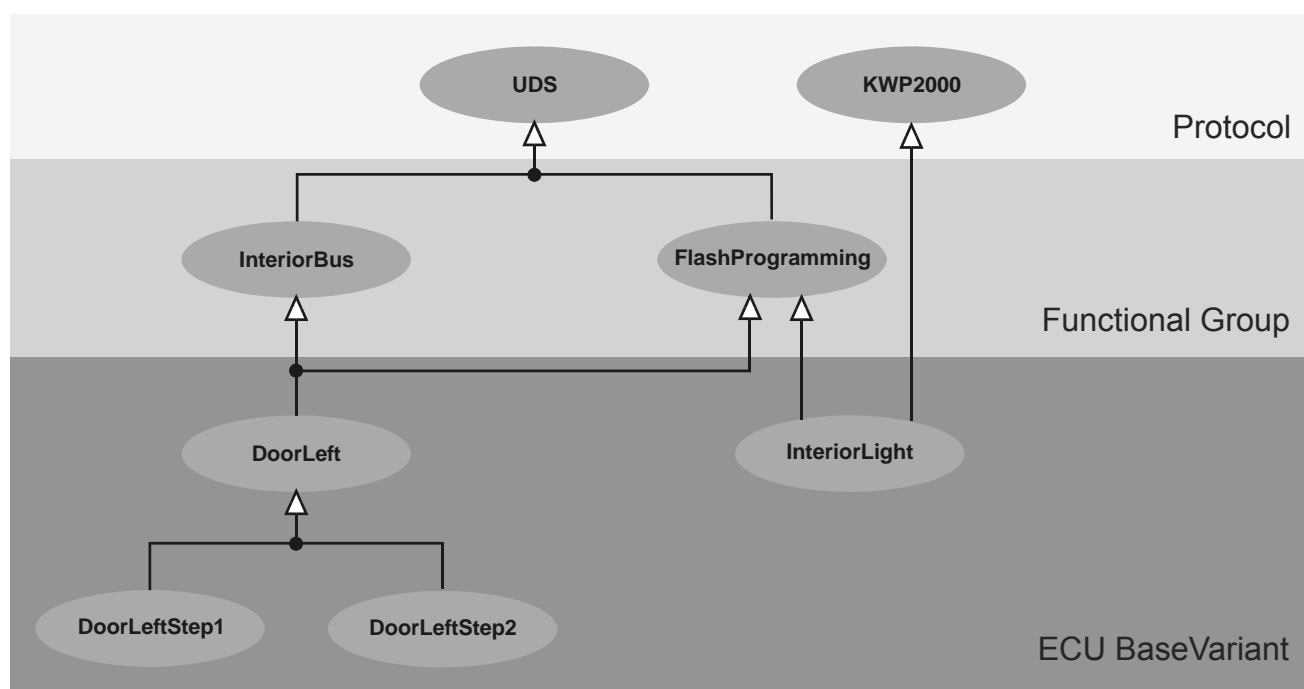
Figure 5 — AccessKey example

Resulting AccessKeys:

⎯ [Protocol]UDS,

⎯ [Protocol]UDS.[FunctionalGroup]InteriorBus,

⎯ [Protocol]UDS.[FunctionalGroup]FlashProgramming,

⎯ [Protocol]UDS.[EcuBaseVariant]DoorLeft,

⎯ [Protocol]UDS.[EcuBaseVariant]DoorLeft.[EcuVariant]DoorLeftStep1,

⎯ [Protocol]UDS.[EcuBaseVariant]DoorLeft.[EcuVariant]DoorLeftStep2,

⎯ [Protocol]UDS.[EcuBaseVariant]InteriorLight,

⎯ [Protocol]KWP2000.[EcuBaseVariant]InteriorLight,

⎯ [Protocol]KWP2000;

7.2.6   Logical Link (LOGICAL-LINK)

A Logical Link is a logical connection to ECUs. Normally this is only one ECU, but in case of a Functional Group it can be more than one ECU. The Logical Link is represented by a short name. Information about a Logical Link is contained in the Logical Link Table. Elements of this table are the AccessKey and the Physical

Vehicle Link or Interface. Logical Links are used to access the same ECU on different ways, or access more than one ECU instance on different links. For more details see in 8.

7.2.7 Physical Interface Link

A Physical Interface Link is a logical connection between MVCI diagnostic server and Interface Connector. The Physical Interface Link is represented by a short name. Information about a Physical Interface Link is contained in the Interface Connector Information Table. In this table the description of the Interface connector is included. The available Physical interface links are defined by the available interfaces of a MVCI diagnostic server.

The Interface Connector Information Table has an entry for each Physical Interface Link and one connector for this Link.

The Interface Connector Information Table uses the standardized short name of a Physical Link.

7.2.8 Physical Link

A physical Link is a Physical Vehicle Link connected to a Physical Interface Link, so it is the connection from the interface of the MVCI diagnostic server to the ECU in the vehicle.

Figure 6 depicts the overall scheme between different links and tables in D.
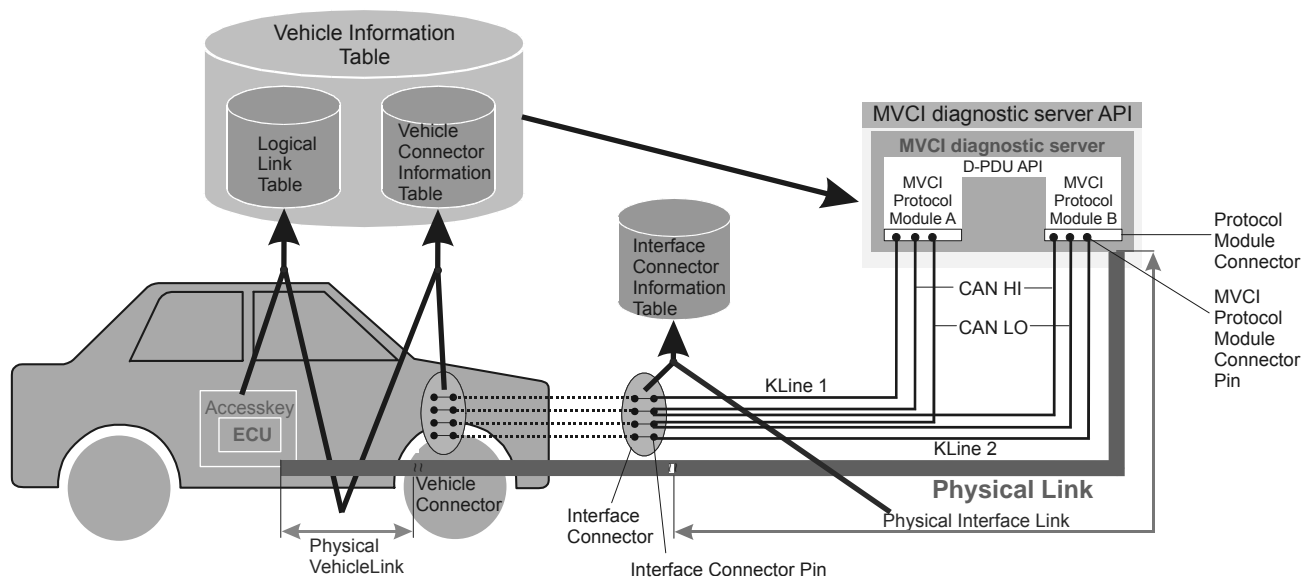
Figure 6 — Overall scheme between different links and tables in D

The pins of the Vehicle Connector are connected to identical pins of the Interface connector.

7.2.9 Physical Vehicle Link (PHYSICAL-VEHICLE-LINK)

The physical vehicle link describes the unique bus system in a vehicle, so it is the connection between the vehicle connector and the ECU. The physical vehicle link is represented by a short name. Information about a vehicle link is contained in the Vehicle Connector Information Table. In this table the Vehicle Connector Information is included.

The Vehicle Connector Information Table has an entry for each Physical Vehicle Link and one or more Vehicle Connectors (Pins) for this Link.

The available Physical vehicle links are defined by the vehicle.

7.2.10 Project

A project is a logical grouping for defined test installations selected by the user. Within a project, all information necessary for a test installation has to be contained. It is only permitted to work within one project, this has to be considered for the logical grouping (e.g. two model series within one project). That's why the project tuple is not part of the AccessKey.

At project level, the forming of manufacturer specific hierarchies is possible, as for this level no standardisation takes place.

Typically, a project contains:

— all static database information (Diagnosis Services, ...)

— jobs

— flash containers and

— configuration files.

Within the MVCI diagnostic server, first the selection of a project out of the list of the existing projects takes place, before any database information can be accessed. Because of this mechanism only one project can be active.

The database of a project is not restricted to a physical file.

7.3 Version information retrieval

The method `MCDSystem::getASAMMCDVersion():MCDVersion` returns the ASAM release number of the interface. For this specification it is the major value 3 and the minor value 0 defined.

The method `MCDSystem::getVersion():MCDVersion` returns the tool version.

The technology version number is available via the interface `MCDSystem::getInterfaceNumber():A_UINT32`.

The version number of the model (specification) and the version number of the reference implementations are kept synchronous. The so-called "interface number (IFN)" is used as a build number for the reference implementations. Each reference implementation maintains an own interface number. This number will be incremented continuously within a minor version. Increments are applied whenever a new reference implementation is generated and shipped. The "interface number" is reset to zero for each new minor or major version. Please, note that this number is completely independent from the version numbers. The technology version number is incremented separately for each technology.

7.4   States of the MCD system

Within the state `eINITIALIZED` the `MCDSystem` object is transmitted to the client by the MVCI diagnostic server. Within this state, the database project descriptions can be listed and general system initialisations can be done.

By selecting the project to be worked on, the system takes the state `ePROJECT_SELECTED`. Within this state the database can be polled for information, but no communication to the hardware is possible.

As soon as the first Logical Link has been created, the `MCDSystem` object takes the state `eLOGICALLY_CONNECTED` and can execute the communication by means of the created Logical Links. If the last Logical Link has been removed from the runtime project, the system automatically takes the state `ePROJECT_SELECTED` again. By means of `deselectProject()` the `MCDSystem` Object is set to the initial state `eINITIALIZED` again.

The diagnostic server changes the state to `eDBPROJECT_CONFIGURATION` by loading a database project at object `MCDDbProjectConfiguration`. In this state is searching in database possible. Configuration (add and remove elements) of the database can also be done in this state. ECUMEMs can be added. ECU-MEMs should be loaded temporary to a MCDDbProject or permanently added to a project configuration. ECU-MEM's are necessary for flashing.

Figure 7 depicts the system state transitions.
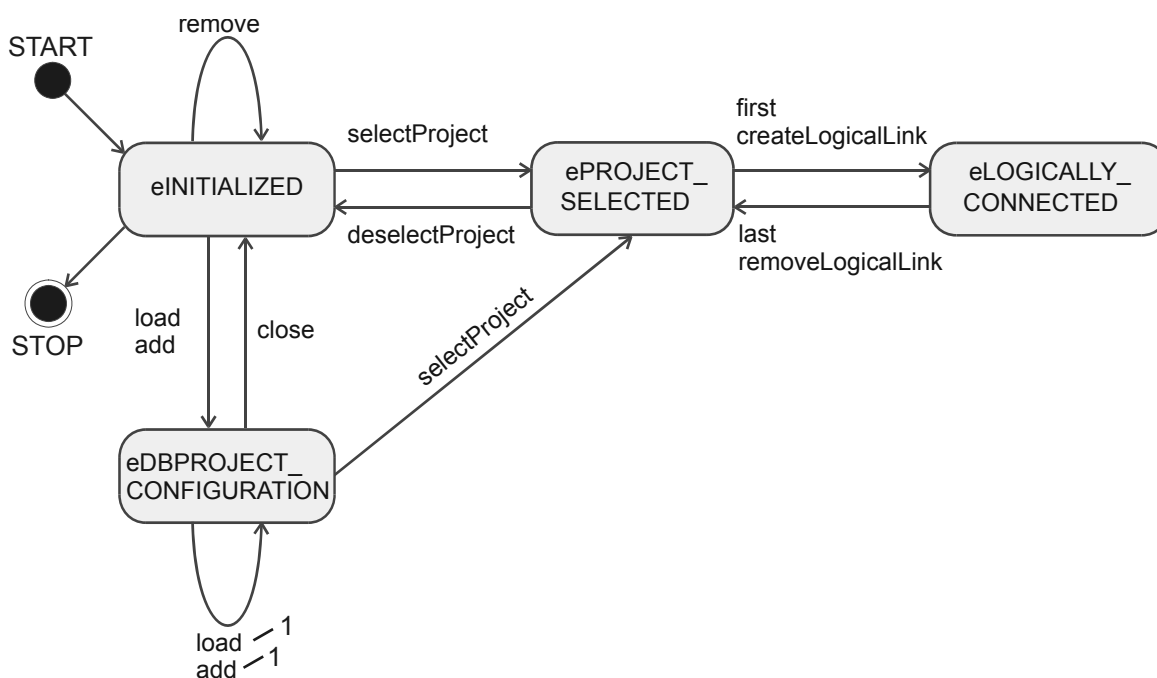
Key

1   `'load'` or `'add'` of `DbProject` includes an implicit close of actual opened `DbProject`.

Figure 7 — System state transitions

It has to be distinguished between the states of clients and the central state of the cerver (internal MCDSystem).

At the server side there may only be one state (eINITIALIZED, eDBPROJECT_CONFIGURATION, ePROJECT_SELECTED, eLOGICALLY_CONNECTED). As soon as a client has successfully called selectProject, the internal MCDSystem transits to the state ePROJECT_SELECTED.

Table 3 defines the system states.

Table 3 — System states

System State	MCDSystem				MCDDbProject Configuration				MCDProject				MCDDb Project
	MCDDbVehicleInformation selected	selectProject	selectProjectByName	deselectProject	load	close	getAdditionalEcuMemNames	getActiveDbProject	selectDbVehicleInformation	selectDbVehicleInformationByName	deselectVehicleInformation	CreateLogicalLink…	loadNewEcuMem
eINITIALIZED	no	X	X		X				state does not occur				
ePROJECT_SELECTED	yes							X	X^a	X^a	X	X	
	no			X				X	X	X			
eLOGICALLY_CONNECTED	yes							X	X^a	X^a		X	
	no							X	state does not occur				
eDB_PROJECT_CONFIGURATION	no	X	X		X	X	X	X					X

a   valid action in case there isn't already selected a different vehicle information, no state transition, otherwise invalid action, an exception will be thrown, see corresponding method definition

Figure 8 depicts the system lock states.
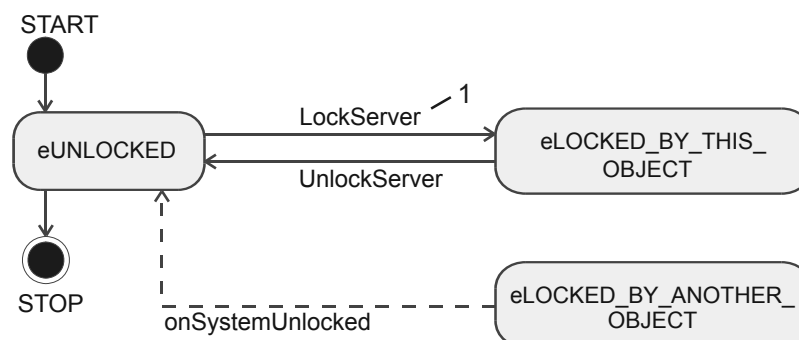
Key

1   this transition can only be done in system state eINITIALIZED.

Figure 8 — System lock states

Each `MCDSystem` object may be locked for exclusive use of the whole MVCI diagnostic server by the client application. The locking of the `MCDSystem` may only take place within the state `eINITIALIZED` and refers to the internal server `MCDSystem` object. No other client can than communicate with the diagnostic server. Within the states `ePROJECT_SELECTED`, `eLOGICALLY_CONNECTED` and `ePROJECT_CONFIGURATION` this is not allowed. If a `MCDSystem` Object is exclusive locked, the locking client exclusively accesses the MVCI diagnostic server. All clients will be informed by system event of locking and unlocking.

`MCDProject/MCDSystem` LongName and Description are server specific values and can be empty. An exception will not be used.

7.5   State changes

The behaviour for methods which should change a state should be always the same, i.e. a call for transition to a state which is available before the call throws an exception without a state change.

This behaviour is used by `MCDSystem, MCDLogicalLink, MCDDiagComPrimitive.`

If any exception occurs during a state changing operation, the state shall not be changed!

7.6   Project configuration

Within the state `ePROJECT_CONFIGURATION` of the `MCDSystem` a database project may be loaded and browsed. Runtime objects may not be created within this state. Once the transition to this state has taken place, this state can only be left by closing down the database project or selecting the corresponding runtime project. This selection implicates the saving of all changes which were made. In this state the used database project can be changed by loading or creating another database project, the so far used project will be implicitly saved if changes were made.

The Method `MCDSystem:getDbProjectConfiguration` returns a `MCDProjectConfiguration` Object, which can be used for

— browsing (without existing RunTimeObjects, MCD: `MCDProjectConfiguration:load`) and

— modification (D: `MCDDbProject:loadNewECUMem`) of DbProjects.

The state transition from `eINITIALIZED` to `ePROJECT_CONFIGURATION` takes place only after a Client opens a `DbProject` by means of `add/load`.

The Methods

— `MCDSystem::selectProject(),`

— `MCDDbProject::loadNewEcuMem()` and

— `MCDDbProjectConfiguration::getAdditionalEcuMemNames`

checks   the   consistence   of   data   read   from   the   data   base.   An   error   of   type `eDB_INCONSISTENT_DATABASE` should be thrown in case inconsistent data has been identified.

Figure 9 depicts the project configuration.

MCDSystem `<<M,C,D>>`

- `<<M,C,D>>` deselectProject()
- `<<D>>` detectInterfaces()
- `<<M,C,D>>` getActiveProject()
- `<<M,C,D>>` getASAMMCDVersion()
- `<<D>>` getConnectedInterfaces()
- `<<D>>` getCurrentInterfaces()
- `<<M,C,D>>` getDbProjectConfiguration()
- `<<M,C,D>>` getDbProjectDescriptions()
- `<<D>>` getEnumValue()
- `<<M,C,D>>` getInterfaceNumber()
- `<<M,C,D>>` getLockState()
- `<<D>>` getProperty()
- `<<D>>` getPropertyNames()
- `<<M,C,D>>` getServerType()
- `<<M,C,D>>` getState()
- `<<D>>` getSystemParameter()
- `<<M,C,D>>` getVersion()
- `<<D>>` isUnsupportedComParametersAccepted()
- `<<M,C,D>>` lockServer()
- `<<D>>` prepareInterface()
- `<<D>>` prepareVciAccessLayer()
- `<<M,C,D>>` releaseEventHandler()
- `<<D>>` resetProperty()
- `<<M,C,D>>` selectProject()
- `<<M,C,D>>` selectProjectByName()
- `<<M,C,D>>` setEventHandler()
- `<<D>>` setProperty()
- `<<M,C,D>>` unlockServer()
- `<<D>>` unprepareInterface()
- `<<D>>` unprepareVciAccessLayer()
- `<<D>>` unsupportedComParametersAccepted()

MCDDbProjectConfiguration `<<M,C,D>>`

- `<<M,C,D>>` close()
- `<<M,C,D>>` getActiveDbProject()
- `<<D>>` getAdditionalECUMEMNames()
- `<<M,C,D>>` load()

MCDProject `<<M,C,D>>`

- `<<D>>` createLogicalLink()
- `<<D>>` createLogicalLinkByAccessKey()
- `<<D>>` createLogicalLinkByAccessKeyAndInterface()
- `<<D>>` createLogicalLinkByAccessKeyAndInterfaceResource()
- `<<D>>` createLogicalLinkByInterface()
- `<<D>>` createLogicalLinkByInterfaceResource()
- `<<D>>` createLogicalLinkByName()
- `<<D>>` createLogicalLinkByNameAndInterface()
- `<<D>>` createLogicalLinkByNameAndInterfaceResource()
- `<<D>>` createLogicalLinkByVariant()
- `<<D>>` createLogicalLinkByVariantAndInterface()
- `<<D>>` createLogicalLinkByVariantAndInterfaceResource()
- `<<D>>` createMonitoringLink()
- `<<D>>` createValue()
- `<<D>>` execIOCtrl()
- `<<M,C,D>>` deselectVehicleInformation()
- `<<D>>` getClampState()
- `<<M,C,D>>` getActiveDbVehicleInformation()
- `<<M,C,D>>` getDbProject()
- `<<D>>` getIOControlNames()
- `<<D>>` removeLogicalLink()
- `<<D>>` removeMonitoringLink()
- `<<M,C,D>>` selectDbVehicleInformation()
- `<<M,C,D>>` selectDbVehicleInformationByName()

MCDDbProject `<<M,C,D>>`

- `<<D>>` getAccessKeys()
- `<<D>>` getDbEcuBaseVariants()
- `<<D>>` getDbEcuMems()
- `<<D>>` getDbElementByAccessKey()
- `<<D>>` getDbFunctionalGroups()
- `<<D>>` getDbFunctionDictionaries()
- `<<D>>` getDbMultipleEcuJobLocation()
- `<<M,C,D>>` getDbPhysicalVehicleLinkOrInterfaces()
- `<<D>>` getDbProtocolLocations()
- `<<M,C,D>>` getDbVehicleInformations()
- `<<M,C,D>>` getVersion()
- `<<D>>` loadNewECUMEM()

Figure 9 — Project configuration

21

7.7 Interface structure of server API

7.7.1 Hierarchical model overview

Figure 10 depicts the hierarchical model of a diagnostic server part I.

Figure 10 — Hierarchical model of a diagnostic server part I

Figure 11 depicts the hierarchical model of a diagnostic server part II.

Figure 11 — Hierarchical model of a diagnostic server part II

Figure 12 depicts the hierarchical model of a diagnostic server part III.

Figure 12 — Hierarchical model of a diagnostic server part III

Figure 13 depicts the hierarchical model of a diagnostic server part IV.
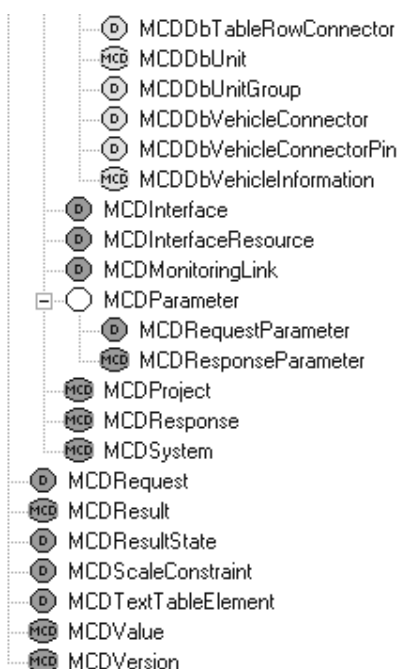
Figure 13 — Hierarchical model of a diagnostic server part IV

As can be seen, all database interfaces are derived from `MCDDbObject`, and all database Collections from `MCDNamedCollection`. The Runtime interfaces usually are derived directly from `MCDObject`. Collections, Exceptions and NamedObjects each form basic classes for a further detailed structuring. Apart from `MCDSystem` and `MCDProject` also all database interfaces belong to the NamedObjects.

The `MCDEventHandler` represents an exception concerning this classification and is not derived from `MCDObject`. It is used as means of communication of the diagnostic server with the Client and provides for Callback methods for the Event sending.

The `MCDEnumValue` is the super class of all classes representing enumerations in this API. It provides two methods to translate between the object and the string representation of the enumeration and vice versa. So string need not to be delivered at the API if there is no need to do and the server can implement enumerations and their representation very efficient with a hash map.

7.7.2   Database and runtime side

7.7.2.1   Basics

Within the whole model there is a separation into database (yellow in black/white the light) and runtime (green in black/white the dark). The database part is invariable and is used as the basis for the creation of the most runtime objects. Database objects / classes contain the sub string "Db" in their names. This sub string signals that the corresponding class is associated with information originating from a data file (ODX, Flash Data, etc.). "Db" objects cannot be modified by the client application. An "Db" object is static, have no status and exist only once.

All access is done by methods of the interfaces. There are no attributes expect in the classes which should behave like an enumeration.

The object oriented model of the API has been separated in communication (runtime) and database part, to provide for a data set with minimum redundancy and for a very slim communication layer.

The database part contains all information which could be detected from ODX, as well as environmental settings relevant for the API. Within the database part, all created objects exist only once to avoid unnecessary redundancies. That means that the respective communication objects only have a reference to its database objects, so that the information does not have to be duplicated.

The database objects are static and do not have an internal status. The once created information content may not be modified by the Client or MVCI diagnostic server. As soon as the project has been selected, all information is available. It does not have to be existing within objects at this moment, but the option to access it has to be guaranteed within an acceptable period of time.

The access to database objects is realized optionally via the short name of the object or via iteration within the database structure.

The diagnostic server should generate suitable shortnames for shortnames are not obtainable from ODX. These shortnames should have a prefix "#RtGen_" that uniquely classifies these shortnames as generated.

Figure 14 depicts the project associations.
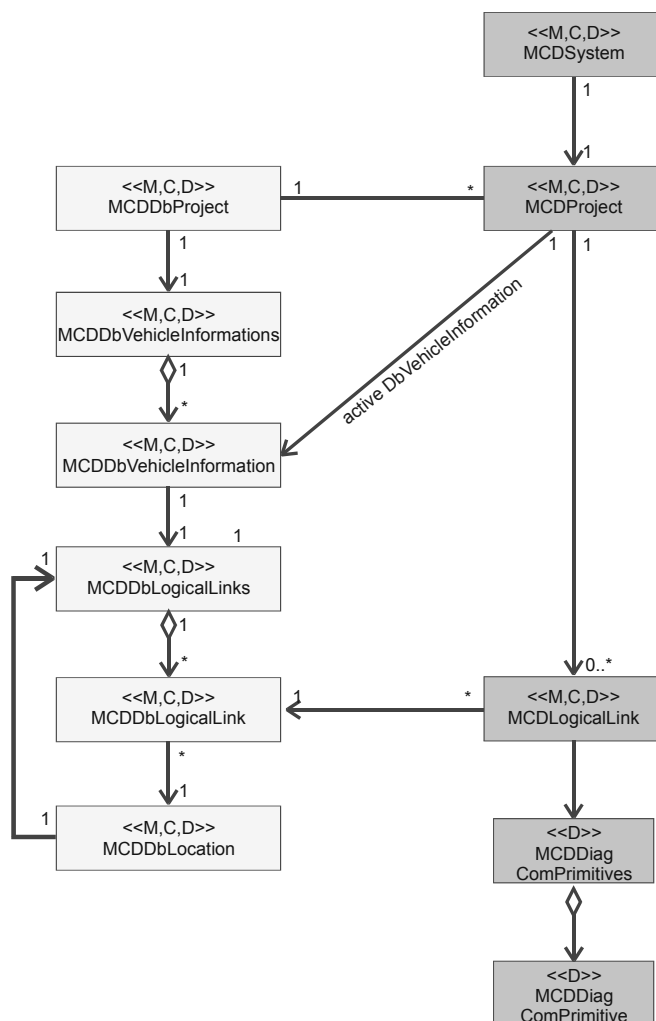
Figure 14 — Project associations

The entry object is the MCDSystem, a runtime object. From this a runtime project will be created on base of it's database project. Selecting the Vehicle Information Table make the access to all database Logical Links possible. This access is necessary to create the runtime Logical Links from which all runtime activities can be done.

The communication part contains all interfaces via which the communication with the ECUs takes place. The majority of the runtime objects is created on the basis of database objects and thus has a direct reference to these. Some objects implementing interfaces of the communication part may be instanced only once (these are unique objects/singletons), others may be instanced multiply (multiple objects). The multiply instanced runtime objects of an identical database object may be parameterised differently.

The runtime objects may be executed (executable), have an internal status and return results.

Table 4 defines the overview RunTimeObjects.

Table 4 — Overview RunTimeObjects

Unique runtime objects /singletons	Multiple runtime objects
`MCDSystem`	`MCDAccessKey`
`MCDProject`	All derived from `MCDDiagComPrimitive`
`MCDJobApi`	`MCDError`
---	`MCDException`
---	All derived from `MCDException`
---	`MCDLogicalLink`
---	`MCDTextTableElement`
---	All derived from `MCDParameter`
---	`MCDResponse`
---	`MCDResult`
---	`MCDResultState`
---	`MCDValue`
---	`MCDRequest`
---	All derived from `MCDDbObject`

7.7.2.2   Structure of the database

The database which can be polled via the Object Model is structured in such a way, that each database element is available only once. Because of this, redundancies are avoided. Connections between the single database elements can be queried via methods. This methods return instances or references.

All database interfaces have methods for the polling of `ShortName, LongName` and `Description`.

A more detailed description can be found with the description of the hierarchic model and the Entity Relationship diagrams.

Database information may be accessed immediately in state `eDBPROJECT_CONFIGURATION` or after the instancing of the RunTime Project.

If there are optional elements not filled in ODX no exception shall be used.

Within the database part templates for all DiagComPrimitives and their parameters, information for ECU (re-) programming and the information concerning the ECUs and the Vehicle Connector Table are deposited.

The meta information of the different communication primitives may be polled by the interfaces derived from `MCDDbDiagComPrimitive`. The objects filled with information from the ODX at the moment of execution are used as template for the runtime communication primitives.

7.7.2.3   Structure of the runtime side

On runtime side the communication actions are carried out by means of DiagComPrimitives, which also include the Services and all three kinds of jobs (FlashJob, SingleEcuJob and MultipleEcuJob). A specific feature of the Diagnostic part is the use of RequestParameters as input for the DiagComPrimitives and the ResultState as Snapshot for the state of the Result.

`MCDDiagComPrimitive` is the superior class for all runtime communication primitives. Communication Primitives represent for example state transitions of the Logical Link (`MCDStartCommunication`) and real communication objects as for example Services and Jobs. Jobs and Services are derived from DataPrimitives and thus they have to be handled alike.

7.7.3   Parent functionality

The parent functionality in objects `MCDError`, `MCDResponseParameter`, `MCDResponseParameters`, `MCDResponse`, `MCDResponses` and `MCDDiagComPrimitive` of the model should give the user the possibility to go through the model not only in top down but also in button up direction. The parent for the object which has the functionality is always the logical parent, e.g. for the DiagComPrimitive is the parent the LogicalLink.

Table 5 defines the deliverd parent.

Table 5 — Get parent functionallity in diagnostic

Class	delivers
MCDResponseParameter	MCDResponseParameters
MCDResponseParameters	MCDResponse or MCDResponseParameters
MCDResponse	MCDResponses
MCDResponses	MCDResult
MCDDiagComPrimitive	MCDLogicalLink
MCDError	MCDResponseParameter or MCDResponse or MCDResult

7.7.4   Entity Relationship Diagrams

7.7.4.1   Objective

Within the Entity Relationship diagrams (ERD) the relations between the single interfaces or the objects to be created from these are visualized.

7.7.4.2   Relation between Vehicle Connector Information Table and Logical Link Table

Figure 15 depicts the DbProject and relation to Logical Link and Vehicle Information Table.
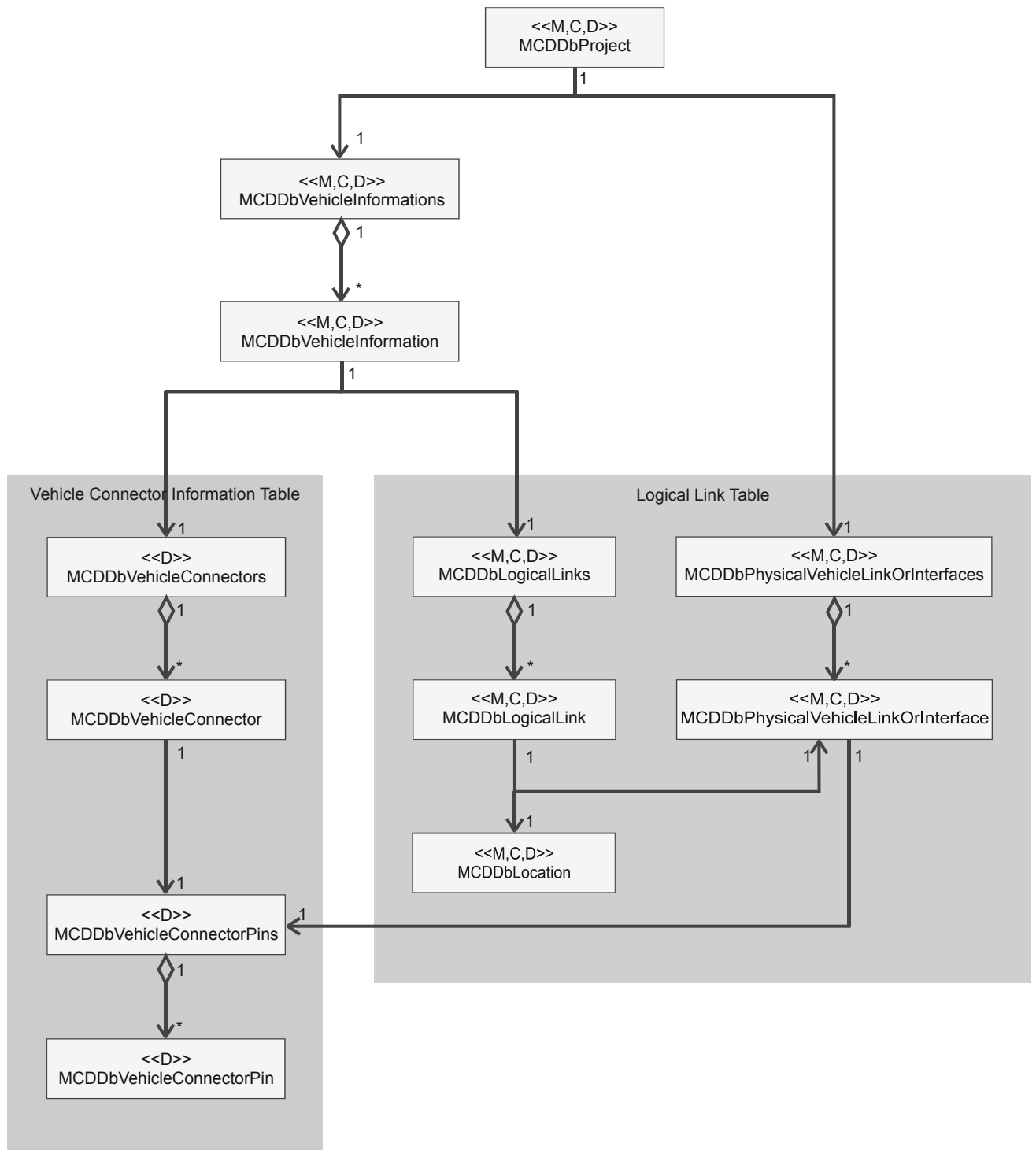
Figure 15 — DbProject and relation to Logical Link and Vehicle Information Table

Within this ERD the relations between the database interface of the project and the ECUs and Vehicle Information contained in the project are shown. It is shown, which information may be listed in collections and in which number the respective objects should be existing within the data base.

In the Logical Link Table each ECU will occur in several Locations if there are different paths to this ECU.

To each Logical Link belong exactly one Location and one Physical Vehicle Link, according to the entry in the Logical Link Table. The Vehicle Connectors and the ConnectorPins according to the Physical Vehicle Link can be listed. Within the Vehicle Information Table the VehicleConnectors with their pin-assignments and the respective Logical Links are listed. As far as the Logical Link is concerned, it references the Physical Vehicle Link which describes the connection between pins and ECU.

Besides the VehicleInformationTables that are defined in ODX, a MCD System may optionally support the concept of runtime generated Vehicle Information Tables.

A runtime generated Vehicle Information Table shall be named with prefix "#RtGen_" and is generated internally by the server. This VIT can be used by a client application like any other VIT. Which DbLogicalLinks will be contained in the VIT is server specific, It could be for example one of the following:

— The VIT contains the collection of all DbLogicalLink which are defined in single VIT's (e.g. #RtGen_SmartVIT).

— The VIT contains the collection of all DbLogicalLinks which can be generated out of the DbLocations of the project (e.g. #RtGen_DefaultVIT).

7.7.4.3   ERD DbLocation

Within this ERD the associations between a DbLocation, which represents an ECU, and the related interfaces are shown. At the DbLocation a number of Collections may be polled, which are filled with items.

Figure 16 depicts the location association for D.
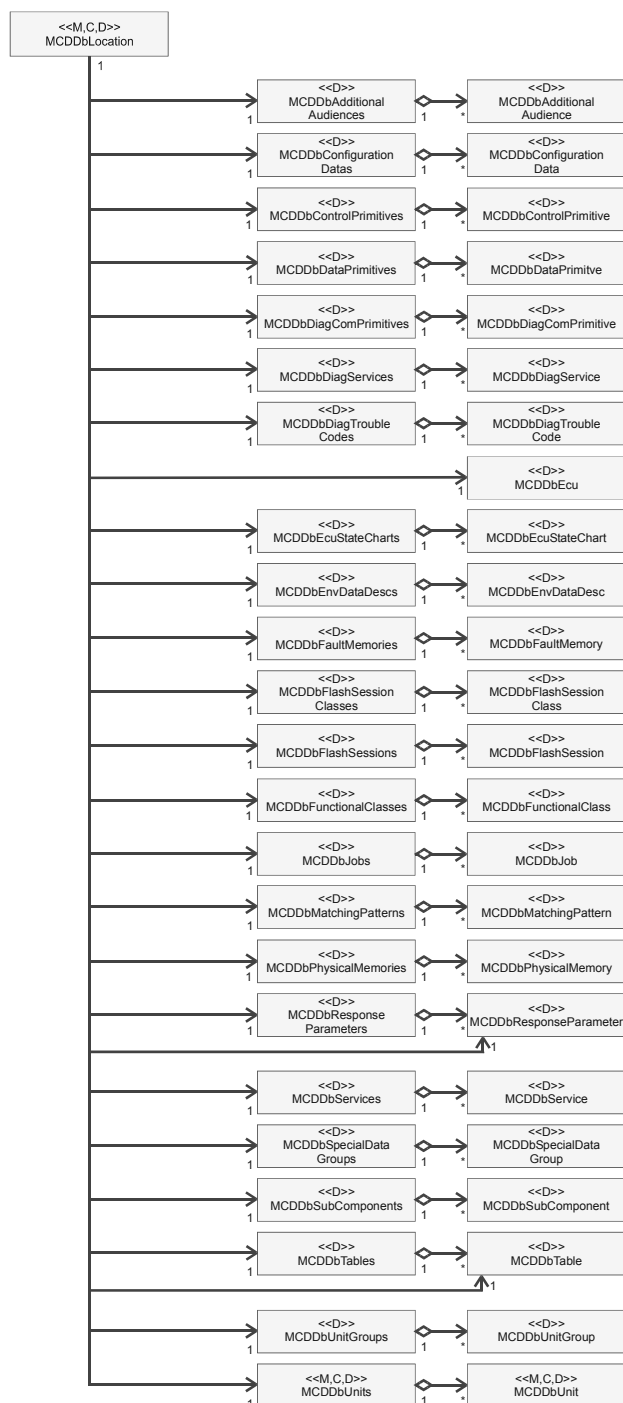
Figure 16 — Location association for D

These associations are created from the information out of ODX description files, whereat `MCDDBLocation` corresponds with ECU.

At this `MCDDBLocation` there is a link to the DbECU Interface and Collections for the different database objects that can be assigned to an ECU. Every Collection contains 0..n objects of the same class or children of a common super class.

7.7.4.4 ERD Logical Link and DiagComPrimitives

Figure 17 depicts the Logical Link associations in function block D.

Figure 17 — Logical Link associations in function block D

Within this Entity Relationship Diagram, the relation between a Logical Link and the Communication Primitives (called DiagComPrimitive) which can be created within it, as well as the relations between the runtime and the database part of the communication primitives is shown.

For each Logical Link can be created 0 to n different DiagComPrimitives at the same time.

The following runtime ControlPrimitives may be created only once per Logical Link:

— MCDStartCommunication,

— MCDStopCommunication,

— MCDVariantIdentification,

— MCDVariantIdentificationAndSelection.

All other DiagComPrimitives may be created as often as you like and exist in parallel. They may be parameterised differently.

Each Runtime DiagComPrimitive of a type uses exactly the database DiagComPrimitive of the same type of this Location. It is not necessary to create a runtime DiagComPrimitive within the Logical Link for each database DiagComPrimitive.

7.7.4.5 ERD Request and Response Parameter associations

Figure 18 depicts the request parameter associations.
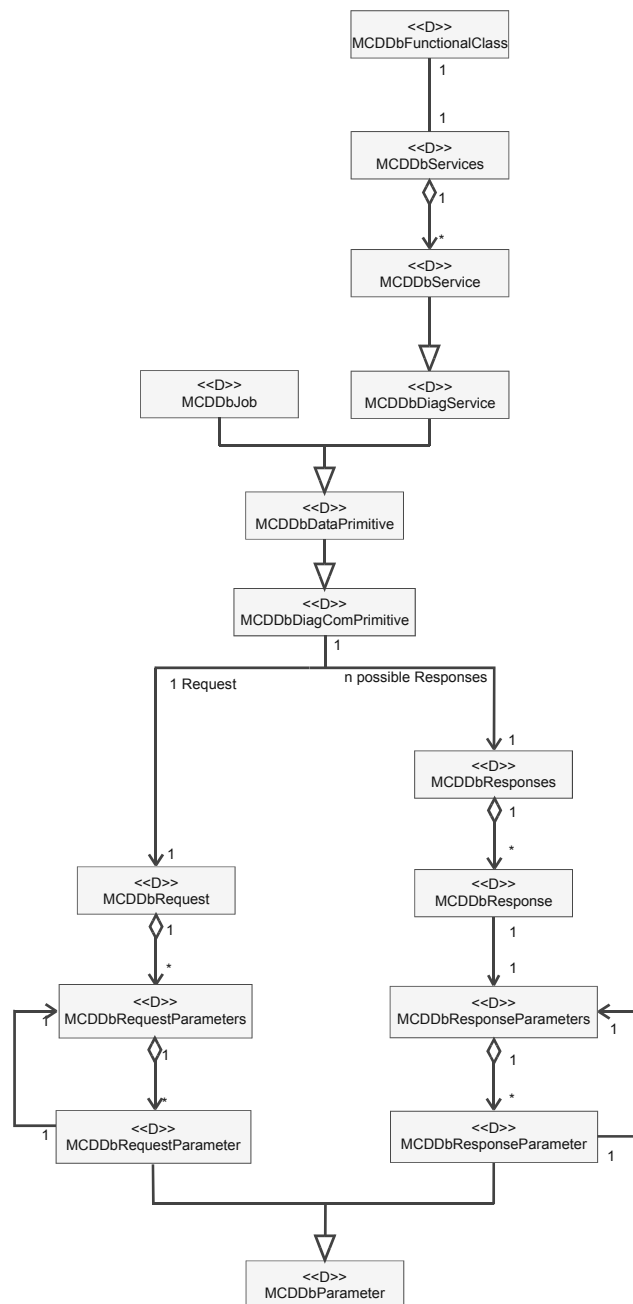
Figure 18 — Request parameter associations

Within this ERD the relations concerning the kinds of parameters of the object model are shown.

Within this ERD the database side is considered in detail. Within the upper part it is shown, that `MCDDbFuncionalClass` contains a collection of Services and that Services belong to the MCDDbDataComPrimitives. To each DataComPrimitive always belong Request Parameters and possibly (that means 0 to as many you like) predefined Responses together with the respective Response Parameters, which describe the general structure of the Response. This is illustrated in Figure 81.

Each Request Parameter may contain a Simple DOP or as Complex DOP a STRUCTURE or EndOfPDU, so it can be build a nested structure. Response Parameter, however, may build any nested structure out of Complex and Simple DOP.

7.7.4.6   ERD result access

Figure 19 depicts the response parameters associations.
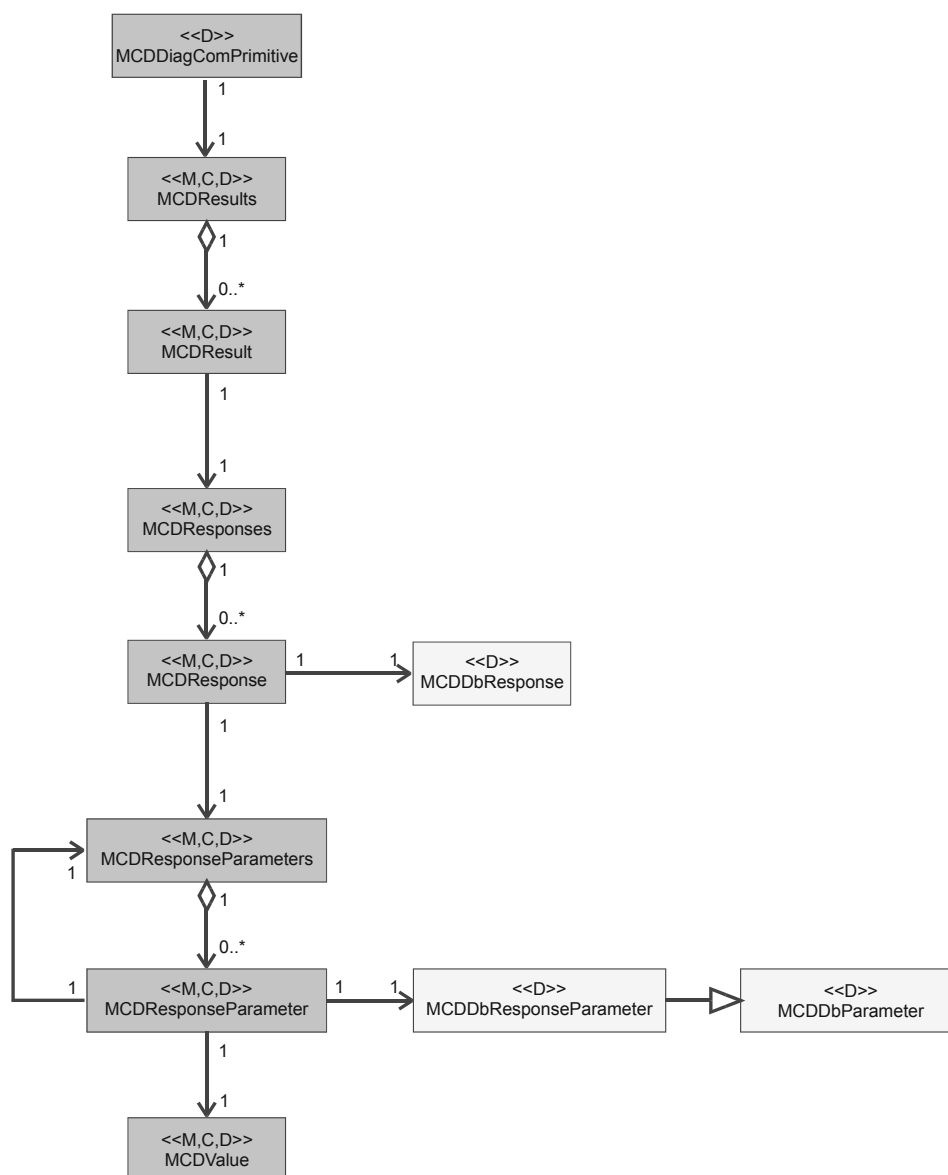
Figure 19 — Response parameters associations

Within this ERD the runtime side of the Result is considered in detail and the relation between DiagComPrimitive, Result, Response and Response Parameter up until the value of the Response Parameter is shown.

This means by calling `fetchResults()` a collection of results is returned. This collection consists of 1..n MCDResults. Each `MCDResult` has for each ECU used by the DiagComPrimitive one response. The response is described in the database with a `MCDDbResponse`. Each `MCDResponse` has a collection of Response Parameters. In the following the Response Parameters can be structured to build the structure of the return values of one ECU. The structure is given by database template.

Figure 80 illustrates the detailed result structure in the D part.

7.7.4.7   ERD Jobs

Within this Entity Relationship Diagram, the relations of the Job interfaces are shown. `MCDDbJob` is derived from `MCDDbDataPrimitive`. `MCDDbMultipleEcuJob`, `MCDDbFlashJob` and `MCDDbSingleEcuJob` are derived from `MCDDbJob`. A relation to the `MCDDbVersion` interface exists.

Figure 20 depicts the Job associations.
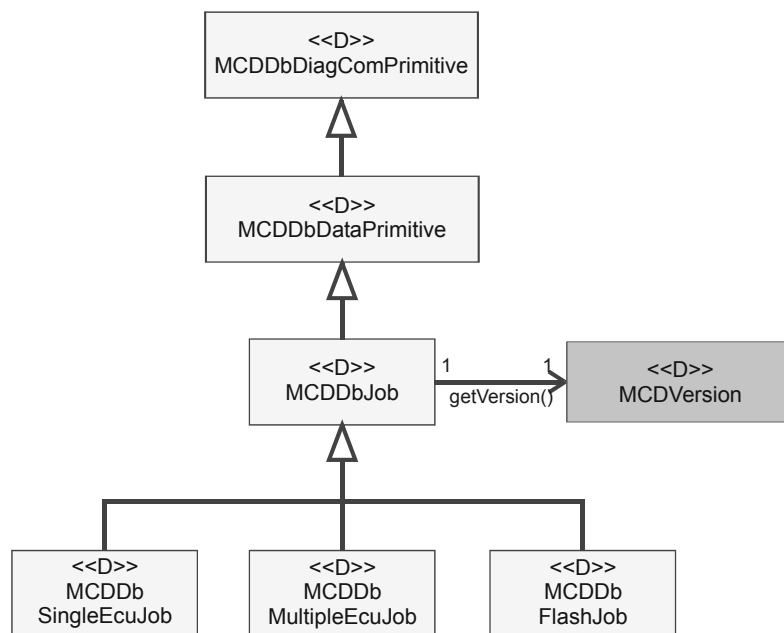
Figure 20 — Job associations

7.7.5   ODX Data Type mapping for database and runtime side

7.7.5.1   Basics

In general runtime data has to be in conformance with the database template for the respective association. If the data type of the parameter is a complex data type, data sections are repeated.

DbObjects are static and will be not be influenced by Run Time side objects. This means that the delivered dbObjects are always the same no matter which access way is used (dbSide or RTSide::`getDbObject`::`getDbParameters`).

For the special complex data type `eTABLE_ROW` the hints are directly included in 8.12.

In general the ODX isVisible attribute is not considered.

Following the complex datatypes

⎯ eSTRUCTURE

⎯ eFIELD, eEND_OF_PDU,

⎯ eMULTIPLEXER and

⎯ eENVDATA, eENVDATADESC are shown in more detail.

7.7.5.2 eSTRUCTURE

A parameter of data type eSTRUCTURE consists of 0 to n elements. Each element can be of simple or complex data type. The parameter collection below the complex eSTRUCTURE parameter contains the MCD(Db)Parameter mappings of ODX PARAM elements that are combined in the ODX STRUCTURE.

Parameters of data type eSTRUCTURE might appear in a request structure as well as in a response structure.

ODX-Data (extract) of the DB-Template

```
<DATA-OBJECT-PROP ID="SimpleDOP_ID">
   <SHORT-NAME>SimpleDOP</SHORT-NAME>
   (…)
   <DIAG-CODED-TYPE (…)>
      (…)
   </DIAG-CODED-TYPE>
   <PHYSICAL-TYPE BASE-DATA-TYPE="A_UNICODE2STRING"/>
</DATA-OBJECT-PROP>
<STRUCTURE ID="SimpleSTRUCT_ID">
   <SHORT-NAME>SimpleSTRUCT</SHORT-NAME>
   <PARAMS>
      <PARAM xsi:type="VALUE">
         <SHORT-NAME>Data_A</SHORT-NAME>
         (…)
         <DOP-REF ID-REF="SimpleDOP_ID"/>
      </PARAM>
      <PARAM xsi:type="VALUE">
         <SHORT-NAME>Data_B</SHORT-NAME>
         (…)
         <DOP-REF ID-REF="SimpleDOP_ID"/>
      </PARAM>
   </PARAMS>
</STRUCTURE>
<DIAG-SERVICE ID="ServiceXYZ_ID">
   <SHORT-NAME>ServiceXYZ</SHORT-NAME>
   (…)
   <REQUEST-REF ID-REF="Req_ID"/>
   (…)
</DIAG-SERVICE>
<REQUEST ID="Req_ID">
   <SHORT-NAME>Req</SHORT-NAME>
```

```
(…)
<PARAMS>
    <PARAM xsi:type="CODED-CONST">
        <SHORT-NAME>SID</SHORT-NAME>
        <DIAG-CODED-TYPE BASE-DATA-TYPE="A_UINT32" (…)>
            (…)
        </DIAG-CODED-TYPE>
    </PARAM>
        <PARAM xsi:type="VALUE">
        <SHORT-NAME>DataItems</SHORT-NAME>
        (…)
        <DOP-REF ID-REF="SimpleSTRUCT_ID"/>
    </PARAM>
</PARAMS>
</REQUEST>
```

Figure 21 depicts the datatype eSTRUCTURE at database and runtime side.
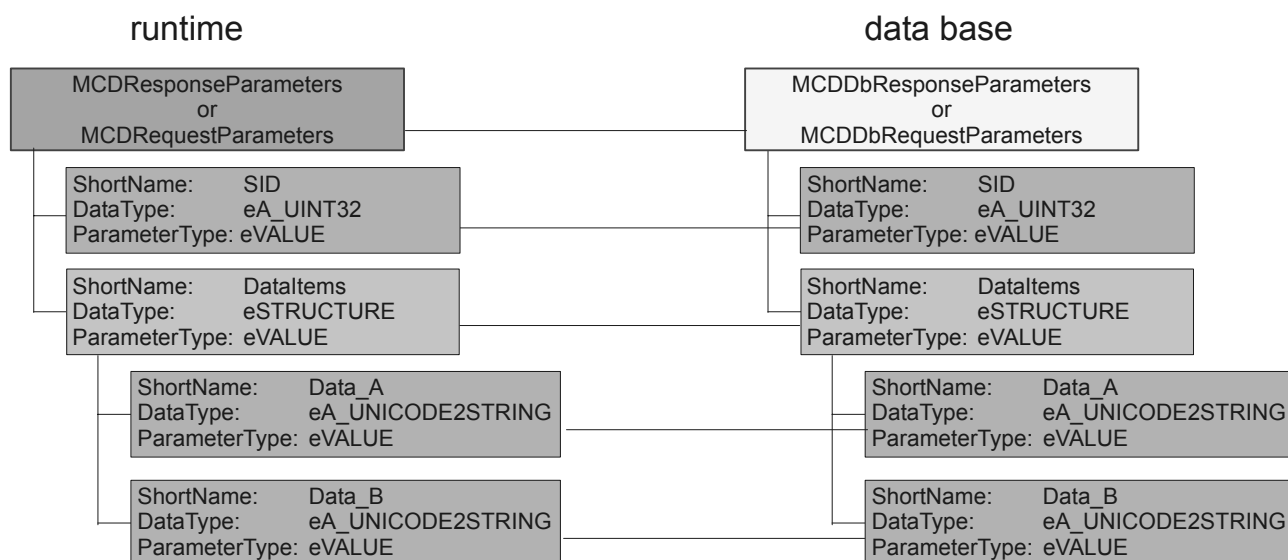
Figure 21 — Datatype eSTRUCTURE at database and runtime side

7.7.5.3 eFIELD

A parameter of data type eFIELD consists of 0 to n elements. Each element contains one element of data type eSTRUCTURE with a short name equal to the SHORT-NAME of the ODX STRUCTURE, this means the STRUCTURE element of ODX will be used independently from the isVisible data. The structure itself consists of arbitrarily simple or complex elements (the actual field content). On database side exactly one field element (eSTRUCTURE with content) is shown. On runtime side the number of included eSTRUCTURE elements depends on how often this structure is contained in the response PDU according to the termination condition of the special field type. Please note, that the short names of the field elements will not be unique.

Parameters of type eFIELD appear in response structures only. They are built from ODX DOPs STATIC-FIELD, DYNAMIC-LENGTH-FIELD, DYNAMIC-ENDMARKER-FIELD, and END-OF-PDU-FIELD.

ODX-Data (Extract) of the DB-Template

```
<DATA-OBJECT-PROP ID="SimpleDOP_ID">
   <SHORT-NAME>SimpleDOP</SHORT-NAME>
   (…)
   <DIAG-CODED-TYPE (…)>
      (…)
   </DIAG-CODED-TYPE>
   <PHYSICAL-TYPE BASE-DATA-TYPE="A_UNICODE2STRING"/>
</DATA-OBJECT-PROP>
<STRUCTURE ID="SimpleSTRUCT_ID">
   <SHORT-NAME>SimpleSTRUCT</SHORT-NAME>
   <PARAMS>
      <PARAM xsi:type="VALUE">
         <SHORT-NAME>Data_A</SHORT-NAME>
         (…)
         <DOP-REF ID-REF="SimpleDOP_ID"/>
      </PARAM>
      <PARAM xsi:type="VALUE">
         <SHORT-NAME>Data_B</SHORT-NAME>
         (…)
         <DOP-REF ID-REF="SimpleDOP_ID"/>
      </PARAM>
   </PARAMS>
</STRUCTURE>
<DYNAMIC-ENDMARKER-FIELD ID="DYN_EM_FIELD_ID">
   <SHORT-NAME>DEMF</SHORT-NAME>
   <BASIC-STRUCTURE-REF ID-REF="SimpleSTRUCT_ID"/>
   (…)
</DYNAMIC-ENDMARKER-FIELD>
<DIAG-SERVICE ID="ServiceXYZ_ID">
   <SHORT-NAME>ServiceXYZ</SHORT-NAME>
   (…)
   <POS-RESPONSE-REF ID-REF="Rsp_ID"/>
   (…)
</DIAG-SERVICE>
<POS-RESPONSE ID="Rsp_ID">
   <SHORT-NAME>Rsp</SHORT-NAME>
   <PARAMS>
      <PARAM xsi:type="CODED-CONST">
         <SHORT-NAME>SID</SHORT-NAME>
         <DIAG-CODED-TYPE BASE-DATA-TYPE="A_UINT32" (…)>
            (…)
         </DIAG-CODED-TYPE>
      </PARAM>
      <PARAM xsi:type="VALUE">
         <SHORT-NAME>RepeatedItems</SHORT-NAME>
         (…)
         <DOP-REF ID-REF="DYN_EM_FIELD_ID">
      </PARAM>
   </PARAMS>
</POS-RESPONSE>
```

Figure 22 depicts the datatype eFIELD at database and runtime side.
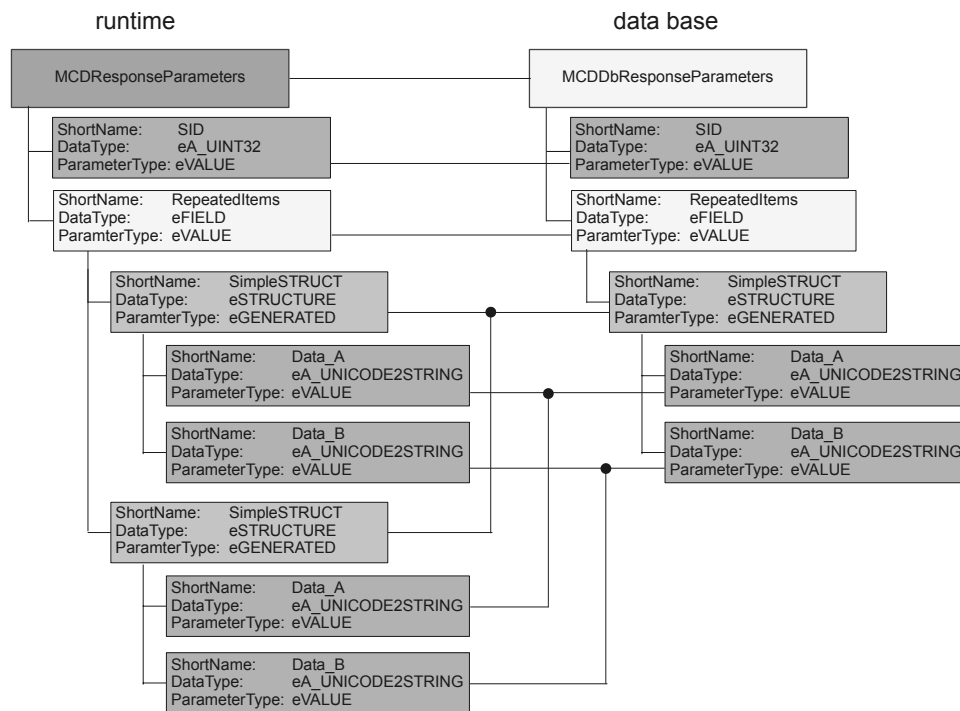
Figure 22 — Datatype eFIELD at database and runtime side

7.7.5.4   eEND_OF_PDU

A parameter of data type eEND_OF_PDU consists of 0 to n elements. Each element contains one element of data type eSTRUCTURE with a short name equal to the SHORT-NAME of the ODX STRUCTURE, this means the STRUCTURE element of ODX will be used independently from the isVisible data. The structure itself consists of arbitrarily simple or complex elements (the actual field content). On database side exactly one field element (eSTRUCTURE with content) is shown. On runtime side the number of included eSTRUCTURE elements depends on how often this structure shall be repeated in the request PDU. Initially, this number is equal to the minimum number of items defined by MIN-NUMBER-OF-ITEMS in ODX. It might be extended calling `MCDRequestParameter::addParameters` until the maximum given by MAX-NUMBER-OF-ITEMS in ODX is reached. Please note, that the short names of the field elements will not be unique.

Parameters of type eEND_OF_PDU appear in request structures only. They are built from ODX END-OF-PDU-FIELD.

ODX-Data (Extract) of the DB-Template

```
<DATA-OBJECT-PROP ID="SimpleDOP_ID">
   <SHORT-NAME>SimpleDOP</SHORT-NAME>
   (…)
   <DIAG-CODED-TYPE (…)>
      (…)
   </DIAG-CODED-TYPE>
   <PHYSICAL-TYPE BASE-DATA-TYPE="A_UNICODE2STRING"/>
</DATA-OBJECT-PROP>
```

```
<STRUCTURE ID="SimpleSTRUCT_ID">
   <SHORT-NAME>SimpleSTRUCT</SHORT-NAME>
   <PARAMS>
      <PARAM xsi:type="VALUE">
         <SHORT-NAME>Data_A</SHORT-NAME>
         (…)
         <DOP-REF ID-REF="SimpleDOP_ID"/>
      </PARAM>
      <PARAM xsi:type="VALUE">
         <SHORT-NAME>Data_B</SHORT-NAME>
         (…)
         <DOP-REF ID-REF="SimpleDOP_ID"/>
      </PARAM>
   </PARAMS>
</STRUCTURE>
<END-OF-PDU-FIELD ID="EOP_FIELD_ID">
   <SHORT-NAME>EOPF</SHORT-NAME>
   <BASIC-STRUCTURE-REF ID-REF="SimpleSTRUCT_ID"/>
   <MAX-NUMBER-OF-ITEMS>4</MAX-NUMBER-OF-ITEMS>
   <MIN-NUMBER-OF-ITEMS>2</MIN-NUMBER-OF-ITEMS>
</END-OF-PDU-FIELD>
<DIAG-SERVICE ID="ServiceXYZ_ID">
   <SHORT-NAME>ServiceXYZ</SHORT-NAME>
   (…)
   <REQUEST-REF ID-REF="Req_ID"/>
   (…)
</DIAG-SERVICE>
<REQUEST ID="Req_ID">
   <SHORT-NAME>Req</SHORT-NAME>
   <PARAMS>
      <PARAM xsi:type="CODED-CONST">
         <SHORT-NAME>SID</SHORT-NAME>
         <DIAG-CODED-TYPE BASE-DATA-TYPE="A_UINT32" (…)>
            (…)
         </DIAG-CODED-TYPE>
      </PARAM>
         <PARAM xsi:type="VALUE">
         <SHORT-NAME>RepeatedItems</SHORT-NAME>
         (…)
         <DOP-REF ID-REF="EOP_FIELD_ID"/>
      </PARAM>
   </PARAMS>
</REQUEST>
```

Figure 23 depicts the datatype eEND_OF_PDU at database and runtime side.
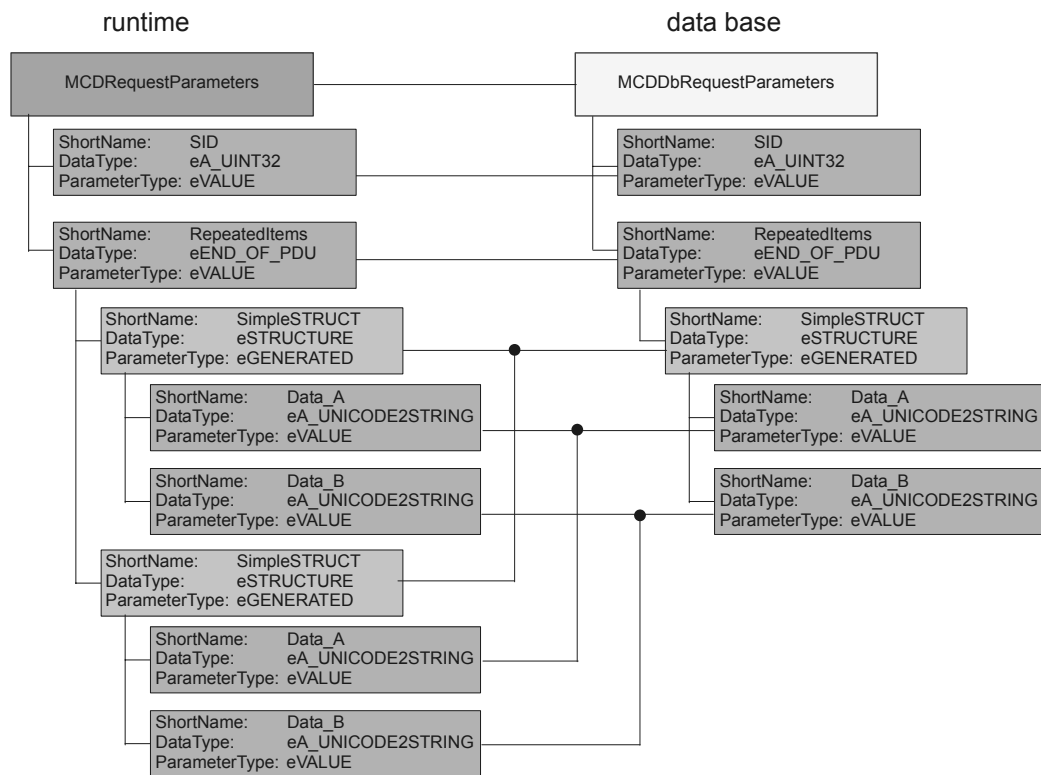
Figure 23 — Datatype eEND_OF_PDU at database and runtime side

7.7.5.5 eMULTIPLEXER

A parameter of data type eMULTIPLEXER consists of 0 to n branches. Each branch contains one element of data type eSTRUCTURE with name equal to CASE SHORT-NAME, this means the STRUCTURE element of ODX will be used independently from isVisible data. The structure itself consists of arbitrarily simple or complex elements. On database side all branches of a multiplexer are described including possibly empty cases which do not reference a STRUCTURE, and the optional default case. On runtime side only one branch is available.

ODX-Data (Extract) of the DB-Template

```
<DATA-OBJECT-PROP ID="SwitchKeyDOP_ID">
   <SHORT-NAME>SwitchKeyDOP</SHORT-NAME>
   (…)
</DATA-OBJECT-PROP>
<DATA-OBJECT-PROP ID="SimpleDOP_ID">
   <SHORT-NAME>SimpleDOP</SHORT-NAME>
   (…)
   <DIAG-CODED-TYPE (…)>
      (…)
   </DIAG-CODED-TYPE>
   <PHYSICAL-TYPE BASE-DATA-TYPE="A_UNICODE2STRING"/>
</DATA-OBJECT-PROP>
```

```
<STRUCTURE ID="DefaultCaseSTRUCT_ID">
   <SHORT-NAME>DefaultCaseSTRUCT</SHORT-NAME>
   <PARAMS>
      <PARAM xsi:type="VALUE">
         <SHORT-NAME>DefaultData_A</SHORT-NAME>
         (…)
         <DOP-REF ID-REF="SimpleDOP_ID"/>
      </PARAM>
      <PARAM xsi:type="VALUE">
         <SHORT-NAME>DefaultData_B</SHORT-NAME>
         (…)
         <DOP-REF ID-REF="SimpleDOP_ID"/>
      </PARAM>
   </PARAMS>
</STRUCTURE>
<STRUCTURE ID="Case1STRUCT_ID">
   <SHORT-NAME>Case1STRUCT</SHORT-NAME>
   <PARAMS>
      <PARAM xsi:type="VALUE">
         <SHORT-NAME>Case1Data_A</SHORT-NAME>
         (…)
         <DOP-REF ID-REF="SimpleDOP_ID"/>
      </PARAM>
      <PARAM xsi:type="VALUE">
         <SHORT-NAME>Case1Data_B</SHORT-NAME>
         (…)
         <DOP-REF ID-REF="SimpleDOP_ID"/>
      </PARAM>
      <PARAM xsi:type="VALUE">
         <SHORT-NAME>Case1Data_C</SHORT-NAME>
         (…)
         <DOP-REF ID-REF="SimpleDOP_ID"/>
      </PARAM>
   </PARAMS>
</STRUCTURE>
<STRUCTURE ID="Case2STRUCT_ID">
   <SHORT-NAME>Case2STRUCT</SHORT-NAME>
   <PARAMS>
      <PARAM xsi:type="VALUE">
         <SHORT-NAME>Case2Data</SHORT-NAME>
         (…)
         <DOP-REF ID-REF="SimpleDOP_ID"/>
      </PARAM>
   </PARAMS>
</STRUCTURE>
<MUX ID="MUX_ID">
   <SHORT-NAME>Multiplexer</SHORT-NAME>
   (…)
   <SWITCH-KEY>
      (…)
      <DATA-OBJECT-PROP-REF ID-REF="SwitchKeyDOP_ID"/>
   </SWITCH-KEY>
   <DEFAULT-CASE>
      <SHORT-NAME>DefaultCase</SHORT-NAME>
```

```
                <STRUCURE-REF ID-REF="DefaultCaseSTRUCT_ID"/>
         </DEFAULT-CASE>
         <CASES>
             <CASE>
                 <SHORT-NAME>Case1</SHORT-NAME>
                 <STRUCTURE-REF ID-REF="Case1STRUCT_ID"/>
                 (…)
             </CASE>
             <CASE>
                 <SHORT-NAME>Case2</SHORT-NAME>
                 <STRUCTURE-REF ID-REF="Case2STRUCT_ID"/>
                 (…)
             </CASE>
             <CASE>
                 <SHORT-NAME>EmptyCase3</SHORT-NAME>
                 (…)
             </CASE>
         </CASES>
    </MUX>
    <DIAG-SERVICE ID="ServiceXYZ_ID">
        <SHORT-NAME>ServiceXYZ</SHORT-NAME>
        (…)
        <POS-RESPONSE-REF ID-REF="Rsp_ID"/>
        (…)
    </DIAG-SERVICE>
    <POS-RESPONSE ID="Rsp_ID">
        <SHORT-NAME>Rsp</SHORT-NAME>
        <PARAMS>
            <PARAM xsi:type="CODED-CONST">
                <SHORT-NAME>SID</SHORT-NAME>
                <DIAG-CODED-TYPE BASE-DATA-TYPE="A_UINT32" (…)>
                    (…)
                </DIAG-CODED-TYPE>
            </PARAM>
            <PARAM xsi:type="VALUE">
                <SHORT-NAME>MuxItems</SHORT-NAME>
                (…)
                <DOP-REF ID-REF="MUX_ID"/>
            </PARAM>
        </PARAMS>
    </POS-RESPONSE>
```

Figure 24 depicts the datatype eMULTIPLEXER at database and runtime side.

Figure 24 — Datatype eMULTIPLEXER at database and runtime side

7.7.5.6   eENVDATADESC

A database parameter of data type eENVDATADESC contains 0 items.

A runtime parameter of data type eENVDATADESC contains 0 to 2 elements of data type eENVDATA. The parameter collection of the eENVDATADESC parameter is empty, if the response PDU does not contain any environment data. Otherwise, the collection contains 0 or 1 common environment data parameter according to the optional ODX ALL-VALUE of the ODX ENV-DATA-DESC followed by 0 or 1 environment data parameter for a specific diagnostic trouble code.

ODX-Data (extract) of the DB-Template

```
<DATA-OBJECT-PROP ID="SimpleDOP_ID">
   <SHORT-NAME>SimpleDOP</SHORT-NAME>
   (…)
   <PHYSICAL-TYPE BASE-DATA-TYPE="A_UINT32"/>
</DATA-OBJECT-PROP>
<ENV-DATA-DESC ID="EnvDataDesc_ID">
   <SHORT-NAME>EnvDataDesc</SHORT-NAME>
   <PARAM-SNREF SHORT-NAME="SwitchKeyDTC"/>
   <ENV-DATAS>
      <ENV-DATA ID="EnvData_DTC_210_and_220_ID"
         <SHORT-NAME>EnvData_DTC_210_and_220</SHORT-NAME>
         <PARAMS>
            <PARAM xsi:type="VALUE">
               <SHORT-NAME>EngineSpeed</SHORT-NAME>
               (…)
               <DOP-REF ID-REF="SimpleDOP_ID"/>
            </PARAM>
            <PARAM xsi:type="VALUE">
               <SHORT-NAME>Voltage</SHORT-NAME>
               (…)
               <DOP-REF ID-REF="SimpleDOP_ID"/>
            </PARAM>
         </PARAMS>
         <DTC-VALUES>
            <DTC-VALUE>210</DTC-VALUE>
            <DTC-VALUE>220</DTC-VALUE>
         </DTC-VALUES>
      </ENV-DATA>
      <ENV-DATA>
         <SHORT-NAME>EnvData_All</SHORT-NAME>
         <PARAMS>
            <PARAM xsi:type="VALUE">
               <SHORT-NAME>EventCounter</SHORT-NAME>
               (…)
               <DOP-REF ID-REF="SimpleDOP_ID"/>
            </PARAM>
         </PARAMS>
         <ALL-VALUE/>
      </ENV-DATA>
   </ENV-DATAS>
</ENV-DATA-DESC>
<DIAG-SERVICE ID="ServiceXYZ_ID">
   <SHORT-NAME>ServiceXYZ</SHORT-NAME>
   (…)
   <POS-RESPONSE-REF ID-REF="Rsp_ID">
   (…)
</DIAG-SERVICE>
<POS-RESPONSE ID="Rsp_ID">
   <SHORT-NAME>Rsp</SHORT-NAME>
   <PARAMS>
      <PARAM xsi:type="CODED-CONST">
         <SHORT-NAME>SID</SHORT-NAME>
         <DIAG-CODED-TYPE BASE-DATA-TYPE="A_UINT32" (…)>
```

```
            (…)
        </DIAG-CODED-TYPE>
    </PARAM>
    <PARAM xsi:type="VALUE">
        <SHORT-NAME>SwitchKeyDTC</SHORT-NAME>
        (…)
        <DOP-REF ID-REF="SimpleDOP_ID"/>
    </PARAM>
    <PARAM>
        <SHORT-NAME>EnvData</SHORT-NAME>
        <DOP-REF ID-REF="EnvDataDesc_ID"/>
    </PARAM>
  </PARAMS>
</POS-RESPONSE>
```

Figure 25 depicts the datatype eENVDATADESC / eENVDATA at database and runtime side.
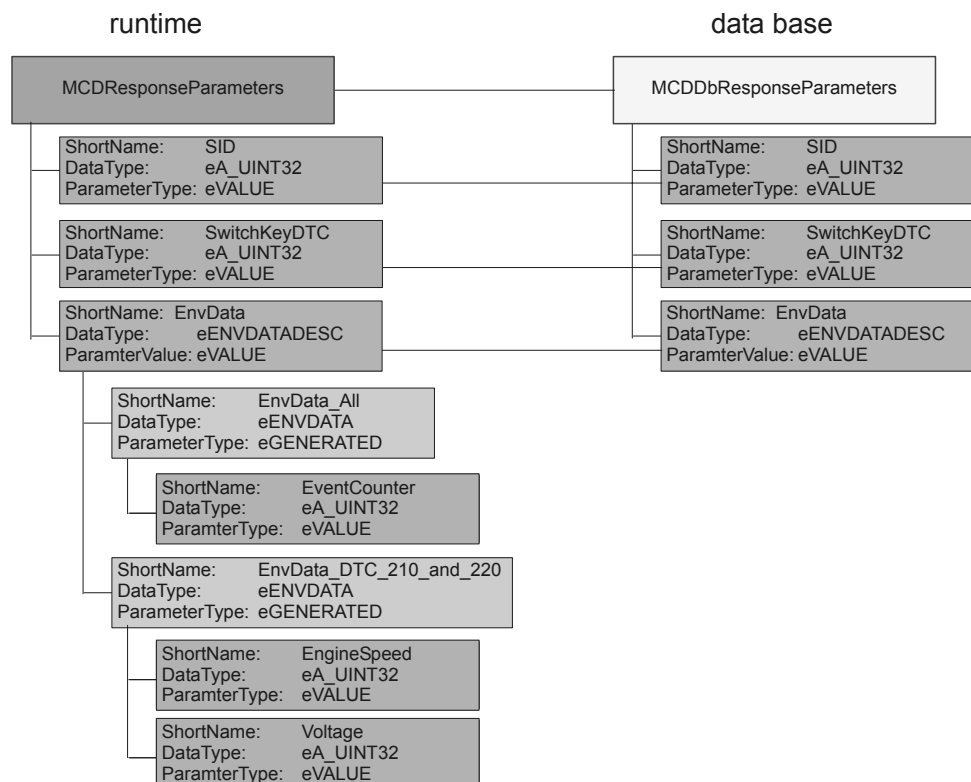
Figure 25 — Datatype eENVDATADESC / eENVDATA at database and runtime side

7.8   Collections

7.8.1   Types and methods

A Collection is a collection of single elements. The ordering criteria are not defined by the MVCI diagnostic server specification. Single objects within the Collection may be accessed by means of an Index. The Index

may also be used to iterate over all elements of the Collection. The counting in collections starts with zero. Anyway, the Collection offers the possibility to query the number of elements currently located within the Collection. The Interface `NamedCollection` has been derived from the general `Collection`. `NamedCollections` additionally offer the possibility to poll a list with the names of all elements currently located within the Collection and to access an element via its name. The kind of elements stored within a Collection can be determined from the InterfaceName. The methods `getItemByIndex` and `getItemByName` are only declared within the specific Interfaces and by this way are declared type safe.

Figure 26 depicts the principle of collections.
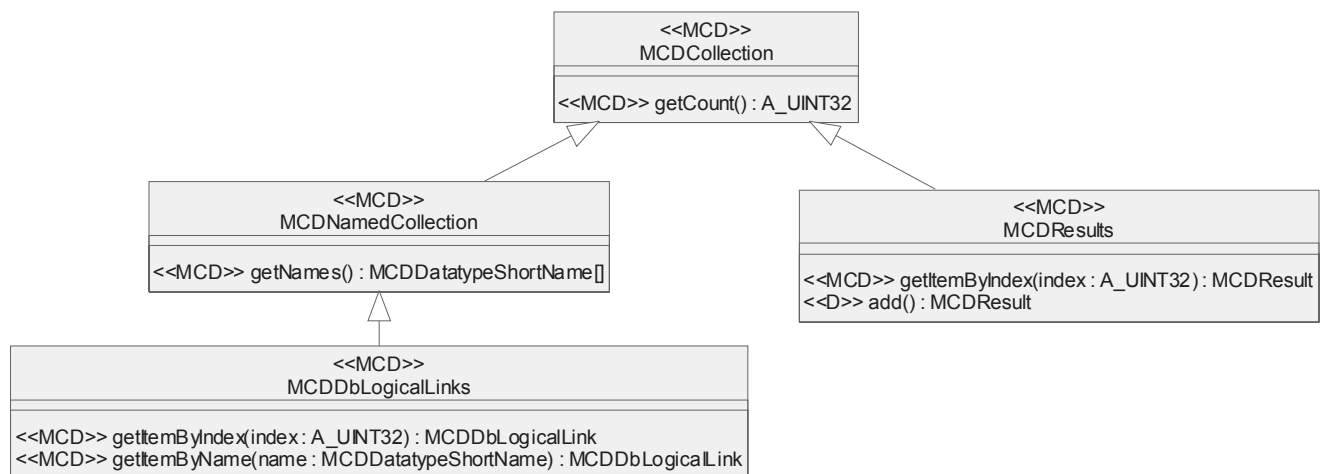
Figure 26 — Principle of collections

7.8.2 RunTime collections

Figure 27 depicts the RunTime collections MCD.
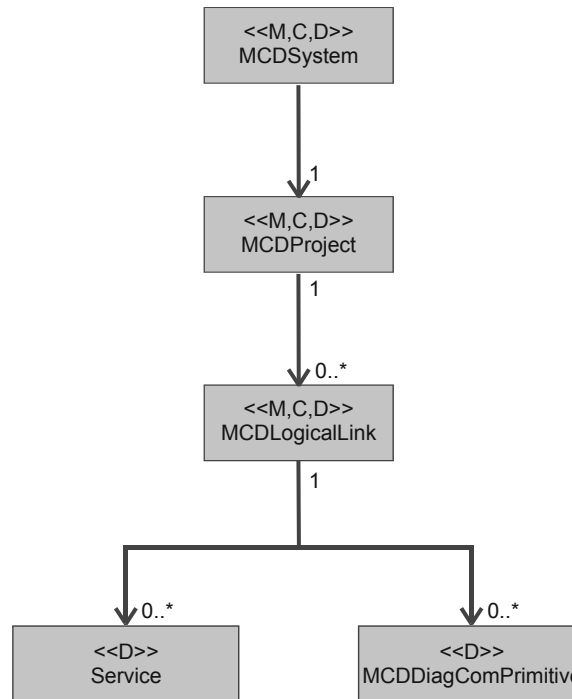
Figure 27 — RunTime collections MCD

Within this ERD the associations between `MCDProject` and its Logical Links are shown. Also the relations of the Logical Link to the DiagComPrimitives and Services are visible.

Logical Links and DiagComPrimitives (incl. Services and Jobs) do not form a Collection that can be polled via the API.

Only the Results and their sub-structures are mapped as Runtime collections. In case of these collections there is no danger that names are assigned twice, as they are created by the server or Job.

7.8.3 Database collections

Collections are used for the listing of database objects with identical interface. The Collections reflect the content of the database of the used project. The database is static; that means nothing can be added or modified. The collections are always derived from `MCDNamedCollection`, so that its items may be accessed via index and via name. It has to be guaranteed for the uniqueness of the names of the items within the data base.

Figure 28 depicts the database collections part I.

```
○ MCDObject
└─○ MCDCollection
      ─Ⓓ MCDDbComponentConnectors
      ─Ⓓ MCDDbEcuStateTransitionActions
      ─Ⓓ MCDDbInterfaceConnectorPins
      ─Ⓓ MCDDbItemValues
      ─Ⓓ MCDDbMatchingParameters
      ─Ⓓ MCDDbMatchingPatterns
      ─Ⓓ MCDDbPreconditionDefinitions
      ─Ⓓ MCDDbSpecialDataGroups
      └─○ MCDNamedCollection
            ─Ⓓ MCDDbAdditionalAudiences
            ─Ⓓ MCDDbCodeInformations
            ─Ⓓ MCDDbConfigurationDatas
            ─Ⓓ MCDDbConfigurationRecords
            ─Ⓓ MCDDbControlPrimitives
            ─Ⓓ MCDDbDataPrimitives
            ─Ⓓ MCDDbDataRecords
            ─Ⓓ MCDDbDiagComPrimitives
            ─Ⓓ MCDDbDiagServices
            ─Ⓓ MCDDbDiagTroubleCodeConnectors
            ─Ⓓ MCDDbDiagTroubleCodes
            ─Ⓓ MCDDbEcuBaseVariants
            ─Ⓓ MCDDbEcuMems
            ─Ⓓ MCDDbEcuStateCharts
            ─Ⓓ MCDDbEcuStates
            ─Ⓓ MCDDbEcuStateTransitions
            ─Ⓓ MCDDbEcuVariants
            ─Ⓓ MCDDbEnvDataConnectors
            ─Ⓓ MCDDbEnvDataDescs
            ─Ⓓ MCDDbFaultMemories
            ─Ⓓ MCDDbFlashChecksums
            ─Ⓓ MCDDbFlashDataBlocks
            ─Ⓓ MCDDbFlashFilters
            ─Ⓓ MCDDbFlashIdents
            ─Ⓓ MCDDbFlashSecurities
            ─Ⓓ MCDDbFlashSegments
            ─Ⓓ MCDDbFlashSessionClasses
            ─Ⓓ MCDDbFlashSessions
            ─Ⓓ MCDDbFunctionalClasses
            ─Ⓓ MCDDbFunctionalGroups
            ─Ⓓ MCDDbFunctionDiagComConnectors
            ─Ⓓ MCDDbFunctionDictionaries
            ─Ⓓ MCDDbFunctionInParameters
            ─Ⓓ MCDDbFunctionNodeGroups
            ─Ⓓ MCDDbFunctionNodes
            ─Ⓓ MCDDbFunctionOutParameters
            ─Ⓓ MCDDbInterfaceCables
            ─Ⓓ MCDDbJobs
            ─Ⓜⅽᵈ MCDDbLocations
            ─Ⓜⅽᵈ MCDDbLogicalLinks
            ─Ⓓ MCDDbObjectItems
            ─Ⓓ MCDDbOptionItems
            ─Ⓓ MCDDbParameters
            ─Ⓓ MCDDbPhysicalMemories
```

Figure 28 — Database collections part I

Figure 29 depicts the database collections part II.


```
MCDDbPhysicalSegments
MCDDbPhysicalVehicleLinkOrInterfaces
MCDDbProjectDescriptions
MCDDbRequestParameters
MCDDbResponseParameters
MCDDbResponses
MCDDbServices
MCDDbSubComponentParamConnectors
MCDDbSubComponents
MCDDbSystemItems
MCDDbTableParameters
MCDDbTableRowConnectors
MCDDbTables
MCDDbUnitGroups
MCDDbUnits
MCDDbVehicleConnectorPins
MCDDbVehicleConnectors
MCDDbVehicleInformations
```

Figure 29 — Database collections part II

7.9 Registering/deregistering of the EventHandler

For each Interface that may use an EventHandler, two methods have to be implemented:

Registering: `setEventHandler (handler.MCDEventHandler): A_UINT32`,

Deregistering: `releaseEventHandler (Id: A_UINT32): void`.

Because of this kind of registering several EventHandler may be connected per object at the same time. These are identified by means of the Id that is assigned at registration.

If the server cannot register another EventHandler because of internal restrictions, e.g. resources or exceeding MAX A_UINT32 (2^{32}) EventHandlers, an `MCDProgramViolationException` will be thrown. It contains the error `eSYSTEM_RESOURCE_OVERLOAD`.

To meet the requirement that the EventHandling has to be also fully functional with only one EventHandler, all methods for the Event reception are expected within each EventHandler. It is up to the Client–Implementation how many EventHandler will be registered and if these EventHandler each implement all methods for the event handling or use some of them only as rudimentary data sinks. It is only required that each registered EventHandler provides for all methods. The functionality derived from the methods is implementation specific.

The EventHandler may be registered at different objects that take a key position of the run time process, for example `MCDSystem` and `MCDLogicalLink`. If for the Logical Link an Event shall be sent to the EventHandler, it is only sent to the EventHandler which is registered at the Logical Link. If no EventHandler is registered at the Logical Link, the Event is sent to the EventHandler of the `MCDSystem`. It is not sent to both EventHandlers, if there are EventHandler registered at both objects.

If one or more event handler are registered at least one of them has to be registered at the `MCDSystem` since all events not handled before are finally sent to the `MCDSystem`.

The registering of an EventHandler at the system is optional. The EventHandler at the MCDSystem also may receive all messages as only EventHandler as long as no other EventHandler is registered.

Figure 30 depicts the using of EventHandler.
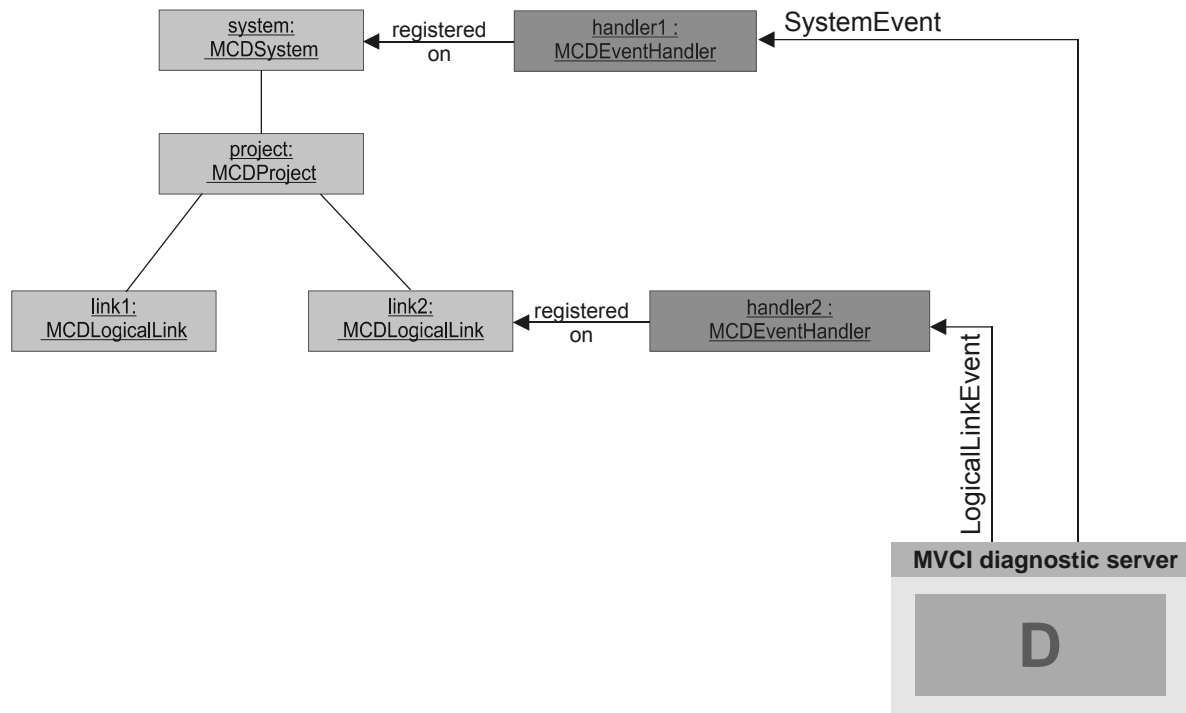
Figure 30 — Using of EventHandler

The `releaseEventHandler` method can be called in any state without throwing an exception (Implementation hint to the client programmer: The `releaseEventHandler` method should be called in the same states where the corresponding `setEventHandler` had been called).

7.10 MCD value

`CreateValue()` returns a reference to a MCDValue object that has been attached to the class/object at which the `createValue()` method has been called. Note that this behaviour is identical for all other `createXXX`-methods (e.g. `createException()`, `createError()`, `createValue()`, `createResult()`, `createLogicalLink`). If there are different calls to the same server object, only new references to same instantiated object are delivered. The object is initialised with the default value stated in the corresponding database template.

`GetValue()` returns a copy of the MCDValue object that has been attached to the class/object the `getValue()` method has been called at. If there is no MCDValue object available, the `getValue()`-method should throw an exception.

`SetValue(MCDValue)` takes a MCDValue object and overwrites the MCDValue object (if present) that is attached to the class/object the method `setValue()` has been called.

Figure 31 depicts the behaviour of MCDRequestParameter (create / get and set value methods).
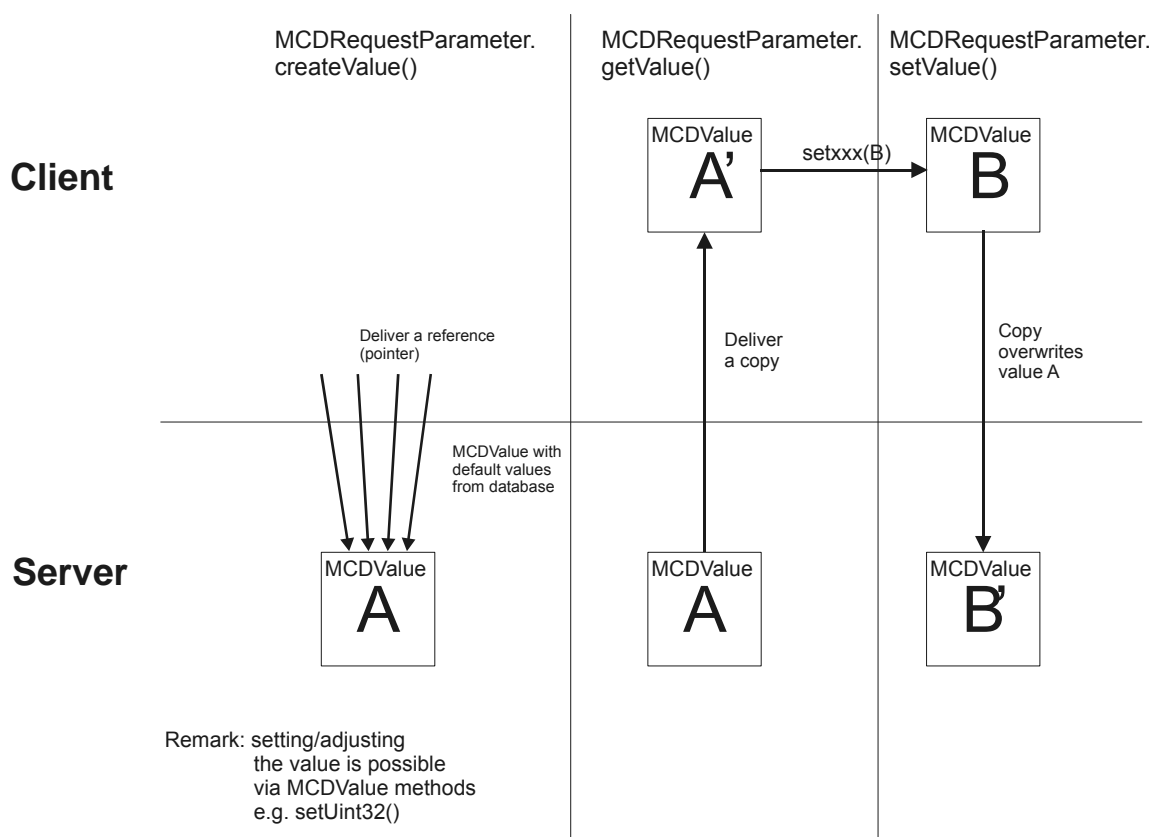
Figure 31 — Behaviour of MCDRequestParameter (create / get and set value methods)

Job INPUT- and OUTPUT PARAMs have references to a DOP-BASE. However DOPs are applied differently from services:

The conversion method is ignored, meaning an input parameter is passed to the job as-is and always as a physical value. PHYSICALDEFAULT- VALUE shall only be used, if INPUT-PARAM references a DATA-OBJECTPROP.

Output parameters are returned by the job as physical values. The physical values returned by the job shall comply with the PHYSICAL-TYPE specification of the associated DOP-BASE.

Special cases are input parameters that have a TEXTTABLE defined as COMPUMETHOD. Here, the valid texts of the TEXTTABLE can be listed within an application. Upon selection, the corresponding text (the physical value) is passed to the job. Hence, the COMPU-METHOD is only used for comfort reasons not for actual conversion of the input parameter from a physical into a coded type. By consequence, the DIAG-CODED-TYPE of the DOP associated by a job parameter (regardless of whether it is an INPUT- or OUTPUT-PARAM) is ignored.

Figure 32 depicts the behaviour of MCDResponseParameter (get value method).

MCDResponseParameter.
getValue()

Client

MCDValue

A'

Deliver
a copy

Server

MCDValue

A
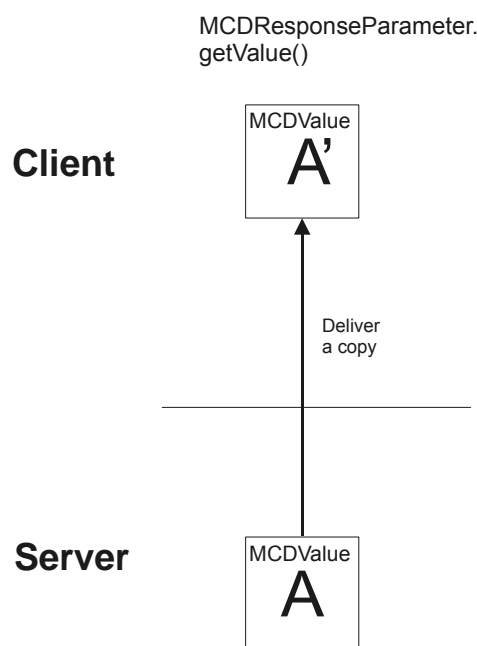
Figure 32 — Behaviour of MCDResponseParameter (get value method)

Figure 33 depicts the MCD Values in jobs.

copy overwrites value

<<M,C,D>>
MCDResponseParameter

① MCDResponseParameter.getType()

② MCDJobAPI.createValue( )

<<M,C,D>>
MCDValue

③ MCDValue.setxxx( )   xxx = Asciistring
                              Bitfield
                              Boolean
                              Bytefield
                              Float32
                              Float64
                              Int8
                              Int16
                              Int32
                              Int64
                              UInt8
                              UInt16
                              UInt32
                              UInt64
                              Unicode2string

④ MCDResponseParameter.setValue( )
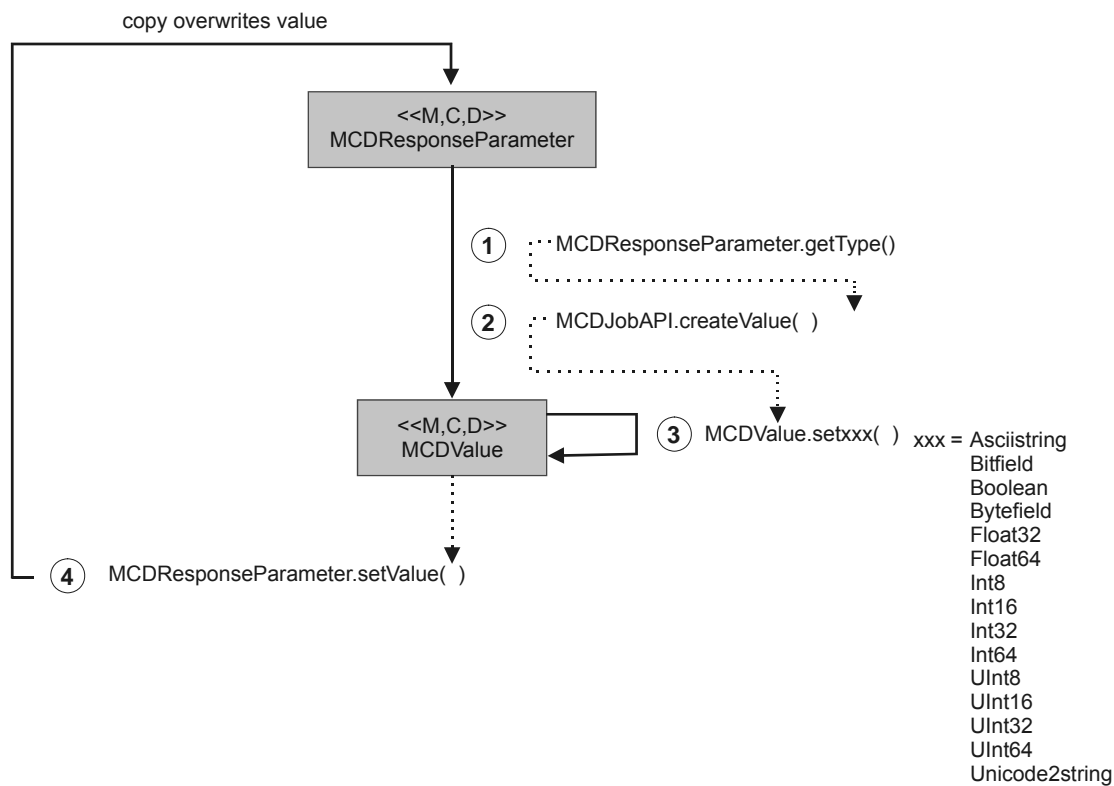
Figure 33 — MCD Values in jobs

7.11 Use cases

7.11.1 View

With the help of Sequence Diagrams the interactive use of the API and the sequences for certain general cases are presented in chronological order.

It is acted from the view of a single client and it is assumed that each action is executed successfully.

7.11.2 Instantiation of projects

Figure 34 depicts the starting work with the MVCI diagnostic server.
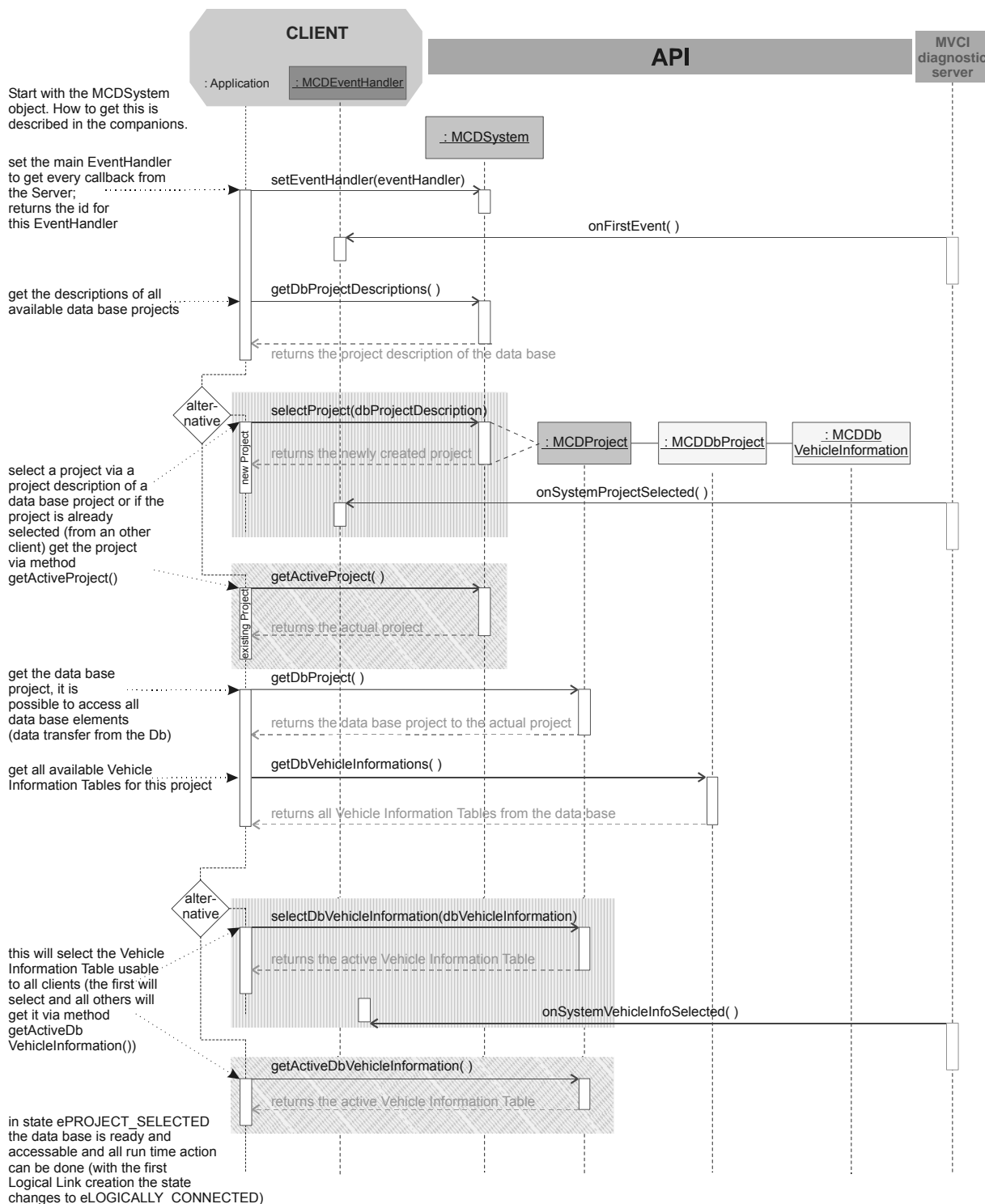
Figure 34 — Starting work with the MVCI diagnostic server

As an entry into the API the Client gets the `MCDSystem` object from the diagnostic server. How this has to be done for the separate mappings of the object model for different programming languages and platforms (ASAM GDI, Java, COM/DCOM, C++) has to be described within the respective documents mapping the MVCI diagnostic server D object model.

The object `MCDSystem` is handed over within the state `eINITIALIZED`. First register the main EventHandler to get all events delivered by the diagnostic server. Now the Client may poll the collection of all projects descriptions available within the data base. In the next step the Client will select one of these projects and thus provide for access to the database contents from the ODX with Vehicle Information Table, Logical Link Table and environmental variables for this project. After this, the MCDSystem object is within the state `ePROJECT_SELECTED`. If another Client has already selected a project this Client ask for the active project, because only one project can be active at one time.

Now, out of the project the desired Vehicle Information Table is selected from the collection of available Vehicle Information Tables on the depending database project. Within this table all `MCDDbLogicalLink` objects are located, which are necessary for the connection to the ECU s.

Alternatively to the procedure mentioned above, the Client might skip the polling for information from the data base, if the short names of all necessary objects are already known. This possibility is shown in the following diagram.

Figure 35 depicts the starting work with the MVCI diagnostic server (via short name).
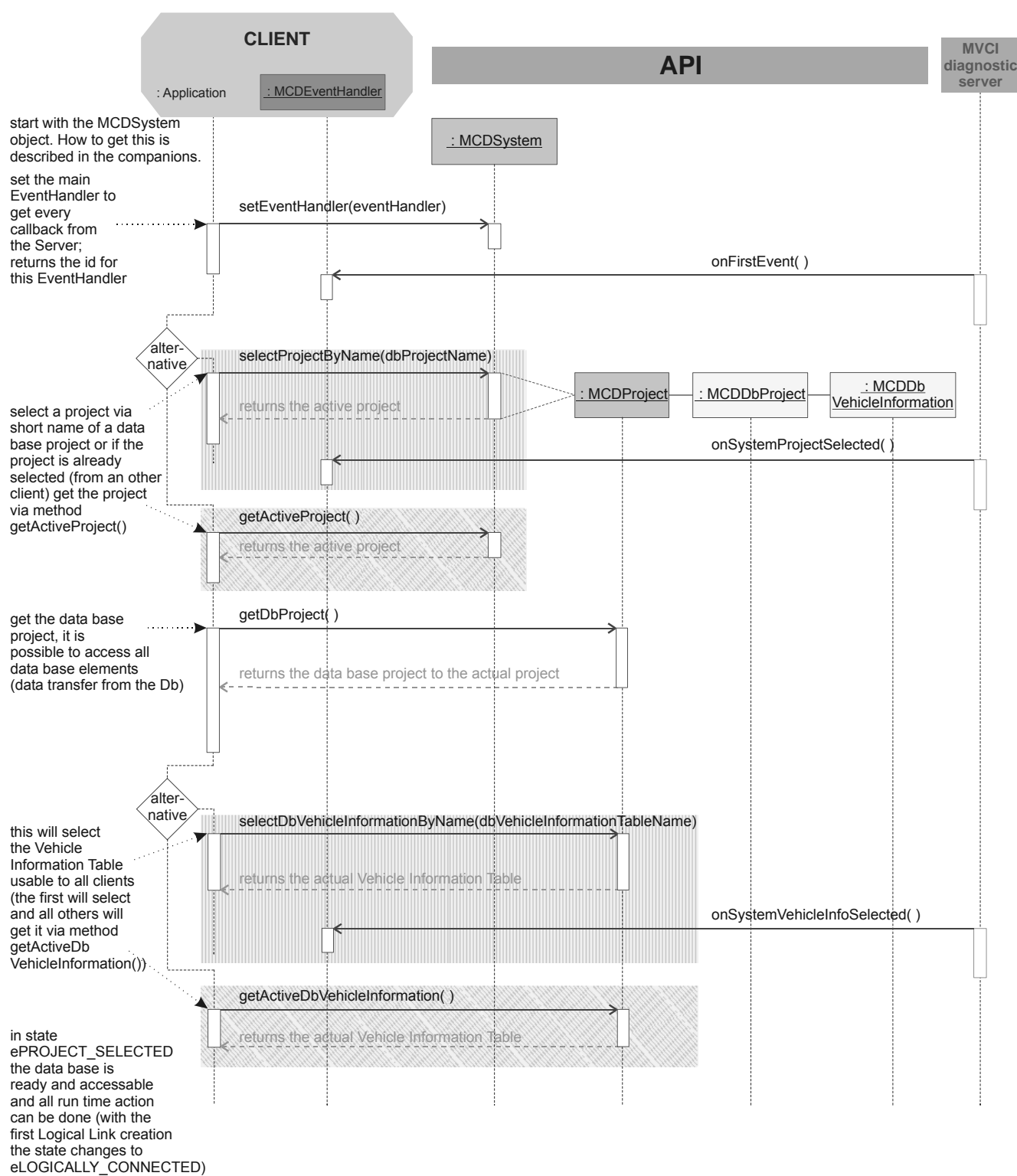
Figure 35 — Starting work with MVCI diagnostic server (via short name)

7.11.3 Database access

For the `MCDDbLogicalLink` selected out of the Logical Link collection, the information stored in the Logical Link Table can be accessed: the ShortName, the Physical Vehicle Link or Interface (and it's type) and from the corresponding DbLocation the AccessKey.

Depending on the type of the DbLocation there are several collections filled with database elements like Services and DiagComPrimitives. Also the information about the Gateway property from the Logical Link Table can be accessed too.

This sequence diagram shows only the access to the top most elements of the database. Which are the elements below that is shown in the corresponding ERD diagrams (see Figure 16).

Figure 36 depicts the database access.
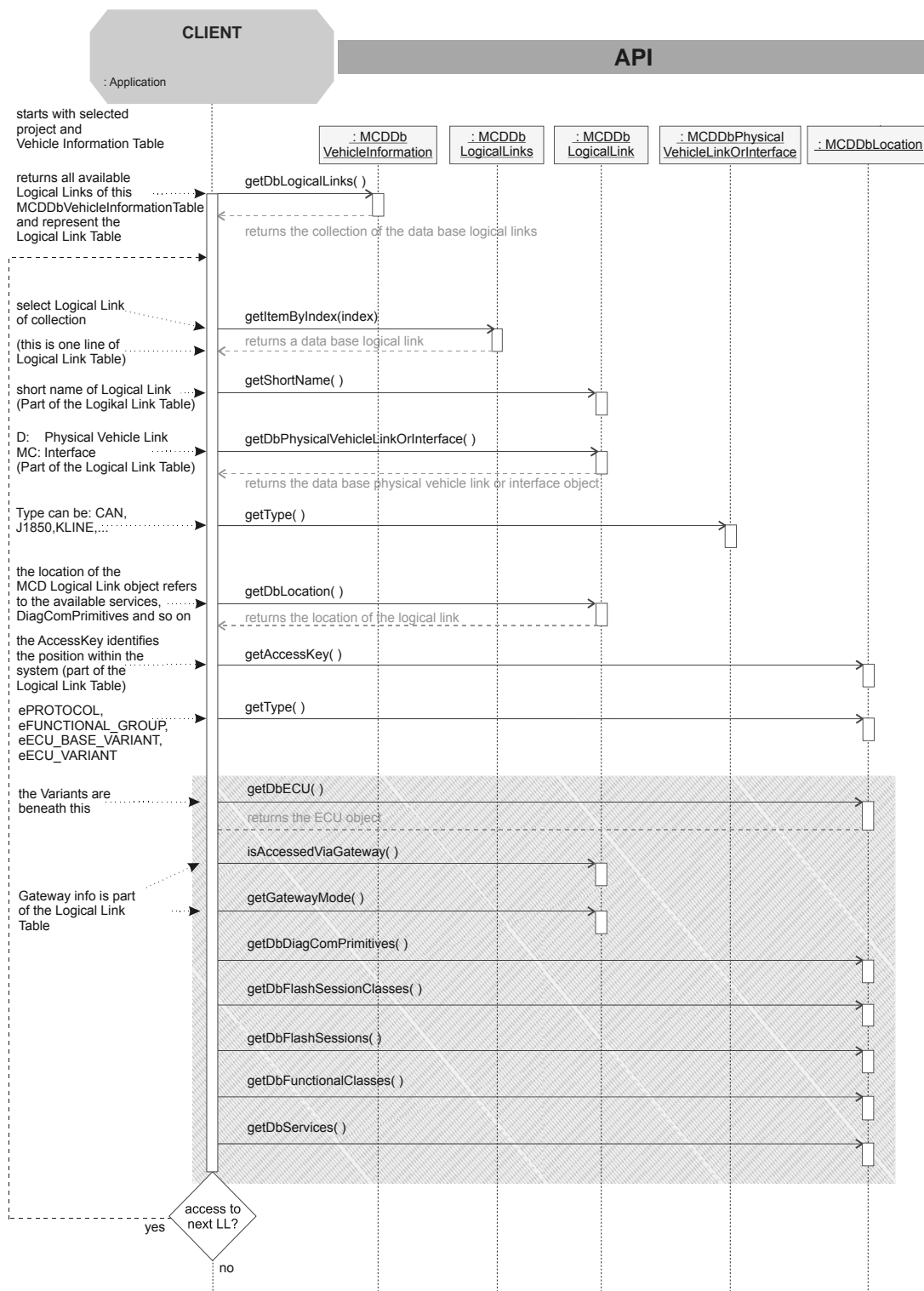
Figure 36 — Database access

7.11.4 Destruction

Figure 37 depicts the destruction.
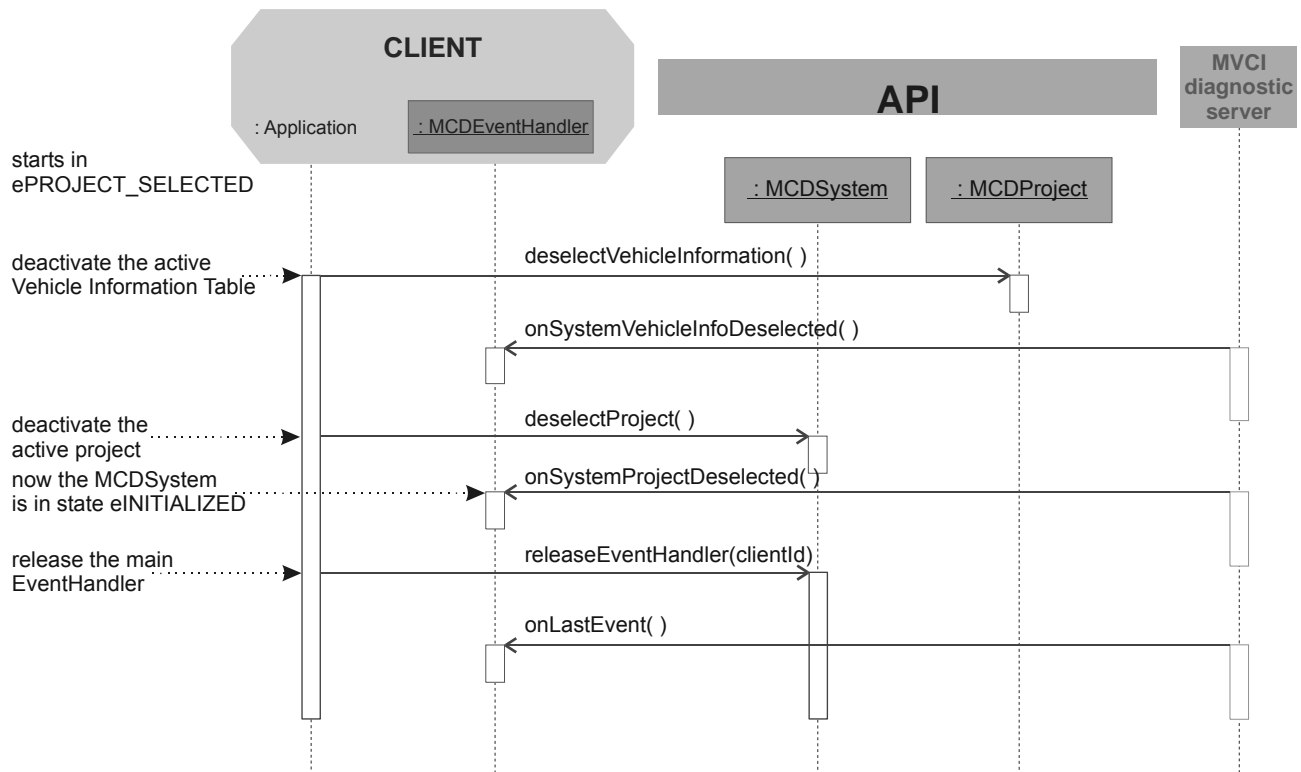
Figure 37 — Destruction

This Sequence Diagram starts in the MCDSystem state ePROJECT_SELECTED and with an active Vehicle Information Table. Logical Links don't exists at this time anymore.

First deactivate the Vehicle Information Table and after successful deactivation an event will be delivered. Then the project can be deactivated and for this state transition of the MCDSystem an event will be delivered too. The last action is to release the main EventHandler.

8   Function block Diagnostic in detail

8.1   Constraints

In ODX, the validity of the internal value can be restricted to a given interval via internal constraints. The physical value can be restricted via physical constraints. The value definition of both kinds of constraints can be obtained by using the methods `getInternalConstraint()` and `getPhysicalConstraint()` at `MCDDbParameter`. With the type of constraint it is defined how the `MCDInterval` and its limits have to be interpreted. Values regarding the internal constraint definition have to be interpreted by the internal data type. The values regarding the physical constraint definition have to be interpreted by the physical data type.

The implicit valid range of a value is defined in ODX by the integral data type. In the case of internal constraint definition the implicit valid range is additionally restricted by the encoding and the size of the parameter, which is defined either by the number of 1 in a condensed bit mask or the bit length. The explicit valid range is defined by the limits of the outer constraint (internal- or physical constraint), which is always a VALID interval, minus all inner scale constraints (SCALE-CONSTR in ODX) where the attribute VALIDITY is not equal to VALID. The attribute VALIDITY of each constraint definition can take the values VALID, NOT-VALID, NOT-DEFINED or NOT-AVAILABLE. The interval of a constraint is defined in ODX by the UPPER-LIMIT and LOWER-LIMIT attribute. If UPPER-LIMIT and/or LOWER-LIMIT are missing, the explicit valid range is not restricted in that direction. In general, the valid range is defined by the intersection of the implicit and the explicit valid range.

If the corresponding interval type is set to `eLIMIT_INFINITE`, the lower and upper limit are defined by the lowest and highest possible value of the integral data type.

For example the method `getUpperLimit()` will return the maximum positive value of the corresponding data type if `getUpperLimitIntervalType()` returns `MCDInterval::eLIMIT_INFINITE`.

In case of a request, the physical values given by the user or pre-defined in ODX shall be checked against the physical constraint by the diagnostic server. If the check is successful the physical values will be converted to the corresponding internal values. At last (after applying the computational method) the diagnostic server will check the internal values against the internal constraints. If successful, the data can be coded into the request message.

If the value violates the physical constraints in case of a request the computational method will not be applied. `MCDRequestParameter::setValue()` will throw a `MCDParameterizationException` with error code `ePAR_INVALID_VALUE`. In this state the methods `getValue()` and `getCodedValue()` will deliver an `MCDValue` that is not initialized.

In case of a response, the internal values extracted from the ECU response message and interpreted by the internal data type shall be checked against the internal constraint by the diagnostic server. If the check is successful, the internal values will be converted to the corresponding physical values by application of the computational method. At last the diagnostic server will check the physical values against the physical constraints.

If the value violates the internal constraints in case of a response the computational method will not be applied. In this state `MCDParameter::getValue()` will throw an `MCDProgramViolationException` with error code `eRT_ELEMENT_NOT_AVAILABLE` and `getCodedValue()` returns the internal value that violates the internal constraints.

The `MCDRangeInfo` will be set according to the validity of the value (request/response).

The valid range of a constraint is defined by the intersection of the implicit and the explicit valid range. The implicit valid range of the internal constraint is defined by the coded data type restricted by the encoding and the size of the parameter, which is defined either by the number of 1 in a condensed BIT-MASK or the BIT-LENGTH. The implicit valid range of the physical constraint is defined by the physical data type only. The explicit valid range is defined by UPPER-LIMIT and LOWER-LIMIT of the constraint minus all SCALE-

CONSTR with VALIDITY != VALID. If UPPER-LIMIT and/or LOWER-LIMIT are missing in the ODX data, the explicit valid range is not restricted in that direction.

The validity of a value is determined by the following multiple step process:

⎯ If the value is outside of the intersection of implicit and explicit valid range, ignoring any SCALE-CONSTR definition, the method MCDParameter::getValueRangeInfo() delivers eVALUE_NOT_VALID.

⎯ If the value is inside of the valid range (intersection of implicit and explicit valid range with respect to the SCALE-CONSTR definitions), the method MCDParameter::getValueRangeInfo() delivers eVALUE_VALID.

⎯ If the value is inside any SCALE-CONSTR value range the method MCDParameter::getValueRangeInfo() returns the validity defined by the SCALE-CONSTR in ODX.

⎯ Figure 38 depicts the application of Constraints during Data Extraction Process.

⎯
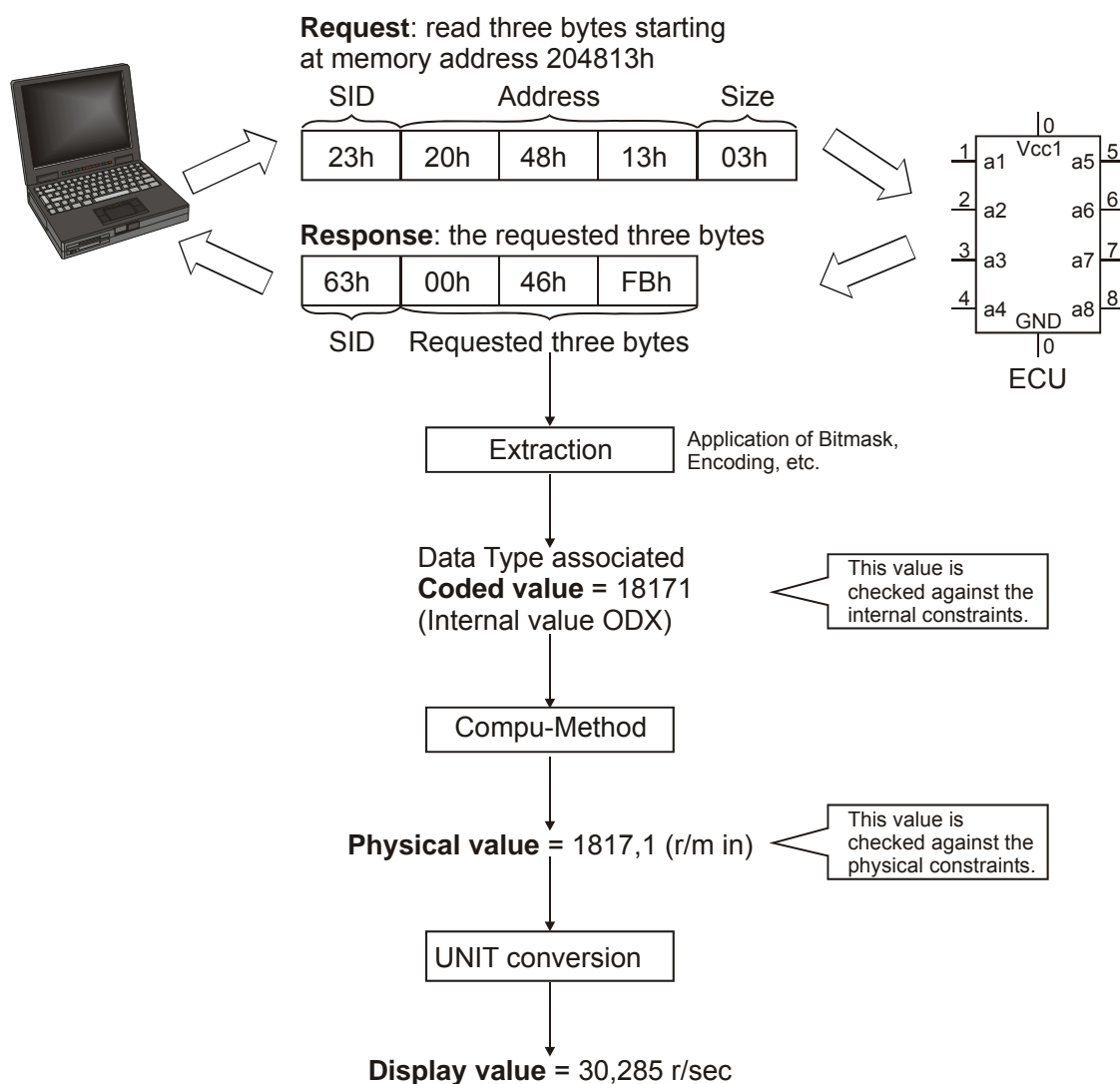
Figure 38 — Application of Constraints during Data Extraction Process

For the examples following, this ODX specification of DIAG-CODED-TYPE is assumed:

```
  <DIAG-CODED-TYPE        xsi:type="STANDARD-LENGTH-TYPE"        BASE-DATA-
TYPE="A_INT32"
    CONDENSED="true" BASE-TYPE-ENCODING="SM">
    <BIT-LENGTH>12</BIT-LENGTH>
    <BIT-MASK>0E07</BIT-MASK>
  </DIAG-CODED-TYPE>
```

Within the message the parameter has a length of 12 bit, but the condensed bitmask restricts the length of the signed integer value to actually 6 bit. Because it is encoded as SM (see ISO 22901-1 for details of encodings), the implicit valid range is [-31, +31].

In the first example the explicit valid range is (-∞, +∞) and therefore the intersection between the explicit and implicit valid ranges is [-31, +31].

Figure 39 depicts the use of constraints: No explicit restrictions (no INTERNAL-CONSTR).

Figure 39 — Use of constraints: No explicit restrictions (no INTERNAL-CONSTR)

In the second example the internal constraint is limited by the lower limit value -3 and the upper limit value 5. Both limits are defined with the interval type CLOSED.

```
<INTERNAL-CONSTR>
    <LOWER-LIMIT INTERVAL-TYPE = "CLOSED">-3</LOWER-LIMIT>
    <UPPER-LIMIT INTERVAL-TYPE = "CLOSED">5</UPPER-LIMIT>
</INTERNAL-CONSTR>
```

The explicit valid range is [-3, +5] and, therefore, the intersection between the explicit and implicit valid ranges is [-3, +5].

Figure 40 depicts the use of constraints: One valid interval.
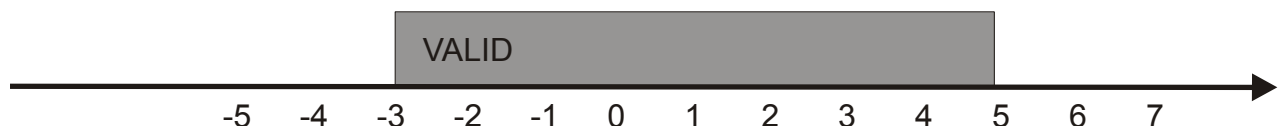
Figure 40 — Use of constraints: One valid interval

In the next example no limits for the internal constraint are defined but two inner scale constraints.

```
<INTERNAL-CONSTR>
    <SCALE-CONSTRS>
        <SCALE-CONSTR VALIDITY = "NOT-DEFINED">
            <LOWER-LIMIT INTERVAL-TYPE = "OPEN">4</LOWER-LIMIT>
            <UPPER-LIMIT INTERVAL-TYPE = "CLOSED">5</UPPER-LIMIT>
        </SCALE-CONSTR>
        <SCALE-CONSTR VALIDITY = "NOT-VALID">
            <LOWER-LIMIT INTERVAL-TYPE = "OPEN">5</LOWER-LIMIT>
            <UPPER-LIMIT INTERVAL-TYPE = "CLOSED">6</UPPER-LIMIT>
        </SCALE-CONSTR>
    </SCALE-CONSTRS>
</INTERNAL-CONSTR>
```

With no limits for the INTERNAL-CONSTR the explicit validity of the internal value is defined in the interval (-∞, +∞). Within this outer interval the interval (4, 5] is declared via SCALE-CONSTRS as NON-DEFINED and the interval (5, 6] is declared as NOT-VALID, which is illustrated in the following figure.

The explicit valid ranges are (-∞, +4] and (+6, +∞), therefore, the intersections between the explicit and implicit valid ranges are [-31, +4] and (+6, +31]. Because the internal type is an integer type, the second range is equivalent to [+7, +31].

Figure 41 depicts the use of constraints: No INTERNAL-CONSTR limits.
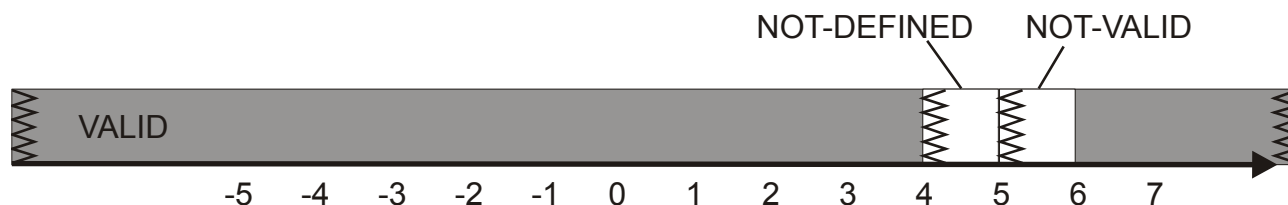
Figure 41 — Use of constraints: No INTERNAL-CONSTR limits

The next and the last example regarding internal constraints is defined by the following ODX data.

```
<INTERNAL-CONSTR>
    <LOWER-LIMIT INTERVAL-TYPE = "CLOSED">0</LOWER-LIMIT>
    <UPPER-LIMIT INTERVAL-TYPE = "INFINITE"/>
    <SCALE-CONSTRS>
        <SCALE-CONSTR VALIDITY = "NOT-DEFINED">
            <LOWER-LIMIT INTERVAL-TYPE = "OPEN">4</LOWER-LIMIT>
            <UPPER-LIMIT INTERVAL-TYPE = "CLOSED">5</UPPER-LIMIT>
        </SCALE-CONSTR>
        <SCALE-CONSTR VALIDITY = "NOT-AVAILABLE">
            <LOWER-LIMIT INTERVAL-TYPE = "OPEN">5</LOWER-LIMIT>
            <UPPER-LIMIT INTERVAL-TYPE = "CLOSED">6</UPPER-LIMIT>
        </SCALE-CONSTR>
    </SCALE-CONSTRS>
</INTERNAL-CONSTR>
```

This ODX data defines the explicit validity of the internal value in the interval [0, +∞) by defining limits for the INTERNAL-CONSTR. Within this outer interval the interval (4, 5] is declared via SCALE-CONSTRS as NON-DEFINED and the interval (5, 6] is declared as NOT-AVAILABLE, which is illustrated in the following figure.

The explicit valid ranges are [0, +4] and (+6, +∞), therefore, the intersections between the explicit and implicit valid ranges are [0, +4] and (+6, +31). Because the internal type is an integer type, the second range is equivalent to [+7, +31].

Figure 42 depicts the use of constraints: Clipping.
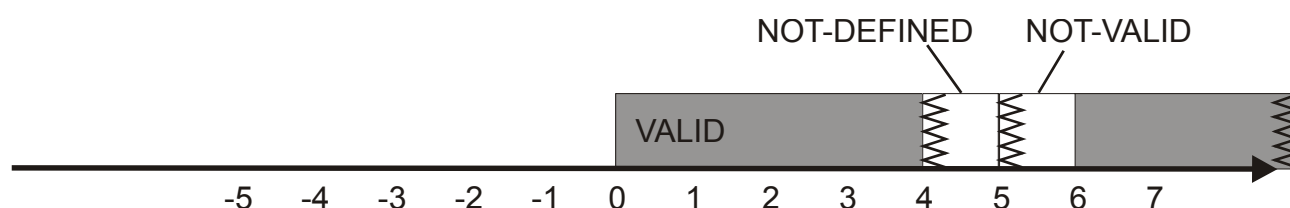
Figure 42 — Use of constraints: Clipping

In a similar way, it is possible to specify constraints for the physical value if the physical type is a numerical type. In this case, the implicit valid range is not restricted by the size or encoding of the internal data type but by the physical data type.

General diagnostic server behaviour for CODED-CONST and PHYS-CONST Parameters:

As CODED-CONST and PHYS-CONST parameters are parameters like any other in ODX, these parameters are also available at the diagnostic server API. That is, these parameters are contained in the collections delivered by `getRequestParameters()` and `getResponseParameters()`. However, it is impossible for a client application or job to change the value of a constant parameter.

TextTable

If you use `MCDTextTableElement::getInterval` the lower and upper limit shall deliver the coded values of the according text table elements in ODX.

Interval information for CODED-CONST and PHYS-CONST Parameters:

CODED-CONST:
The diagnostic server generates a `MCDInterval` object where the return value of the methods `MCDInterval::getLowerLimit()`, `MCDInterval::getUpperLimit()`, and `MCDDbParameter::getDefaultValue()` is the same value, namely the physical value of this constant parameter.

The methods `MCDInterval::getLowerLimitIntervalType()` and `MCDInterval::getUpperLimitIntervalType()` both return `MCDLimitType::eLIMIT_CLOSED`.

In case of a CODED-CONST parameter, the methods `MCDInterval::getLowerLimit()` and `MCDInterval::getUpperLimit()` both return the same coded value – the constant coded value.

PHYS-CONST:

In case of a PHYS-CONST parameter, an `MCDInterval` object is returned which corresponds to the information in the ODX data.

The method `getCodedValue()` always delivers an object of type `MCDValue` containing the coded value of the `MCDResponseParameter` object. The types used to store a coded value in a `MCDValue` object are the appropriate coded ODX types.

NRC-CONST

Parameters of type CODED-CONST or PHYS-CONST possess only one possible value and may appear at both request and response parameters. In contrast to that, parameters of type NRC-CONST consist of a collection of possible CODED-CONST values and may only appear at response parameters. Due to that, parameters of type NRC-CONST behave differently from parameters of type CODED-CONST or PHYS-CONST and throw the exception MCDProgramViolationException with error code `eRT_VALUE_NOT_AVAILABLE` for the following methods:

— MCDDbParameter::getDefaultValue(),

— MCDDbParameter::getCodedDefaultValue(),

— MCDDbParameter::getInternalConstraint(),

— MCDDbParameter::getPhysicalConstraint();

Parameters of type NRC-CONST may only appear at response parameters of services. They may not appear in response parameters of jobs.

Matching of result parameters can be done in one of the following ways:

— MATCHING-REQUEST-PARAM (repetition of data from request),

— CODED-CONST (matching with fixed coded-value),

— NRC-CONST (matching with one of the fixed coded-values),

— PHYS-CONST (matching with fixed physical value);

NRC-CONST is used only in negative responses. All values have to be transmitted to the PDU API for each execution and a high number of values can have a strong negative influence on the performance. It describes a set of negative response codes as coded values. The response must contain one of the given coded values in the encoding defined by the DIAG-CODED-TYPE. It differs from a CODED-CONST only in that it describes a set of values and must not be used in a REQUEST or in a POS-RESPONSE. The values are used in matching a negative response. For example you might define a VALUE parameter that uses a TEXTTABLE COMPU-METHOD to map all possible negative response codes to descriptive texts. An additional NRC-CONST parameter can make the NEG-RESPONSE match if one of a specific subset of these codes is actually returned.

The NRC-CONST is located at the same PDU position as the VALUE parameter. As a VALUE parameter is never used to match a RESPONSE, the parameter (and its associated TEXTTABLE) does not suffice to select the response.

Behaviour of MCDResponseParameter::getCodedValue():

— In case a MCDResponseParameter contains a valid coded value, a copy of the corresponding MCDValue object containing the server-internal coded value is returned.

— In case a MCDResponseParameter does not contain a valid coded value and therefore no coded type, MCDResonseParameter::getCodedValue() returns a MCDValue object of type A_BITFIELD which is initialized with the bits representing the corresponding parameter in the response PDU.

— In case the parameter is of type CODED-CONST, the method getCodedValue() returns the constant coded value as defined in ODX. For coded const parameters, this value is mandatory.

— In case the parameter is of type PHYS-CONST, the method getCodedValue() returns an object of type MCDValue which is initialised with the coded value calculated from the constant physical value.

— In case the parameter is of type RESERVED, the method getCodedValue() returns the coded value as extracted from the response PDU. If no coded value and therefore no coded type is defined in ODX, getCodedValue() returns a MCDValue object of type A_BITFIELD which is initialized with the bits representing the corresponding parameter in the response PDU. Any coded value defined for a reserved response parameter in ODX is ignored.

— For complex parameters, the method getCodedValue() returns the same values as defined for the method MCDParameter::getValue().

Behaviour of MCDRequestParameter::getCodedValue():

The method `getCodedValue()` always delivers an object of type MCDValue containing the coded value of the `MCDRequestParameter` object. The types used to store a coded value in a MCDValue object are the appropriate coded ODX types.

— In case a MCDRequestParameter contains a valid coded value (set via `setValue()` or `setCodedValue()`), a copy of the corresponding server-internal coded value is returned in form of a MCDValue object.

— In case `setValue()` or `setCodedValue()` have not been called before but a physical default value is defined in ODX, an object of type MCDValue is returned which is initialised with the coded value calculated from the physical default value.

— In case the parameter is of type CODED-CONST, the method `getCodedValue()` returns the constant coded value as defined in ODX. For coded const parameters, this value is mandatory.

— In case the parameter is of type PHYS-CONST, the method getCodedValue() returns an object of type MCDValue which is initialised with the coded value calculated from the constant physical value.

— In case the parameter is of type RESERVED, the method `getCodedValue()` returns the coded value as defined in ODX. If no coded value is defined in ODX, `getCodedValue()` returns a correctly typed MCDValue object (coded ODX value type), which is initialized with the value which would be used by the server when sending the request. This means that the server is not to alter the coded value after it has been returned to a client via `getCodedValue()` once.

— In case `setValue()` or `setCodedValue()` have not been called before and no physical default value is defined in ODX, the method `getCodedValue()` returns a correctly typed MCDValue object (coded ODX value type). The internal state of this MCDValue is "uninitialized" and its value is "undefined". That is, calling a getXXX()-Method at this MCDValue (where XXX represents the type of this MCDValue) will result in an exception being thrown. The type of this exception is MCDProgramViolationException with error code `eRT_VALUE_NOT_INITIALIZED`.

— For complex parameters, the method getCodedValue() returns the same values as defined for the method MCDParameter::getValue().

Behaviour of MCDRequestParameter::setCodedValue():

If `MCDRequestParameter::setCodedValue(MCDValue)` is called, the coded value passed is used to alter the server-internal PDU accordingly. Also the physical value of the affected parameter has to be updated (backward conversion), if possible. If it's not possible to perform backwards conversion of the coded value to a corresponding physical value (e.g. an appropriate conversion formula is missing), the range information of that parameter is set to `eVALUE_CODED_TO_PHYSICAL_FAILED`. Thereafter, the physical value of that request parameter with range information set to `eVALUE_CODED_TO_PHYSICAL_FAILED` is considered invalid. However, it is still possible to read the coded value via `MCDRequestParameter::getCodedValue()`.

Behaviour of setting values for CODED-CONST and PHYS-CONST Parameters:

— Parameters of type CODED-CONST or PHYS-CONST cannot be changed by using the method MCDRequestParameter::setValue(). So, calling method MCDRequestParameter::setValue() leads to an MCDParameterizationException with error code `ePAR_INVALID_VALUE`.

— The same applies for the method MCDResponseParameter::setValue(), which can be called by a job to fill the response parameter structure. In case a job tries to overwrite a CONST value using this method, a MCDParameterizationException with error code `ePAR_INVALID_VALUE` will be thrown.

— Reponses returned by an ECU that don't match values for CONST values, similarly to wrong values for other types of parameters in ODX, should be marked as erroneous. This is achieved by a flag which can be queried by hasError() at various classes, e.g. `MCDResult::hasError()`, `MCDResponse::hasError()`, and `MCDResponseParameter::hasError()`. The error code in this case is `eRT_VALUE_OUT_OF_RANGE`.

Retrieval of all valid physical or internal intervals:

Usually, constraints and computational methods of a parameter defined in ODX should be consistent. That means, for each value passing a constraint there should be a suitable COMPU-METHOD that could be used to compute a physical value from the coded one or vice versa. But this cannot be guaranteed. Furthermore, a client can get the valid physical intervals of a parameter according to the physical constraints before setting the value, but the value possibly does not pass the internal constraints or vice versa. Due to the lack of access to the COMPU-METHODs a client cannot retrieve all necessary information in advance needed to find out, whether a value is actually valid or not. Depending on the COMPU-METHOD, implicit and explicit value ranges there might be zero, one or more intervals of valid values for one parameter. These intervals can be retrieved by calling `MCDDbParameter::getValidPhysicalIntervals` or `MCDDbParameter::getValidInternalIntervals`. The following example shows how the valid internal and physical intervals are computed from a certain ODX parameter definition.

ODX Data

```
<DIAG-CODED-TYPE          xsi:type="STANDARD-LENGTH-TYPE"          BASE-DATA-
TYPE="A_UINT32">
  <BIT-LENGTH>12</BIT-LENGTH> --> value restricted to interval [0..4095]
</DIAG-CODED-TYPE>
<PHYSICAL-TYPE BASE-DATA-TYPE="A_UINT32"/>
<COMPU-METHOD>
  <CATEGORY>SCALE-LINEAR</CATEGORY>
    <COMPU-INTERNAL-TO-PHYS>
      <COMPU-SCALES>
      <COMPU-SCALE> (1) --> phys = 2 + 2 x coded
        <LOWER-LIMIT INTERVAL-TYPE="CLOSED">0</LOWER-LIMIT>
        <UPPER-LIMIT INTERVAL-TYPE="CLOSED">100</UPPER-LIMIT>
        <COMPU-RATIONAL-COEFFS>
```

```
                <COMPU-NUMERATOR><V>2</V><V>2</V></COMPU-NUMERATOR>
                <COMPU-DENOMINATOR><V>1</V></COMPU-DENOMINATOR>
            </COMPU-RATIONAL-COEFFS>
            </COMPU-SCALE>
          <COMPU-SCALE> (2) --> phys = 3 + 2 x coded
            <LOWER-LIMIT INTERVAL-TYPE="CLOSED">102</LOWER-LIMIT>
            <UPPER-LIMIT INTERVAL-TYPE="CLOSED">300</UPPER-LIMIT>
            <COMPU-RATIONAL-COEFFS>
                <COMPU-NUMERATOR><V>3</V><V>2</V></COMPU-NUMERATOR>
                <COMPU-DENOMINATOR><V>1</V></COMPU-DENOMINATOR>
            </COMPU-RATIONAL-COEFFS>
            </COMPU-SCALE>
          <COMPU-SCALE> (3) --> phys = 4 + 2 x coded
            <LOWER-LIMIT INTERVAL-TYPE="CLOSED">1900</LOWER-LIMIT>
            <UPPER-LIMIT INTERVAL-TYPE="CLOSED">2020</UPPER-LIMIT>
            <COMPU-RATIONAL-COEFFS>
                <COMPU-NUMERATOR><V>4</V><V>2</V></COMPU-NUMERATOR>
                <COMPU-DENOMINATOR><V>1</V></COMPU-DENOMINATOR>
            </COMPU-RATIONAL-COEFFS>
            </COMPU-SCALE>
        </COMPU-SCALES>
    </COMPU-INTERNAL-TO-PHYS>
</COMPU-METHOD>
<INTERNAL-CONSTR>
    <UPPER-LIMIT INTERVAL-TYPE="CLOSED">5000</UPPER-LIMIT>
  <SCALE-CONSTRS>
     <SCALE-CONSTR VALIDITY="NOT-DEFINED"> (a)
      <LOWER-LIMIT INTERVAL-TYPE="CLOSED">0</LOWER-LIMIT>
      <UPPER-LIMIT INTERVAL-TYPE="CLOSED">100</UPPER-LIMIT>
    </SCALE-CONSTR>
     <SCALE-CONSTR VALIDITY="VALID"> (b)
      <LOWER-LIMIT INTERVAL-TYPE="CLOSED">101</LOWER-LIMIT>
      <UPPER-LIMIT INTERVAL-TYPE="CLOSED">333</UPPER-LIMIT>
    </SCALE-CONSTR>
     <SCALE-CONSTR VALIDITY="NOT-AVAILABLE"> (c)
      <LOWER-LIMIT INTERVAL-TYPE="CLOSED">334</LOWER-LIMIT>
      <UPPER-LIMIT INTERVAL-TYPE="CLOSED">513</UPPER-LIMIT>
    </SCALE-CONSTR>
     <SCALE-CONSTR VALIDITY="VALID"> (d)
      <LOWER-LIMIT INTERVAL-TYPE="CLOSED">514</LOWER-LIMIT>
      <UPPER-LIMIT INTERVAL-TYPE="CLOSED">1000</UPPER-LIMIT>
    </SCALE-CONSTR>
     <SCALE-CONSTR VALIDITY="NOT-VALID"> (e)
      <LOWER-LIMIT INTERVAL-TYPE="CLOSED">1500</LOWER-LIMIT>
      <UPPER-LIMIT INTERVAL-TYPE="CLOSED">2000</UPPER-LIMIT>
    </SCALE-CONSTR>
  </SCALE-CONSTRS>
</INTERNAL-CONSTR>
<PHYS-CONSTR>
    <UPPER-LIMIT INTERVAL-TYPE="CLOSED">5000</UPPER-LIMIT>
  <SCALE-CONSTRS>
     <SCALE-CONSTR VALIDITY="NOT-DEFINED"> (A)
      <LOWER-LIMIT INTERVAL-TYPE="CLOSED">200</LOWER-LIMIT>
      <UPPER-LIMIT INTERVAL-TYPE="CLOSED">250</UPPER-LIMIT>
      </SCALE-CONSTR>
```

```
      <SCALE-CONSTR VALIDITY="VALID"> (B)
       <LOWER-LIMIT INTERVAL-TYPE="CLOSED">251</LOWER-LIMIT>
       <UPPER-LIMIT INTERVAL-TYPE="CLOSED">500</UPPER-LIMIT>
      </SCALE-CONSTR>
      <SCALE-CONSTR VALIDITY="NOT-AVAILABLE"> (C)
       <LOWER-LIMIT INTERVAL-TYPE="CLOSED">501</LOWER-LIMIT>
       <UPPER-LIMIT INTERVAL-TYPE="CLOSED">550</UPPER-LIMIT>
      </SCALE-CONSTR>
      <SCALE-CONSTR VALIDITY="VALID"> (D)
       <LOWER-LIMIT INTERVAL-TYPE="CLOSED">551</LOWER-LIMIT>
       <UPPER-LIMIT INTERVAL-TYPE="CLOSED">4020</UPPER-LIMIT>
      </SCALE-CONSTR>
      <SCALE-CONSTR VALIDITY="NOT-VALID"> (E)
       <LOWER-LIMIT INTERVAL-TYPE="CLOSED">4021</LOWER-LIMIT>
       <UPPER-LIMIT INTERVAL-TYPE="INFINITE"/>
      </SCALE-CONSTR>
    </SCALE-CONSTRS>
</PHYS-CONSTR>
```

Valid internal intervals:

A_UINT32 + BIT-LENGTH 12: [0..4095], no restrictions based on INTERNAL-CONSTR interval

[0..5000]

less not VALID SCALE-CONSTRs (a), (c), (e):

[101..333], [514..1499], [2001, 4095]

Intersection with available COMPU-SCALEs: (2): [102..300] und (3) [2001, 2020]

Calculated physical intervals:

[207, 603], [4006, 4044]

less not VALID SCALE-CONSTR (A), (C), (E):

Interval 1: [251..500]

Interval 2: [551..603]

Interval 3: [4006..4020]

8.2   System Properties

A global mechanism to handle system properties is provided by the methods `getPropertyNames()`, `getProperty(A_ASCIISTRING propertyName)`, `setProperty(A_ASCIISTRING propertyName, MCDValue value)` and `resetProperty(A_ASCIISTRING propertyName)` at the interface `MCDSystem`. The methods allow retrieval and modification of all system properties defined within the diagnostic server. System properties are separated into *mandatory* and *optional* properties.

Every vendor or OEM is free to define his own additional server properties. To avoid namespace conflicts, the shortnames of these properties have to be prefixed according to Sun's rules for java package names, e.g. starting with com.company.<PROPERTY_NAME>.

For all properties defined by a vendor or OEM, the following information shall be included in the corresponding server documentation:

— Name of the property,

— Type of the value,

— Valid values,

— Default value,

— Description,

— Mandatory / Optional,

— Precondition;

8.3 Diagnostic DiagComPrimitives and Services

8.3.1 Diagnostic DiagComPrimitives

8.3.1.1 DCP types and hierarchy

These are the types of DiagComPrimitives declared in the API:

`MCDHexService`	performs a diagnostic hexservice (low level service)
`MCDService`	performs a diagnostic service
`MCDProtocolParameterSet`	provides the set of protocol parameters defined for the location
`MCDStartCommunication`	performs protocol specific initialisation
`MCDStopCommunication`	performs protocol specific termination
`MCDVariantIdentification`	performs a variant identification
`MCDVariantIdentification AndSelection`	performs a variant identification and selects the identified variant
`MCDSingleEcuJob`	performs a job on a single ECU
`MCDMultipleEcuJob`	performs a job on multiple ECUs
`MCDFlashJob`	performs a Flash Job
`MCDDynIdDefineComPrimitive`	Define the data structure for a dynamically defined identifier
`MCDDynIdReadComPrimitive`	read the service, which used a dynamically defined identifier data structure
`MCDDynIdClearComPrimitive`	Delete the DynId and the assigned data structure

Figure 43 depicts the hierarchy of inheritance of DiagComPrimitive.
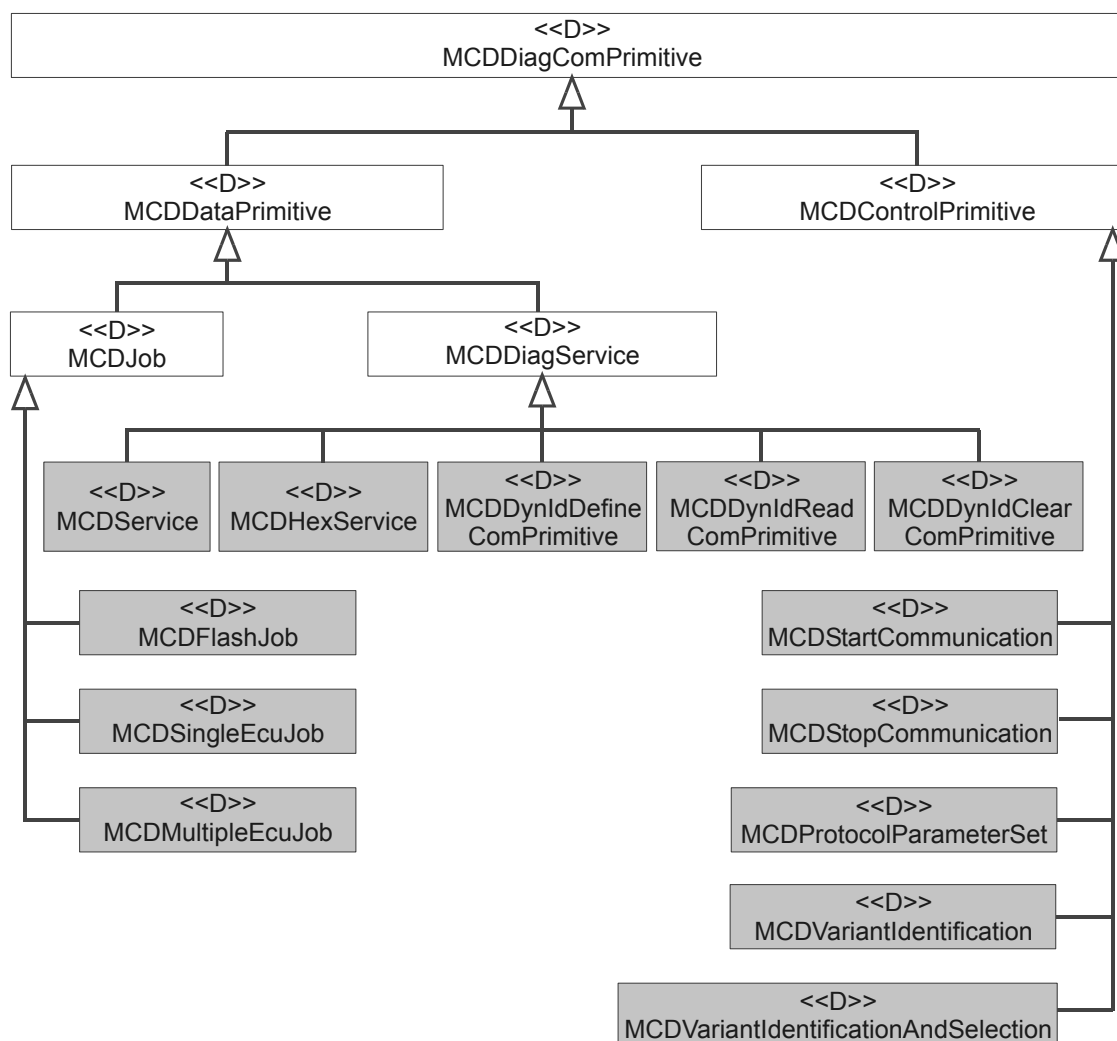
Figure 43 — Hierarchy of inheritance of DiagComPrimitive

The DiagComPrimitives are divided in two different types: DataPrimitives and ControlPrimitives.

ControlPrimitives perform state transitions, protocol settings or recognize the real ECU. They exist only once per Logical Link. While execution of one of these ControlPrimitives the Logical Link Activity Queue has to be empty and the Activity state has to be `eACTIVITY_IDLE`. They cannot be executed asynchronously.

DataPrimitive perform action on the ECU. They are divided in such with PDU information like the service and such without PDU information, the jobs. All DataPrimitives can (if the DataPrimitive carries the IS-CYCLIC flag within the data base) executed in cyclic mode and/or asynchronously. Only DataPrimitivesWithPDU can be executed in repetition mode. That's why only DataPrimitive can resize the ResultBuffer.

All communication primitives have the equivalent result structure as diagnostic services. Each noncyclic DiagCom Primitive has exactly one result record per execution.

In case a DiagComPrimitive, e.g. `MCDDynIdDefineComPrimitive`, `MCDDynIdReadComPrimitive`, or `MCDDynIdClearComPrimitive`, is not available (either in the

MVCI diagnostic server or in the ODX data base), an exception of type `eDB_ELEMENT_NOT_AVAILABLE` is thrown.

List of MCDObjectTypes for which an object can be created by calling `MCDLogicalLink::createDiagComPrimitiveByType` (only object types for which no additional information is required during the creation, that is, no additional parameters):

— eMCDHEXSERVICE,

— eMCDPROTOCOLPARAMETERSET,

— eMCDSTARTCOMMUNICATION,

— eMCDSTOPCOMMUNICATION,

— eMCDVARIANTIDENTIFICATION,

— eMCDVARIANTIDENTIFICATIONANDSELECTION;

If the method is called for a different MCDObjectType than for the ones not in this list an exception of type `ePAR_INVALID_OBJECTTYPE_FOR_DIAGCOMPRIMITIVE` is thrown.

8.3.1.2 States of DiagComPrimitives

Each runtime DiagComPrimitive has two states, `eIDLE` and `ePENDING`. It is created within the state `eIDLE` and may be deleted within this state only. Furthermore, within this state the parameterisation and the evaluation of the results can be carried out. As soon as the objects has been started is takes the state `ePENDING` and is executed by the MVCI diagnostic server. After this or after the execution has been stopped by `cancel()` it returns to the state `eIDLE`.

Figure 44 depicts the state diagram MCDDiagComPrimitive.
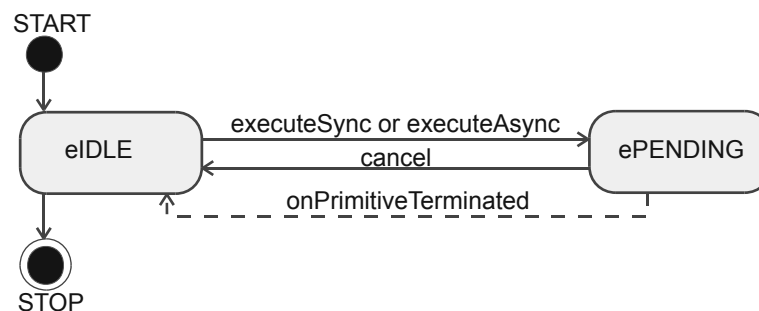
Figure 44 — State diagram MCDDiagComPrimitive

`MCDService` will be considered in more detail within the next Chapter.

8.3.2   Service overview

A diagnostic service will be executed by the Data Processor of the MVCI diagnostic server.

Table 6 defines a subdivision of the diagnostic services.

Table 6 — Types of diagnose services

RuntimeMode	RepetitionMode	Transmission Mode	Kind of Execution
`eNONCYCLIC`	`eSINGLE`	`eSEND_AND_RECEIVE`	Async and Sync
		`eSEND_ONLY`	Async and Sync
		`eRECEIVE_ONLY`	Async and Sync
	`eREPEATED`	`eSEND_AND_RECEIVE`	Async
		`eSEND_ONLY`	Async
		`eRECEIVE_ONLY`	Async
`eCYCLIC`	`eSINGLE`	`eSEND_AND_RECEIVE`	Async
		`eSEND_ONLY`	Async
		`eRECEIVE_ONLY`	Async
	Reflection in Repetition State		

Generally, Jobs are handled like Diag services.

The features of the services are polled by means of the RunTimeMode and the RepetitionMode of the database template.

If the feature Repeated is reported, this service can be executed by means of `startRepetition()` within the Repetition Mode or `executeSync() or executeAsync()` within the Single Mode. If the MVCI diagnostic server does not support the RepetitionMode, the error message "Function not supported" is returned to this function call.

If a service with the Repetition Mode Single is called by `startRepetition()`, an error message is returned.

A non cyclic diagnostic service returns exactly one result record. Non cyclic services end with the returning of the result or are terminated by a Timeout. Non cyclic services may be started synchronously or asynchronously. A non cyclic diagnostic service (no Job !) may be executed in Repetition Mode (if supported by the MVCI diagnostic server and permitted by the available data, start by means of the method `startRepetition()`). Thus, after its execution (Event: `onPrimitveHasResult`), the non cyclic diagnostic service is automatically started anew by the server. The execution is repeated until the method `stopRepetition()` is called. The time interval between two automatically executions of the non cyclic diagnostic service by the diagnostic server may be set by the Client (RepetitionTime). The Activity Queue of the Logical Link is not bothered by the automatic starting of the non cyclic diagnostic service by the diagnostic server. But however, a delay of the repeated execution may occur, if other methods within the Activity Queue are executed.

Cyclic diagnostic services return complete result records within non equidistant time intervals. The structure of the independent result records is in accordance with the database template. Cyclic diagnostic services usually end with the call of the method `cancel()` (`onPrimitiveCanceledDuringExecution`) or after an

internal Timeout (`onPrimitiveTerminated`). Cyclic diagnostic services may be started asynchronously only.

Diagnostic services which are started by `executeSync()`, deliver the Result State as return value and no Event `onPrimitiveTerminated`, while diagnostic services which are executed by `executeAsync()` get the Event `onPrimitiveTerminated` and evaluate the Result State by means of this event.

The Service is derived from DiagComPrimitive (via DataPrimitive and DiagService) and thus also has the two states eIDLE and `ePENDING`, which are used in the same way. Additionally there are the two states `eNOT_REPEATING` and `eREPEATING`. Within the state `eNOT_REPEATING` the Service may be parameterised for the execution within the repeated mode. As soon as the Service starts its execution by `startRepetition()`, it takes the state `eREPEATING`. Within the state `eREPEATING` it also may change between `eIDLE` and `ePENDING`, if for example its RequestParameter are set anew. It may be parameterised anew but not start a new execution. After finishing the execution within repeated mode by `stopRepetition()`, the Service returns to the state `eNOT_REPEATED` and `eIDLE`.

Figure 45 depicts the state diagrams MCDService.
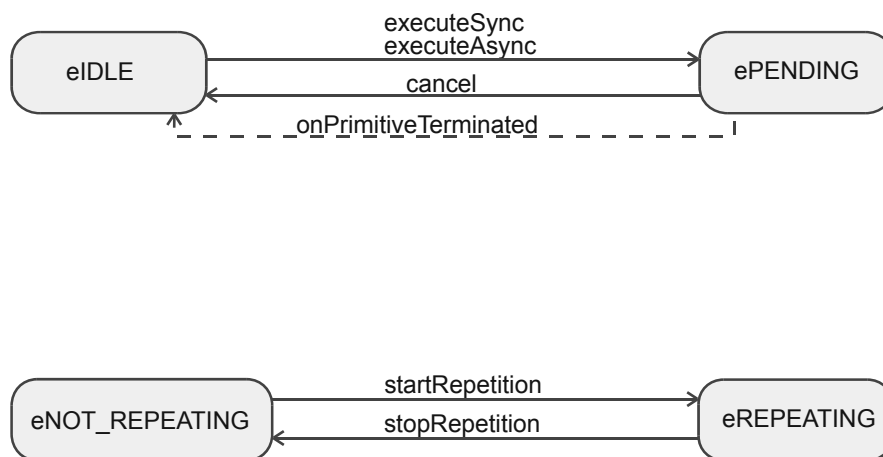
Figure 45 — State diagrams MCDService

NOTE   The instruction queue shown in the following pictures is represented by Activity state.

The handling of pending responses should always be performed below the diagnostic server because otherwise the diagnostic server would need to interpret responses. As a result the diagnostic server and/or the client would need to implement protocol interpreters. The protocol drivers used with a diagnostic server shall not be allowed to switch off the response pending handling.

Table 7 defines the SingleShot diagnostic services.

Table 7 — SingleShot diagnostic services

EM [a]		Termination (T)	AM [b]		Intermediate Result (IR)	Termination Events	
Synchronous	Asynchronou		Physical	Functional		SPR = Single-Part Response	MPR = Multi-Part Response
X		Response or Timeout or VCL	X		cannot occur	1 MCDResult (to caller) In case of VCL the empty result contains the error eCOM_LOST_COMM_TO_VCI	Not allowed
X		Timeout or VCL		X	cannot occur	1 MCDResult (to caller) with <= m MCDResponses	Not allowed
	X	Response (only if not multi-part) or Timeout or VCL	X		cannot occur	1 MCDResult with 1 MCDResponse 1 x onPrimitiveTerminated	1 MCDResult with p MCDResponses 1 x onPrimitiveTerminated
	X	Cancel	X		cannot occur	No MCDResult 1 x onPrimitiveCanceledDuringExecution or 1 x onPrimitiveCanceledFromQueue	No MCDResult 1 x onPrimitiveCanceledDuringExecution or 1 x onPrimitiveCanceledFromQueue
	X	Timeout or VCL		X	cannot occur	1 MCDResult with <= m MCDResponses 1 x onPrimitiveTerminated	1 MCDResult with <= (p *m) MCDResponses 1 x onPrimitiveTerminated
	X	Cancel		X	cannot occur	No MCDResult 1 x onPrimitiveCanceledDuringExecution or 1 x onPrimitiveCanceledFromQueue	No MCDResult 1 x onPrimitiveCanceledDuringExecution or 1 x onPrimitiveCanceledFromQueue

[a]   Execution Mode

[b]   Addressing Mode

m   number of ECUs in functional group

p   number of available multi-part responses per ECU

n   number of execution cycles during cyclic or repeated execution

Table 8 defines the cyclic diagnostic services.

Table 8 — Cyclic diagnostic services

EM [a]			AM [b]			Termination Events	
Synchronous	Asynchronou	Termination (T)	Physical	Functional	Intermediate Result (IR)	SPR = Single-Part Response	MPR = Multi-Part Response
	X	Cancel	Physical		cannot occur	n MCDResults with 1 MCDResponse each n x onPrimitiveHasResult 1 x onPrimitiveCanceledDuringExecution	n MCDResults with p MCDResponses each n x onPrimitiveHasResult 1 x onPrimitiveCanceledDuringExecution
	X	Timeout / BusError or VCL	Physical		cannot occur	n MCDResults with 1 MCDResponse each n x onPrimitiveHasResult 1 x onPrimitiveTerminated	n MCDResults with p MCDResponses each n x onPrimitiveHasResult 1 x onPrimitiveTerminated
	X	Cancel		Functional	cannot occur	n MCDResults with <= m MCDResponses each n x onPrimitiveHasResult 1 x onPrimitiveCanceledDuringExecution	n MCDResults with <= (p * m) MCDResponses each n x onPrimitiveHasResult 1 x onPrimitiveCanceledDuringExecution
	X	Timeout / BusError or VCL		Functional	cannot occur	n MCDResults with <= m MCDResponses each n x onPrimitiveHasResult 1 x onPrimitiveTerminated	n MCDResults with <= (p * m) MCDResponses each n x onPrimitiveHasResult 1 x onPrimitiveTerminated
[a] Execution Mode [b] Addressing Mode m number of ECUs in functional group p number of available multi-part responses per ECU n number of execution cycles during cyclic or repeated execution							

Table 9 defines the Repeated diagnostic services.

Table 9 — Repeated diagnostic services

| EM[a] | | Termination (T) | AM[b] | | Intermediate Result (IR) | Termination Events | |
Synchronous	Asynchronou		Physical	Functional		SPR = Single-Part Response	MPR = Multi-Part Response
	X	Stop-Repetition	X		cannot occur	n MCDResults with 1 MCDResponse each n x onPrimitiveHasResult 1 x onPrimitiveRepetitionStopped	n MCDResults with p MCDResponses each n x onPrimitiveHasResult 1 x onPrimitiveRepetitionStopped
	X	Stop-Repetition		X	cannot occur	n MCDResults with <= m MCDResponses each n x onPrimitiveHasResult 1 x onPrimitiveRepetitionStopped	n MCDResults with <= (p * m) MCDResponses each n x onPrimitiveHasResult 1 x onPrimitiveRepetitionStopped
	X	Cancel	X		cannot occur	n MCDResults with 1 MCDResponse each n x onPrimitiveHasResult 1 x onPrimitiveCanceledDuringExecution or 1 x onPrimitiveCanceledFromQueue	n MCDResults with p MCDResponses each n x onPrimitiveHasResult 1 x onPrimitiveCanceledDuringExecution or 1 x onPrimitiveCanceledFromQueue
	X	Cancel		X	cannot occur	n MCDResults with <= m MCDResponses each n x onPrimitiveHasResult 1 x onPrimitiveCanceledDuringExecution or 1 x onPrimitiveCanceledFromQueue	n MCDResults with <= (p * m) MCDResponses each n x onPrimitiveHasResult 1 x onPrimitiveCanceledDuringExecution or 1 x onPrimitiveCanceledFromQueue

[a] Execution Mode

[b] Addressing Mode

m   number of ECUs in functional group

p   number of available multi-part responses per ECU

n   number of execution cycles during cyclic or repeated execution

Table 10 defines the Java-Job services.

Table 10 — Java-Job services

EM [a]		Termination (T)	AM [b]		Intermediate Result (IR)	Termination Events	
Synchronous	Asynchronou		Physical	Functional		**SPR = Single-Part Response**	**MPR = Multi-Part Response**
X		Response or Timeout	X		can occur	1 MCDResult (to caller) x Intermediate results x onPrimitiveHasIntermediateResult	Not allowed
	X	Response (only if not multi-part) or Timeout	X		can occur	1 MCDResult x Intermediate results x onPrimitiveHasIntermediateResult 1 x on PrimitiveTerminated	Not allowed
	X	Cancel	X		can occur	No MCDResult x Intermediate results x onPrimitiveHasIntermediateResult 1 x onPrimitiveCanceledDuringExecution or 1 x onPrimitiveCanceledFromQueue	Not allowed
X				X		Not allowed	Not allowed
	X			X		Not allowed	Not allowed
[a] Execution Mode							
[b] Addressing Mode							
m number of ECUs in functional group							
p number of available multi-part responses per ECU							
n number of execution cycles during cyclic or repeated execution							

Table 11 defines the Repeated Java-Job services.

Table 11 — Repeated Java-Job services

EM [a]		Termination (T)	AM [b]		Intermediate Result (IR)	Termination Events	
Synchronous	Asynchronou		Physical	Functional		SPR = Single-Part Response	MPR = Multi-Part Response
	X	Stop-Repetition	X		can occur	x Intermediate results x onPrimitiveHasIntermediateResult n MCDResults with 1 MCDResponse each n x onPrimitiveHasResult 1 x onPrimitiveRepetitionStopped	Not allowed
	X	Cancel	X		can occur	x Intermediate results x onPrimitiveHasIntermediateResult n MCDResults with 1 MCDResponse each n x onPrimitiveHasResult 1 x onPrimitiveCanceledDuringExecution or 1 x onPrimitiveCanceledFromQueue	Not allowed
	X			X		Not allowed	Not allowed
[a] Execution Mode							
[b] Addressing Mode							
m number of ECUs in functional group							
p number of available multi-part responses per ECU							
n number of execution cycles during cyclic or repeated execution							

Figure 46 shows the principle result handling in case of physical / functional services for Single and Multi Part Responses. Please note that in case of Multi part Responses the Responses Collection in the Result Object can contain more than 1 Response for each ECU.
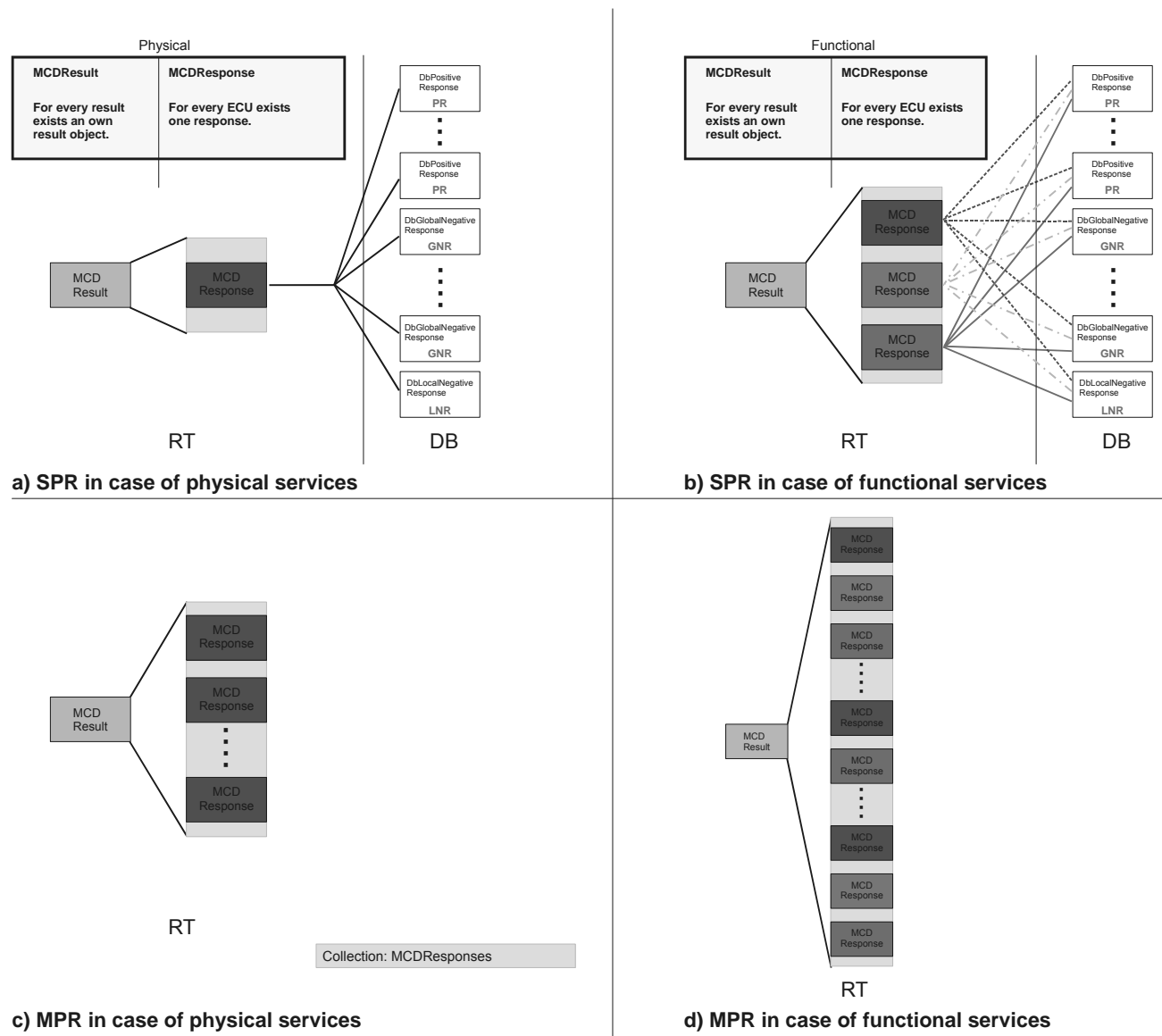
Figure 46 — Principle result handling in case of Physical and Functional Services for SPR and MPR

8.3.3 Non cyclic single diagnostic service

Figure 47 depicts the non cyclic single diagnostic service (asynchronous executed outside jobs).
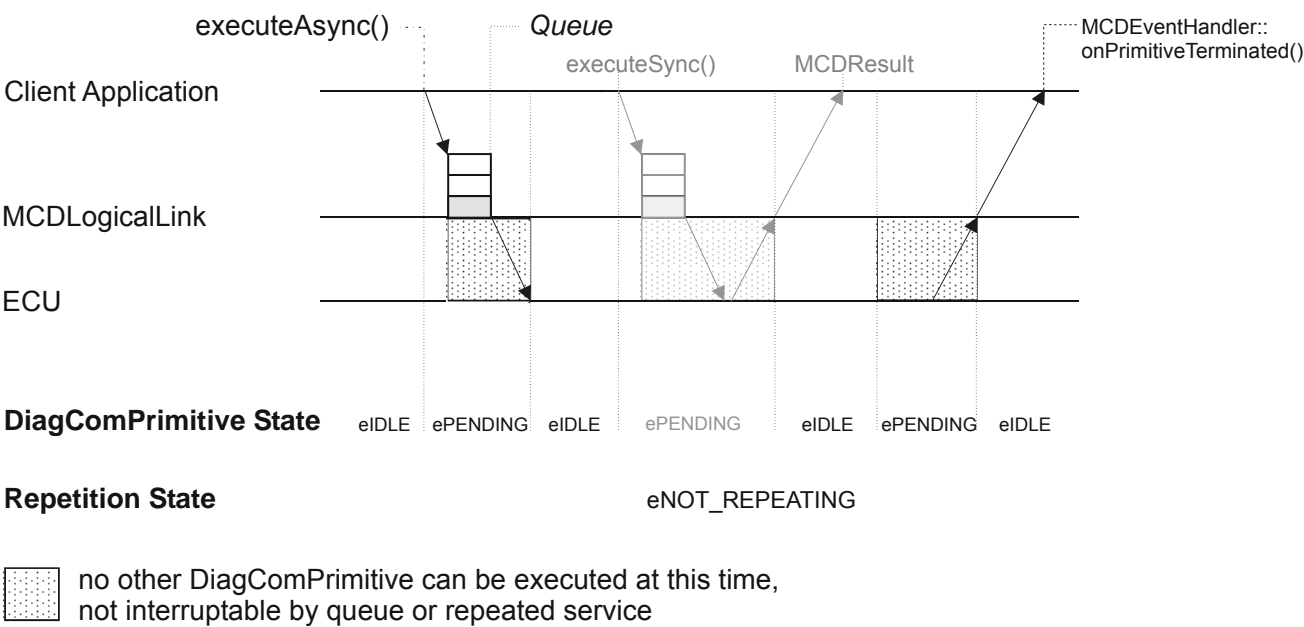
Figure 47 — Non cyclic single diagnostic service (asynchronous executed outside jobs)

IMPORTANT — After sending the request it is possible to start other services from the queue to the ECU. This is depended from the used protocol, this means it is not necessary to wait until the response is available.

Figure 48 depicts the non cyclic single diagnostic service (synchronous executed outside jobs).
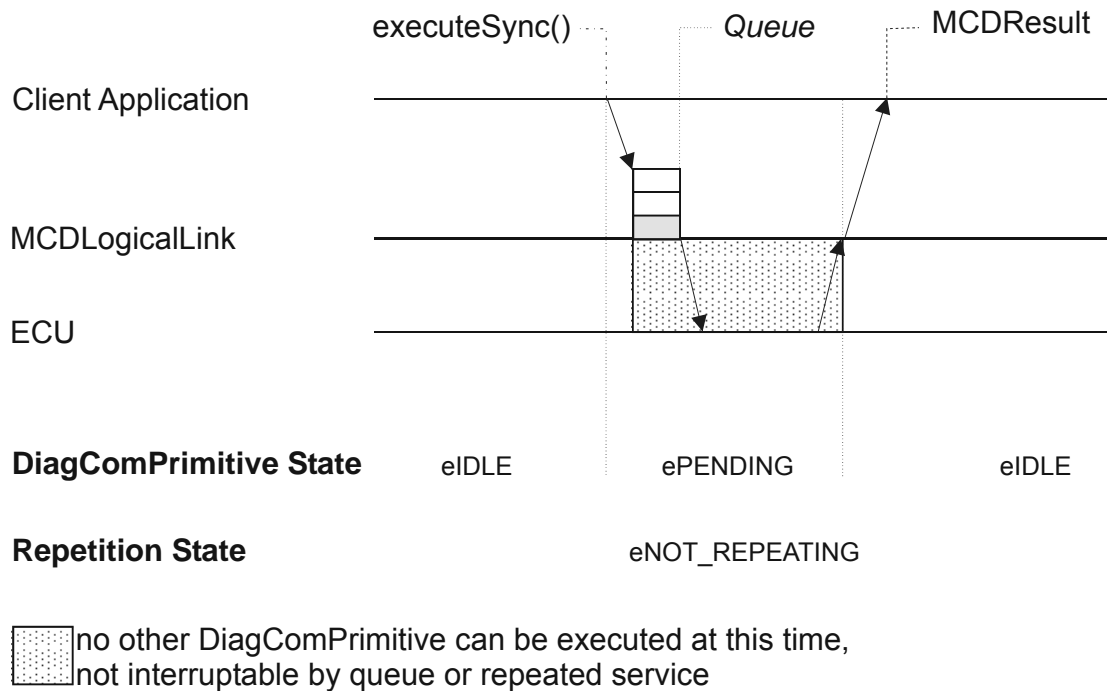
Figure 48 — Non cyclic single diagnostic service (synchronous executed outside jobs)

Sample: normal diagnostic service as specified in ISO 14230-3 KWP 2000 (e.g. ReadDTC)

DiagComPrimitive method description:

executeSync()   synchronous start of DiagComPrimitive execution

executeAsync()   asynchronous start of DiagComPrimitive execution

cancel()   quit DiagComPrimitive execution as fast as possible or remove it from execution activity queue

States:

The repetition state of the DiagComPrimitive execution is eNOT_REPEATING.

The DiagComPrimitive state changes from eIDLE (initially; state before starting DiagComPrimitive execution) to ePENDING (state while execution) back to eIDLE (state after execution). The states of the DiagComPrimitive are set by the MVCI diagnostic server.

Results:

There can be only 0 or 1 result. There cannot be intermediate results. The result is the return value of the synchronous execution. The result state can be requested from the Service itself.

Functional addressing delivers also only one complete result.

Figure 49 depicts the non cyclic single diagnostic service or job (executed inside jobs).
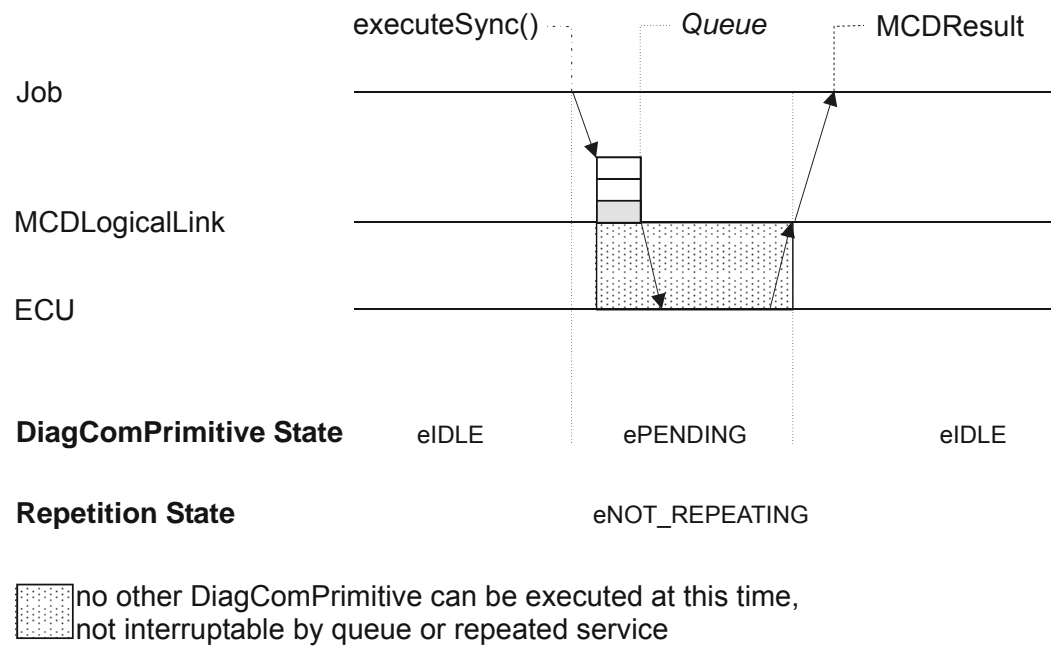
Figure 49 — Non cyclic single diagnostic service or job (executed inside jobs)

Remark:

Inside a job only non cyclic single diag services and jobs can be executed and these must be started synchronous.

Results:

There can be only 0 or 1 result. There cannot be intermediate results.

The result is the return value of the synchronous execution. The result state can be requested from the Service itself.

Functional addressing delivers also only one complete result.

The results of this service will be evaluated inside the job (see 8.19).

8.3.4 Cyclic diagnostic service

Figure 50 depicts the Cyclic diagnoctic services.
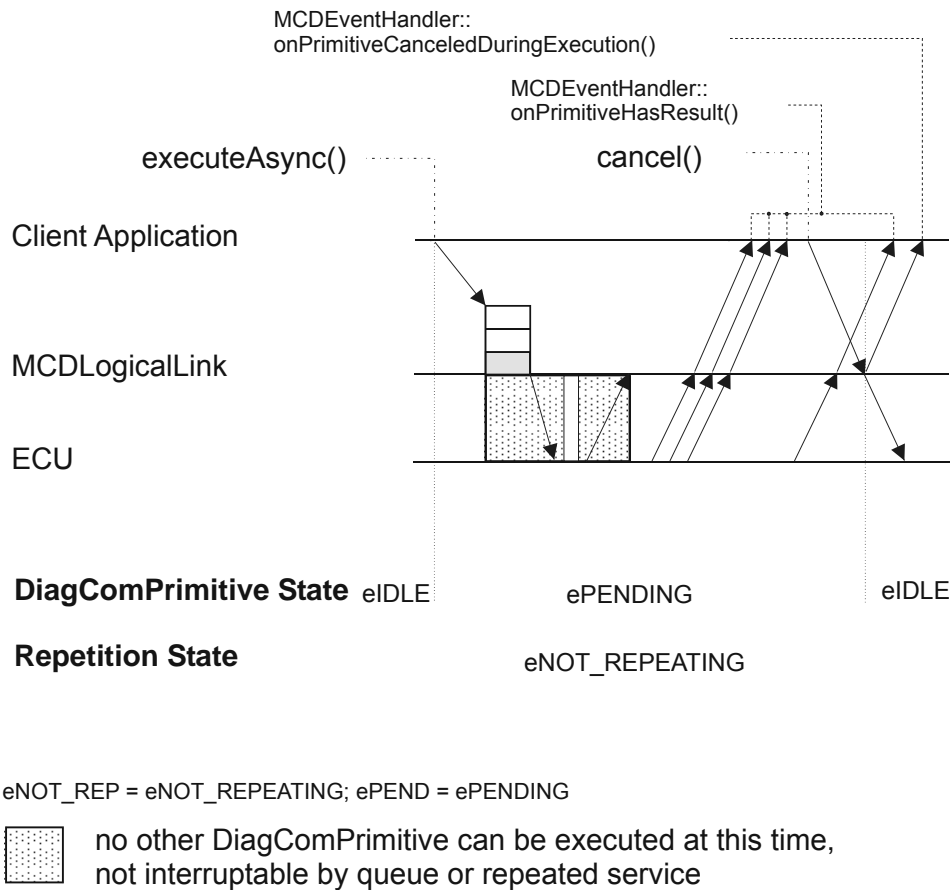
Figure 50 — Cyclic diagnostic services

Sample: ISO 14230-3 KWP 2000 periodic transmission

DiagComPrimitive method description:

executeAsync()   asynchronous start of DiagComPrimitive execution

cancel()   quit DiagComPrimitive execution as fast as possible or remove it from execution queue

States:

The repetition state of the DiagComPrimitive execution is eNOT_REPEATING.

The DiagComPrimitive state changes from eIDLE (initially; state before starting DiagComPrimitive execution) to ePENDING (state while execution) back to eIDLE (state after execution). The states of the DiagComPrimitive are set by the MVCI diagnostic server. While ePENDING several results can be reported via event and requested from ring buffer. The execution will normally be terminated by cancel(). But a termination

with TimeOut or BusError is also possible and not in all use cases an indication of errors (e.g. in ECU a loop with 100 data registrations). In this case a `onPrimitiveTerminated` event will be sent.

Results:

There can be several (zero or more) results, stored in the ring buffer. For every result there is an event sent to the Client Application. The results do not have to occur in equidistant time intervals.

There cannot be intermediate results. Requesting the result state, the number of results or the result(s) is allowed after getting one of the events `onPrimitiveHasResult` or `onPrimitiveCanceledDuringExecution`.

8.3.5 Repeated diag service

Figure 51 depicts the Repeated diagnostic service.
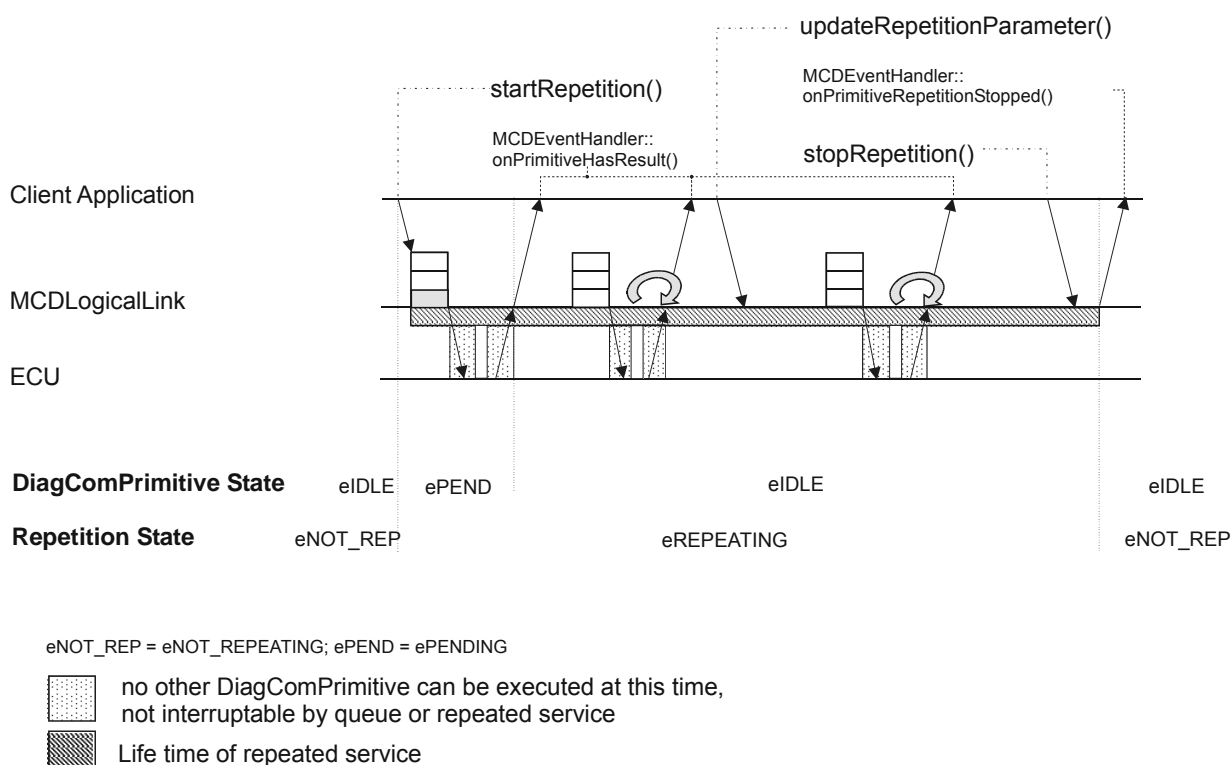
Figure 51 — Repeated diagnostic service

Description: Repeated execution of a NON CYCLIC DIAG SERVICE

The time between two repeated executions will be set by the client application and is not stored in data base.

DiagComPrimitive method description:

`startRepetition()`   start of DiagComPrimitive execution, after passing the queue, the service will live in a loop and start action (not through the queue)

```
stopRepetition()                        quit DiagComPrimitive execution

cancel()                                quit DiagComPrimitive execution as fast as possible

updateRepetitionParameter ()            in the state eIDLE the service parameter can be changed,
                                        this method goes not through the queue
```

States:

The repetition state changes from `eNOT_REPEATING` (before `startRepetition()`) to `eREPEATING` (after `startRepetition()` and back to `eNOT_REPEATING` (after `stopRepetition()` or `cancel()`).

The DiagComPrimitive state changes from `eIDLE` to `ePENDING` is made every time the Client Application starts a method, that goes through the queue till end of this method (e.g. `startRepetition()`), not for repeated execution, `updateRepetionParameters` and `stopRepition`.

Results:

There can be one or more results, stored in the ring buffer. There cannot be intermediate results. Requesting the result state, the number of results or the result(s) is allowed after getting one of the events `onPrimitiveHasResult` or `onPrimitiveRepetitionStopped`.

8.3.6 Repeated send only diag service

Figure 52 depicts the Repeated send only diagnostic service.
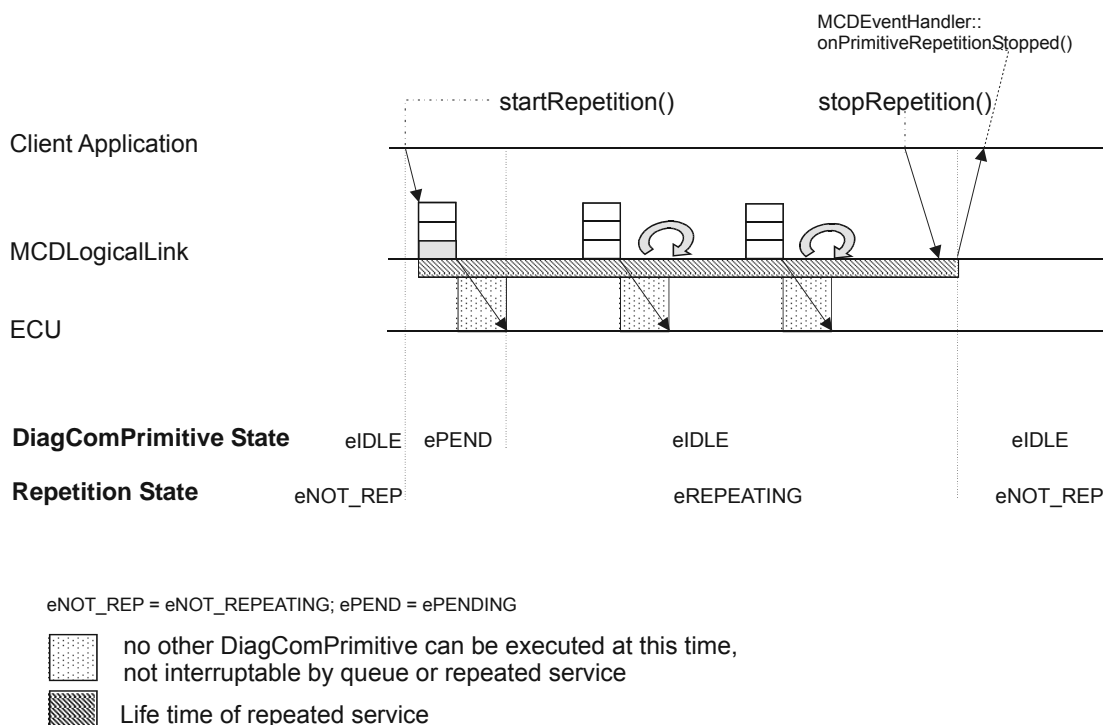
Figure 52 — Repeated send only diagnostic service

Sample: 1. send single CAN frames (ISO 14229-3 UDSonCAN unacknowledged unsegmented data transfer)

2. RestBus Simulation

Description: Special case of Repeated Diagnostic Service

Results: There are no results.

8.3.7   Repeated receive only diag service

Figure 53 depicts the Repeated receive only diagnostic service.
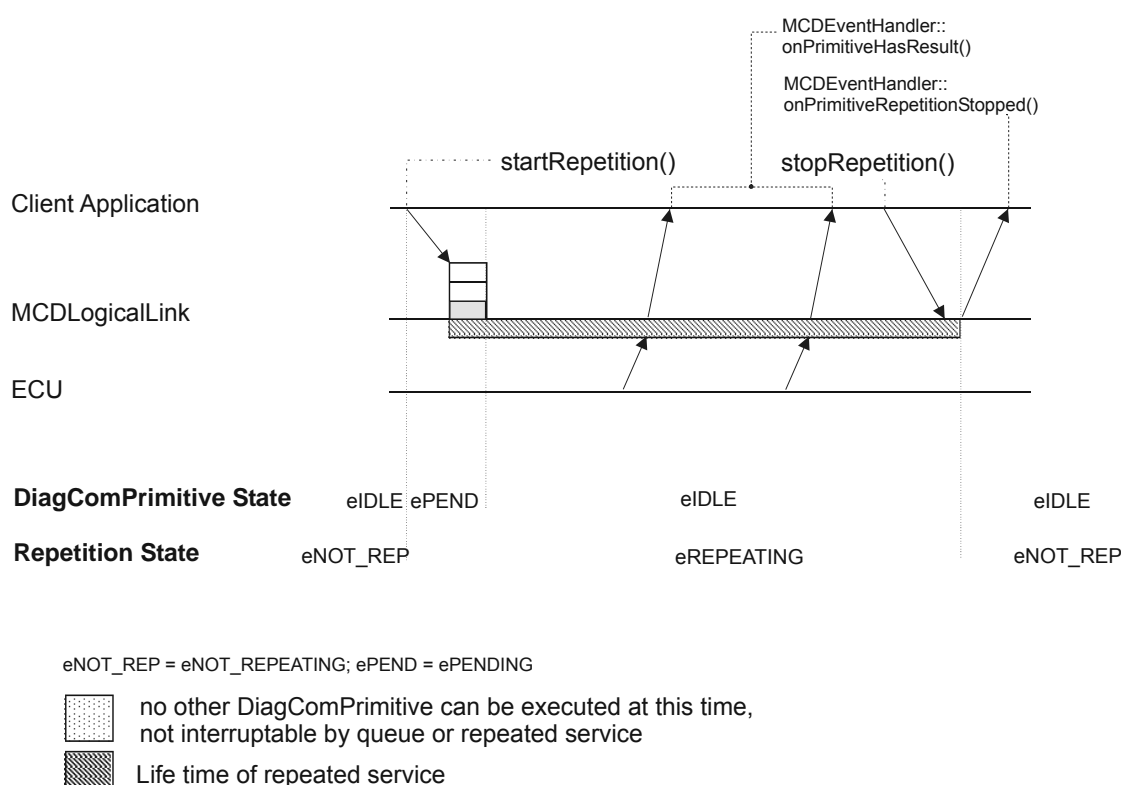
Figure 53 — Repeated receive only diagnostic service

Sample: 1. ISO 15765 receive single CAN frames

2. ISO 14229 / Version 2.0.0.0 response on event

Description: Special case of Repeated Diag Service

Results:

There can be one or more results, stored in the ring buffer. There cannot be intermediate results. Requesting the result state, the number of results or the result(s) is allowed after getting one of the events `onPrimitiveHasResult` or `onPrimitiveRepetitionStopped`.

8.3.8   Summary

Table 12 shows the overview about methods and events.

Table 12 — Overview about methods and events

ServiceType	Start method	Stop method	Event
NonCyclicDiagService	executeSync executeAsync		onPrimitiveTerminated
CyclicDiagService	executeAsync	cancel	onPrimitiveCanceldDuringExecution
RepeatedDiagService	startRepetition	stopRepetition	onPrimitiveRepetitionStopped

Table 13 defines the events in case of single or repeated execution of DiagService and Jobs.

Table 13 — Events in case of single or repeated execution of DiagService and Jobs

	Single	Repeated
DiagService	onPrimitiveTerminated	onPrimitiveHasResult
		onPrimitiveRepetitionStopped
Job	onPrimitiveHasIntermediateResult ②	onPrimitiveHasIntermediateResult ②
	onPrimitiveTerminated ①	onPrimitiveHasResult ①
		onPrimitiveRepetitionStopped

The Job API's `sendFinalResult` ① raises an event of type `onPrimitiveTerminated` in case of non repeated execution and raises `onPrimitiveHasResult` in case of repeated execution. The Job API's `sendIntermediateResult` ② raises an event of type `onPrimitiveHasIntermediateResult` independent from case of single or repeated execution.

8.3.9   Protocol parameters

8.3.9.1   General

For a diagnostic server to exchange data with an ECU, it needs to be able to correctly set up the communications link to be used, e.g. regarding baud rate, protocol timings, tester present behaviour and so on. To be able to handle these kinds of settings in a protocol-independent manner, the concept of protocol parameters (also called communication parameters in this document) has been introduced. Protocol parameters are used to define all settings that are relevant for diagnostic communication, and described by specific elements within an ODX data set. The diagnostic server is agnostic concerning the contents of protocol parameters, and usually passes them on to its protocol layer implementation (e.g. a D-PDU API layer), which contains the according logic to understand and correctly use protocol parameter semantics. This section describes how protocol parameters are represented at the MVCI diagnostic server API, and how client applications can modify and use protocol parameters to configure a communication link's behaviour.

8.3.9.2 Introduction related to ODX 2.2

Figure 54 depicts the PROT-STACKS and its COMPARAM-SUBSETs.
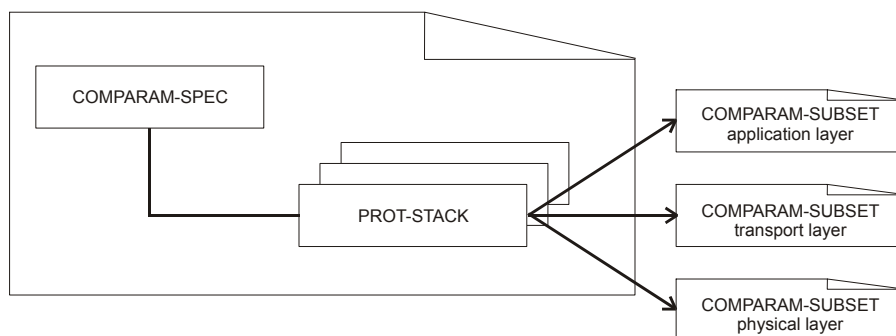
Figure 54 — PROT-STACKS and its COMPARAM-SUBSETs

Protocol parameters are defined within an ODX part called a PROT-STACK. Within a PROT-STACK, communication parameters are grouped into different COMPARAM-SUBSETs, usually with respect to the OSI layer they belong to (e.g. physical-, transport- or protocol-layer). At the MVCI diagnostic server API, communication parameters are not distinguished according to the different COMPARAM-SUBSET/OSI layers they belong to; rather, a superset of all available communication parameters (within the active PROT-STACK) will be delivered to the client application. Exactly one PROT-STACK is used during runtime and defines the active communication parameter set for a communication link. The communication parameter values defined within the PROT-STACK (default values) can be redefined at different contexts like diagnostic layers (DIAG-LAYER), diagnostic service (DIAG-SERVICE), logical link and physical vehicle link. There are dependencies between the different redefinition contexts. For example, the redefinition of a communication parameter at PROTOCOL level is also applied in all the more specialized DIAG-LAYERS (e.g. BASE-VARIANT) using this PROTOCOL. A complete description of redefinition of communication parameters can be found in ISO 22901-1. In general, two types of communication parameter exist in ODX: The simple COMPARAM holding a single value of a simple data type and the COMPLEX-COMPARAM, which contains a complex structure of values. The following list shows the general mapping between communication parameter types defined in ODX and types of MCDDbProtocolParameters.

— A COMPLEX-COMPARAM with ALLOW-MULTIPLE-VALUES set to 'true' results in a protocol parameter with datatype eSTRUCT_FIELD containing an arbitrary number of COMPLEX-COMPARAMs. The shortnames of the inner structure elements are generated by the diagnostic server according to the following pattern: #RtGen_<NameOfComplexComParam>_<unique_number>.

— A COMPLEX-COMPARAM with ALLOW-MULTIPLE-VALUES set to 'false' results in a protocol parameter with datatype eSTRUCTURE containing an arbitrary number of simple COMPARAMs or COMPLEX-COMPARAMs.

— A simple COMPARAM results in a protocol parameter with parameter type eVALUE. The datatype of such a protocol parameter is defined by the DOP of this parameter in ODX.

In case of a COMPLEX-COMPARAM with ALLOW-MULTIPLE-VALUES set to true (mapped to eSTRUCT_FIELD), the COMPLEX-COMPARAM value can be redefined by multiple-value structures within one context, which results in multiple-field items.

Such a COMPLEX-COMPARAM cannot be redefined partially, i.e. it is not possible to overwrite only parts of a COMPLEX-COMPARAM that is inherited from a higher layer. A redefinition will always substitute the entire COMPLEX-COMPARAM, regardless whether the new definition contains less data than the overridden one. For example, consider the COMPLEX-COMPARAM CP_SessionTimingOverride, which contains an array of structures containing session timing data, one structure for each defined session. If this parameter is redefined in a DIAG-LAYER (overriding a definition that e.g. contains timing data for sessions A and B), the new definition again has to provide array entries for all sessions where a timing override should be applied at runtime. If CP_SessionTimingOverride is redefined for session A but not for session B, no timing override is applied at runtime for session B.

In contrast to COMPLEX-COMPARAMs with ALLOW-MULTIPLE-VALUES set to false, the order of sub-parameters (field items) of a COMPLEX-COMPARAM with ALLOW-MULTIPLE-VALUES set to true doesn't carry any information. Therefore, a diagnostic server does not have to guarantee the same sequence for such field items as defined in ODX. Redefinition of communication parameter values is done by the ODX element COMPARAM-REF with an attached value definition. Only a COMPLEX-COMPARAM with ALLOW-MULTIPLE-VALUES set to true might be referenced by a COMPARAM-REF without a value definition to reset the communication parameter value to an empty field.

At the diagnostic server API, communication parameters are represented by `MCDDbRequestParameter` or `MCDRequestParameter` objects. In the following, the `MCDDbRequestParameter` and `MCDRequestParameter` objects are both called ProtocolParameters if there is no need to distinguish between both types.

Note: the `MCDRequestParameters` of a `MCDProtocolParameterSet` control primitive represent the protocol parameters of the Logical Link.

As an example, the protocol parameter structure at the diagnostic server API level for the complex protocol parameter CP_SessionTimingOverride is illustrated in the figure below, followed by the appropriate protocol parameter data set in ODX.

Figure 55 depicts the complex Comparam CP_SessionTimingOverride at MVCI diagnostic server level.
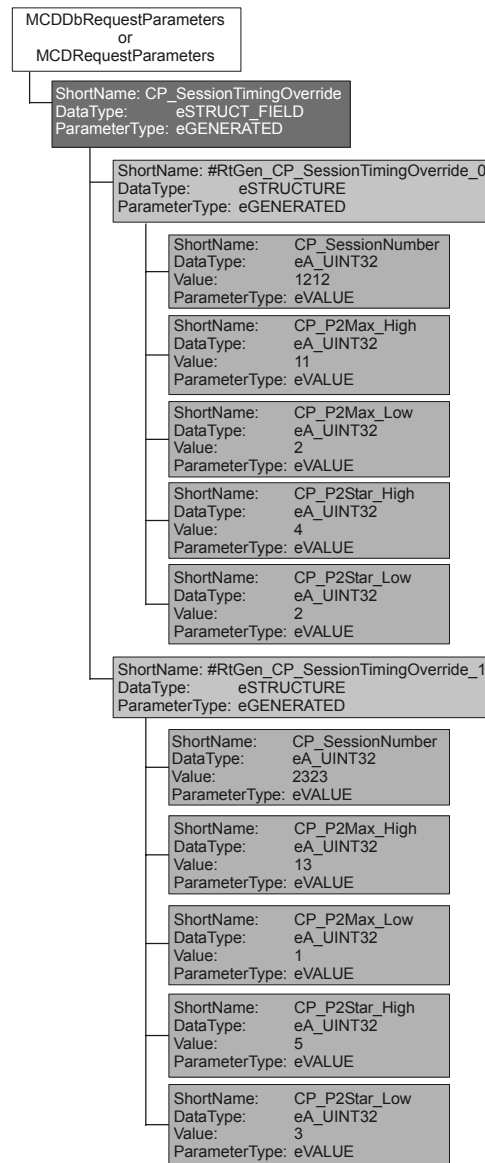
Figure 55 — Complex Comparam CP_SessionTimingOverride at MVCI diagnostic server level

```
<COMPARAM-REFS>
   <COMPARAM-REF ID-REF="ISO_15765_3.CP_SessionTimingOverride">
         <COMPLEX-VALUE>
      <SIMPLE-VALUE>1212</SIMPLE-VALUE>
      <SIMPLE-VALUE>11</SIMPLE-VALUE>
      <SIMPLE-VALUE>2</SIMPLE-VALUE>
      <SIMPLE-VALUE>4</SIMPLE-VALUE>
      <SIMPLE-VALUE>2</SIMPLE-VALUE>
         </COMPLEX-VALUE>
   </COMPARAM-REF>
```

```
    <COMPARAM-REF ID-REF="ISO_15765_3.CP_SessionTimingOverride">
          <COMPLEX-VALUE>
      <SIMPLE-VALUE>2323</SIMPLE-VALUE>
      <SIMPLE-VALUE>13</SIMPLE-VALUE>
      <SIMPLE-VALUE>1</SIMPLE-VALUE>
      <SIMPLE-VALUE>5</SIMPLE-VALUE>
      <SIMPLE-VALUE>3</SIMPLE-VALUE>
          </COMPLEX-VALUE>
      </COMPARAM-REF>
</COMPARAM-REFS>
```

The ODX definition of the complex protocol parameter CP_SessionTimingOverride used by the example above looks as follows:

```
<COMPLEX-COMPARAM                    ID="ISO_15765_3.CP_SessionTimingOverride"
CPTYPE="OPTIONAL"  CPUSAGE="TESTER"  PARAM-CLASS="TIMING"  ALLOW-MULTIPLE-
VALUES="true">
   <SHORT-NAME>CP_SessionTimingOverride</SHORT-NAME>
   <COMPARAM       ID="ISO_15765_3.CP_SessionNumber"       CPTYPE="OPTIONAL"
CPUSAGE="TESTER" PARAM-CLASS="TIMING">
      <SHORT-NAME>CP_SessionNumber</SHORT-NAME>
      <PHYSICAL-DEFAULT-VALUE>0</PHYSICAL-DEFAULT-VALUE>
      <DATA-OBJECT-PROP-REF ID-REF="ISO_15765_3.DOP_IDENTICAL_16Bit"/>
   </COMPARAM>
   <COMPARAM         ID="ISO_15765_3.CP_P2Max_High"        CPTYPE="OPTIONAL"
CPUSAGE="TESTER" PARAM-CLASS="TIMING">
      <SHORT-NAME>CP_P2Max_High</SHORT-NAME>
      <PHYSICAL-DEFAULT-VALUE>0</PHYSICAL-DEFAULT-VALUE>
      <DATA-OBJECT-PROP-REF ID-REF="ISO_15765_3.DOP_IDENTICAL_8Bit_1ms"/>
   </COMPARAM>
   <COMPARAM          ID="ISO_15765_3.CP_P2Max_Low"        CPTYPE="OPTIONAL"
CPUSAGE="TESTER" PARAM-CLASS="TIMING">
      <SHORT-NAME>CP_P2Max_Low</SHORT-NAME>
      <PHYSICAL-DEFAULT-VALUE>0</PHYSICAL-DEFAULT-VALUE>
      <DATA-OBJECT-PROP-REF ID-REF="ISO_15765_3.DOP_IDENTICAL_8Bit_1ms"/>
   </COMPARAM>
   <COMPARAM         ID="ISO_15765_3.CP_P2Star_High"       CPTYPE="OPTIONAL"
CPUSAGE="TESTER" PARAM-CLASS="TIMING">
      <SHORT-NAME>CP_P2Star_High</SHORT-NAME>
      <PHYSICAL-DEFAULT-VALUE>0</PHYSICAL-DEFAULT-VALUE>
      <DATA-OBJECT-PROP-REF                                            ID-
REF="ISO_15765_3.DOP_LINEAR_8Bit_Resolution_10ms"/>
   </COMPARAM>
   <COMPARAM         ID="ISO_15765_3.CP_P2Star_Low"        CPTYPE="OPTIONAL"
CPUSAGE="TESTER" PARAM-CLASS="TIMING">
      <SHORT-NAME>CP_P2Star_Low</SHORT-NAME>
      <PHYSICAL-DEFAULT-VALUE>0</PHYSICAL-DEFAULT-VALUE>
      <DATA-OBJECT-PROP-REF                                            ID-
REF="ISO_15765_3.DOP_LINEAR_8Bit_Resolution_10ms"/>
   </COMPARAM>
  <COMPLEX-PHYSICAL-DEFAULT-VALUE></COMPLEX-PHYSICAL-DEFAULT-VALUE>
</COMPLEX-COMPARAM>
…..
<DATA-OBJECT-PROP ID="ISO_15765_3.DOP_IDENTICAL_8Bit_1ms">
   <SHORT-NAME>DOP_IDENTICAL_8Bit_1ms</SHORT-NAME>
```

```
    <LONG-NAME>DOP_IDENTICAL_8Bit_1ms</LONG-NAME>
    <COMPU-METHOD>
        <CATEGORY>IDENTICAL</CATEGORY>
    </COMPU-METHOD>
    <DIAG-CODED-TYPE  BASE-DATA-TYPE="A_UINT32"  xsi:type="STANDARD-LENGTH-
TYPE">
        <BIT-LENGTH>8</BIT-LENGTH>
    </DIAG-CODED-TYPE>
    <PHYSICAL-TYPE BASE-DATA-TYPE="A_UINT32"/>
    <UNIT-REF ID-REF="ISO_15765_3.ms"/>
</DATA-OBJECT-PROP>
<DATA-OBJECT-PROP ID="ISO_15765_3.DOP_LINEAR_8Bit_Resolution_10ms">
    <SHORT-NAME>DOP_LINEAR_8Bit_Resolution_10ms</SHORT-NAME>
    <LONG-NAME>DOP_LINEAR_8Bit_Resolution_10ms</LONG-NAME>
    <COMPU-METHOD>
        <CATEGORY>LINEAR</CATEGORY>
        <COMPU-INTERNAL-TO-PHYS>
        <COMPU-SCALES>
        <COMPU-SCALE>
        <COMPU-RATIONAL-COEFFS>
        <COMPU-NUMERATOR>
        <V>0</V>
        <V>10</V>
        </COMPU-NUMERATOR>
        <COMPU-DENOMINATOR>
        <V>1</V>
        </COMPU-DENOMINATOR>
        </COMPU-RATIONAL-COEFFS>
        </COMPU-SCALE>
        </COMPU-SCALES>
        </COMPU-INTERNAL-TO-PHYS>
    </COMPU-METHOD>
    <DIAG-CODED-TYPE  BASE-DATA-TYPE="A_UINT32"  xsi:type="STANDARD-LENGTH-
TYPE">
        <BIT-LENGTH>8</BIT-LENGTH>
    </DIAG-CODED-TYPE>
    <PHYSICAL-TYPE BASE-DATA-TYPE="A_UINT32"/>
    <UNIT-REF ID-REF="ISO_15765_3.ms"/>
</DATA-OBJECT-PROP>
<DATA-OBJECT-PROP ID="ISO_15765_3.DOP_IDENTICAL_16Bit">
    <SHORT-NAME>DOP_IDENTICAL_16Bit</SHORT-NAME>
    <LONG-NAME>DOP_IDENTICAL_16Bit</LONG-NAME>
    <COMPU-METHOD>
        <CATEGORY>IDENTICAL</CATEGORY>
    </COMPU-METHOD>
    <DIAG-CODED-TYPE  BASE-DATA-TYPE="A_UINT32"  xsi:type="STANDARD-LENGTH-
TYPE">
        <BIT-LENGTH>16</BIT-LENGTH>
    </DIAG-CODED-TYPE>
    <PHYSICAL-TYPE BASE-DATA-TYPE="A_UINT32"/>
</DATA-OBJECT-PROP>
```

In ODX, protocol parameters are classified into three different categories:

— MCDProtocolParameterClass (ODX: PARAM-CLASS),

— MCDProtocolParameterType (ODX: CPTYPE),

— MCDProtocolParameterUsage ODX: CPUSAGE);

The enumeration MCDProtocolParameterClass comprises the following items:

— eBUSTYPE: This class of parameters is used to define bus type specific parameters (e.g. baud rate). Most of these parameters affect the physical hardware. These parameters can only be modified by the first Logical Link that acquired the physical resource. When a second Logical Link is created for the same resource, these parameters that were previously set will be active for the new Logical Link.

— eCOM: General communication parameters.

— eERRHDL: Parameter defining the behaviour of the runtime system in case an error occurred, e.g. the runtime system could either continue communication after a timeout was detected, or stop and reactivate the communication link.

— eINIT: Parameters for initiation of communication e.g. trigger address or wakeup pattern. These parameters shall not be overwritten within an ECU-Variant layer in any way.

— eTESTER_PRESENT: This type of communication parameter is relevant for the various aspects of tester present functionality, e.g. they determine tester present timeout settings or tester present message contents.

— eTIMING: Message flow timing parameters, e.g. inter-byte-time or time between request and response.

— eUNIQUE_ID: This type of communication parameter is used to indicate to both the ComLogicalLink and to the application that the information is used for protocol response handling from a physical or functional group of ECUs to uniquely identify an ECU response.

The enumeration MCDProtocolParameterType comprises the following items:

— eSTANDARD: A communication parameter belonging to a standardized protocol that has to be supported by a runtime system implementing this standardized protocol. Diagnostic data using a protocol not supported by the runtime system cannot be executed by the diagnostic server.

— eOPTIONAL: This communication parameter does not have to be supported by the runtime system. If a DIAG-COMM uses an unsupported communication parameter of this type, the parameter can be ignored and the DIAG-COMM can be executed nevertheless.

— eOEM-SPECIFIC: The communication parameter is part of a non-standardized OEM-specific protocol; nevertheless it is required to be implemented by the runtime system. Diagnostic data using an OEM-specific protocol not supported by the runtime system cannot be executed by the diagnostic server.

— eOEM-OPTIONAL: This communication parameter is a non-standardized, OEM-specific parameter that does not have to be supported by the protocol layer implementation of the runtime system (D-PDU API).

— As stated above, DiagComPrimitives with associated protocol parameters of type STANDARD or OEM-SPECIFIC will not be executed by the diagnostic server if one or more of these protocol parameters are not supported by the protocol driver (D-PDU API). In this case, a MCDCommunicationException of type eCOM_COMPARAM_NOT_SUPPORTED will be thrown when the client application tries to execute such a DiagComPrimitive. The same applies to the method MCDLogicalLink::gotoOnline() if at least one of the protocol parameters to be set is not supported by the protocol driver.

The enumeration MCDProtocolParameterUsage comprises the following items:

⎯ eECU-COMM parameters are relevant for basic properties of the communication channel with an ECU (e.g. timings and addresses).

⎯ eECU-SOFTWARE parameters are only used for ECU software generation and configuration. Communication parameters of this type shall be ignored by the diagnostic server.

⎯ eAPPLICATION parameters are only evaluated by the client application. Communication parameters of this type shall not be passed down to the D-PDU API by the diagnostic server, but they shall be accessible via the diagnostic server API.

⎯ eTESTER parameters are only valid to the tester during diagnostic communication; they are not relevant to the use case of ECU software generation and calibration.

The DISPLAY-LEVEL of COMPARAMs is used to restrict the visibility (and therefore changeability) of the COMPARAMs in a client application. Therefore the diagnostic server only delivers the DISPLAY-Level to the client application, which has to implement any functionality based on display level information.

8.3.9.3   Inheritance of protocol parameters

During runtime a diagnostic layer is linked with a PROT-STACK to import a valid set of communication parameters (COMPARAM and COMPLEX-COMPARAM elements). This unambiguous link is defined via the PROTOCOL or the LOGICAL-LINK. The COMPARAM values are inherited between the different layers. An inherited COMPARAM value is identical to that in the parent layer, i.e. it has the same value. A communication parameter has to be overridden to change its value.

Figure 56 depicts the inheritance of COMPARAMs between different layers.
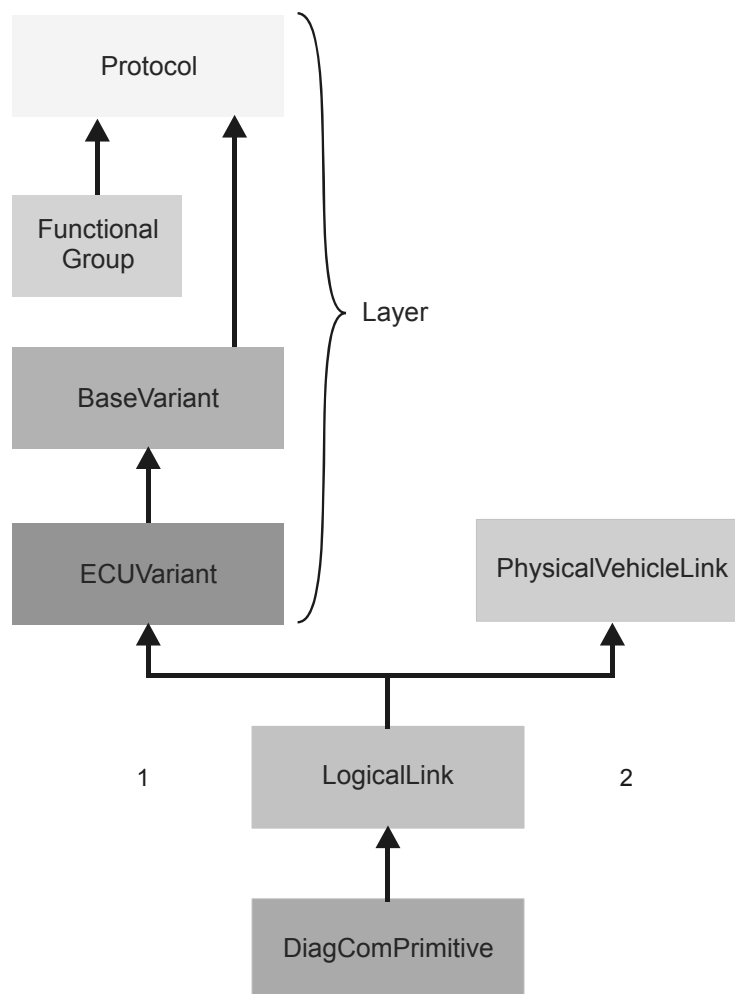
Key

1   If there is a protocol parameter defined at the layer and the the physical vehicle link, it will be taken from the link.

2   The logical link cannot redefine protocol parameters from the physical vehicle link (checker-rule).

Figure 56 — Inheritance of COMPARAMs between different layers

Protocol parameters can be overwritten at different layers to change their value. The following issues have to be considered:

⎯ Overwritten communication parameter values on a FUNCTIONAL-GROUP level are not inherited by lower layers (i.e. the inheriting base variants).

⎯ Base variants inherit their set of communication parameter values directly from the protocol layer, and can then override them with locally defined values.

⎯ Since a logical link always includes one dedicated protocol, all multiple inheritance issues can be resolved unambiguously at runtime.

A simplified inheritance example for simple communication parameters without redefinition of communication parameters at Physical Vehicle Link and Logical Link is illustrated in Figure 57, which shows a PROTOCOL instance (PROT-A) referencing the PROT-STACK instance PS-B as the active PROT-STACK. As a consequence, the valid set of communication parameters consists of A, B and C. Because COMPARAM A value is overridden within the protocol layer, the valid values in the scope of PROT-A are A=4, B=5 and C=15.
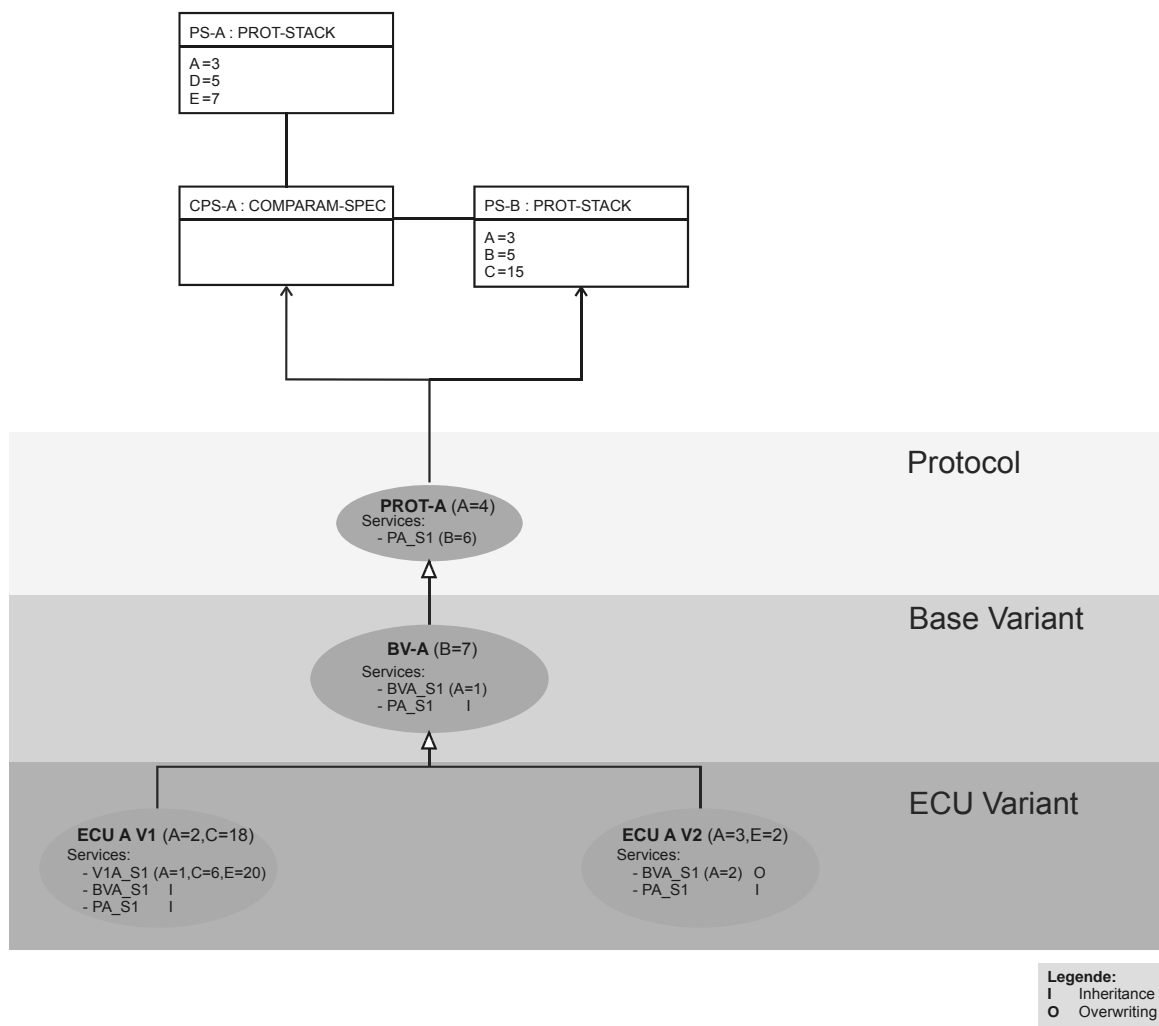
Figure 57 — UML representation of inheritance of communication parameters (example)

These values apply to all the DIAG-COMM that do not override a communication parameter value themselves. The DIAG-SERVICE with the identifier PA_S1 in the PROTOCOL PROT-A shown in the example overrides the COMPARAM B value. This is done within the data by a COMPARAM-REF element. The COMPARAM set for this special DIAG-SERVICE results in [A=4, B=6, C=15].

The BASE-VARIANT instance BV-A overrides the COMPARAM B value. All other COMPARAM values are derived from the protocol layer PROT-A. The COMPARAM set in the scope of BV-A is [A=4, B=7, C=15]. Two DIAG-COMMs are available in this base variant layer. The DIAG-SERVICE BVA_S1 defined within BV-A and the DIAG-SERVICE PA_S1 derived from the parent layer PROT-A. The COMPARAM set for PA_S1 at the base variant level is [A=4, B=6, C=15] and COMPARAM set for BVA_S1 is [A=1, B=7, C=15].

There are two instances of an ECU-VARIANT both inherited from BV-A. For ECU-A-V1 the COMPARAM set is [A=2, B=7, C=18]. There are three DIAG-SERVICEs available at ECU-A-V1. The inherited PA_S1 with [A=2, B=6, C=18], the inherited BVA_S1 with [A=1, B=7, C=18] and locally defined V1A_S1 with [A=1, B=7, C=6]. Because the COMPARAM E is not part of the current communication parameter set the COMPARAM-REF element at V1A_S1 that tries to override the COMPARAM E value exclusively for the DIAG-SERVICE has no affect and can be ignored. The ECU-VARIANT instance ECU-A-V2 applies the COMPARAM values [A=3, B=7, C=15]. E is ignored because it's not part of the active PROT-STACK. The available DIAG-SERVICEs are the value inherited PA_S1 with [A=3, B=6, C=15] and the locally defined BVA_S1 which overrides the BVA_S1 defined at the layer above. The COMPARAM set for BVA_S1 in the scope of ECU-A-V2 is [A=2, B=7, C=15].

8.3.9.4   Runtime part

Local and global protocol parameters

At runtime, it is distinguished between "local" protocol parameters at diagnostic services or certain control primitives (`MCD(Db)DiagService`, `MCD(Db)StartCommunication` and `MCD(Db)StopCommunication`), and "global" protocol parameters which apply to entire Logical Links. For the ease of reading, we will refer to DiagComPrimitives in the remainder of this section instead of explicitly stating that protocol parameters are only available for elements of type `MCD(Db)DiagService`, `MCD(Db)StartCommunication` and `MCD(Db)StopCommunication`.

The local protocol parameter collection of a DiagComPrimitive is a subset of all protocol parameters of the related Logical Link. The values of all Logical Link protocol parameters that are temporarily valid for a DiagComPrimitive at execution time are determined through the rules described in the following paragraphs.

In general, a DiagComPrimitive is executed in the context of the protocol parameter values that are currently valid at the Logical Link – except when the local protocol parameters collection of the DiagComPrimitive is not empty; in this case, the value of each local protocol parameter overwrites the value of the corresponding global protocol parameter. The diagnostic server has to set the DiagComPrimitive specific protocol parameters before executing the DiagComPrimitive, and has to restore the original protocol parameter settings afterwards. These temporary changes of protocol parameters are only valid during this DiagComPrimitive's execution time. All other DiagComPrimitives are executed in their own context of protocol parameters consisting of the protocol parameters currently valid at the Logical Link and their own temporarily overwritten protocol parameters.

Global protocol parameters are set at the logical link by using the `MCDProtocolParameterSet` control primitive.

DiagComPrimitives of type `MCDProtocolParameterSet` are not handled differently from other DiagComPrimitives. That is, these ControlPrimitives can only be executed in Logical Link states `eONLINE` and `eCOMMUNICATION`. If a Logical Link needs to be prepared via protocol parameters before going into communication, the client application has to take care to execute the proper `MCDProtocolParameterSet` prior to all other DiagComPrimitives, especially `MCDStartCommunication`.

The protocol parameters for a Logical Link are set via the `MCDProtocolParameterSet::executeSync()` method. The effect is a global change of the protocol parameters of the link the `MCDProtocolParameterSet` primitive was created on. It is not allowed to execute a ControlPrimitive of type `MCDProtocolParameterSet` in case at least one repeated DiagComPrimitive is currently executed on the same Logical Link. In this case, an exception of type `MCDProgramViolationException` with error code `eRT_REPEATED_SERVICE_RUNNING` shall be thrown.

Figure 58 depicts the relation between database and hardware for the MCDRequestParameter.
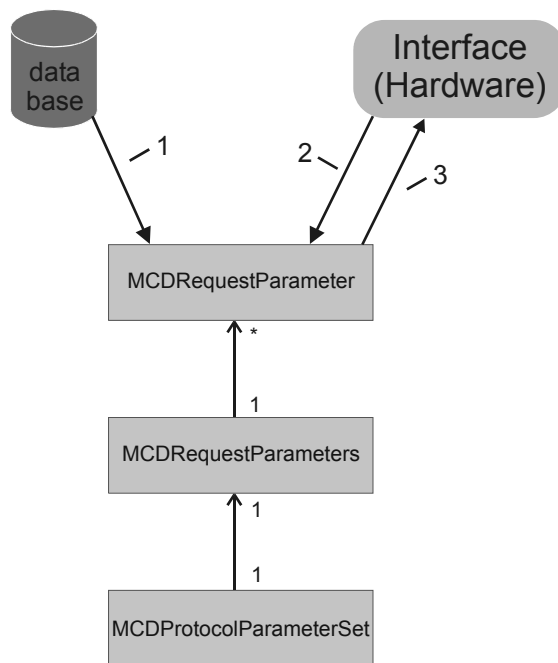
Key

1   `MCDProtocolParameterSet::resetToDefaultValues()` or
    `MCDProtocolParameterSet::resetToDefaultValues(parameterName)`

2   `MCDProtocolParameterSet::fetchValuesFromInterface()` or
    `MCDProtocolParameterSet::fetchValuesFromInterface(parameterName)`

3   `MCDProtocolParameterSet::executeSynch()`

Figure 58 — Relation between database and hardware for MCDRequestParameter

By means of the method `fetchValueFromInterface()` the current values from the interface will be set to the RequestParameters of the `MCDProtocolParameterSet` primitive.

Behaviour in case of non-exclusive setting of Protocol Parameters:

— The diagnostic server binds the current protocol parameters (current parameters in the interface, plus temporary parameters from ODX) to the DiagComPrimitive during its execution. This binding is valid as long as the DiagComPrimitive is active (the Logical Link the DiagComPrimitive is executed on is in state `MCDActivityState::eACTIVITY_RUNNING`, where the state change from an to eACTIVITY_IDLE has been caused by the execution of the DiagComPrimitive).

— If the diagnostic server, the protocol engine and the ECU allow nested/parallel execution of DiagComPrimitives, each DiagComPrimitive may have its own set of protocol parameters, and they are treated in the same way as above.

— If these conditions are valid, the setting of protocol parameters can be queued.

— The ProtocolParameterSet control primitive can only be executed synchronously.

When using `MCDProtocolParameterSet::executeSync()`, only those communication parameters will be set which have to be changed at the protocol layer (D-PDU API). That is, the ProtocolParameterSet will always be sent as an incomplete set (set of parameter to be changed) of communication parameters to the protocol layer (D-PDU API).

The diagnostic server maintains the mirror values of the protocol parameters of every Logical Link, and keeps track of all changes to the values since the last execution of the `MCDProtocolParameterSet` control primitive. On the next execution of `MCDProtocolParameterSet` control primitive, only the changed values will be written to the interface.

There are two ways for the diagnostic server to deal with protocol parameters that are not supported by its D-PDU API implementation. If the setting of a protocol parameter fails, the diagnostic server can continue operating and executing diagnostic services on the communications link that caused the failure. This behavior is most useful in engineering environments, where bus topologies and diagnostic data sets are still under development. Alternatively, the diagnostic server can refuse the execution of diagnostic services on a link when the setting of a protocol parameter has not been successful, as could be appropriate in more restricted environments (e.g. an after sales workshop), to ensure consistent behavior and functionality of vehicle diagnostics.

It is up to the client application whether to use OEM specific protocol parameters. With the method `MCDSystem::isUnsupportedComParametersAccepted()` it can be determined if the diagnostic server will be accepting non-standard protocol parameters. The method `MCDLogicalLink::unsupportedComParametersAccepted(A_BOOLEAN)` can be used to forbid the usage of OEM-specific protocol parameters for this specific Logical Link, but this will only have an effect if protocol parameters are accepted system wide; It is not possible to allow non-standard protocol parameters for a logical link when they are disallowed globally.

For a diagnostic server implementation, this has the following consequences: as the `MCDProtocolParameterSet` control primitive only allows the client application to set all protocol parameters that are valid for a link in one single operation, the diagnostic server has to provide protocol parameter consistency analogous to the transaction concept of a database system.

For example, imagine the case when a client application executes a `MCDProtocolParameterSet` primitive on a logical link, which includes five protocol parameters. For each of these five parameters, the diagnostic server in turn has to call a D-PDU API method, telling the protocol driver layer to set the desired value for each respective protocol parameter. Now, the setting of the third protocol parameter fails because it is not supported by the protocol driver. In case the diagnostic server has been configured to accept unsupported protocol parameters, it can continue with setting the remaining two parameters and then use that particular link for doing diagnostic communication.

However, if the diagnostic server is configured to not accept unsupported protocol parameters, it can't simply abort the execution of the `MCDProtocolParameterSet` at this point and return the corresponding error to the client application, because two of the five protocol parameters have already been successfully modified. Instead, it first has to reset the first two protocol parameters to their original values, so that the value set of protocol parameters of that link remains in a consistent state. This implies that in this case, the diagnostic server has to cache the original values of all protocol parameters that will be modified by a `MCDProtocolParameterSet` primitive, so it can undo any modifications in case the setting of any parameter fails. Please note that this kind of consistency also is required in case protocol parameters are set implicitly by the diagnostic server, e.g. when opening a new logical link, or when executing a `MCDDiagComPrimitive` that as associated overwritten protocol parameters.

The class `MCDProtocolParameterSet` inherits from class `MCDControlPrimitive`. Therefore, the execution of a `MCDProtocolParameterSet` primitive needs to generate an appropriate response object. This response object is empty (on the runtime as well as on the database side), that is the response collection of the `MCDProtocolParameterSet` has zero entries. Instead, the result can be obtained from the result collection, which can deliver the error code `eRT_PROTOCOLPARAMETERSET_FAILED` if the execution of a `MCDProtocolParameterSet` failed.

The Execution states in case of MCDProtocolParamterSet are:

— eALL_NEGATIVE: Setting of protocol parameters failed for at least one protocol parameter.

— eALL_POSITIVE: Setting of protocol parameters succeeded for at all protocol parameters.

`MCDDbProtocolParameterSet` is a runtime-generated object. Therefore the following methods

⎯ getDbResponses(),

⎯ getDbResponsesByType(…),

all deliver an empty collection.

MCDProtocolParameterSet

If a Java-Jobs needs to alter the currently valid protocol parameters of the Logical Link, it should use and execute a `MCDProtocolParameterSet` from within its code. Please note that all changes to protocol parameters caused by a `MCDProtocolParameterSet` executed within a Java-Job will be persistent after this job has terminated – just as if a client application would have issued the changes. So, a clean Job implementation has to restore the protocol parameters at the end of the Job execution. Please note that the usage of `MCDProtocolParameterSet` in a Java-Job is considered harmful, as it could cause undocumented and therefore unexpected changes to the protocol parameters of a logical link at runtime.

8.4   Suppress positive response

The suppress positive response feature allows to ask the ECU not to send any positive response to the request of the current DiagComPrimitive. This allows to decrease the load on the communication bus and the processing time in the protocol layer and MVCI diagnostic server. However, a negative response may be sent by the ECU any time. In addition, the ECU may send a positive response after it had sent a response pending as first answer just before.

If the suppress positive response feature is enabled for a `MCDService` or `MCDStart/Stop Communication` primitive, the "suppress positive response" bit in the protocol data stream is set by the protocol layer, e.g. the D-PDU API, if applicable. In an MVCI diagnostic server implementation, this is achieved by setting the corresponding flags in the communication data structures it passes to the protocol layer. If the protocol layer is a D-PDU API, bit 6 in byte 15 of the PDU header structure needs to be set (see ISO 22900-2 for more information).

The diagnostic server internally applies the bit-mask given in the ODX data to the referenced request parameter (see corresponding ODX structure) before the PDU which describes the service is passed to the D-PDU API for sending.

The solution proposed above has the major benefit that it is independent of the concrete protocol. As a result, the major goal of designing a protocol independent diagnostic server has been met. Furthermore, the solution is also capable of handling future versions of the suppress positive response feature where the bit mask manipulates several non-contiguous bits in possibly multiple contiguous bytes. Finally, this solution does not require to inspect all parameters of a MCDService to find out whether there is a parameter for switching on and off the suppress positive response feature at a service. Instead, this can be directly obtained from the MCDService-object itself.

The suppress positive response feature is not used on level MCDHexServices and MCDDynIdxxxComPrimitives. With MCDHexServices being plain services not controlled by the diagnostic server but by the application, the suppress positive response feature can be used independent of the server. However, as with all other MCDHexServices the application needs to have all the knowledge and all the control structures to react to every possible answer to a MCDHexService by the server, e.g. response message, exception, and timeout. Removing the suppress positive response support from the API for MCDHexServices just prevents inconsistencies between the server internal status and the data contained in a MCDHexService's plain PDU. Furthermore, it prevents the server from interpreting MCDHexService-PDUs to overcome this deficit. Please note that interpreting PDUs of a MCDHexService would impose protocol dependencies into the server, which is out of scope by design rule.

8.5   eEND_OF_PDU as RequestParameter

8.5.1   Database side

If a MCDDbRequestParameter within the Collection of MCDDbRequestParameters has the data type eEND_OF_PDU, the count can be polled using the method `getmaxNumberOfItems()`. This method delivers 1 in case of a normal parameter type. In case of eEND_OF_PDU, a number between 0 and MAX_UINT32 is returned. If the field's maximum length is not defined within ODX, MAX_UINT32 is returned. No exception is raised here.

Figure 59 depicts the RequestParameter eEND_OF_PDU on database side.
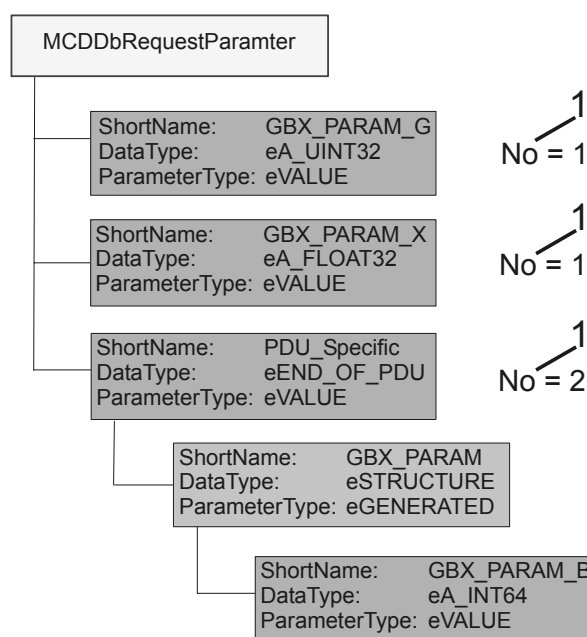
Key

1   result of method getMaxNumberOfItems

Figure 59 — RequestParameter eEND_OF_PDU on database side

8.5.2   Runtime side

By means of the method `MCDRequestParameter:addParameters (A_UINT32 count)` parameters can be inserted. If the type of the `MCDRequestparameter`, where addParameters is called, is not `eEND_OF_PDU`, the exception `ePAR_MCD_NO_DYNAMIC_FIELD` is thrown. The diagnostic server should consider the mininum number of items as defined in ODX for this END-OF-PDU parameter, that is, after the request has been created, the END-OF-PDU parameter already contains <min-number-of-items> structure parameters. If the parameter count plus the number of already existing parameters (`MCDRequestParameter::getRequestParameters()->getCount()`) exceeds the value returned by `getmaxNumberOfItems()`, the exception `ePAR_VALUE_OUT_OF_RANGE` is thrown.

After new sets of parameters (several calls of `addParameters()` can be executed in sequence) have been added, the method `getParameters()` should be used to fetch the whole set of parameters, and deliver a collection that contains an arbitrary number of such elements (zero number of `eEND_OF_PDU` elements is allowed too), for further processing (fill-in of values etc.).

Figure 60 depicts the RequestParameter eEND_OF_PDU on runtime side.
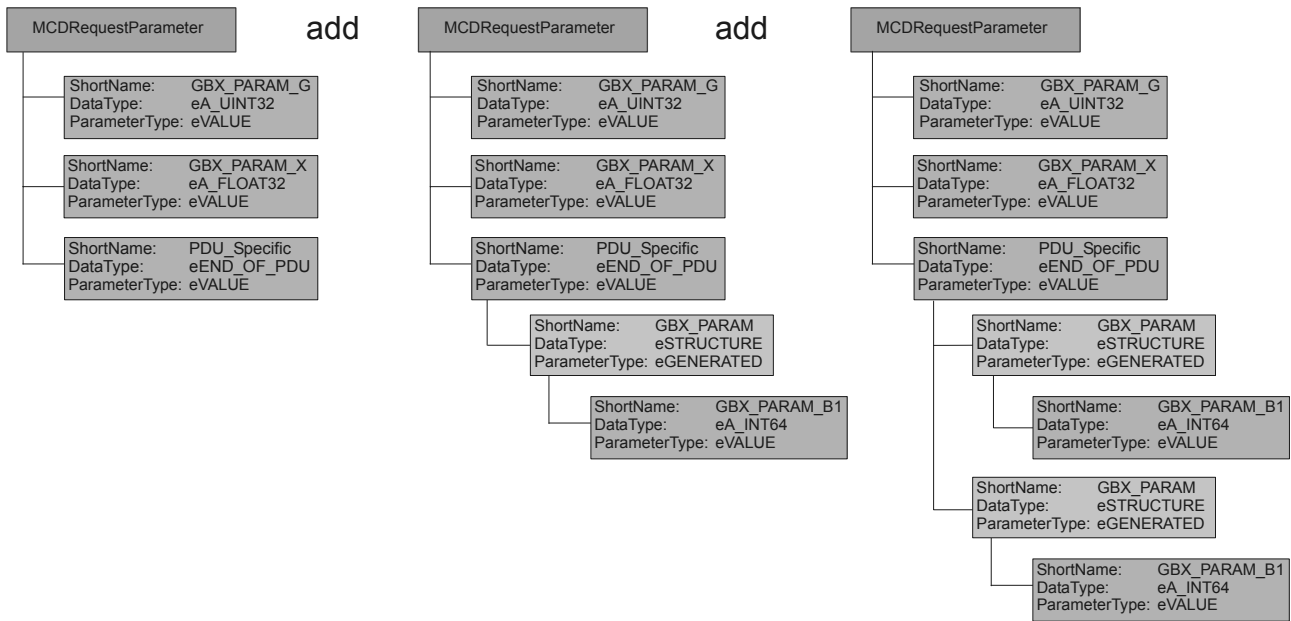
a) repeated add of single parameter
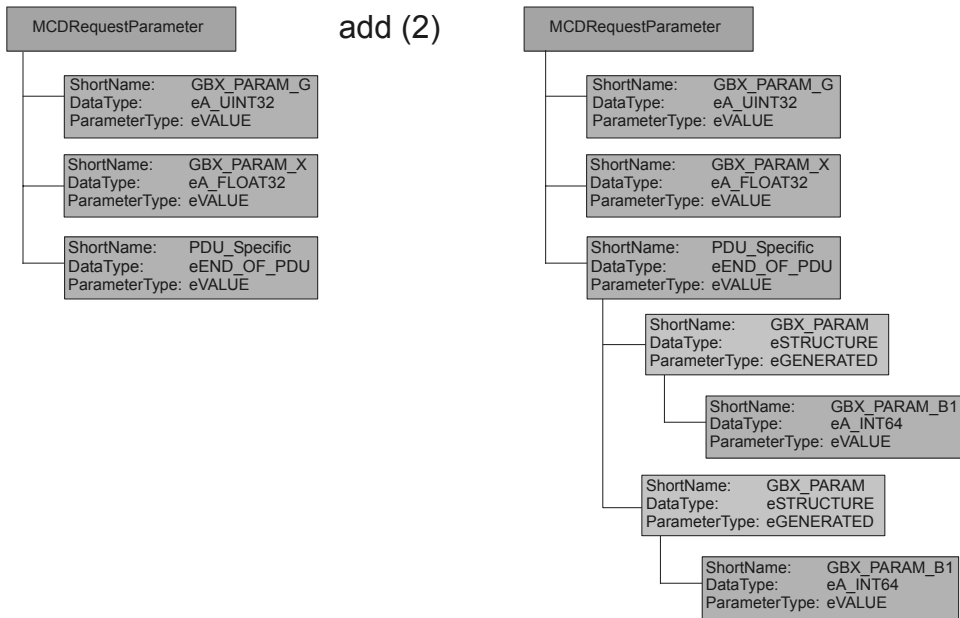
b) add of multiple parameters

Figure 60 — RequestParameter eEND_OF_PDU on runtime side – repeated add of single parameter and add of multiple parameters

8.6 Variable length parameters

Some diagnostic protocols (e.g. UDS) provide services that have parameters of variable length. The size of such a parameter is defined by another parameter in the same message, the so-called length key parameter. In ODX, length key parameters and variable length parameters have a one-to-one relationship, that is, for each variable length parameter, there is one length key parameter. For this length key parameter, the simple `MCDParameterType eLENGTH_KEY` is used in diagnostic server. As defined in ODX, parameters of type `eLENGTH_KEY` code the length of a variable length parameter in bits. However, if a diagnostic protocol requires the length to be given in a different format in the PDU, there needs to be a corresponding conversion between the coded (protocol dependent) and physical value (protocol independent, size in bits) in the ODX data. A diagnostic server shall only consider the physical value of a length key parameter for determining the length of the referenced variable length parameter. At diagnostic server side the data type of a length key parameter is mapped to `eA_UINT32`.

In case an `MCDRequest` or `MCDResponse` object contains a parameter that is of variable length, the corresponding `MCDRequestParameter` or `MCDResponseParameter` object of the request's or response's parameter list shall return true when queried with the `MCDRequestParameter.isVariableLength()` or `MCDResponseParameter.isVariableLength()` method, respectively. If that method returns true, the method `MCDRequestParameter.getLengthKey()` or `MCDResponseParameter.getLengthKey()` can be used to retrieve the parameter with parameter type `eLENGTH_KEY` that contains the associated parameter length. Both methods isVariableLength() and getLengthKey() are also available at the corresponding database objects `MCDDbRequestParameter` and `MCDDbResponseParameter`.

The client is responsible for setting the value of the corresponding length key parameter prior to setting the value of a variable length parameter. Setting the corresponding length key parameter value of a variable length parameter shall result in the following diagnostic server internal actions:

— The diagnostic server checks the new length key value and, if valid, sets the server internal length key value to its new value. In case an invalid length key value is set at an `MCDRequestParameter` or `MCDResponseParameter`, an exception is thrown when calling the method setValue() (MCDParameterizationException, `ePAR_INVALID_VALUE`) and the diagnostic server internal length key value shall not be set to the invalid value.

— The state of the diagnostic server internal MCDValue object of the variable length parameter is set to "uninitialized".

— The D-PDU API position of all subsequent parameters is recalculated as far as possible.

Setting a new value at a variable length parameter that does not match the size defined by the value of its corresponding length key parameter shall result in an exception being thrown by the diagnostic server (MCDParameterizationException, `ePAR_INVALID_VALUE`). In case the value of the corresponding length key parameter is not yet initialized, a client is not allowed to set the value of the variable length parameter. Doing so shall result in an exception being thrown by the diagnostic server (MCDProgramViolationException, `eRT_WRONG_SEQUENCE`).

If a length key parameter is constant, the client is not allowed to change the size of the corresponding variable length parameter. In that case, the size of the variable length parameter is determined by the predefined default value of its corresponding length key parameter.

The example in Figure 61 shows the D-PDU API of a request message with two variable length parameters. Each of these two variable length parameters has a corresponding length key parameter, which defines the size of its variable length parameter in bits. The bit size of both length key parameters LengthOfMemoryAddress and LengthOfMemorySize is 4 bits. Together they occupy byte #1 of the D-PDU API. The physical value n of parameter `LengthOfMemoryAddress` defines the length of parameter MemoryAddress in bits. The physical value m of parameter `LengthOfMemorySize` defines the length of

parameter MemorySize in bits. In order for parameters MemoryAddress and MemorySize to occupy only complete bytes the physical value of both n and m shall be devisable by 8 without remainder. It is obvious that in case parameter LengthOfMemoryAddress has no default value assigned to it, the parameter position of parameter MemorySize may only be calculated at runtime. In order to fulfill the above for parameters `LengthOfMemoryAddress` and `LengthOfMemorySize` a DOP with a Linear conversion by 8 is used.

Figure 61 depicts an example with variable length parameters.
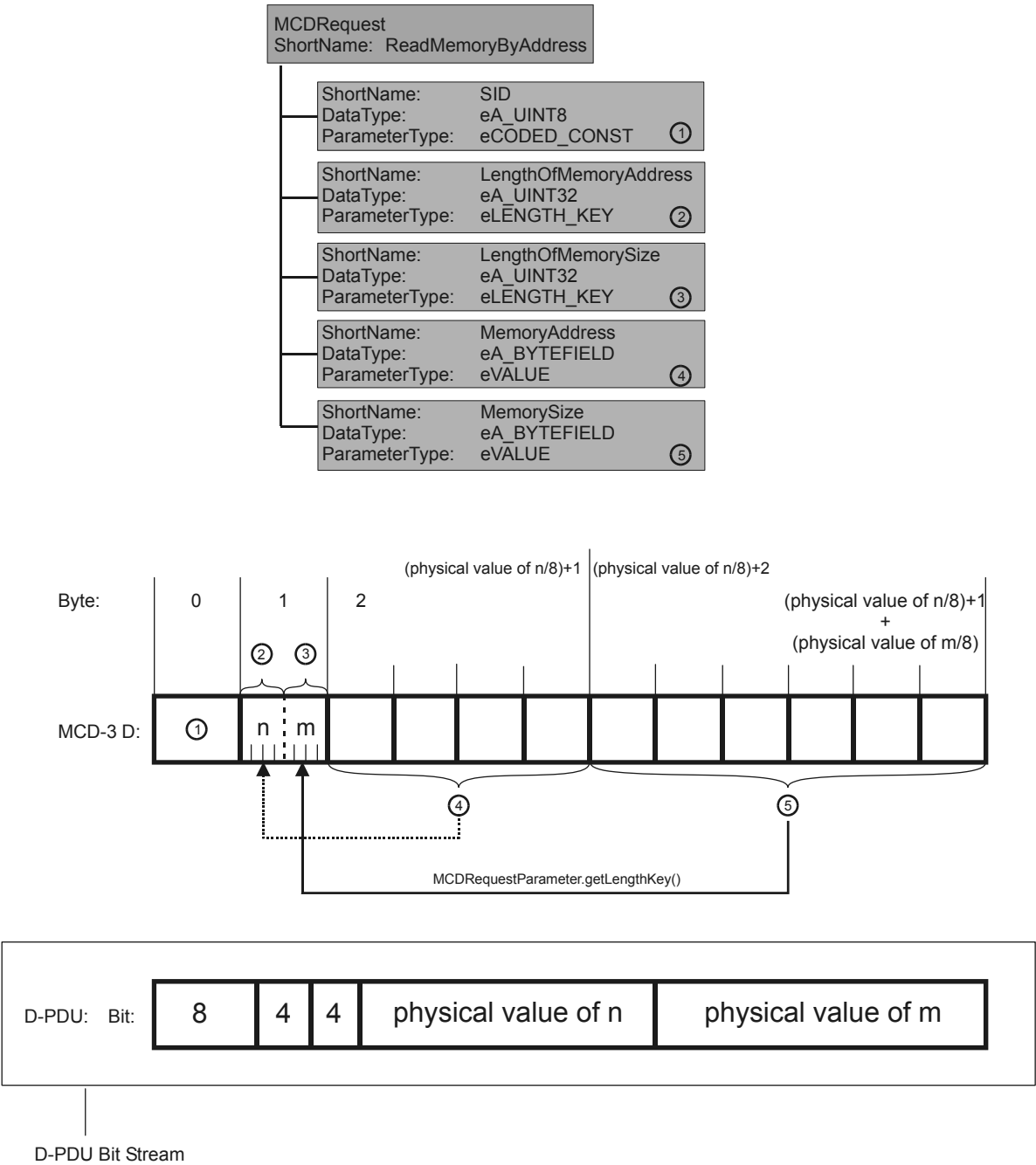
Figure 61 — Example variable length parameters

8.7 Variant identification

8.7.1 Interpretation algorithm

Per ECU, for each Physical Vehicle Link only one RunTime Logical Link for Base Variants and Variants is permitted. That means that either only one Base Variant or one Variant may be instanced. The instancing is independent from the variant correspondence. A multiple instancing may be done by the application, whereas references are assigned by the instanced RunTime Logical Link. The RunTime Logical Link may only be modified by the VariantIdentificationAndSelection. For this, it is of no importance, at which level the VariantIdentificationAndSelection has been carried out (Functional Group, Base Variant, Variant). Functional Groups and BaseVariants may be instanced at any time, independently from already accompanying instanced Variants of the Members. After a Variant selection, in spite of an instanced Base Variant, which has been converted to a Variant by means of VariantIdentificationAndSelection, also the corresponding Variant may be instanced.

In contrast to VariantIdentificationAndSelection, by means of Variant Identification the corresponding Variant is only detected and reported. An automatic switching of the database does not take place, that means the behaviour before the Variant identification will be maintained after its execution. The Variant Identification is allowed at each level (Functional Group, Base Variant und Variant). Each ECU of a functional group will have its own response, so there can be no duplicate names.

All References of a RunTime Logical Link will be informed by an event, if a variant is identified or selected. This is also valid for all members of a Functional Group. If one member of a Functional Group is identified or selected, the Functional Group will also be informed by an event.

If no variant was identified, `MCDLogicalLink::getIdentifiedVariantAccessKeys` delivers an empty collection.

`MCDLogicalLink::getSelectedVariantAccessKeys` delivers the selected / instantiated Variant or the BaseVariant.

The delivered collection depends on the location type:

— BaseVariant or ECUVariant: the collection has one element. This is the selected location, independent if VIS was called or not.

— Protocol: delivers always an empty collection

— Functional Group: the collection returns only Accesskeys to MCDEcuVariant Locations that belong to this Functional Group. If no variant is selected or instantiated the collection is empty.

The VariantIdentification and VariantIdentificationAndSelection are executed like diagnostic services and are permitted at all locations.

The database settings at the levels BaseVariant, Variant, Protocol and MultipleECUJob are always valid for both the transmitting and receiving data transfer direction together. At the Functional Group level, the interpretation on the transmitter side always takes place at this level, whereas on the receiver side the level detected or set for the respective ECU is used. This has been illustrated within the following figures by means of a data pot containing an arrow in direction of the MVCI diagnostic server. An interpretation algorithm to be used is given in the following.

Independent from the interpretation mechanism, according to the actual data the mechanisms of overwriting, eliminating and inheritance will be used.

8.7.2 Identification algorithm

In order to perform Variant Identification or Variant Identification and Selection, a runtime instance of the corresponding control primitive – `MCDVariantIdentification` or

`MCDVariantIdentificationAndSelection` – needs to be created at the Logical Link the VI or VIS is to be executed on. Then, this control primitive is executed synchronously by means of `MCDDiagComPrimitive::executeSync()`. Now, the diagnostic server internally executes the DiagComPrimitives referenced from the matching parameters in the matching patterns that have been used to generate the variant identification (and selection) control primitive in a certain order. More details on this order and the algorithm can be found below.

One important requirement for the execution of a VI(S) is that it needs to be quick, as the concept of an automated variant identification feature only retains its usefulness when the execution time is kept as low as possible – after all, deliberately complex and elaborate usage scenarios can be handled, e.g., by a Java job and should not be part of a general standard. For this reason, the following rules apply for the execution of a variant identification (and selection) control primitive by a diagnostic server.

VI(S) pattern matching has to be done in a specific order which is defined by the matching patterns and matching parameters in the ODX data. In principle, this matching is a three-step process:

— All ECU Variants inheriting from the ECU Base Variant which is to be resolved to a variant have to be considered.

— For each ECU Variant, the associated Variant Patterns (matching patterns) defined in ODX have to be checked.

— For each Variant Pattern, all Matching Parameters defined in ODX have to be checked.

ECU Variants have to be tested during VI or VIS in ascending alphabetical order of their short names. For each ECU Variant, the diagnostic server then considers the `ECU-VARIANT-PATTERN`s of the `ECU-VARIANT` in the order they are defined in the processed ODX data. The first matching `ECU-VARIANT-PATTERN` determines that this ECU Variant is present in the vehicle. In this case, the ECU Variant this matching pattern belongs to is considered identified. After a match, the other `ECU-VARIANT-PATTERN`s of an ECU Variant will not to be tested any more (short-cut resolution). As an ECU Variant has been identified successfully, no further ECU Variants' matching patterns will be tested. In case of VIS, the identified ECU Variant is automatically selected. That is, the `MCDDbLogicalLink` used for VIS is switched to the `MCDDbLocation` of the identified `MCDDbEcuVariant`.

The `MATCHING-PARAMETER`s defined in an `ECU-VARIANT-PATTERN` need to be tested in the order they are defined in ODX. Furthermore, all parameters referenced from a set of `MATCHING-PARAMETER`s shall be checked, short-cut resolution is forbidden here. All `MATCHING-PARAMETER`s of an `ECU-VARIANT-PATTERN` need to match in order to consider the pattern as matching the current ECU Variant.

For the execution of diagnostic services for performing a VI(S) according to the ordering rules described above, the diagnostic server has to use the following algorithm:

Check if a variant pattern is available. In case no variant pattern is available, the variant identification or selection fails (The execution state will be `eALL_FAILED`)

— For each ECU Variant (in ascending alphabetical order of their short names), iterate through the ECU-VARIANT-PATTERNs of the ECU Variant valid for the current Logical Link.

— For each ECU-VARIANT-PATTERN, iterate the MATCHING-PARAMETERs in the order they are defined in ODX.

— Execute the DIAG-COMM referenced from the current MATCHING-PARAMETER if it has not been executed in the context of a previous ECU-VARIANT-PATTERN or MATCHING-PARAMETER of the same execution cycle of VI(S).

— Match the value of the response parameter referenced from this MATCHING-PARAMETER with its expected value. Either use a temporarily stored response of a previous execution of the

respective DIAG-COMM in of the same execution cycle of VI(S) or use the current response. Store the result of the match until the current ECU-VARIANT-PATTERN has been processed and evaluated completely.

— Store the result of the execution of the DIAG-COMM (including its responses) for the rest of the current execution cycle of VI(S).

— Continue with the next MATCHING-PARAMETER in the order defined in the ODX data.

— Check whether all MATCHING-PARAMETERs of the current ECU-VARIANT-PATTERN have matched their expected values.

— If the current ECU-VARIANT-PATTERN has matched completely, mark this corresponding ECU Variant as identified and exit the variant identification algorithm completely. In case of VIS, switch the Logical Link to the identified ECU Variant.

— If the current ECU-VARIANT-PATTERN has not matched completely, continue with the next ECU-VARIANT-PATTERN in the order defined in the ODX data.

— Repeat pattern matching as described above (mind the MATCHING-PARAMETER ordering as defined in ODX) with the next ECU Variant in the ascending order of short names if no ECU Variant has been identified yet. As soon as an ECU-VARIANT-PATTERN matches completely, the ECU Variant is considered as successfully identified, and the VI(S) can be stopped.

— If VI(S) terminates successfully (an ECU Variant has been identified), a positive response is to be generated by the diagnostic server. This positive response is then returned within a corresponding result to the client application.

— If VI(S) terminates without having successfully identified an ECU Variant, a negative response is to be generated by the diagnostic server. This negative response is then returned within a corresponding result to the client application.

For engineering use cases, it needs to be possible for the client application to identify which variant pattern has matched during the ECU variant identification process. If the variant identification fails, it can be important to know for an engineering tester which variant-patterns are available in ODX. To this end, the MVCI diagnostic server API provides means to access the variant patterns available for an ECU Variant through the method `MCDDbLocation::getDbVariantPatterns()`.

A short digression concerning the demand that diagnostic services have to be executed in a defined order depending on individual matching parameter order for each ECU variant pattern and each ECU Variant as found in the ODX data: From the diagnostic server's point of view, this approach is not feasible for several reasons and would cripple the VI/VIS feature for the large majority of use cases. First, there is the fact that the diagnostic server has to build request and result templates for VI/VIS services (if enabled by the corresponding system property). As it is now, the diagnostic server will have to include, e.g., five request structures in the result object, one for each of the five executed DiagComs as defined by an exemplary ODX data set according to the rules described above and in 8.7.3. As the execution order of these services is not defined, they will be executed only once on ECU Base Variant level for data gathering while VI/VIS is executed, and the request and response object structures will remain manageable and execution time will be short. In contrast to that, the demand that the execution order of VI/VIS DiagComs has to conform to the order of the matching parameters in each of the ECU variant patterns of all possible ECU Variants for a Base Variant link would lead to the following scenario: First of all, in the worst case (no matching ECU Variant is built into the vehicle), the five DiagComs from the example above would have to be executed anew for each possible permutation of ECU Variants, variant patterns, and matching parameter sets. For example, for a Base Variant with 50 ECU Variants, with 10 variant patterns each and five matching parameters per variant pattern, that would mean that 50*10*5=2500 services would have to be executed for a single VI/VIS. Secondly, when assuming that any of the services that is used for variant identification could cause an ECU to change its state/behaviour, it would be necessary to perform the appropriate reset/state changing action after (unsuccessful) testing of each matching parameter set such that such an ECU would be in the correct state for the next test iteration. Hence, an ECU reset or similar service would have to be part of each VI/VIS pattern.

Imagining the time that would be necessary to perform a VI/VIS that executes 2500 services, where each fifth service causes an ECU reset, is left as an exercise to the reader. On top of these obvious problems, there is a more subtle one: as the diagnostic server has to provide request and result structures for VI/VIS, it would have to provide different request/result structures for each of the possible permutations of service order as described above. When imagining an ECU with 50 variants, 10 variant patterns, 5 matching parameter sets/associated DiagComs, with an average of 3 parameters each, that would lead to a request template containing 50*10*5*3=7500 parameters! When doing the same calculation for the result structure, there are not only possible positive responses (and response parameters) to take into account but also negative and global negative response structures. Again, the exercise of imagining the resulting response templates is left to the reader. The general conclusion is, however, that defining an execution order for VI/VIS DiagComs on the matching parameter level is not a desirable solution for a diagnostic server and will therefore not be part of a standard solution. For the few cases where this kind of behaviour is needed for variant identification, the user can always write an appropriate Java job or application logic to handle these specific use cases.

Figure 62 depicts the interpretation algorithm for variant identification.
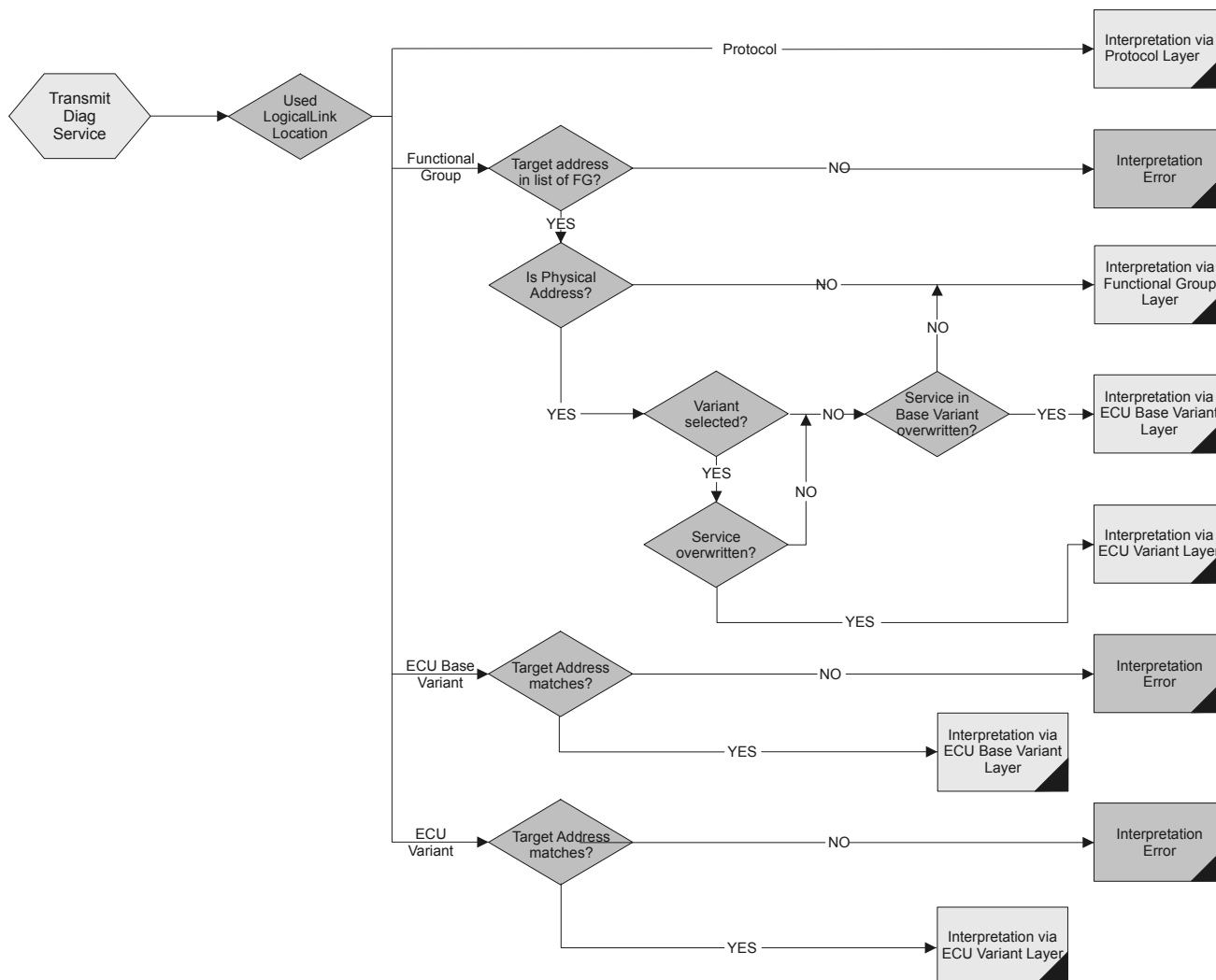
Figure 62 — Interpretation algorithm for variant identification

Figure 63 depicts the example about location hierarchy.
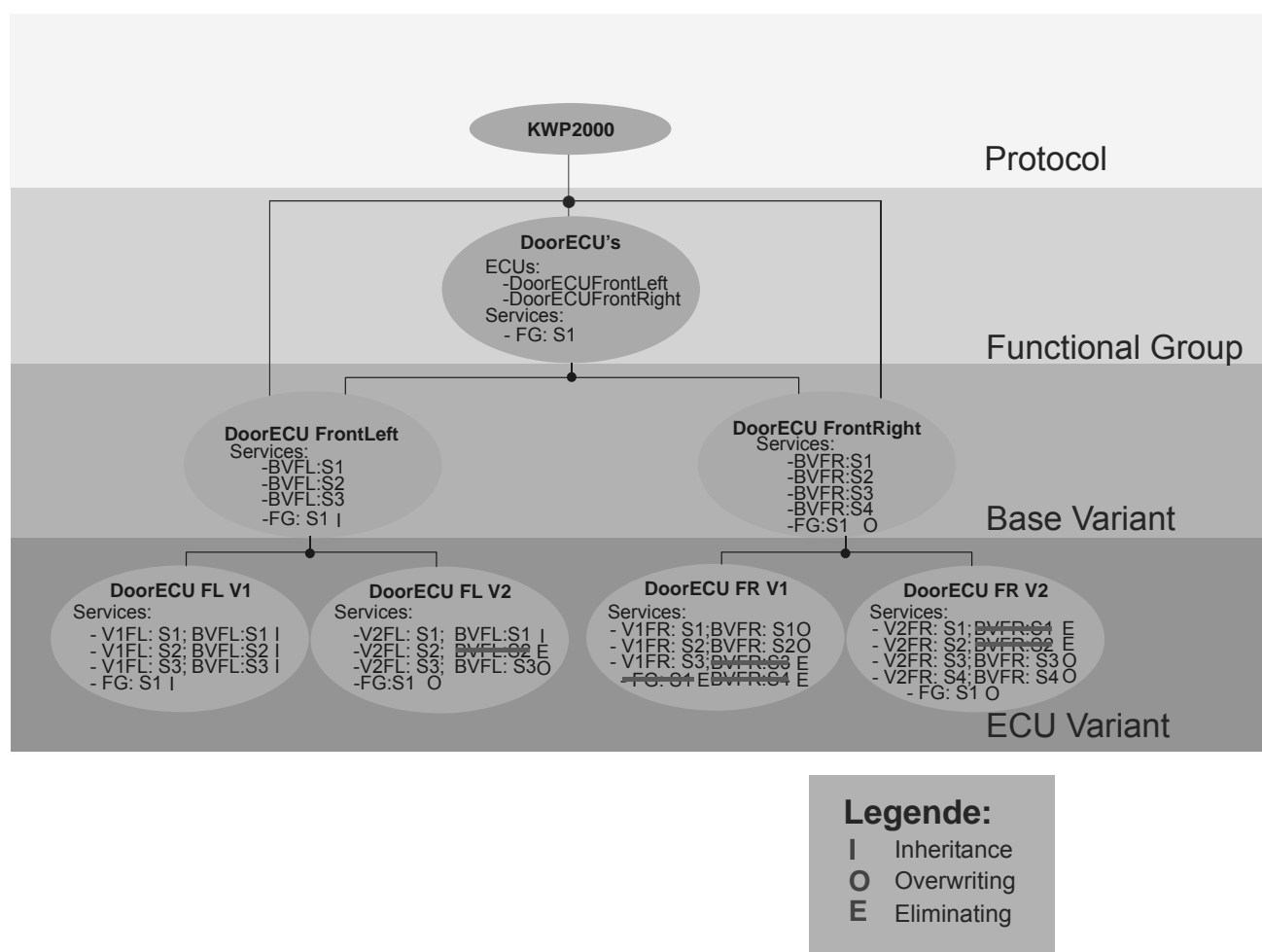
Figure 63 — Example location hierarchy

8.7.3   Request and ResponseParameter of VI and VIS

8.7.3.1   General

The communication primitives `MCDVariantIdentification` and
`MCDVariantIdentificationAndSelection` typically consist of a number of diagnostic services referenced by the MATCHING-PARAMETERs of the relevant ECU VARIANT-PATTERN definitions. Because VI and VIS on the API are represented as single communication primitives they can have only one request parameter structure and only one response structure. That's why there is a need to combine the needed request parameter of the used services to a new request structure and to combine the responses of the used services to a new response structure. The result of the whole communication primitive is independent of the execution order of its diagnostic services, because in the ODX database there is no order defined for the execution.

8.7.3.2 RequestParameter structure

The request parameter collection of a communication primitive (also for VI and VIS) should contain all needed values to execute the communication primitive. In case of VI and VIS it means that all request parameters needed for the used services should be merged in a collection. Since the ODX database provides the possibility of structured request parameters it is easy to do.

Remark:

The short names of objects within one collection must be unique. The short names of objects in different collections need not to be unique, even if they belong to the same service.

Building Rule:

a) For each used diagservice of the variant identification (and selection) which has got own request parameters a `MCDDbRequestParameter` of data type `eSTRUCTURE` and parameter type `eGENERATED` will be added to the collection of request parameters of the variant identification (and selection) primitive.The short name of this request parameter have to be the short name of the *MCDDbService* object it belongs to.

b) The collection of request parameters of this structure must be the same as the collection of the diagservice.

c) The variant identification (and selection) primitive has got a collection of request parameters containing these structures in alphabetic order of their short names.

Figure 64 depicts the RequestParameter of VI / VIS.
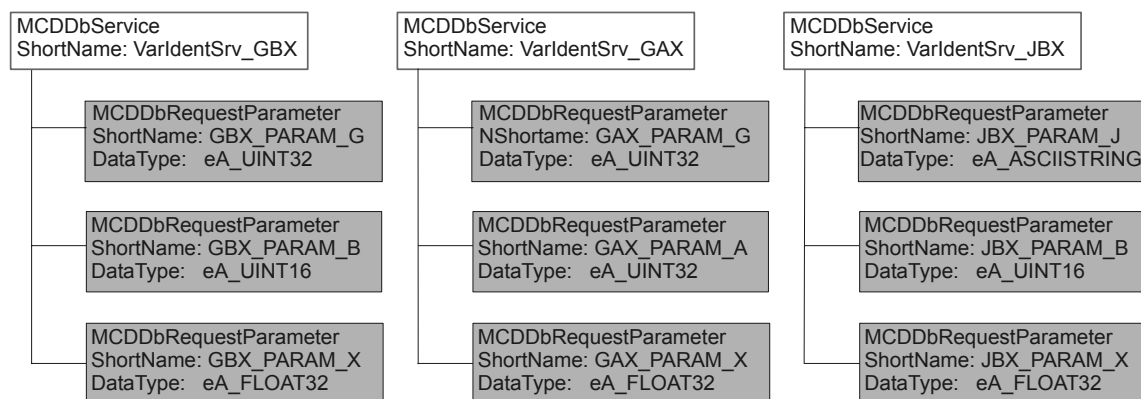
Figure 64 — RequestParameter of VI / VIS – RequestParameters of services

Figure 65 depicts the RequestParameter of VI / VIS – Separation of RequestParameters.
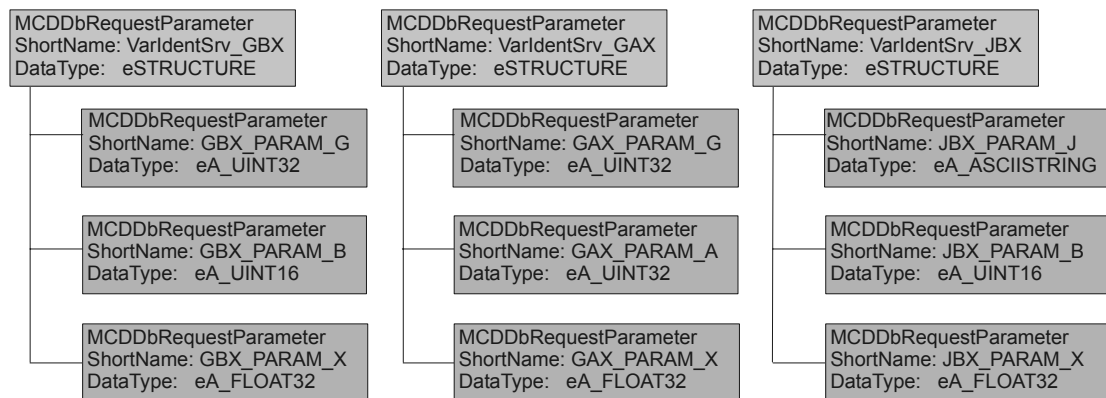
Figure 65 — RequestParameter of VI / VIS – Separation of RequestParameters

Figure 66 depicts the RequestParameter of VI / VIS – Combination of the result of VariantIdentification.
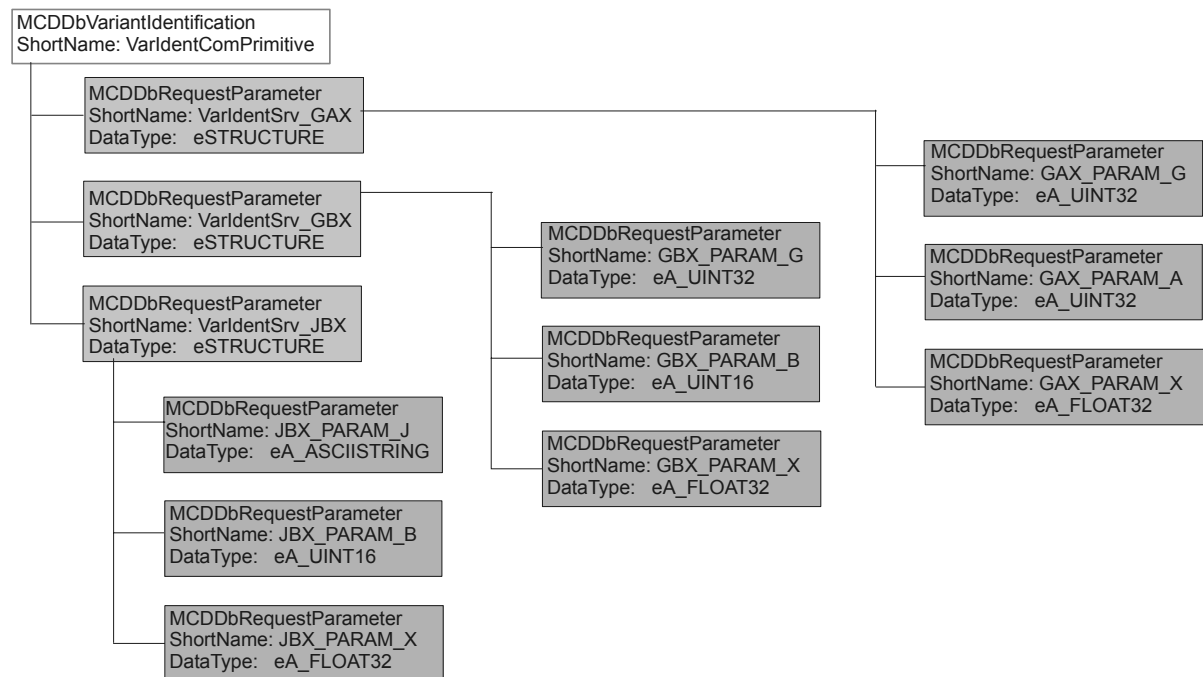
Figure 66 — RequestParameter of VI / VIS – Combination of the result of VariantIdentification

8.7.3.3   ResponseParameter structure

A service that is used for variant identification shall contain at least one single positive response. If the db-template contains more than one response, only the first, according to the ODX order, response is considered.

Negative responses are not considered at all.

The result of a communication primitive (also for VI and VIS) has got one response for each concerned ECU. In case of VI and VIS this ECU can be used by more than one of the used diagnostic services. Each runtime system using the same ODX database must built the database response for VI and VIS in the same way to guarantee interchange ability.

Building Rule

a) The `MCDDbResponse` object of each diagnostic service of the variant identification have to be replaced by a `MCDDbResponseParameter` object of data type `eSTRUCTURE` and parameter type `eGENERATED`, where the short name of this response parameter have to be the short name of the `MCDDbService` object which has been replaced.

b) The collection of response parameters of this structure must be the same as the collection of the former response.

c) The response of the variant identification (and selection) has got a collection of response parameters containing these structures in alphabetic order of their short names.

Figure 67 depicts the Result of VI / VIS – Results of services.
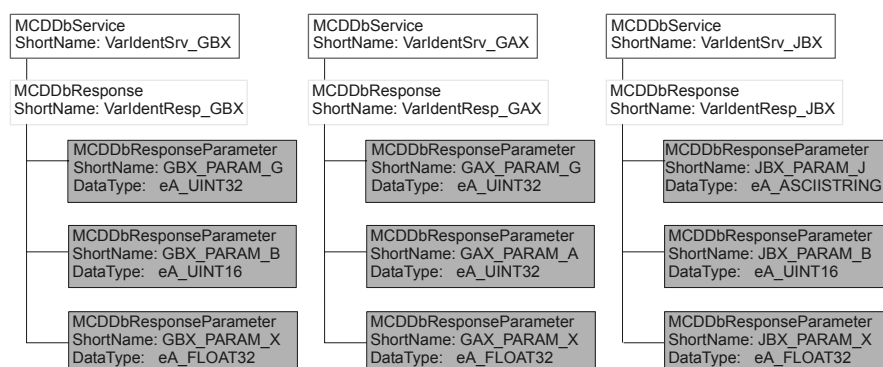
Figure 67 — Result of VI / VIS – Results of services

Figure 68 depicts the Result of VI / VIS – Separation of responses.
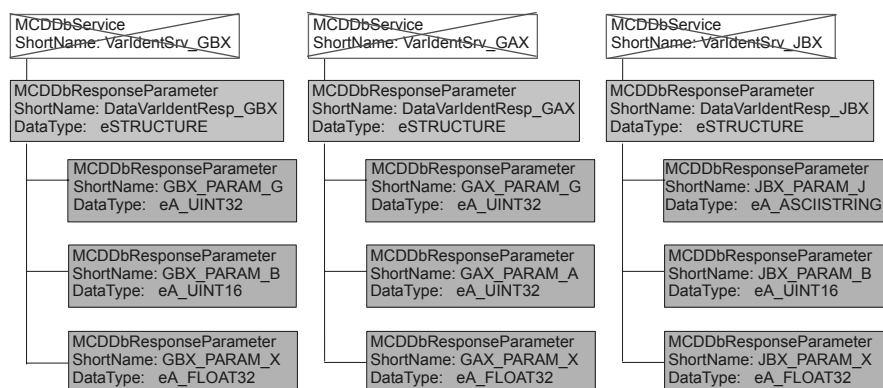
Figure 68 — Result of VI / VIS – Separation of responses

Figure 69 depicts the Result of VI / VIS – Combination of the result of VariantIdentification.
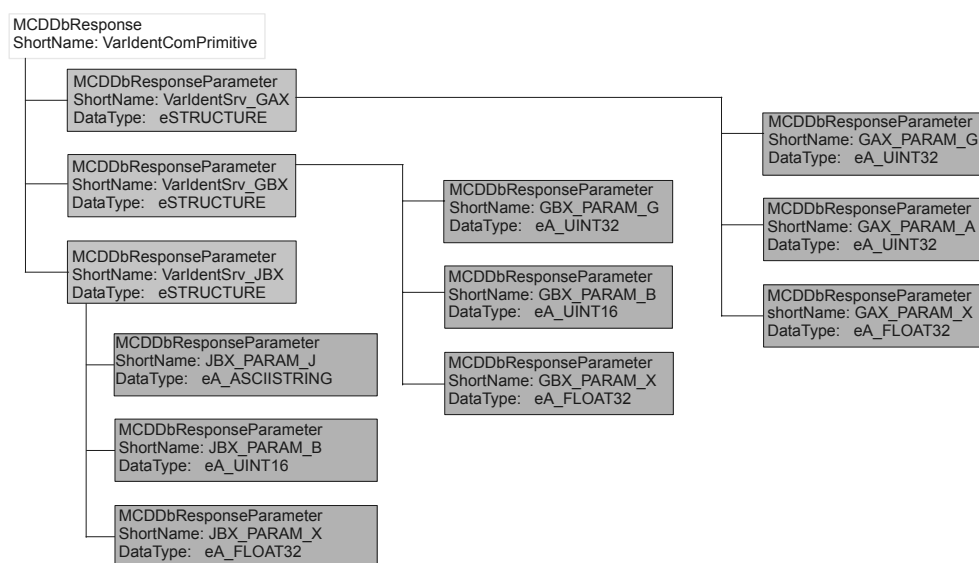
Figure 69 — Result of VI / VIS – Combination of the result of VariantIdentification

8.7.4 Service handling in case of different locations

The addressing mode PHYSICAL means explicit communication with a selected ECU.

The addressing mode FUNCTIONAL is broadcast communication via a Functional Group which can contain several ECUs. Each ECU of a functional group will have it's own response, so there can be no duplicate names.

Figure 70 depicts the address mode in case of physical or functional communication.
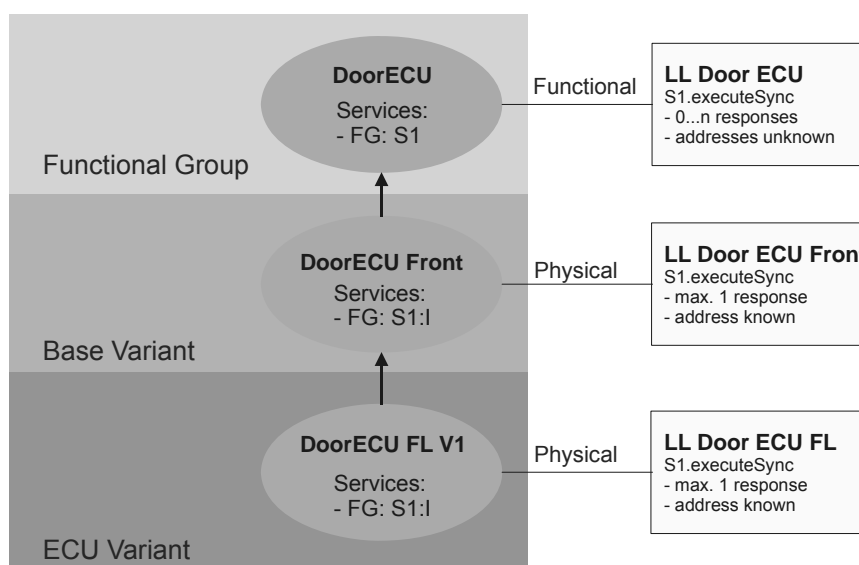
Figure 70 — Address mode in case of physical or functional communication

Internally, the MVCI diagnostic server chooses the correct communication mode (`PHYSICAL` or `FUNCTIONAL`) according to the resources it has allocated.

The MVCI diagnostic server is able to decide from the data in ODX and the current Location which addressing mode of a service has to be used if this service is defined to "`FUNCTIONAL-OR-PHYSICAL`". If the Location is a Functional Group, the service can only be executed functionally. If the Location is a Base Variant or a Variant, then the service can only be executed physically.

Services executed functionally are automatically sent to all ECUs of a functional group. Hence, functional addressing can always result in multiple responses.

As functional addressing is often used for, e.g. Testerpresent or Sleep commands, it is necessary to open a logical link to a specific ECU and a second logical link to a functional group this ECU is contained in. Then, the logical link to the functional group is used for functional addressing and the logical link to the ECU is used for physical addressing.

8.7.5   Variant Patterns and Matching Parameters

Technically, variant identification is performed by the diagnostic server by executing a set of DiagComPrimitives, and then matching the ECU responses to a set of patterns that are defined in the ODX data set. Only the first positive response can be used for matching parameters. Negative response parameters are not supported inside matching patterns and are ignored by the MVCI diagnostic server. Which services have to be executed and which parameters are required to evaluate to which values is defined in so-called matching patterns (ODX: ECU-VARIANT-PATTERN). In this specification, a matching pattern is represented by an object of type MCDDbMatchingPattern (see Figure 71). An ordered collection of such matching patterns can be obtained from each MCDDbLocation of MCDLocationType eECU_VARIANT. Every matching pattern references an ordered collection of matching parameters (ODX: MATCHING-PARAMETER). Every matching parameter – represented by an object of type MCDDbMatchingPattern in the diagnostic server – refers to a DiagComPrimitive, a Response Parameter of this DiagComPrimitive, and an expected value for this Response Parameter.
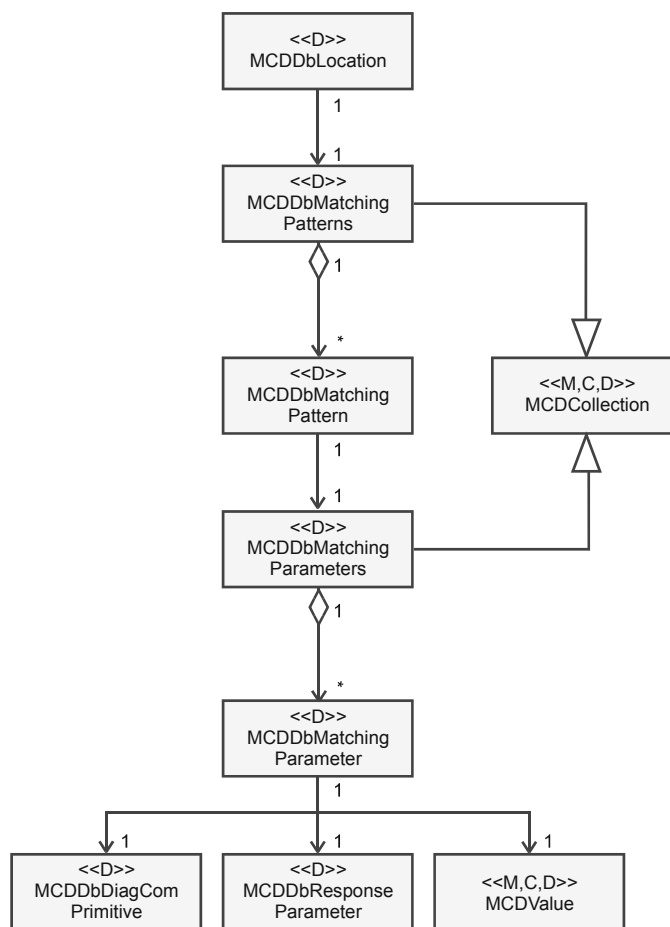
Figure 71 — Matching patterns for VI and VIS

8.8   Use cases

8.8.1   Create Logical Link and use DiagComPrimitives

Figure 72 depicts the instantiation of Logical Links and DiagComPrimitive (Construction).
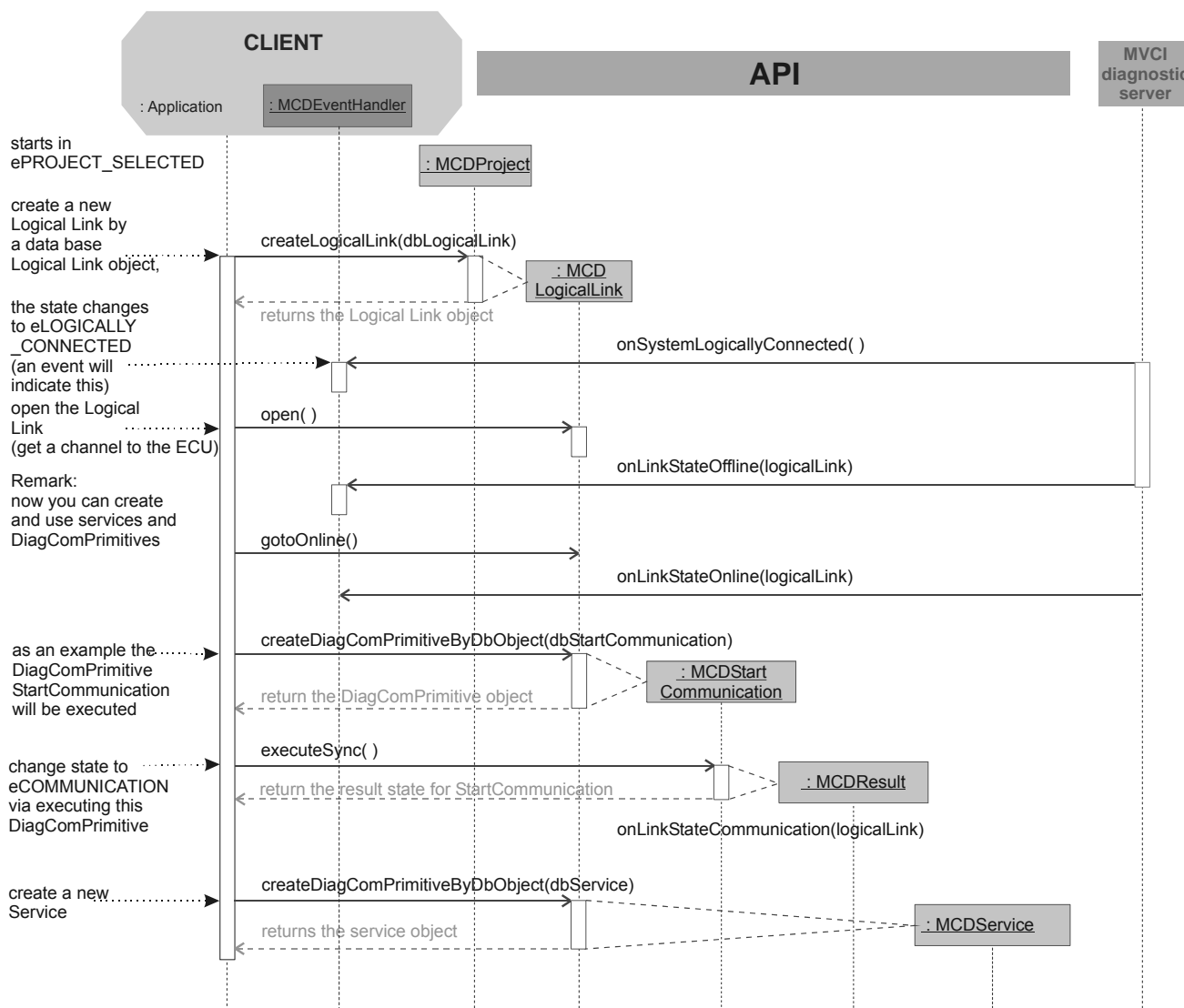
Figure 72 — Instantiation of Logical Links and DiagComPrimitive (Construction)

At first, the Client creates the runtime Logical Links according to the selected database templates and opens the connection of the Logical Link for the communication. The state of the Logical Link after open is `eOFFLINE`. As soon as the first Logical Link has been created, the `MCDSystem` object take the state `eLOGICALLY_CONNECTED`.

The first DiagComPrimitive created and executed in this diagram is `MCDStartCommunication` which performs the state transition to `eCOMMUNICATION` for the Logical Link. Subsequently, the runtime DiagComPrimitives and Services are created for the Logical Links according to their database templates.

Figure 73 depicts the ERD diagram: Possibilities for creation of Logical Links.

VARIOUS CREATIONS OF LOGICAL LINK

(1) createLogicalLinkByName(shortNameLL)
(2) createLogicalLink(databaseObject)
(3) createLogicalLinkByVariant(shortNameLLBaseVariant, shortNameVariant)
(4) createLogicalLinkByAccessKey(accessKeyString, shortNamePhysicalVehicleLink)
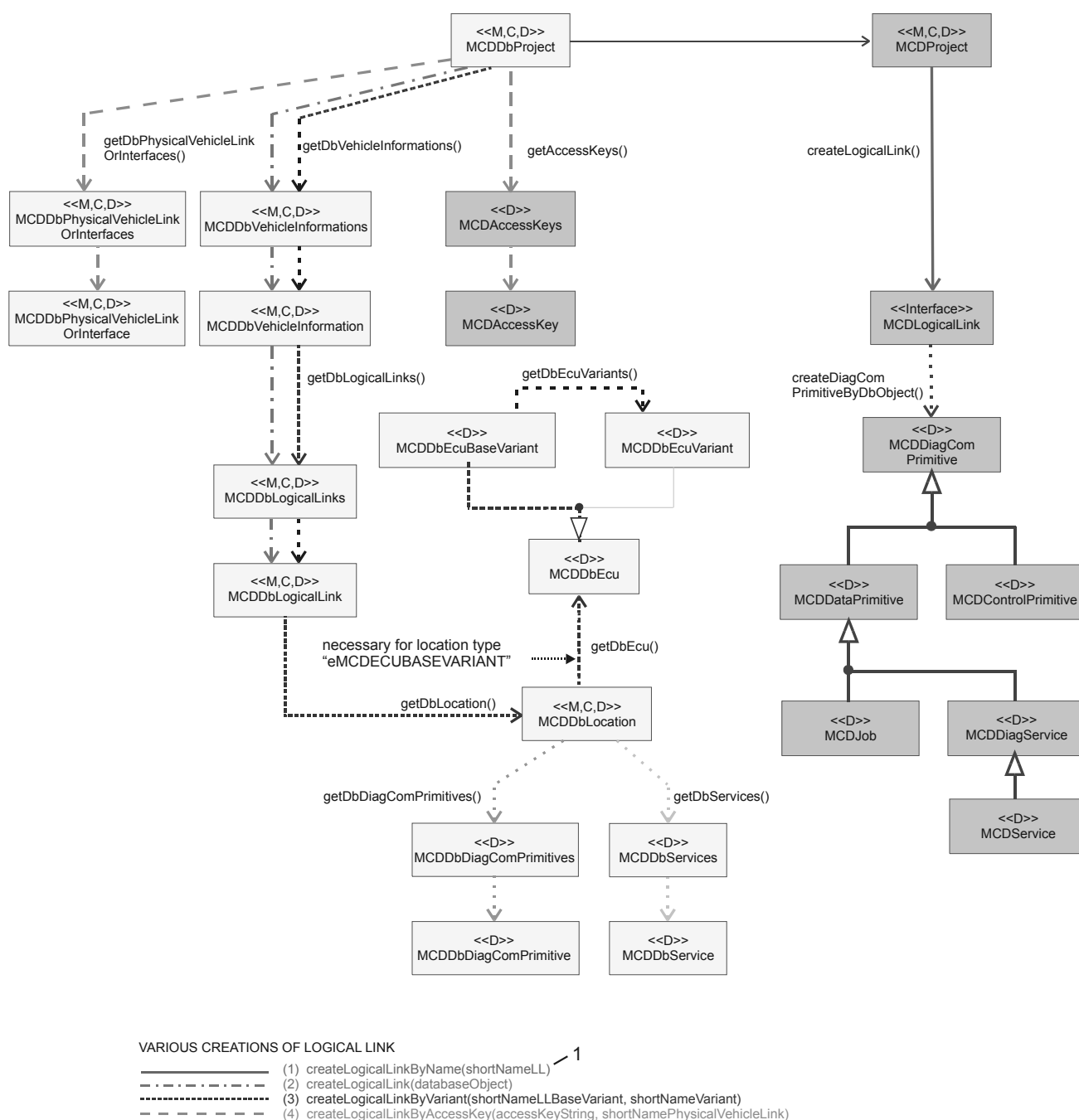
Key

1 - The client has to know the required shortname for this method. No database searching is necessary

Figure 73 — ERD diagram: Possibilities for creation of Logical Links

In this ERD-like diagram only the possibilities for information acquisition from the database for the purpose of Logical Link creation are shown. The four separate possibilities are each marked with different colours.

8.8.2 Removal of communication objects

The deleting of all objects used for the communication takes place in reverse order to the setting up. First to change the state from `eCOMMUNICATION` to `eONLINE` execute the DiagComPrimitive `MCDStopCommunication`. Then remove each of the Services and DiagComPrimitives of this Logical Link with the method `MCDLogicalLink::removeDiagComPrimitive (comPrimitive: MCDDiagComprimitive)`. This action will take place for each Logical Link. Then disconnected the Logical Link on hardware side from the ECU by means of `close()`and get the event `onLinkStateCreated()`for this.

After this, every Logical Link will be removed from the project by means of `MCDProject::removeLogicalLink(link:MCDLogicalLink)`. If all Logical Links have been deleted, the state transition of the `MCDSystem` to `ePROJECT_SELECTED` is announced by the event `onLogicallyDisconnected()`.

Figure 74 depicts the removal of DiagComPrimitives and Logical Links (Destruction).
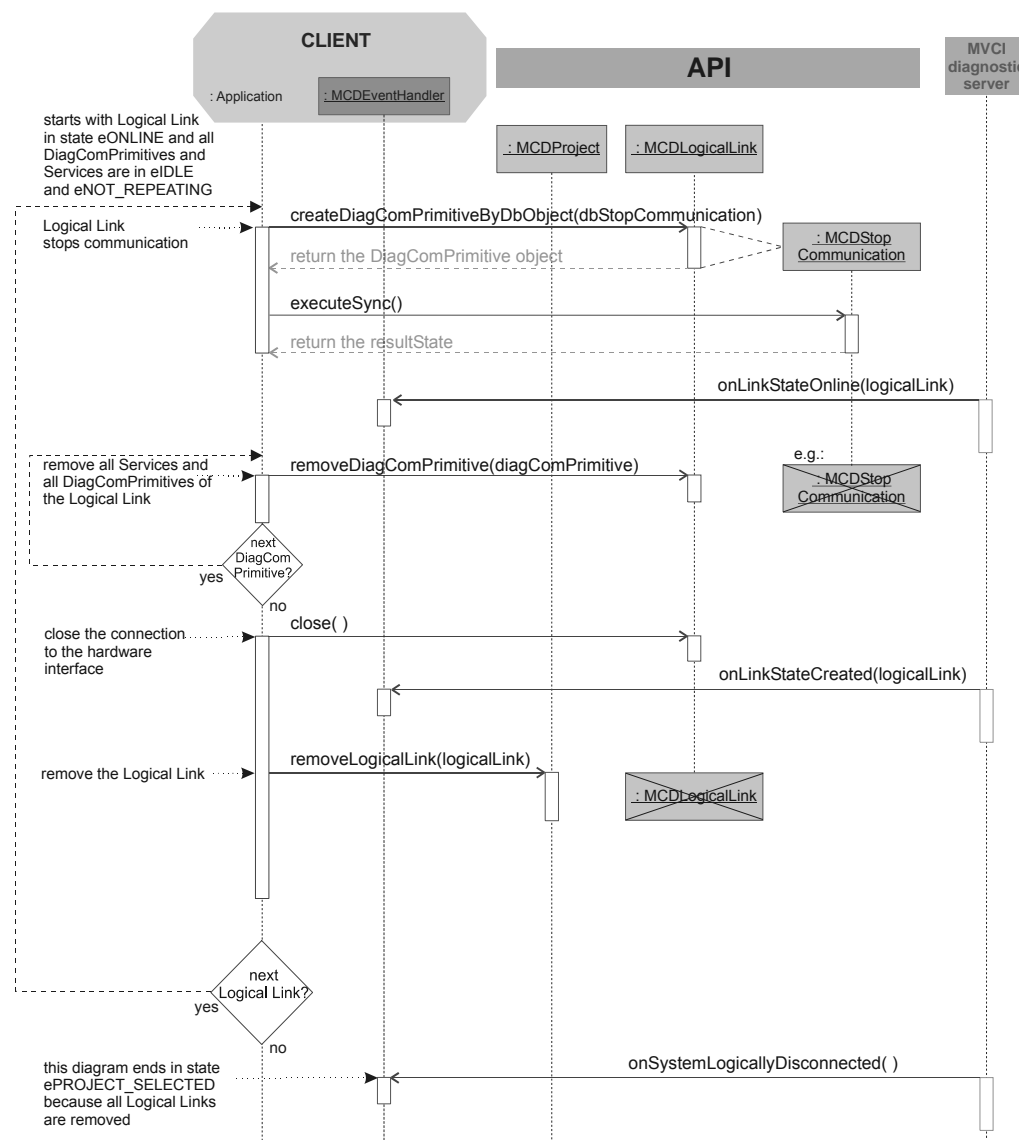
Figure 74 — Removal of DiagComPrimitives and Logical Links (Destruction)

8.8.3 Service Handling

8.8.3.1 Non cyclic diag service execution

This Sequence Diagram shows the usual sequence of a non cyclic diag services in asynchronous execution. Firstly, the Request Parameters of the Service are set, in case the default values preset by the database shall not be used. After this, the Service execution starts with the method `executeAsync()`. Following this, the Service will be put into the execution queue of the Logical Link and normally will be executed within the MVCI diagnostic server within a finite period of time. While executing the Service, the MVCI diagnostic server creates an object for the ResultState and an object for the Result. After finishing the Service, the MVCI diagnostic server sends the event

`onPrimitiveTerminated(primitive:MCDDiagComPrimitive, link:MCDLogicalLink, resultstate:MCDResultState)` to the EventHandler of the Client. The event delivers the created ResultState object. The ResultState Object is asked for the ExecutionState. In case of the correct execution of the Service, the Result-Collection with one Result object is polled by the Service and analysed. Figure 75 depicts the non cyclic diagnostic service execution (asynchronous).
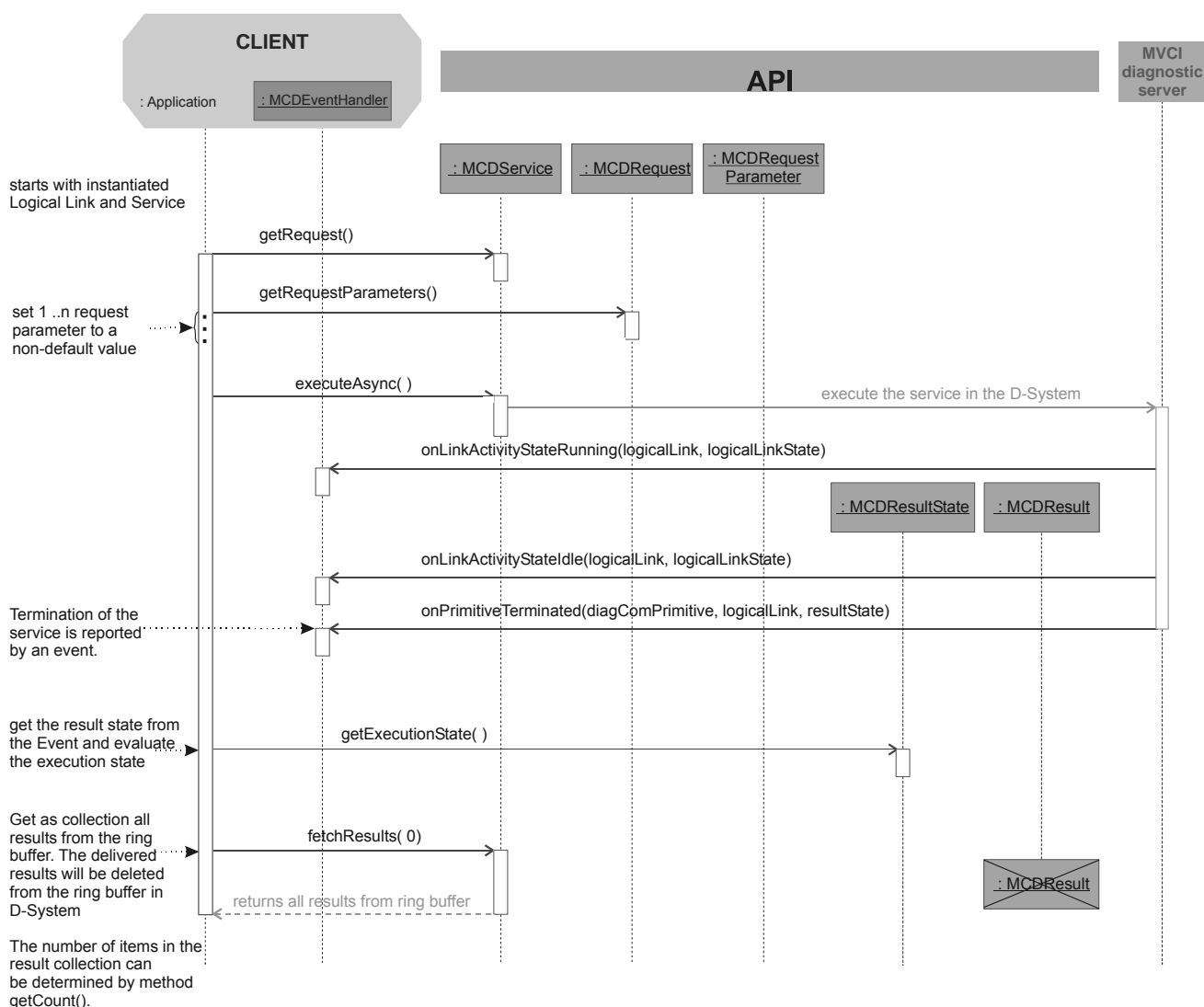
Figure 75 — Non cyclic diagnostic service execution (asynchronous)

In case of synchronous execution of a non cyclic diag service the result state but no event is delivered as return value. Figure 76 depicts the non cyclic diagnostic service execution (synchronous).
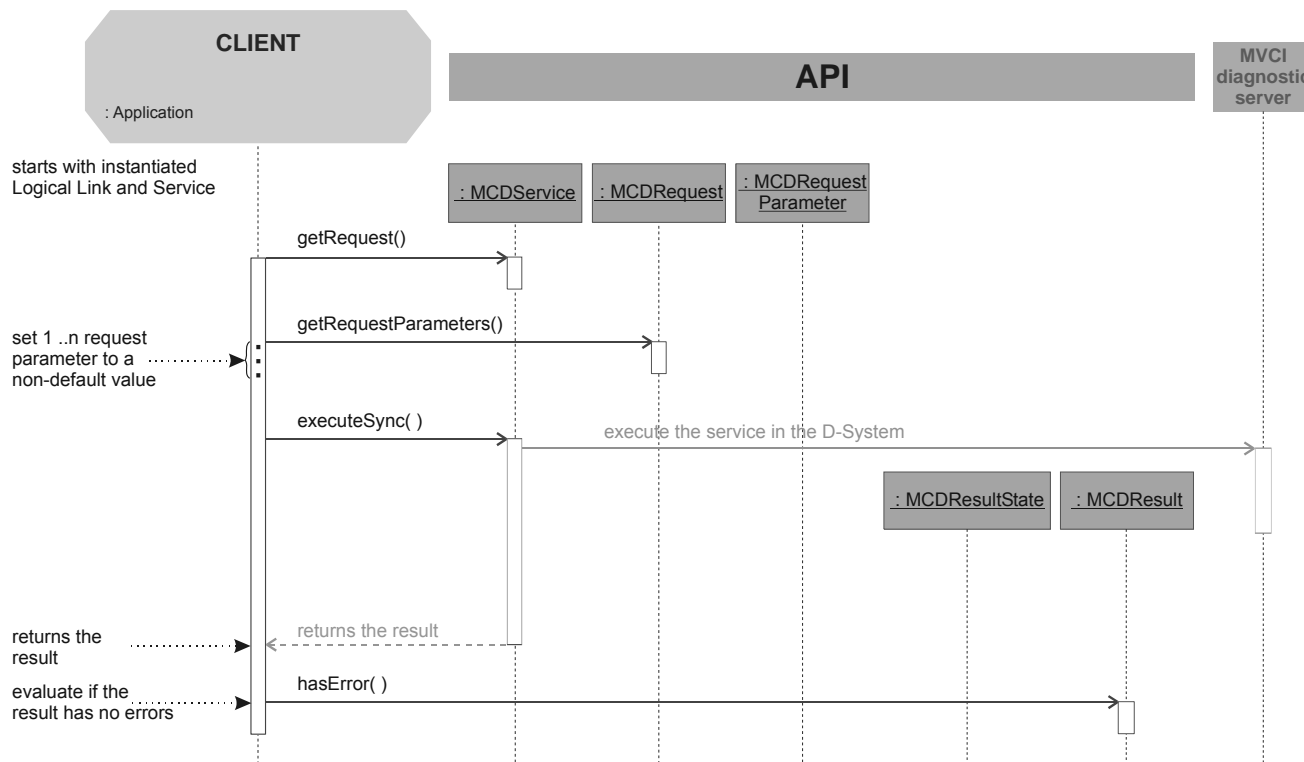
Figure 76 — Non cyclic diagnostic service execution (synchronous)

8.8.3.2   Cyclic diag service execution

This Sequence Diagram shows the usual sequence of a cyclic diag service in asynchronous execution. At first, the Request Parameters of the Service are set with the values necessary for execution, in case the default values preset by the database shall not be used. After this, the service execution is started with the method `executeAsync()`. Following this, the Service will be put into the execution queue of the Logical Link and normally will be executed within the MVCI diagnostic server a infinite period of time. It will normally stopped by method `cancel()`. While executing the Service, the MVCI diagnostic server creates tuples of objects for the ResultState and objects to store the cyclically occurring Results. The generated Results are stored to a ring buffer with defined size. For each Result the MVCI diagnostic server sends the event `onPrimitiveHasResult (primitive:MCDDiagComPrimitive, link:MCDLogicalLink, resultstate:MCDResultState)` with the current `MCDResultState` to the client. Each event is related to a created `MCDResultState` object, and the ResultState Object is asked for the ExecutionState. In case of the correct execution of the Service, the already generated ResultObject(s) is(are) polled by the Service and analysed.

The `MCDResultState` objects reflect the current state of execution of the DiagComPrimitive. The Execution State located within these objects shows how the execution has been running so far. As soon as any error has cropped up at any time the Execution State cannot be `eALL_POSITIVE` anymore. To find out if any error has occurred within the current result, this can directly be polled from the each read in `MCDResult`.

Figure 77 depicts the Cyclic diagnostic service execution.
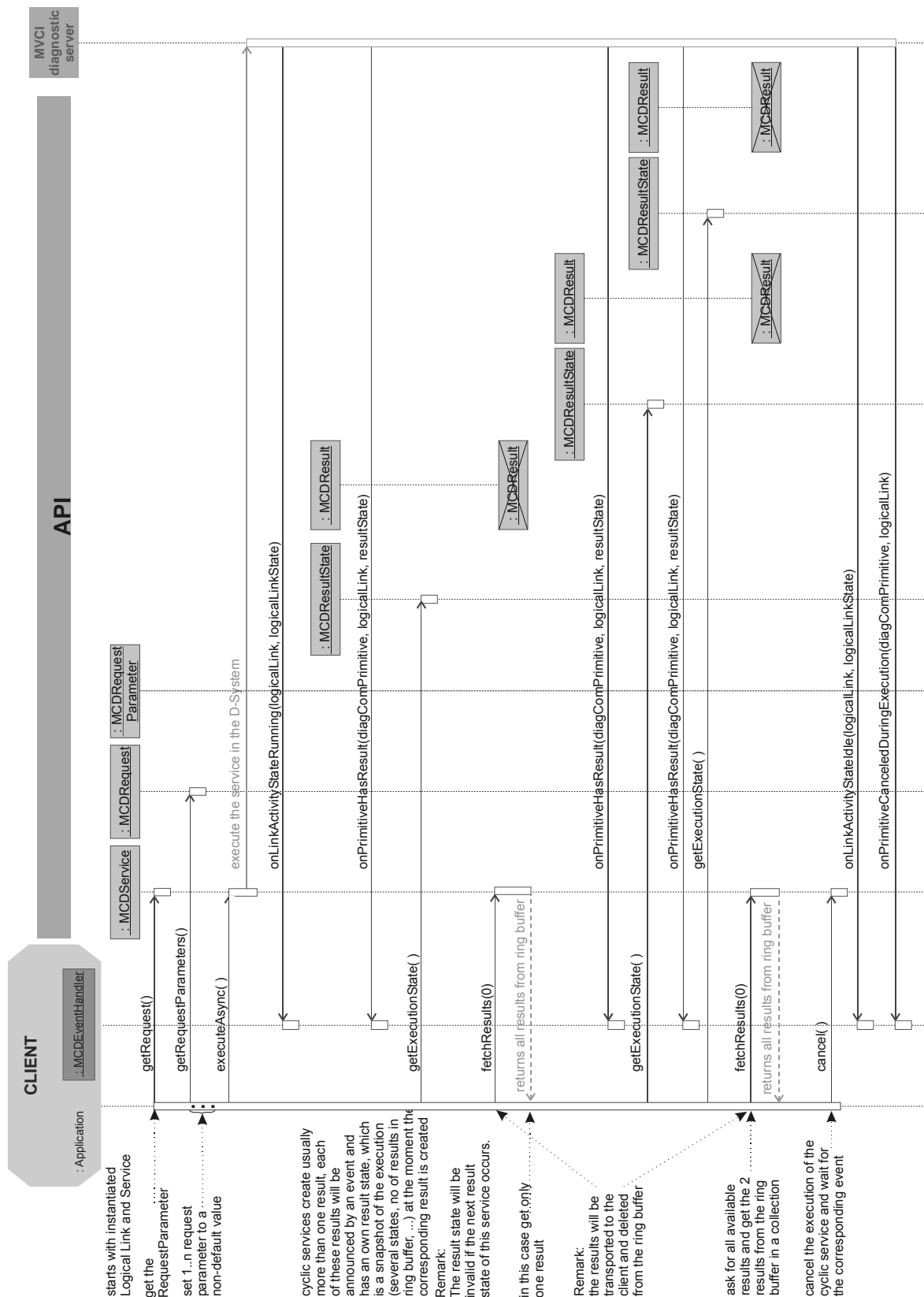
Figure 77 — Cyclic diagnostic service execution

8.8.4   Result access

After each execution of a Service or ComPrimtive the MVCI diagnostic server creates a result. The following Sequence Diagram shows the evaluation of a result of a Service.

To make clear, wherefrom which information on the execution exists at the end of the Service, the asynchronous start of the Service has been chosen as starting point for the diagram.

As soon as the event `onPrimitiveTerminated(primitive:MCDDiagComPrimitive, link:MCDLogicalLink, resultstate:MCDResultState.)` is reported to the Client, the evaluation can be started. The status of the execution is detected by asking the event delivered ResultState Object. If this is `eALL_POSITIVE` as assumed in the sequence diagram, the result collection is polled with method `fetchResults(numberOfRequestedResults)` from the Service. Within the example exactly one result exists.

Now, the result object may be polled for the created results. Using method `getResponses()`, the Client gets a response collection with one response for each ECU response (valid for SPR, for MPR see Figure 46) belonging to the Logical Link. Usually this will be exactly one ECU, in the special case Functional Group several ECU s will answer. The method `MCDResponse::getAccessKeyOfLocation()` can only be realized in the MVCI diagnostic server if the PDU API delivers, e.g. CAN-Ids back to the MVCI diagnostic server. The name of an ECU is coded in the shortname of a Base Variant. Each single response consists of a collection with at least one response parameter, within which the actual structuring of the result according to the tree concept has been set up. The leaves of the tree contain the actual values, the nods of the tree symbolize the structure elements. To get to the single values of the result it has to be iterated through the tree and each element is asked for type and name (`getDataType()` / `getShortName()`). Node elements are polled for sub-elements (`getResponseParameters()`) and leaf elements for the value (`getValue()`). Thus, the Client can analyse result of any structure.

Remark:

If the Service is executed synchronously, the result is immediately delivered to the Client as the return value and not stored in the ring buffer. So that are only results from asynchronous execution will be stored in the ring buffer.

The result structure of a service or job consists of a `MCDResult` object with one `MCDResponse` object for each ECU response. An ECU may answer with a positive or negative response (e.g. a service requires the execution of another service before its execution). In ODX there is a list of positive responses and a list of negative responses at the DIAG-LAYER. The correspondent object of the DIAG-LAYER at MVCI diagnostic server API is the `MCDDbLocation`. The collection of `MCDDbResponse` objects at the class `MCDDbDiagComPrimitive` contains at least two responses per ECU, a positive and at least one negative response.

First, the positive responses are matched against the current response according to their ordering. In case none of the positive response templates matches, the specific negative responses are matched against the current response according to their ordering. If no specific negative response matches, the global negative responses are matched against the current response. Again their ordering is considered.

In case an ECU responds with a PDU which cannot be mapped onto one of the Response Templates defined for this ECU – negative and positive –, the execution state of this service with respect to this ECU is `eINVALID_RESPONSE` (functional addressing) or `eALL_INVALID_RESPONSE` (functional / physical addressing). If a more severe error occurs, the result might have an result state of `eFAILED`. The result contains an empty response object (MCDResponse) with a shortname #RtGen_Response. For this MCDResponse object, the method `getResponseMessage` returns the non-matching PDU (MCDValue of type bytefield). The new error `eRT_INVALID_RESPONSE` is returned by the method `MCDResponse::getError()`.

8.8.5 Error handling in results

Within a Result errors may occur at any place. As not for every error the execution will be aborted and an exception thrown, the errors are delivered not until the Result is returned.

To query all errors together within a Result the method `MCDDiagComPrimitive::getErrors():MCDErrors` can be used. This method returns all error objects of the last Result as a collection. Each of these error objects via the parent functionality has a reference to the object (Result, Response or ResponseParameter) at which the error had occurred. This way, the error unambiguously references the place of error within the structure of the Result.

The MCDResult Object contains the error with the highest severity found at the result, any response or response parameter. Errors in result, any response or response parameters are static. That is, regardless if the result (responses, response parameter) elements are accessed or the method `hasError` is called.

`MCDDiagComPrimitive::getErrors():MCDErrors` always delivers an error code if present, even if no result will be delivered.

Figure 78 depicts the result access (Part 1).
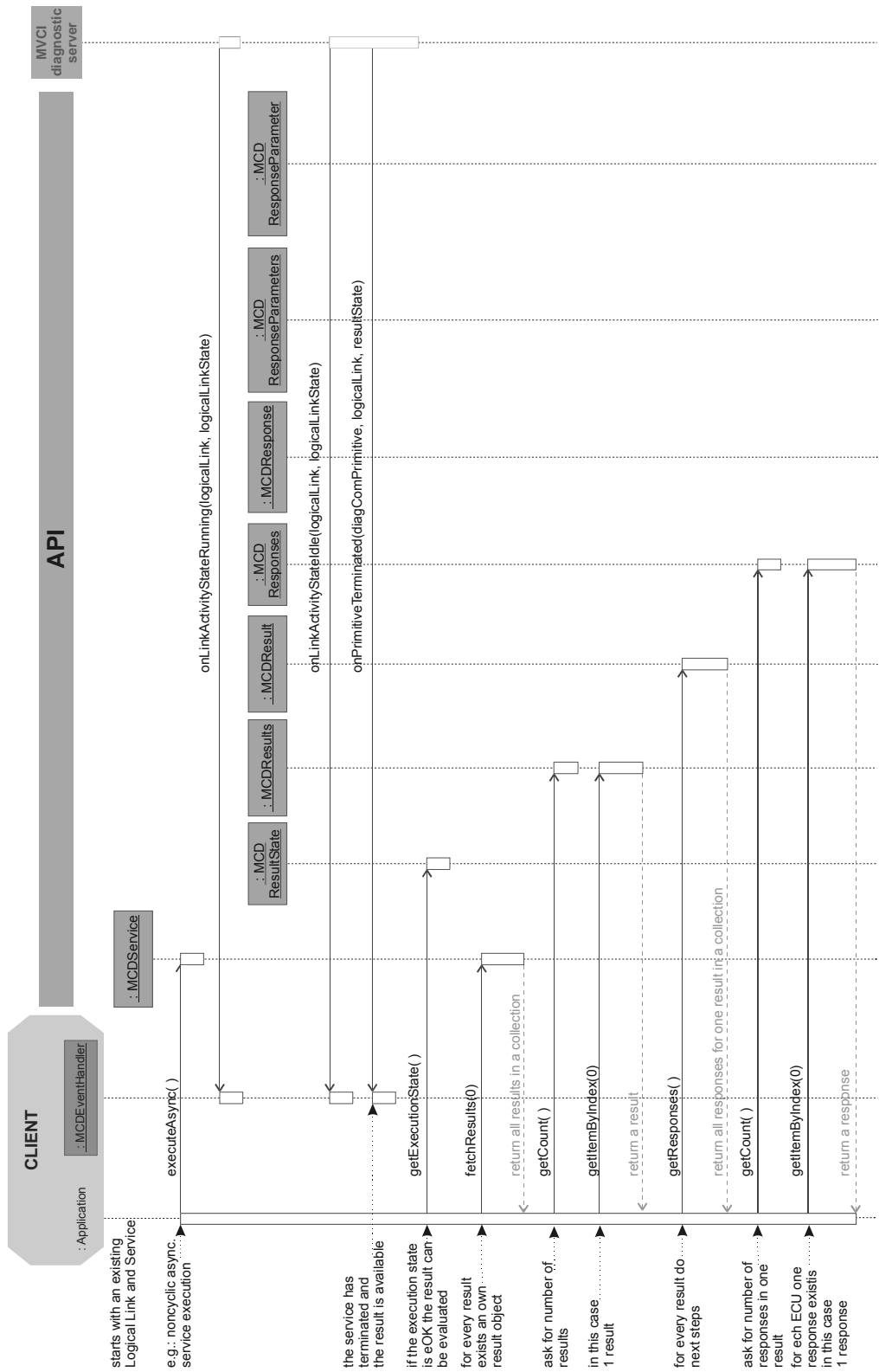
Figure 78 — Result access (Part 1)

Figure 79 depicts the result access (Part 2).
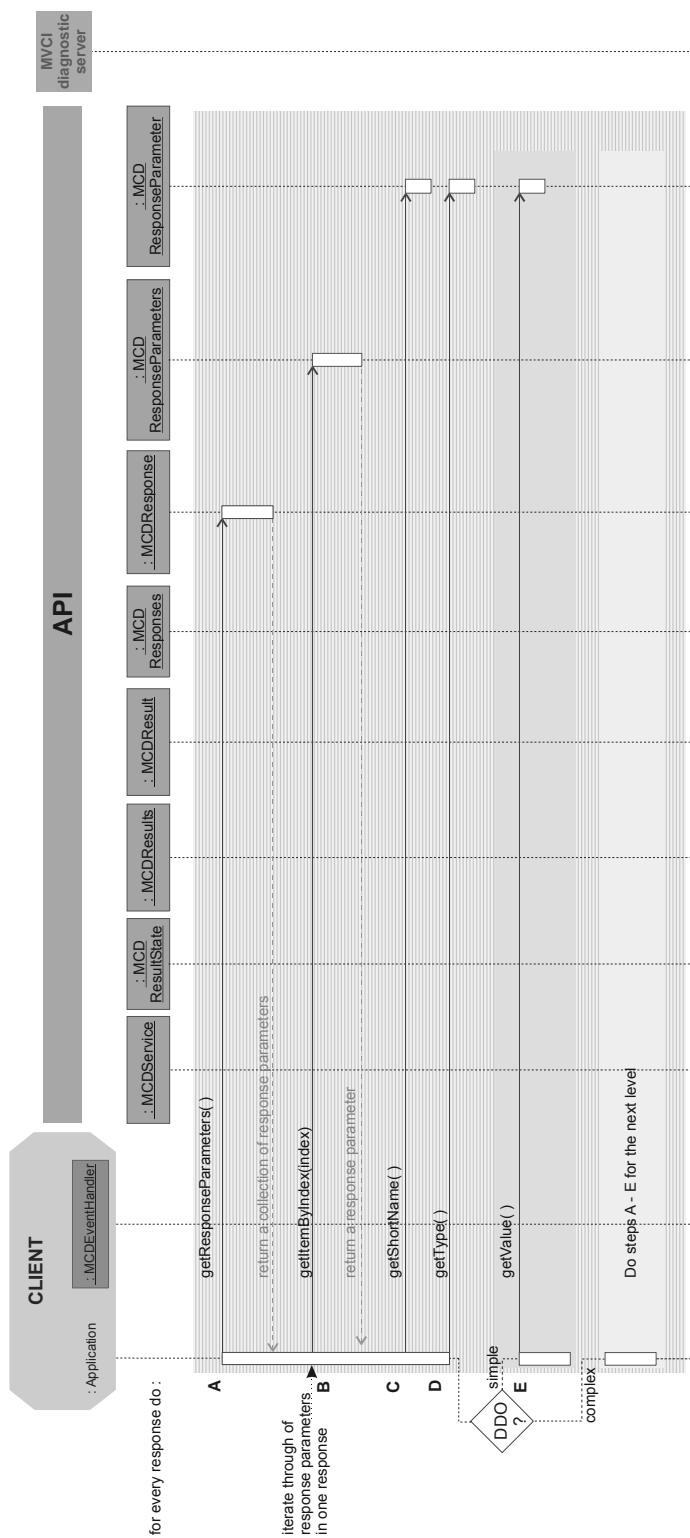
Figure 79 — Result access (Part 2)

Generally is valid:

The availability of results of DiagServices is reported via Events. Exactly one Event is reported per complete result record. The following Events are used:

⸺ OnPrimitiveTerminated (can, but does not have to return a result),

⸺ onPrimitiveHasIntermediateResult (for complete result data records of Services in Jobs),

⸺ onPrimitiveHasResult (for complete result data records of cyclic diagnostic services or for complete result data records of non cyclic diagnostic services in execution mode Repetition),

⸺ onPrimitiveRepetitionStopped.

After one of these events has occurred it is possible to poll the number of complete result data records in hand.

The results of asynchronous execution for each Service/ DiagComPrimitive are stored to a ring buffer with the following features:

RING BUFFER

One default value for the size in the whole MVCI diagnostic server:

⸺ for every Service different values can be set,

⸺ Range: >= 1, maximum 232 – 1,

⸺ read results are deleted from the ring buffer (decrementing number of results).

The result ring buffer is internal to the MVCI diagnostic server and can contain 0 to N result entries. If the ring buffer is full, any further result will overwrite the oldest result.

The ring buffer will not be emptied if a new execution starts and so all results from former executions which had not been fetched by the Client or had not been overwritten (in case of buffer overflow) are in the ring buffer. The results can be fetched at any time the Client wants to fetch.

The method `fetchResult()` supports with parameter `numReq` different possibilities of result access:

Table 14 defines the result access possibilities.

Table 14 — Result access possibilities

Value of numReq	Meaning
- n (n ∈ N, n <> 0)	returns n results. If n > m, where m is the number of available results in the buffer, m results will be delivered. The results in the delivered collection are ordered by their timestamp. The element with the lowest index has got the newest timestamp. All results will be removed from the queue.
0	returns the whole buffer. After this the buffer is empty. The results in the delivered collection are ordered by their timestamp. The element with the lowest index has got the oldest timestamp.
+ n (n ∈ N, n <> 0)	returns n results. If n > m, where m is the number of available results in the buffer, m results will be delivered. The results in the delivered collection are ordered by their timestamp. The element with the lowest index has got the oldest timestamp.

The complete result data records are taken from the `MCDResult` Object. This contains one `MCDResponse` Object for each ECU response participating in the result. Each MCD ResponseObject is recursively composed of `MCDResponseParameter` objects in collections, because of which structured results may be decomposed up until its comprised elementary components.

For structured results the elements `eFIELD` (arrays or sequences), `eSTRUCTURE` and `eMULTIPLEXER` (union) are used.

The relation between the `MCDResponseParameter` objects and ODX are described in 7.7.5 and 8.12.2.

Figure 80 depicts the result structure DTC from example.
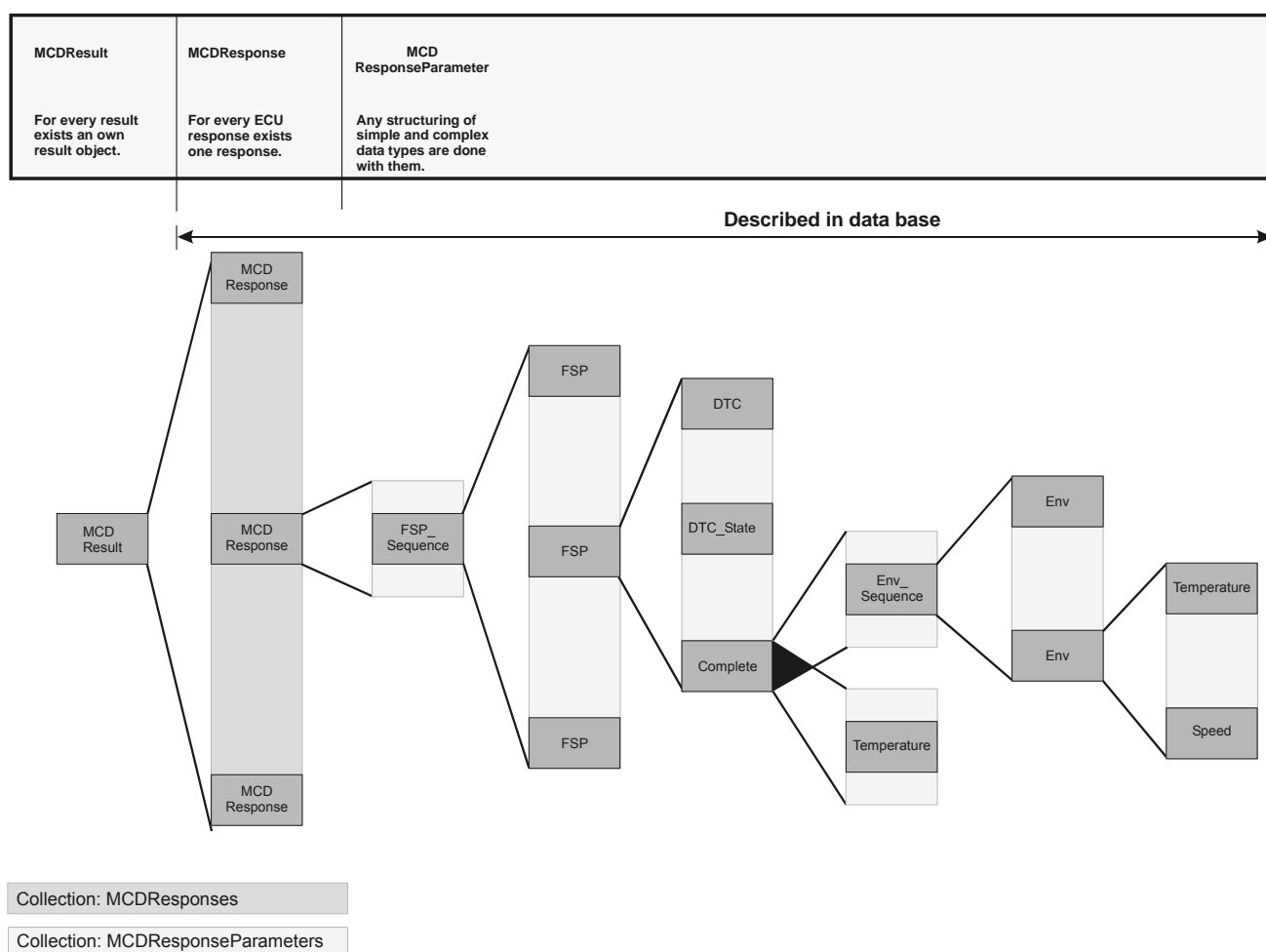
Figure 80 — Result structure DTC from example

Figure 80 depicts the result structure DTC.
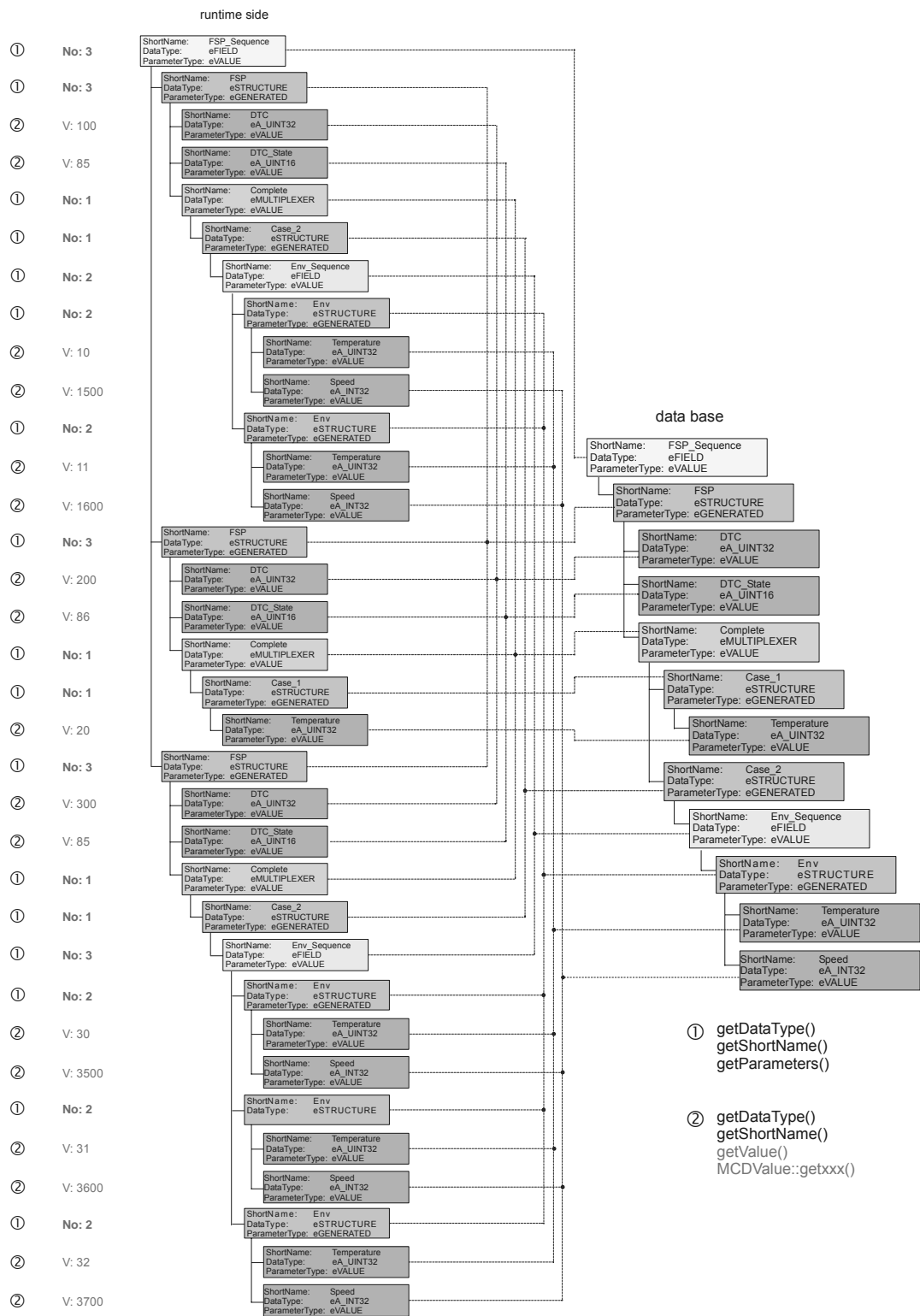
In case of Multiplexer the method getValue() delivers the switch type (index of the branch).

Figure 81 — Result structure DTC

ODX data for database template result structure DTC

```
<PARAM xsi:type="VALUE">
    <SHORT-NAME>FSP_Sequence</SHORT-NAME>
    <DOP-REF ID-REF="EOP_DOP_ID"/>
</PARAM>
<END-OF-PDU-FIELDS>
    <END-OF-PDU-FIELD IS-VISIBLE="true" ID="EOP_DOP_ID">
        <SHORT-NAME>EOP_DOP</SHORT-NAME>
        <BASIC-STRUCTURE-REF ID-REF="FSP_DOP_ID"/>
    </END-OF-PDU-FIELD>
</END-OF-PDU-FIELDS>
<STRUCTURE ID="FSP_DOP_ID">
    <SHORT-NAME>FSP</SHORT-NAME>
    <PARAMS>
        <PARAM xsi:type="VALUE">
            <SHORT-NAME>DTC</SHORT-NAME>
        </PARAM>
        <PARAM xsi:type="VALUE">
            <SHORT-NAME>DTC_State</SHORT-NAME>
        </PARAM>
        <PARAM xsi:type="VALUE">
            <SHORT-NAME>Complete</SHORT-NAME>
            <DOP-REF ID-REF="MUX_DOP_ID"/>
        </PARAM>
    </PARAMS>
</STRUCTURE>

<MUX ID="MUX_DOP_ID">
    <SHORT-NAME>Mux_DOP</SHORT-NAME>
    <SWITCH-KEY>
        <DATA-OBJECT-PROP-REF ID-REF="SwitchKey_DOP_ID"/>
    </SWITCH-KEY>
    <CASES>
        <CASE>
            <SHORT-NAME>Case_1</SHORT-NAME>
            <STRUCTURE-REF ID-REF="Case1_DOP_ID"/>
        </CASE>
        <CASE>
            <SHORT-NAME>Case_2</SHORT-NAME>
            <STRUCTURE-REF ID-REF="Case2_DOP_ID"/>
        </CASE>
    </CASES>
</MUX>

<STRUCTURE ID="Case1_DOP_ID">
    <SHORT-NAME>StructCase_1</SHORT-NAME>
    <PARAMS>
        <PARAM xsi:type="VALUE">
            <SHORT-NAME>Temperature</SHORT-NAME>
        </PARAM>
    </PARAMS>
</STRUCTURE>

<STRUCTURE ID="Case2_DOP_ID">
```

```
    <SHORT-NAME>StructCase_2</SHORT-NAME>
    <PARAMS>
        <PARAM xsi:type="VALUE">
            <SHORT-NAME>Env_Sequence</SHORT-NAME>
            <DOP-REF ID-REF="Field_DOP_ID"/>
        </PARAM>
    </PARAMS>
</STRUCTURE>

<DYNAMIC-ENDMARKER-FIELD ID="Field_DOP_ID">
    <SHORT-NAME>Field_DOP</SHORT-NAME>
    <BASIC-STRUCTURE-REF ID-REF="Env_DOP_ID"/>
</DYNAMIC-ENDMARKER_FIELD>

<STRUCTURE ID="Env_DOP_ID">
    <SHORT-NAME>Env</SHORT-NAME>
    <PARAMS>
        <PARAM xsi:type="VALUE">
            <SHORT-NAME>Temperature</SHORT-NAME>
        </PARAM>
        <PARAM xsi:type="VALUE">
            <SHORT-NAME>Speed</SHORT-NAME>
        </PARAM>
    </PARAMS>
</STRUCTURE>
```

All runtime responses and Response Parameters of the Results are based upon the respective database templates, which contain the related meta information. Thus, the name (semantic assignment) as well as the type is predefined. Concerning its structure, a RunTime result always has to be in conformance with the database template.

Firstly, always the type of the Response Parameter has to be requested. If this is a simple data type, the value and short name of the element can be polled (for the semantic interpretation of the result).

In case of a complex data type the number of the Response Parameters on the next hierarchical level is polled and following the list of these Response Parameters is read in.

This is repeated as often as no further Response Parameter is in hand.

In case of a Multiplexer the Value contains the select value (branch index beginning at 0) which has been used for the respective branch. The numbering of mux branches is determined by the ordering of the CASE elements of ODX data. If the ODX data defines a DEFAULT-CASE element, that element is appended as the last mux branch.

If an empty CASE (CASE without DOP) is used within a Multiplexer (MUX), it is not shown at the DbTemplate. The RunTime Result delivers a Multiplexer Element without a following ResponseParameter.

Parametername of Parameter in Mux shall be the Name of the Case in ODX (the cases reference to a structure, which have an own name, the elements of this structure will be used at next level of parameters at MVCI diagnostic server API).

Additionally, there is a possibility to poll the result for the related Service and the Service Parameters used for the execution.

Table 15 defines the overview about Request-, Response- and Protocol parameter data types.

Table 15 — Overview about Request-, Response- and Protocol parameter data types

DataType as delivered by getDataType()	Type	Included in Request Parameter	Included in Response Parameter	Physical data type of the MCDValue delivered by getValue()	Content of the Value delivered by getValue()
eFIELD	complex	---	Yes	A_UINT32	Number of structure entries directly contained in the field
eMULTIPLEXER	complex	---	Yes	A_UINT32	Branch index, beginning at 0
eSTRUCTURE	complex	Yes	Yes	A_UINT32	Number of entries directly contained in the structure
eSTRUCT_FIELD	complex	Yes (only at Protocol Parameters)	---	A_UINT32	Number of structure entries directly contained in the field
eENVDATA	complex	---	Yes	A_UINT32	Value of eDTC
eENVDATADESC	complex	---	Yes	A_UINT32	Number of entries directly contained in the ENVDATADESC parameter
eDTC	simple	Yes	Yes	A_UINT32	Value of the corresponding element TROUBLE-CODE
eEND_OF_PDU	complex	Yes	---	A_UINT32	Number of entries directly contained in the END_OF_PDU parameter
eTEXTTABLE	simple	Yes	Yes	A_UNICODE2STRING	Text of current conversion
eTABLE_ROW	complex	Yes	Yes	A_UINT32	Number of entries directly contained in the table row

Complex means that the data type is structured. The next hierarchical level of parameters can contain elements. So in case of an complex (structured) data type the next collection of parameters must be taken over with `getResponseParameters/getParameters`.

Table 16 defines parameter types for parameters generated by the MVCI diagnostic server (only if no direct relation to a parameter in ODX):

Table 16 — Parameter types

DataType as delivered by getDataType()	ParameterType As delivered by getMCDParameterType()
eSTRUCTURE	eGENERATED (in case the structure represents a MUX-CASE or is directly contained in a FIELD, eSTRUCT_FIELD or an END_OF_PDU)
eTABLE_ROW	eGENERATED

Table 17 defines the parameter types for protocol parameters.

Table 17 — Table of parameter types for protocol parameters

Type of Protocol Parameter with respect to ODX	ParameterType as delivered by getParameterType()	DataType as delivered by getDataType()
COMPARAM	eVALUE	depends on referenced DOP in ODX
COMPLEX-COMPARAM	eGENERATED	eSTRUCT_FIELD

Notes concerning the example:

For each ECU n errors may occur. Each error is symbolized by the ResponseParameter FSP. The errors of one ECU at a certain instant of time have been framed in the FSP Sequence for comparison with older versions of this specifiaction. Using the object model, the framing of errors into the FSP Sequence would not be necessary, as each error may be handed over as independent Response Parameter. Within the figure in hand, each Response contains only one Response Parameter (FSP_Sequence).

Basically, the result structure of a Service is determined by the data, to which the RunTime Result has to be in conformance with. However, generally it can be applied, that for each complete result data record (of a Cyclic Diag Service or non cyclic Service in Repetition mode) one ResultObject is created, that means that the number of result data records is identical to the number of ResultObjects.

The example "read diagnostic trouble code" is also used as Job example. For this, every minute the ECU is polled, but only every two minutes an intermediate result of the Job is given out. In this case also, the number of Results is two.

Subsequently an example for a simple data type (e.g. temperature) is shown. In this case the Response Structure only contains the element with the basic data type.

Figure 82 depicts the response structure DTC for only one ResponseParameter.

* MCDResponses
** MCDResponseParameters
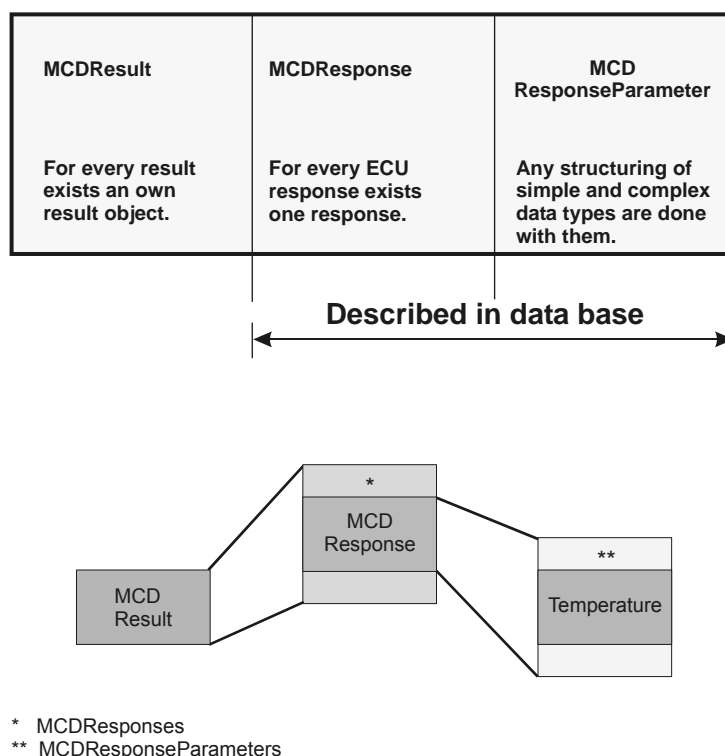
Figure 82 — Response structure DTC for only one ResponseParameter

Remark for Result handling in case of Functional Groups

For each ECU a DbResponse is available (the DbResponses of different ECU s must be identical) and so for a Functional Group exactly one DbResponse is available

Database:

— 1 DbFunctional Group,

— 1 DbService,

— 1 DbResponse (ODX: n DbResponses).

Runtime:

— 1 Functional Group,

— 1 Service,

— n Responses.

Figure 83 depicts an example for one service.

example for 1 Service:
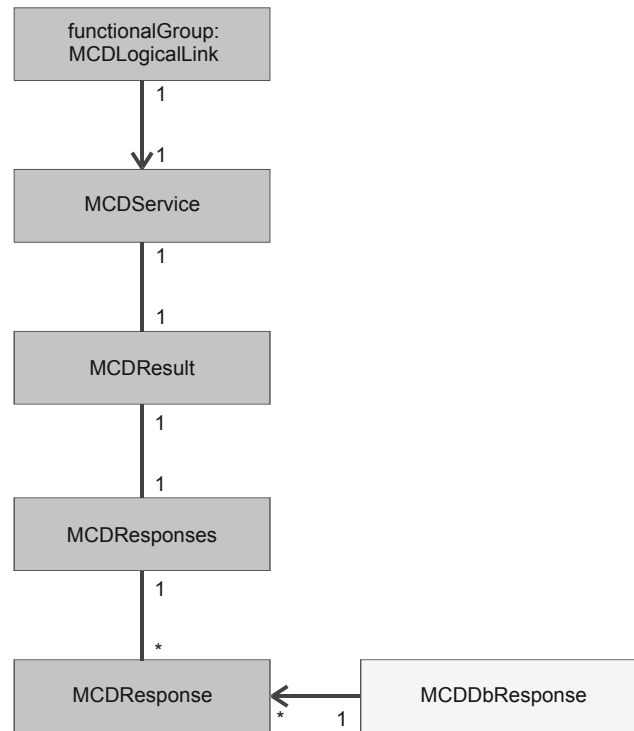
Figure 83 — Example for one service

The physical type of a Response must be identical. The conversion type (coding type) can be ECU specific. The rule for interpretation in implementations is shown in Figure 62.

— In case of functional communication the access key of the responding ECU is part of it's response:

— In case of response with the physical address the access key will be of base variant or variant.

— In case of response with functional address the access key will be of functional group.

8.9 Read DTC

8.9.1 ODX Data for Example Read DTC

ODX-Data (Extract) of the DB-Template

```
<DTC-DOP ID="DTC_DOP_ID">
   <SHORT-NAME>DTC_DOP</SHORT-NAME>
   (…)
<DIAG-CODED-TYPE    xsi:type="STANDARD-LENGTH-TYPE"    BASE-DATA-TYPE="A_
UINT32">
      <BIT-LENGTH>16</BIT-LENGTH>
```

```
        </DIAG-CODED-TYPE>
        <PHYSICAL-TYPE BASE-DATA-TYPE="A_UINT32"/>
        <DTCS>
            <DTC ID="DTC_110_ID">
                <SHORT-NAME>DTC_110</SHORT-NAME>
                <TROUBLE-CODE>110</TROUBLE-CODE>
                (…)
            </DTC>
            <DTC ID="DTC_120_ID">
                <SHORT-NAME>DTC_120</SHORT-NAME>
                <TROUBLE-CODE>120</TROUBLE-CODE>
                (…)
            </DTC>
            <DTC ID="DTC_130_ID">
                <SHORT-NAME>DTC_130</SHORT-NAME>
                <TROUBLE-CODE>130</TROUBLE-CODE>
                (…)
            </DTC>
            (…)
        </DTCS>
<PHYSICAL-TYPE BASE-DATA-TYPE="A_UINT32"/>
</DTC-DOP>

<DATA-OBJECT-PROP ID="SimpleDOP_Uint_ID">
    <SHORT-NAME>SimpleDOP_Uint</SHORT-NAME>
    (…)
<PHYSICAL-TYPE BASE-DATA-TYPE="A_UINT32"/>
</DATA-OBJECT-PROP>

<DATA-OBJECT-PROP ID="SimpleDOP_Uint16_ID">
    <SHORT-NAME>SimpleDOP_Uint16</SHORT-NAME>
    (…)
<PHYSICAL-TYPE BASE-DATA-TYPE="A_UINT16"/>
</DATA-OBJECT-PROP>

<DATA-OBJECT-PROP ID="SimpleDOP_Int_ID">
    <SHORT-NAME>SimpleDOP_Int</SHORT-NAME>
    (…)
<PHYSICAL-TYPE BASE-DATA-TYPE="A_INT32"/>
</DATA-OBJECT-PROP>

<ENV-DATA-DESC ID="EnvDataDesc_ID">
<SHORT-NAME>EnvDataDesc</SHORT-NAME>
<PARAM-SNREF SHORT-NAME="DTC"/>
<ENV-DATAS>
    <ENV-DATA ID="EnvData_DTC_110_ID"
        <SHORT-NAME>EnvData_A</SHORT-NAME>
        <PARAMS>
            <PARAM xsi:type="VALUE">
                <SHORT-NAME>Temperature</SHORT-NAME>
                (…)
                <DOP-REF ID-REF="SimpleDOP_Uint_ID"/>
            </PARAM>
        </PARAMS>
        <DTC-VALUES>
```

```
                    <DTC-VALUE>110</DTC-VALUE>
            </DTC-VALUES>
      <ENV-DATA>
      <ENV-DATA ID="EnvData_DTC_120_ID"
            <SHORT-NAME>EnvData_B</SHORT-NAME>
            <PARAMS>
                  <PARAM xsi:type="VALUE">
                        <SHORT-NAME>Env_Sequence</SHORT-NAME>
                        (…)
                        <DOP-REF ID-REF="SimpleFieldDOP_ID"/>
                  </PARAM>
            </PARAMS>
            <DTC-VALUES>
                  <DTC-VALUE>120</DTC-VALUE>
            </DTC-VALUES>
      <ENV-DATA>
      <ENV-DATA ID="EnvData_DTC_130_ID"
            <SHORT-NAME>EnvData_C</SHORT-NAME>
            <PARAMS>
                  <PARAM xsi:type="VALUE">
                        <SHORT-NAME>Temperature</SHORT-NAME>
                        (…)
                        <DOP-REF ID-REF="SimpleDOP_Uint_ID"/>
                  </PARAM>
                  <PARAM xsi:type="VALUE">
                        <SHORT-NAME>Speed</SHORT-NAME>
                        (…)
                        <DOP-REF ID-REF="SimpleDOP_Int_ID"/>
                  </PARAM>
            </PARAMS>
            <DTC-VALUES>
                  <DTC-VALUE>130</DTC-VALUE>
            </DTC-VALUES>
      <ENV-DATA>
      <ENV-DATA>
            <SHORT-NAME>Common</SHORT-NAME>
            <PARAMS>
                  <PARAM xsi:type="VALUE">
                        <SHORT-NAME>Env_Sequence</SHORT-NAME>
                        (…)
                        <DOP-REF ID-REF="CommonFieldDOP_ID"/>
                  </PARAM>
            </PARAMS>
            <ALL-VALUE/>
      </ENV-DATA>
</ENV-DATAS>
</ENV-DATA-DESC>

<DYNAMIC-ENDMARKER-FIELD ID="SimpleFieldDOP_ID">
      <SHORT-NAME>DEMF_Simple</SHORT-NAME>
      <BASIC-STRUCTURE-REF ID-REF="EnvStruct_ID">
      (…)
</DYNAMIC-ENDMARKER-FIELD>
```

```
<DYNAMIC-ENDMARKER-FIELD ID="CommonFieldDOP_ID">
   <SHORT-NAME>DEMF_Common</SHORT-NAME>
   <BASIC-STRUCTURE-REF ID-REF="CommonStruct_ID">
   (…)
</DYNAMIC-ENDMARKER-FIELD>


<STRUCTURE ID="EnvStruct_ID">
   <SHORT-NAME>Env</SHORT-NAME>
   (…)
<PARAMS>
   <PARAM xsi:type="VALUE">
      <SHORT-NAME>Temperature</SHORT-NAME>
      (…)
      <DOP-REF ID-REF="SimpleDOP_Uint_ID"/>
   </PARAM>
</PARAMS>
</STRUCTURE>


<STRUCTURE ID="CommonStruct_ID">
   <SHORT-NAME>CommonStruct</SHORT-NAME>
   (…)
<PARAMS>
   <PARAM xsi:type="VALUE">
      <SHORT-NAME>Temperature</SHORT-NAME>
      (…)
      <DOP-REF ID-REF="SimpleDOP_Uint_ID"/>
   </PARAM>
   <PARAM xsi:type="VALUE">
      <SHORT-NAME>Speed</SHORT-NAME>
      (…)
      <DOP-REF ID-REF="SimpleDOP_Int_ID"/>
   </PARAM>
</PARAMS>
</STRUCTURE>


<STRUCTURE ID="FSP_ID">
   <SHORT-NAME>FSP</SHORT-NAME>
   (…)
<PARAMS>
   <PARAM xsi:type="VALUE">
      <SHORT-NAME>DTC</SHORT-NAME>
      (…)
      <DOP-REF ID-REF="DTC_DOP_ID"/>
   </PARAM>
   <PARAM xsi:type="VALUE">
      <SHORT-NAME>DTC_State</SHORT-NAME>
      (…)
      <DOP-REF ID-REF="SimpleDOP_Uint16_ID"/>
   </PARAM>
   <PARAM xsi:type="VALUE">
      <SHORT-NAME>EnvRelation</SHORT-NAME>
      (…)
      <DOP-REF ID-REF="EnvDataDesc_ID"/>
   </PARAM>
</PARAMS>
```

```
</STRUCTURE>

<END-OF-PDU-FIELD ID="EOP_ID">
    <SHORT-NAME>EOP</SHORT-NAME>
(…)
    <BASIC-STRUCTURE-REF ID-REF="FSP_ID"/>
</END-OF-PDU-FIELD>
```

8.9.2   Reading without FaultMemories

All DTC values available for a Location may be read out directly using `MCDDbLocation::getDbDTCs()`. The DTC's have a manufacturer specific priority structure. For each DTC value all environment data can be polled with `MCDDbLocation::getDbEnvDataByTroubleCode`. The result MCDResponseParameters contains only one MCDResponseParameter of type eENVDATA for each DTC, which is corresponding ENVDATA to this trouble code.

In general, a DB template for environment data could contain several DTC DOPs and several ENVDATADESC DOPs. These DOPs can be structured arbitrarily . They can even reside on different hierarchical levels in the DB template.

In case of DTC there are three additional ResponseParameter types.

The type `eDTC` is a Simple DOP. For a ResponseParameter of the type `eDTC`, `getValue` returns the TroubleCode of the DTC as A_UINT32 encapsulated in MCDValue.

The type `eENVDATA` on the other hand is a Complex DOP, which represents the environment data. This type combines the advantages of a structure with the characteristic features of a Multiplexer. `eENVDATA` returns a Collection of ResponseParameters, which according to the occurring data types (Simple or Complex DOP) may contain data or further Collections of ResponseParameters. In case of a complex type only eFIELD, eSTRUCTURE, or eMULTIPLEXER are allowed. For a ResponseParameter of the type `eENVDATA` method `getValue` returns the Switch-Param as A_UINT32. This value is equal to the value of `eDTC`.

This shall be demonstrated with help of parts of the Database Template Figure 81.

Figure 84 depicts the different eEnvData blocks for an eDTC element I.
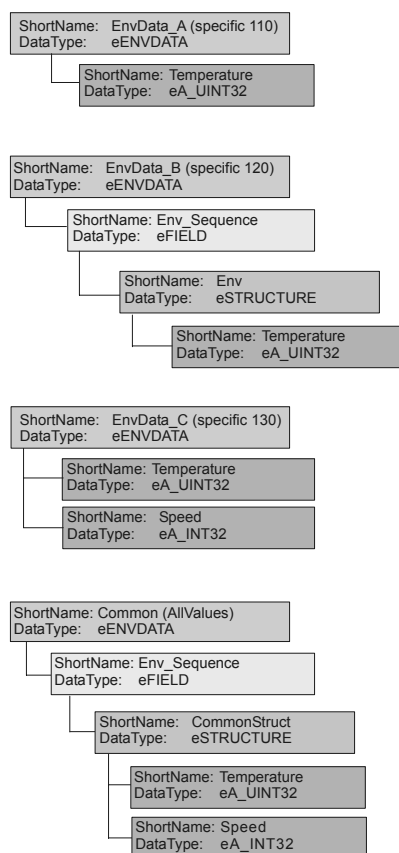
Figure 84 — Different eEnvData blocks for an eDTC element I

The type `eENVDATADESC` is a complex DOP, which announces the include of an environment data (`eENVDATA`) block. Inside a Db result template `eENVDATADESC` and `eDTC` must be on the same hierarchical level.

Every path in a response structure starting at its root element down to a leaf shall contain at most one element of type `eENVDATADESC`.

The collection of DbResponseParameters at this DOP (`eENVDATADESC`) consists of zero and the collection of ResponseParameters (run time side) consist of zero or one till two elements of type `eENVDATA`, i.e. the collection is empty if there is no environment data available. The method delivers first 0 or 1 ALL-VALUE ENV_Data and following 0 or 1 ENV-Data for the specific trouble code.

Figure 85 depicts the usage of Type eEnvDataDesc together with eDTC I.

```
ShortName: FSP_Sequence
DataType:   eFIELD
    │
    └── ShortName: FSP
        DataType:   eSTRUCTURE
            │
            ├── ShortName: DTC
            │   DataType:   eDTC
            │
            ├── ShortName: DTC_State
            │   DataType:   eA_UINT16
            │
            └── ShortName: EnvRelation
                DataType:   eENVDATADESC
```
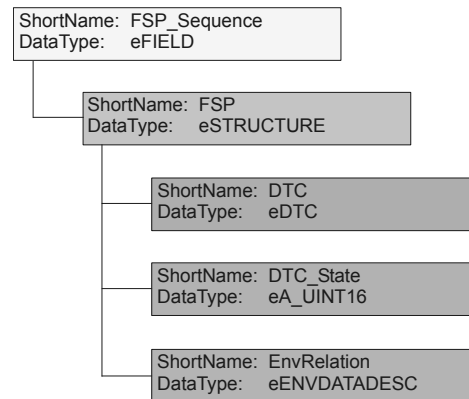
Figure 85 — Usage of Type eEnvDataDesc together with eDTC I

The names of all Response Parameters of the type `eENVDATA` within a Database Template have to be unique. Advantage of the separation between the database template finished with `eENVDATADESC` and the different `eENVData` constructs is, that independent from the Response Parameter `eDTC` (the DTC Value), all variants of the EnvironmentData, which may occur via the selected Service for this Location, must not be included within the Database Template.

Figure 86 depicts the relation between database template and different environment data blocks I.

```
ShortName: FSP_Sequence                    ShortName: Common (AllValues)        ShortName: EnvData_A (specific 110)
DataType:   eFIELD                         DataType:   eENVDATA                 DataType:   eENVDATA
    │                                          │                                    └── ShortName: Temperature
    └── ShortName: FSP                         └── ShortName: Env_Sequence              DataType:   eA_UINT32
        DataType:   eSTRUCTURE                     DataType:   eFIELD
            │                  V:110                   │              V:110        ShortName: EnvData_B (specific 120)
            ├── ShortName: DTC V:120                   └── ShortName: CommonStruct V:120  DataType:   eENVDATA
            │   DataType: eDTC V:130                       DataType:   eSTRUCTURE V:130       └── ShortName: Env_Sequence
            │                                                  │                                  DataType:   eFIELD
            ├── ShortName: DTC_State                           ├── ShortName: Temperature            │
            │   DataType: eA_UINT16                            │   DataType: eA_UINT32               └── ShortName: Env
            │                                                  │                                         DataType: eSTRUCTURE
            └── ShortName: EnvRelation                         └── ShortName: Speed                          │
                DataType: eENVDATADESC                             DataType: eA_INT32                         └── ShortName: Temperature
                                                                                                                 DataType: eA_UINT32

                                                                                 ShortName: EnvData_C (specific 130)
                                                                                 DataType:   eENVDATA
                                                                                     ├── ShortName: Temperature
                                                                                     │   DataType: eA_UINT32
                                                                                     └── ShortName: Speed
                                                                                         DataType: eA_INT32
```

Figure 86 — Relation between database template and different environment data blocks I

On the runtime side, the result is populated dynamically. Here, the value of the `eDTC` element defines which `eENVDATA` block is to be used in the runtime response structure.

Figure 87 depicts the RunTime result for DTC example I.

① getValue()
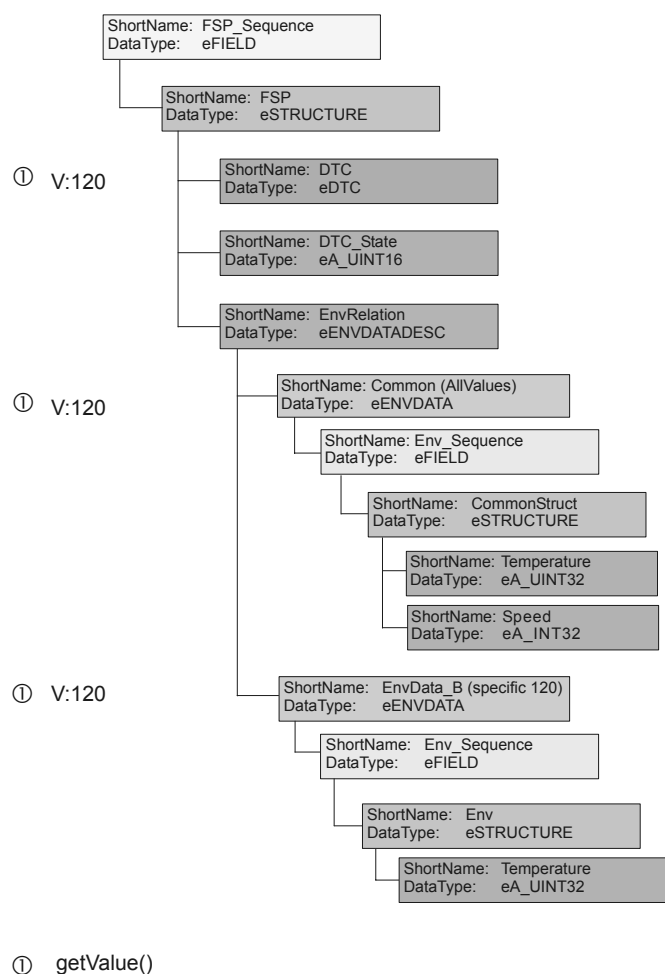
Figure 87 — RunTime result for DTC example I

8.9.3 Reading with FaultMemories

Per location (DIAG-LAYER) exits different fault memories each containing a set of DTCs. In diagnostic server all variant related DTCs should be returned for the database part. An element of type MCDDbFaultMemory (ODX: DTC-DOP) contains a collection of type MCDDbDiagTroubleCodes. The members of this collection are elements of type MCDDbDiagTroubleCode (ODX: DTC).

Figure 88 depicts the relation between FaultMemory and EnvDataDesc.
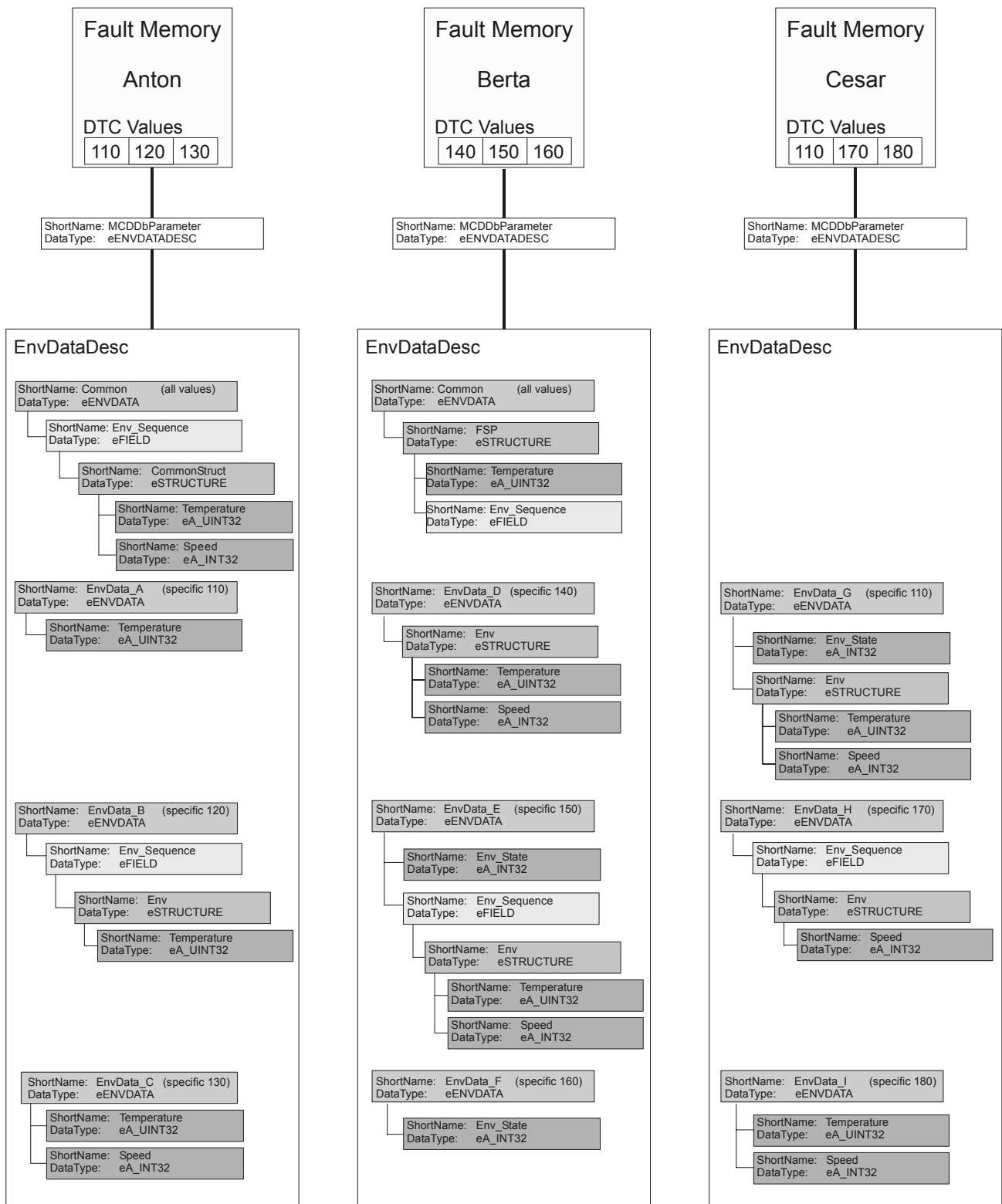
Figure 88 — Relation between FaultMemory and EnvDataDesc

In ODX, several DTC-DOPs can be referenced from one ECU. Furthermore, DTC-DOPs can be linked into another DTC-DOP. This allows to compose a new DTC-DOP from existing DTC-DOPs. At the interface of the diagnostic server, any DTC-DOP to be presented, that is, every DTC-DOP referenced by an ECU, is already converted into a flat list of DTCs – all links have been resolved.

In addition, there can be several eENVDATADESCs in ODX. Therefore, a response parameter of type eENVDATADESC in diagnostic server links together a eENVDATADESC from ODX with a response parameter of type eDTC. The value of this response parameter of type eDTC is then used in the server to calculate the sub-structures of the corresponding response parameter of type eENVDATADESC at runtime.

With respect to ODX, elements of type ENV-DATA are COMPLEXDOPs of type BASIC-STRUCTURE. Therefore, a response parameter of type eENVDATADESC can contain more than one complex response parameter of type eENVDATA.

Per ENV-DATA-DESC there can be at most one ENV-DATA applying to all DTCs, all others need to be DTC-specific. Every DTC can have at most one specific ENV-DATA applying to it. The order of the corresponding structures is defined as: ALL-VALUE ENVDATA comes first (named common in diagnostic server and may be empty, which means no data in ODX), then the DTCspecific ENV-DATA.

ENVDATA's are returned as collection of type MCDDbResponseParameters.

The method `MCDbEnvDataDesc::getCompleteDbEnvDatasByDiagTroubleCode (A_UNIT32 troubleCode)` returns a collection of type MCDDbResponseParameters. This collection contains at most a MCDDbResponseParameter of type eENVDATA which contains response parameters representing common environment data and at most a MCDDbResponseParameter of type eENVDATA which contains response parameters representing environment data specific to a certain DTC value (mind the order).

The method `MCDbEnvDataDesc::getCommonDbEnvDatas()` returns a collection of type MCDDbResponseParameters. This collection contains at most a MCDDbResponseParameter of type eENVDATA which contains response parameters representing common environment data.

Figure 89 depicts the common eEnvData block for Fault Memory with name "Anton".
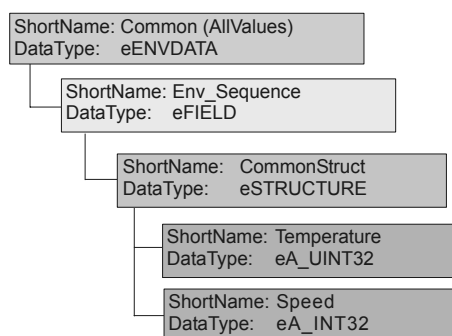
Figure 89 — Common eEnvData block for Fault Memory with name "Anton"

The method `MCDbEnvDataDesc::getSpecificDbEnvDatasByDiagTroubleCode (A_UNIT32 troubleCode)` returns a collection of type MCDDbResponseParameters. This collection contains at most a MCDDbResponseParameter of type eENVDATA which contains response parameters representing environment data specific to a certain DTC value.

Figure 90 depicts the Different eEnvData blocks for an eDTC element II.
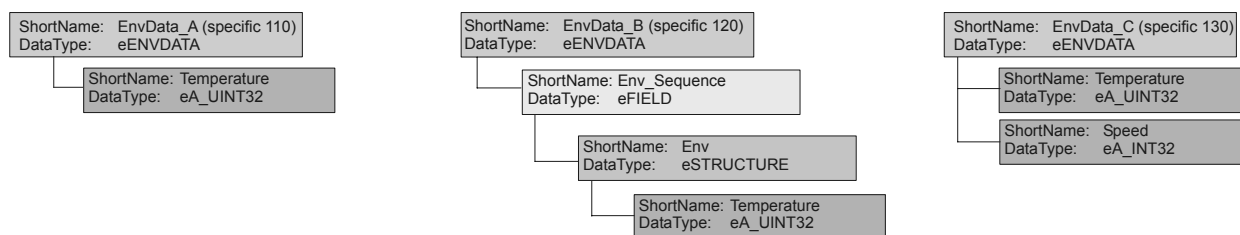
Figure 90 — Different eEnvData blocks for an eDTC element II

For details and example see 8.9.2.

8.9.4 DTC Read Service

The SEMANTIC-Attribute „DEFAULT_FAULT_READ", which is allowed to occur only once for each Location, is used for the instancing of the Read DTC Service by means of the method `MCDLogicalLink::createDiagComPrimtiveBySemanticAttribute(semantic:A_ASCII STRING):MCDDiagComPrimitive`.

The method `MCDDbLocation::getDbServicesBySemanticAttribute (semantic_A_ASCIISTRING):MCDDbServices` is used to filter out Services from the database on the basis of the Semantic Attributes defined within the data base. To leave open the possibility to define further Semantic Attributes, the Semantic Attribute is handed over as String.

In case of the existence of more than one service for one semantic attribute the method `MCDLogicalLink::createDiagComPrimitiveBySemanticAttribut` does not create any service but returns the error `eRT_NO_UNIQUE_SEMANTIC_ATTRIBUTE`. Some semantic attributes are unique (e.g. STARTCOM, STOPCOM).

8.10 Logical Link

8.10.1 Connection overview

Information about a Logical Link is contained in the Logical Link Table. Elements of this table are the AccessKey, which includes protocol, and the Physical Vehicle Link (because it is only a description of the vehicle side). Logical Links are used to access the same ECU on different ways, or access more than one ECU instance on different links.

By selecting a Logical Link, the selection of a Location and the respective access path within the Client takes place.

The application can use the short name of Logical Links to instance and work with ECU s. The short name is defined in the Logical Link Table. One method of the Logical Link will deliver the access key.

Logical Links are used to access the same ECU on different ways or access more than one ECU instance on different links. Different Logical Links can share the same PhysicalLink to different ECU s. Each Logical Link is assigned an own instruction queue (represented by Activity state) for the execution of DiagComPrimitives.

Within the Logical Link Table, only the BaseVariant is entered for each ECU, which describes an unambiguous access path to the ECU. The Variant may be polled or identified, the instancing of a Variant is also possible via the Logical Link. For every instantiated Logical Link only one Location can be active at one point in time for one ECU.

8.10.2 State diagram of Logical Link

At the Logical Link it is distinguished between the Logical Link States and the Activity (Queue) States. The state diagrams are influenced by the states of the Primitives / Services (see Table 12).

All state transitions are indicated by means of Events. Non state changing operations, for Logical Link State and the Activity (Queue) State, will not produce an event.

The <D> method `MCDLogicalLink:reset()`called in state `eCREATED` does not produce an event or an exception. This is a state transition inside the state.

Figure 91 depicts the state diagram Logical Link in function block D.
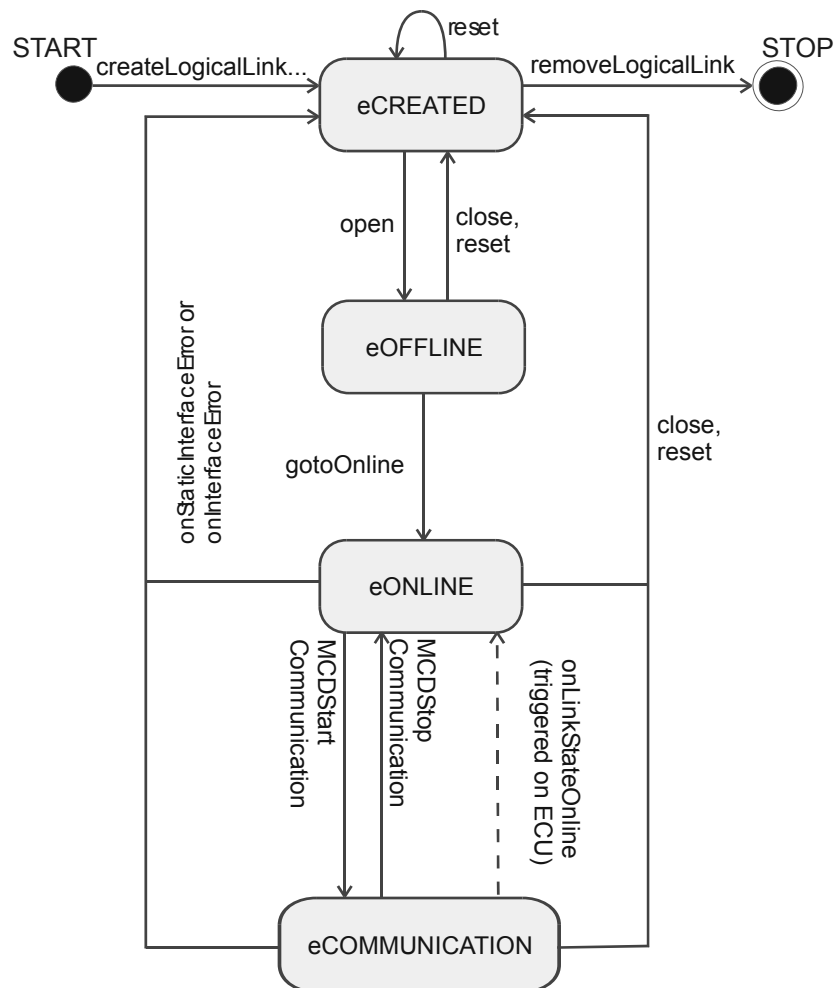
Figure 91 — State diagram Logical Link in function block D

Table 18 defines the Logical Link states.

Table 18 — Logical Link states

System State	LogicalLink State	MCD Project	MCDLogicalLink							MCDStart Communication	MCDStop Communication	MCD Interface
		removeLogicalLink	open	close	reset	getInterfaceResource	gotoOnline	sendBreak	createDiagComPrimitive	executeSync	executeSync	disconnect
eLOGICALLY_ CONNECTED	eCREATED	X	X		X	X	---	---	X	---	---	Xª
	eOFFLINE	---	---	X	X	X	X	X	X	---	---	---
	eONLINE	---	---	X	X	X	---	X	X	X	---	---
	eCOMMUNICATION	---	---	X	X	X	---	X	X	---	X	---

ª valid action in case all MCDLogicalLinks, referencing this MCDInterface, are in state eCREATED, otherwise invalid action, an exception will be thrown, see corresponding method definition

In case of an error `MCDLogicalLink::close()` throws an exception and does not perform a state change. `reset()` never throws an exception. The same holds true for `MCDLogicalLink::open()` and `MCDLogicalLink::gotoOnline().MCDLogicalLink::reset()` never throws an exception.

If any exception occurs during a state changing operation, the state shall not be changed in general! For successful state changes the MCDExecutionState shall be eALL_POSITIVE. State is not changed, if MCDExecutionState is eNEGATIVE / eALL_NEGATIVE or eFAILED.

Table 19 defines the Logical Link state description.

Table 19 — Logical Link state description

Logical Link State	Description
eCREATED Event: onLinkStateCreated	Logical Link has been created, but is not ready for operation.
eOFFLINE Event: onLinkStateOffline	The Logical Link has opened a hardware channel. No logical connection to the ECU exists, that means no communication has taken place.
eONLINE Event: onLinkStateOnline	A logical connection to the ECU exists, but no DiagComPrimitive or Service is executed.
eCOMMUNICATION Event: onLinkStateCommunicating	A logical connection to the ECU exists, at least one DiagComPrimitive or Service is executed.

Figure 92 depicts the state diagram ACTIVITY (QUEUE) states.
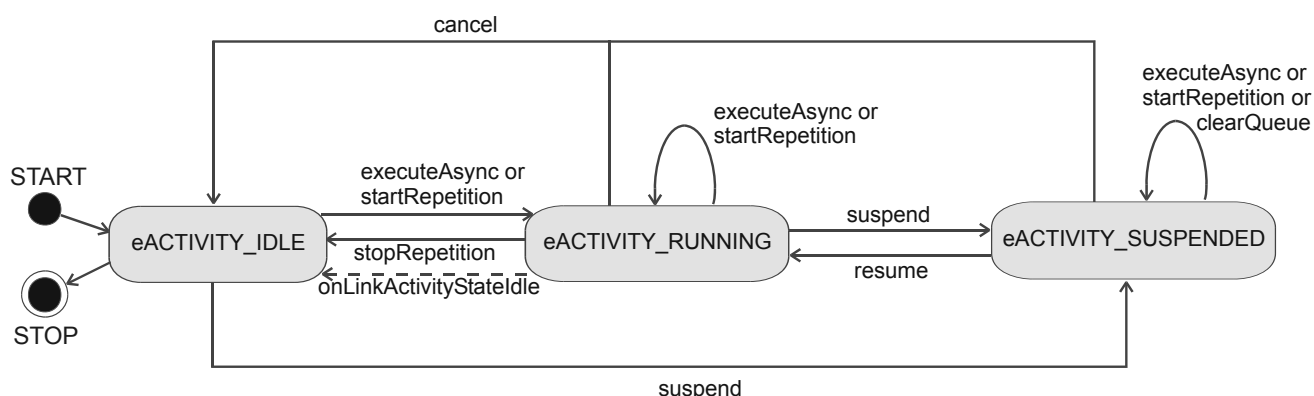
Figure 92 — State diagram ACTIVITY (QUEUE) states

Table 20 defines the Activity states.

Table 20 — Activity states

System State	LL State	Activity State	MCDLogicalLink			
			suspend	resume	clearQueue	getQueueState
eLOGICALLY_CONNECTED	eCREATED	eACTIVITY_IDLE	-	-	-	-
		eACTIVITY_RUNNING	-	-	-	-
		eACTIVITY_SUSPENDED	-	-	-	-
	eOFFLINE	eACTIVITY_IDLE	-	-	-	-
		eACTIVITY_RUNNING	-	-	-	-
		eACTIVITY_SUSPENDED	-	-	-	-
	eONLINE & eCOMMUNICATION	eACTIVITY_IDLE	X	-	-	X
		eACTIVITY_RUNNING	X	-	-	X
		eACTIVITY_SUSPENDED	-	X	X	X

The method `MCDLogicalLink::suspend()` has effect on all services, also on repeated services. Results coming in while the Activity queue is suspended will be deleted and not be given to the client application. Noncyclic single diagnostic services currently running on the ECU are completed and the result is delivered to the client. The results of cyclic diagnostic services are not delivered to the application but the cyclic service will be continued on the ECU. The result of currently running repeated services are not delivered to the application, repetition of the service will not continue until a call `MCDLogicalLink::resume()`. The repetition timer of a repeated service is unaffected by suspension. If the repetition time is reached during suspension no action is taken and the repetition timer is rescheduled. New services from the client and services already in repetition shall be executed intermittently by the MVCI diagnostic server after `resume`.

Repetition of services is performed by a scheduler in the diagnostic server. Thus, `stopRepitition()` does not require to send any information to the ECU. Rather, `stopRepitition()` is a method that tells the diagnostic server internal scheduler not to insert the respective service into the queue again to avoid

another repetition cycle. In contrast, the method `startRepitition()` inserts a service into the queue which is then to be repeated by the diagnostic server. For `updateRepitition()` the same applies as for `stopRepitition()`. As a result, `stopRepitition()` and `updateRepitition()` are not queued nor do these methods queue any service.

The method `MCDLogicalLink::clearQueue()` has no effect on running repeated and cyclic services. This method does not change the state `eACTIVITY_SUSPENDED`.

Calls of the methods `executeAsync()` and `startRepetition()` are allowed in state `eACTIVITY_SUSPENDED`, the execution will not be started unless `resume()` has been called.

A call of `resume()` changes the state from `eACTIVITY_SUSPENDED` to `eACTIVITY_RUNNING`. In case of an empty queue the state `eACTIVITY_IDLE` is reached with an event `OnLinkActivityStateIdle`.

Running cyclic services and repeated services are stopped by the MVCI diagnostic server through `cancel` or `stopRepetition` calls, respectively. The state will be changed from `eACTIVITY_SUSPENDED` to `eACTIVITY_IDLE`.

`VariantIdentification(AndSelection)` are also allowed in logical link state `eONLINE`.

Table 21 defines the Activity queue states description.

Table 21 — Activity queue states description

Activity State	Description
`eACTIVITY_IDLE` Event: onLinkActivityStateIdle	A logical connection to the ECU exists, but no DiagComPrimitive or Service is executed (DiagComPrimitiveState: eIDLE, RepetitionState: eNOT_REPEATING).
`eACTIVITY_RUNNING` Event: onLinkActivityStateRunning	A logical connection to the ECU exists, at least one DiagComPrimitive or Service is executed (DiagComPrimitiveState: ePENDING, RepetitionState: eREPEATING or eNOT_REPEATING). As soon as all services have been finished, the state switches back to eACTIVITY_IDLE. New services from the client and services already in repetition shall be executed intermittently by the MVCI diagnostic server.
`eACTIVITY_SUSPENDED` Event: onLinkActivityStateSuspended	The execution via activity queue has been stopped. Within this state, a DiagComPrimitive which is already in the activity queue can be cancelled or a DiagComPrimitive may be put into the activity queue asynchronously or by means of `startRepetition()`. All services are affected, also repeating and cyclic. Results of repeating or cyclic services will not be transferred to the application. Non-cyclic single services currently running on the ECU get completed and the result is delivered to the client. Cyclic services are continuing on the ECU and have to be cancelled separately. Repeating services get stopped until `resume`. The repetition timer is unaffected by the suspension. If the repetition time is reached during suspension, no action is taken and the repetition timer is rescheduled. `ClearQueue` deletes all single and repeating services.

Figure 93 depicts the Logical Link state diagram in function block D including ACTIVITY (QUEUE) state.
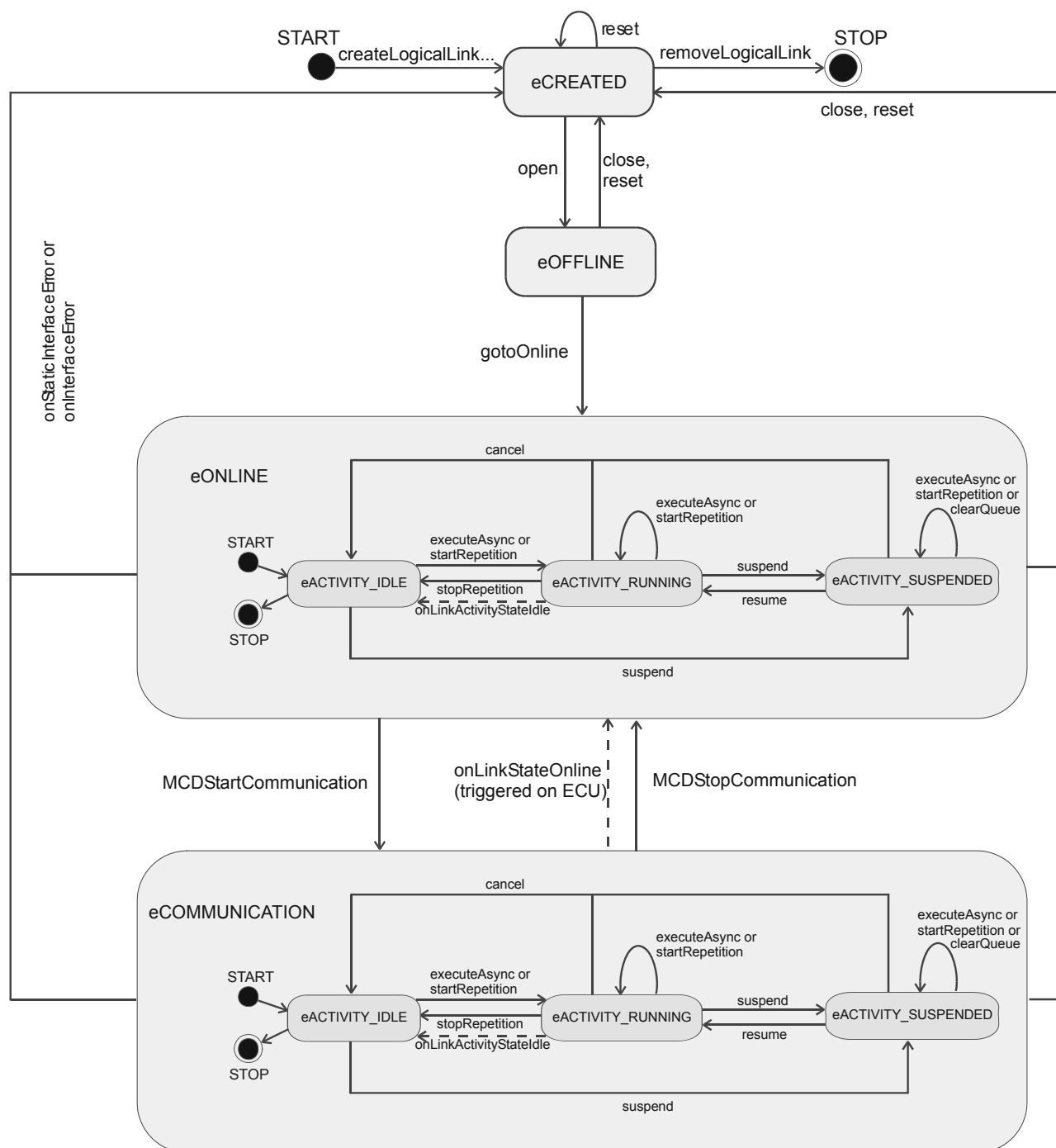
Figure 93 — Logical Link state diagram in function block D including ACTIVITY (QUEUE) state

The synchronous execution of Services takes place directly within the ACTIVITY States eACTIVITY_IDLE and eACTIVITY _RUNNING and does not cause a state change.

Table 22 defines the relations between states and actions.

Table 22 — Relations between states and actions

	Logical Link	ECU	Data base	Activity
eCREATED	Logical Link Object created	Not connected	Location accessed, no DB changes	---
eOFFLINE	Channel allocated	Connected communication	Location accessed, no DB changes	---
eONLINE / eIDLE	Communicating	Connected communication	VI/VIS and DB changes possible	No service running
eCOMMUNICATION eACTIVITY_IDLE	Communicating	Communicating	VI/VIS and DB changes possible	No service running
eACTIVITY_RUNNING	Communicating	---	No DB changes	Services running
eACTIVITY_SUSPENDED	Communicating	---	No DB changes	Suspended

Remark:

The location gets reset to the firstly defined one within the state change from eCOMMUNICATION to eONLINE or eCREATED.

For `setQueueSize` the logical link state must be in state `eOFFLINE` or a further state. In case the new queue size is smaller than the current queue size, the reduction becomes effective as soon as the actual activity queue is below the new threshold. The size is given as the number of DiagComPrimitives.

For `setEventHandler`, `releaseEventHandler` (for the logical link event) and `removeDiagComPrimitive` the activity queue must not be running.

At the first creation of a Logical Link, within the MVCI diagnostic server a object is created and the reference to this object is returned to the Client. If this or another Client creates a further Logical Link with the same combination of the `MCDDbLocigalLink`, `MCDDbLocation`, `MCDInterface` and `MCDInterfaceResource`, only a reference to the already created Logical Link within the MVCI diagnostic server is returned.

Only for methods `createLogicalLinkByInterfaceResource`, `createLogicalLinkByNameAndInterfaceResource` and `createLogicalLinkByVariantAndInterfaceResource` the instance of an LogicalLink is unique in case of qualitative identical parameters. The usage of methods `createLogicalLinkBy…Interface` (without MCDInterfaceResource) is implementation specific, because it depends on how the underlying communication layer selects a certain interface resource.

Furthermore, the combination of `accesskey` and `physicalVehicleLink` may not uniquely identify a certain `logicalLink`, e.g. if the vehicleInformation contains several logicalLinks, having the same accesskey and physicalVehicleLink but different communication parameters. In that case a deterministic creation of logicalLinks is not possible. Thus, for supporting such use-cases other methods than `createLogicalLinkByAccessKey...` should be used instead.

Because of this, all DiagComPrimitives and Services which are executed on this Logical Link are put into the same activity queue and are executed there. Thus, no overlapping and undesired parallel executions of DiagComPrimitives may occur. As many diagnostic services as desired may be executed via the Activity Queue per Logical Link. This provides for the possibility, to execute diagnostic services parallel.

The MVCI diagnostic server has to make sure, that the results of the DiagComPrimitive- or Service-executions get to the right reference of the Logical Link.

As long as there is no DiagComPrimitive or Service within the Activity Queue, the Activity Queue is within state `eACTIVITY_IDLE`. If a DiagComPrimitive or Service is put into the Activity Queue for execution, the Activity Queue takes the state `eACTIVITY_RUNNING`.

8.10.3 VCI communication lost handling

8.10.3.1   Basics

If the D-PDU API or any proprietary diagnostic VCI connection detects a connection lost to the VCI, one of the events `MCDEventHandler::onInterfaceError` or `MCDEventHandler::onStaticInterfaceError` is fired. The client can detect that the communication with the VCI was lost and should take appropriate actions.

After firing one of the two events, all ComPrimitives running on logical links on the VCI are cancelled. All opened logical links on this VCI change their state into `eOFFLINE`.

It is important for the client to know, that all consecutive calls to `MCDLogicalLink::gotoOnline()` will fail with an exception as the internal logical link cannot be recovered (as specified in D-PDU API) and thus the link can never reach the state `eONLINE` anymore. Even if the communication to the VCI would be recovered meanwhile, the logical link must be removed and created again to continue the communication.

Upon receiving the `onInterfaceError()` or `onStaticInterfaceError` all comprimitives are canceled and the MCDResult object is available and can be processed. Running comprimitives will be terminated by the MVCI diagnostic server itself. In this case MCDExecutionState will be set to `eCANCELED_DURING_EXECUTION` and the result contains the error `eCOM_LOST_COMM_TO_VCI`.

8.10.3.2   Example how a client could behave upon receiving `onInterfaceError` or `onStaticInterfaceError`

For all comprimitives:

— `MCDResult` processing (if necessary)

— `MCDLogicalLink::removeDiagComPrimitive()`

For all logical links:

— `MCDLogicalLink::reset()`

— `MCDProject::removeLogicalLink()`

Case static VCIs (call to `MCDSystem::prepareInterface()`):

One time

— `MCDSystem::unprepareInterface()` in case of static VCI

To recover the communication

— Repeated call to `MCDSystem::prepareInterface() / prepareVCIAccessLayer()`

— Create new logical links and continue the communication

Case of dynamic VCIs (call to `MCDSystem::prepareVCIAccessLayer()`):

— `MCDInterface::disconnect()`
  Upon receiving the event `MCDSystem::onInterfacesChanged()`, call `MCDSystem::getCurrentInterfaces`, select VCI and continue. A call to `MCDSystem::prepareVCIAccessLayer` is not necessary.

8.10.4 Logical Link examples

The Logical Link Table consists of five columns. The first column is just a counter for the row. The following three columns (Shortname of Logical Link, AccessKey, Shortname of PhysicalVehicleLink) are part of a Logical Link. The last two columns (Logical Link ShortName of Gateway and GetGatewayMode) are for information and initialisation purposes.

— The unique short name of a Logical Link is used by the application/job to access the ECU.

— The MVCI diagnostic server uses the unique short name of a Logical Link to find the related row in the Logical Link Table. From this row the MVCI diagnostic server reads the AccessKey and the Physical Vehicle Link needed to physically address the ECU.

— Only ECUs which are referred by Logical Links in the Logical Link Table can be accessed by applications/jobs.

— The Logical Link Table is vehicle dependent.

EXAMPLE I:

in database must exist:

— one reference of the ECM;

— a test with the instances of four ECMs and their different Logical Links;

Figure 94 depicts the four ECMs of same type in one test in one project.
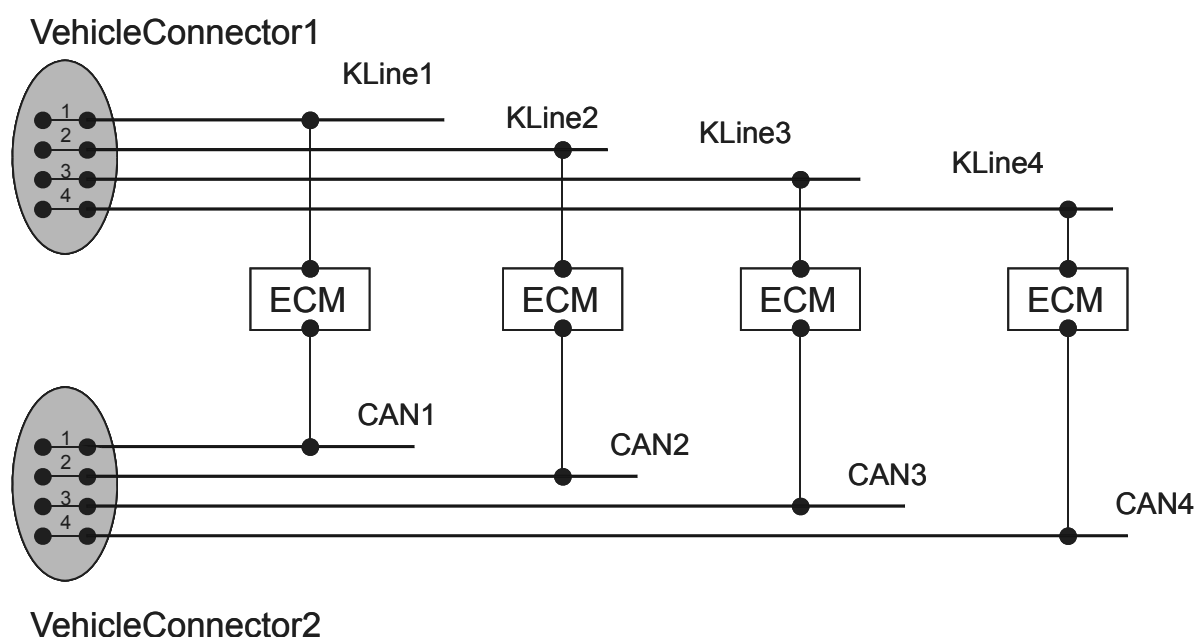
Figure 94 — Four ECMs of same type in one test in one project

Table 23 defines the Logical Link Table for example I.

Table 23 — Logical Link Table for example I

ShortName	AccessKey V2.0	ShortNamePhysical Vehicle Link	Logical Link ShortName of Gateway	Get Gateway Mode
ECM-KLine1	[Protocol]KWP2000.[ECUBaseVariant].ECM	KLINE1	---	---
ECM-KLine2	[Protocol]KWP2000.[ECUBaseVariant].ECM	KLINE2	---	---
ECM-KLine3	[Protocol]KWP2000.[ECUBaseVariant].ECM	KLINE2	---	---
ECM-KLine4	[Protocol]KWP2000.[ECUBaseVariant].ECM	KLINE4	---	---
ECM-CAN1	[Protocol]DiagOnCan.[ECUBaseVariant].ECM	CAN1	---	---
ECM-CAN2	[Protocol]DiagOnCan.[ECUBaseVariant].ECM	CAN2	---	---
ECM-CAN3	[Protocol]DiagOnCan.[ECUBaseVariant].ECM	CAN3	---	---
ECM-CAN4	[Protocol]DiagOnCan.[ECUBaseVariant].ECM	CAN4	---	---

The Vehicle Connector Information Table has five columns. The first column is just a counter for the row. The following two columns (Physical Vehicle Link and Vehicle Connector Information) describe the one to many relation (1-n) between the Physical Vehicle Links and the Pins of the VehicleConnector.

— Because one Physical Vehicle Link could be accessed by two different connectors (see row 1 and 2) the MVCI diagnostic server could find multiple rows in this table.

— The entries for connector and pins in the column Vehicle Connector Information are unique, because two Physical Vehicle Links cannot be connected.

— One Physical Vehicle Link, i. e. CAN, could use more than one pin at a connector.

— Table 24 defines the Vehicle Connector Information Table for example I.

Table 24 — Vehicle Connector Information Table for example I

No	ShortName of Physical Vehicle Link	VehicleConnectorInformation	Type	LongName
1	KLine1	Connector1_Pin1	KLINE	Diagnostic Line
2	KLine2	Connector1_Pin2	KLINE	Diagnostic Line
3	KLine2	Connector1_Pin3	KLINE	Diagnostic Line
4	KLine4	Connector1_Pin4	KLINE	Diagnostic Line
5	CAN1	Connector2_Pin1	CAN	Body CAN High Speed
6	CAN2	Connector2_Pin2	CAN	Body CAN High Speed
7	CAN3	Connector2_Pin3	CAN	Body CAN High Speed
8	CAN4	Connector2_Pin4	CAN	Body CAN High Speed

EXAMPLE II:

Figure 95 depicts the example Logical Link with functional group.
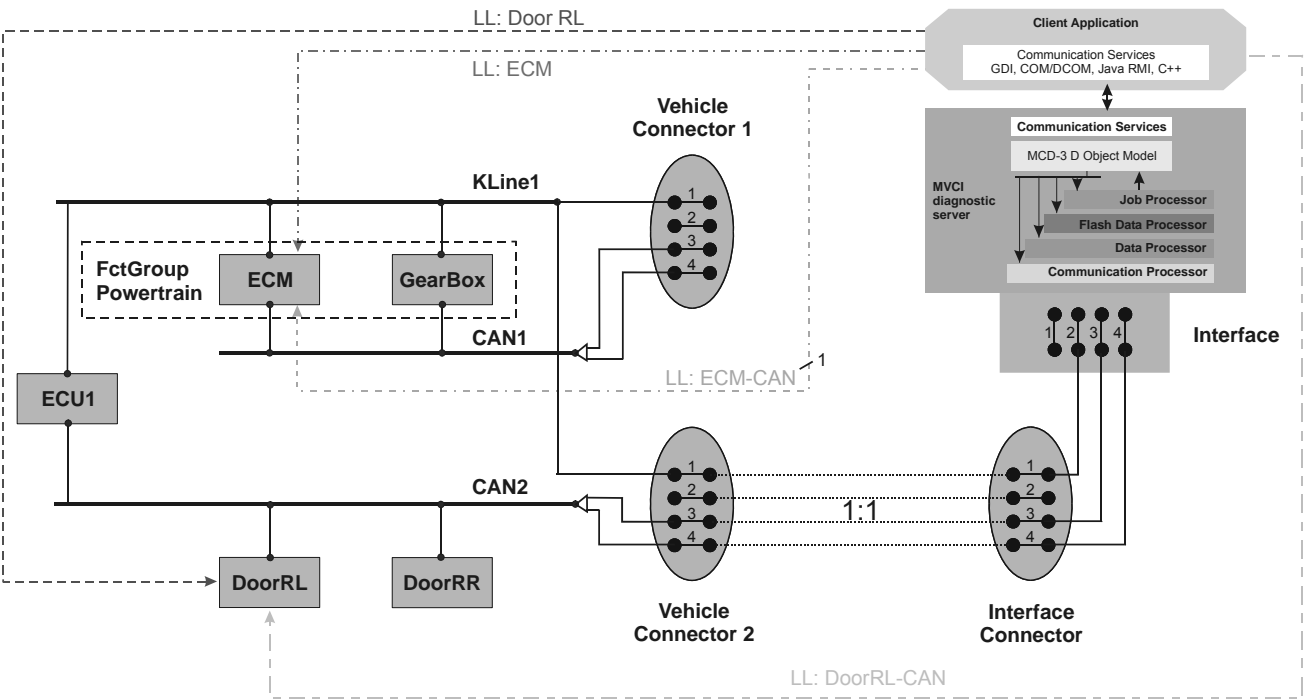
Key

1   not possible at moment, because Vehicle Connector 2 is connected to Interface Connector

Figure 95 — Example Logical Link with functional group

Table 25 defines the Logical Link Table for example II.

Table 25 — Logical Link Table for example II

No.	ShortName of Logical Link	AccessKey V2.0	ShortName Physical Vehicle Link	Logical Link ShortName of Gateway	Get Gateway Mode
1	ECM	[Protocol]KWP2000. [ECUBaseVariant]ECM	KLINE1	---	---
2	Powertrain	[Protocol]KWP2000[FunctionalGroup]Powertrain	KLINE1	---	---
3	GearBox	[Protocol]KWP2000. [ECUBaseVariant]GearBox	KLINE1	---	---
4	ECU1	[Protocol]KWP2000. [ECUBaseVariant]ECU1	KLINE1	---	visible
5	DoorRR	[Protocol]KWP2000. [ECUBaseVariant]DoorRR	KLINE1	ECU1	---
6	DoorRL	[Protocol]KWP2000. [ECUBaseVariant]DoorRL	KLINE1	ECU1	--
7	ECM-CAN	[Protocol]DiagOnCAN. [ECUBaseVariant]ECM	CAN1	---	---
8	GearBox-CAN	[Protocol]DiagOnCAN. [ECUBaseVariant]GearBox	CAN1	---	---
9	ECU1-CAN	[Protocol]DiagOnCAN. [ECUBaseVariant]ECU1	CAN2	---	---
10	DoorRR-CAN	[Protocol]DiagOnCAN. [ECUBaseVariant]DoorRR	CAN2	---	---
11	DoorRL-CAN	[Protocol]DiagOnCAN. [ECUBaseVariant]DoorRL	CAN2	---	---
12	CAN	[Protocol]DiagOnCAN	CAN1	---	---
13	KWP	[Protocol]KWP2000	KLINE1	---	---

It is distinguished between

— visible and

— transparent

gateways. In case of transparent gateways Logical Links of ECUs behind the gateway are used in the same way like ECUs without gateways in the Logical Link (AccessKey). The sever handles this automatically, so that the gateway handling is not visible for the application.

Before communication with a ECU, it must be checked if the ECU is a member. In this case all referenced gateways of the ECU will be listed. If the corresponding gateway is already open, it will be used. Otherwise, the gateway will be opened before.

A Client creates a Logical Link either to an ECU or to a Gateway! In case of a Logical Link to an ECU, the MVCI diagnostic server internally opens the necessary communication to the corresponding Gateway. This process is transparent for the Client, that is, the Client does not know about the composition this physical communication connection (Internal management in MVCI diagnostic server: check of Members opened via a certain Gateway).

Gateway will be closed by the MVCI diagnostic server, if no Logical Link to a Member exists and also no Logical Link to the Gateway.

If the communication to a Gateway is interrupted/broken, an Event (`onLinkStateOnline`) is sent to the Client for each accessed Member (open Logical Link) of the Gateway.

If there is any open member the protocol parameters of the gateway cannot change.

Table 26   Vehicle Connector Information Table for example II

No.	ShortName of Physical Vehicle Link	VehicleConnectorInformation	Type	LongName
1	KLine1	Connector1_Pin1	KLINE	Diagnostic Line
2	KLine1	Connector2_Pin1	KLINE	Diagnostic Line
3	CAN2	Connector2_Pin3CanH Connector2_Pin4CanL	CAN	Body CAN High Speed
4	CAN1	Connector1_Pin3CanH Connector1_Pin4CanL	CAN	Powertrain CAN

8.11 Functional addressing

Functional addressing is a communication mode similar to a multicast in established network environments. While a physically addressed service is targeted at (and answered by) only one ECU, a functionally addressed service is usually received and answered by multiple ECUs. An example use case would be an OBD scenario where all ECUs that are emission-critical are part of one functional group and can all be addressed by a single functional service. To use functional addressing, a logical link to a functional group has to be opened. This logical link can then be used for functional communication with the set of ECUs that are part of the functional group. It is also possible to use functional addressing alongside physical addressing. To this end, a logical link to a specific ECU and a second logical link to the functional group this ECU belongs to have to be opened. Then, the logical link of the functional group can be used for functional addressing and the logical link of the ECU can be used for physical addressing.

The following pseudo code shows an exemplary interaction between a client application and the diagnostic server for using functional addressing [assuming the initial logical link is pointing to an ECU Base Variant]*:

1. `MCDDbLogicalLink.getDbLocation().getAccesskey().getProtocol()`
   `=> MCDDbDatatypeShortName` *protocolNameBV*

2. `MCDDbLogicalLink.getDbLocation().getDbEcu() => MCDDbEcu` *dbEcu*

3. `[IF dbEcu.getObjectType() EQUALS eMCDDBECUBASEVARIANT]*`
   `((MCDDbEcuBaseVariant)dbEcu).getDbFunctionalGroups()`
   `=> MCDDbEcuFunctionalGroups dbFunctionalGroups`

4. For each `MCDDbFunctionalGroup` in `dbFunctionalGroups`
   `MCDDbFunctionalGroup.getDbLocations() => MCDDbLocations dbLocations`

5. For each `MCDDbLocation` in `dbLocations MCDDbLocation.getAccessKey()`
   `=> MCDAccessKey accessKey`

6. `accessKey.getProtocol() => MCDDatatypeShortname` *protocolNameFG*

7. If *protocolNameBV* equals *protocolNameFG*

8. `MCDDbLogicalLink.getDbPhysicalVehicleLinkOrInterface.getShortname()` => *baseVariantPhysicalVehicleLink*

9. `MCDProject.createLogicalLinkByAccessKey(accesskeyLL,` baseVariant*PhysicalVehicleLink*)

When communicating functionally, the request of the functionally executed service is used as defined at the functional group. However, the response of this service could be overridden on BaseVariant or EcuVariant level. If this is the case, the responses of ECUs will be evaluated using the response patterns of the most specific diagnostic layer that is applicable (EcuVariant layer if ECU variant identification has been performed beforehand, the BaseVariant layer otherwise). If the request of a service defined on a functional level is overridden on the BaseVariant or EcuVariant level, this overridden request is only used when the service is executed physically (the service is created and executed on a logical link that points to a BaseVariant or EcuVariant location).

A functional group can contain services with and without responses. Services without responses (send-only services) can always be executed on the functional group level. For these services, no definition of physical addresses is required, that is, no information on the physical ECUs is necessary, e.g. to identify which ECUs have responded. An example for such a service would be a functional tester present message. For services with responses, response interpretation takes place on the FunctionalGroup, BaseVariant (no variant identified) or EcuVariant (variant identified) level. In case a base variant overrides the response of a functional group service, the service's response is interpreted on the base variant level for this ECU. The decision whether to use the response pattern defined on the functional group level or the one defined on the base variant level needs to be made individually for every ECU at runtime. This means that information on the physical addresses of all possibly responding ECUs is required. The physical addresses need to be defined on the base variant level, e.g. by means of corresponding communication parameter definitions. In case a functional service is overridden in an EcuVariant, a flag is set on the corresponding BaseVariant instance. This flag informs the diagnostic server not to interpret the response of this ECU until variant identification has taken place. If in this context an ECU variant cannot be identified, the responses of that ECU are not interpreted at all. Instead, an error is put in the result for this respective response and the response's PDU is returned without interpretation. Setting the flag when overriding a functional service on an EcuVariant level is considered mandatory.

There are two ways for managing the physical response addresses of ECUs which can potentially respond to a functional request – either a list of physical response addresses on the functional group level or a communication parameter at each individual BaseVariant layer. In case of a list of physical response addresses at the functional group, no ECU base variants need to be defined in order to be able to perform functional communication (OBD use case). When interpreting a response, the response addresses on the base variant level are considered first. If none of these addresses matches, the list of response addresses stored on the functional group level is considered. Effectively, at runtime the superset of both lists is used (where duplicates have been removed).

If a response to a functional request is received which does neither match

⎯ the response template at the EcuVariant level, if variant identification and selection has taken place,

⎯ nor the response template defined on the base variant level,

⎯ nor the response template defined on the functional group level (mind the order),

an interpretation error will occur. In this case, the diagnostic server passes the erroneous response PDU to the application along with the error. Please note: Every response to a functional request can be defined as multi-part in ODX. In this case, there can be several responses per physical ECU to a functional request.

Example for implementation with D-PDU API

To implement the functionality of a list of physical response addresses in the context of ODX data definitions and the D-PDU API specification, a complex communication parameter called *CP_UniqueRespIdTable* has been introduced by ODX (the equivalent concept is called URID Table in the D-PDU API specification ISO 22900-2). The purpose of this communication parameter is to provide a table that allows the diagnostic server to map the ECU responses to functional requests to their physical source ECUs. Within the *CP_UniqueRespIdTable* communication parameter, the parameter *CP_ECULayerShortName* is also of special importance. The value of the communication parameter *CP_ECULayerShortName* is used to reference the diagnostic layer a response to a functional request belongs to. Therefore, a diagnostic server implementation has to be aware of the special semantics of these communication parameters *CP_ECULayerShortName* and *CP_ECULayerShortName* as allow for a protocol-independent implementation of functional addressing. In principle, functional addressing works as follows: When asked to execute a functional service, the diagnostic server composes the unique response id table according to the rules defined in 7.4.9.4 (Sequence of Events for Functional Addressing) in ISO 22901-1. Basically, the dagnostic server creates a table entry for each ECU base variant that is part of the functional group in question and assigns a unique identifier to each of the entries. Then, the diagnostic server passes the unique response id table to the D-PDU API implementation alongside the request to execute the functional service in question. The D-PDU API sends the functional request and matches each ECU response to the unique response id table. If a match is found (i.e. the sender of a response to the functional request is identified to be an entry in the unique response id table), the D-PDU API tags that match with the unique identifier of the answering ECU. This then allows the diagnostic server to relate each response to a specific ECU from the functional group and use this ECU's data for interpretation of the response PDU. For more detail and information about dealing with deviations from the simplified procedure described above, please see the relevant sections of the ODX and D-PDU API specification ISO 22901-1, ISO 22900-2).

8.12 Tables

8.12.1 General

Tables have been introduced in ODX to support the concept of data identifiers and parameter identifiers. This concept describes the association of a data structure definition to a unique identifier. Thus, tables are used to describe e.g. lists of:

— Measurement values which can be read by the same (set of) DiagComPrimitive(s)

— Actuator values which can be written by the same (set of) DiagComPrimitive(s)

Tables shall be browseable in the database part of diagnostic server independently of a specific DiagComPrimitive, that means, without having a `MCDDbDiagComPrimitive` selected. Therefore the interfaces `MCDDbTable(s)` and `MCDDbTableParameter(s)` have to be used (see Figure 96).

Each ODX table is represented by a `MCDDbTable` within a diagnostic server. All tables defined in the database for a certain location can be obtained by calling `MCDDbLocation::getDbTables()`. A subset of this collection will be delivered by `MCDDbLocation::getDbTablesBySemanticAttribute(A_ASCIISTRING semantic)` which returns all tables of the given semantic.
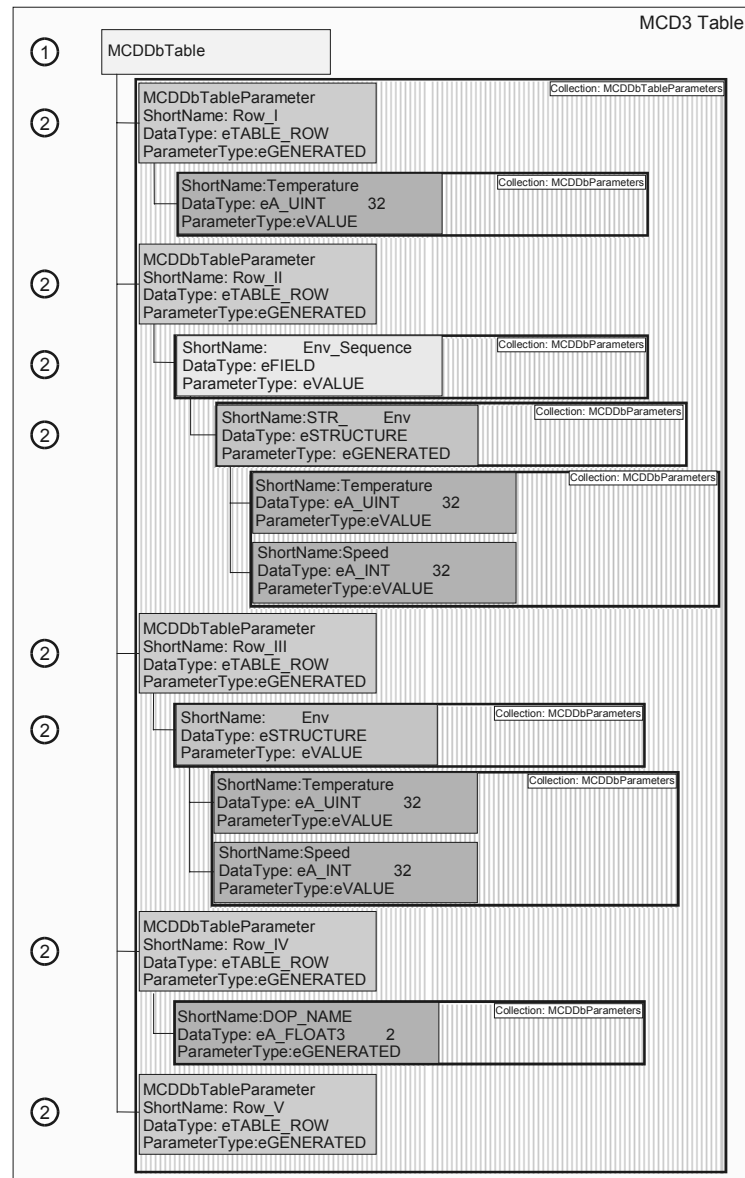
Key

1   getDbTableRows

2   getDbParameters

Figure 96 — Browse through a MCDDbTable

ODX-Data (Extract) of the DB-Template

```xml
<?xml version="1.0" encoding="UTF-8"?>
<ODX    MODEL-VERSION="2.2.0"    xsi:noNamespaceSchemaLocation="odx.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <DIAG-LAYER-CONTAINER ID="DLC_BV_BV1">
        <SHORT-NAME>DLC_BV_BV1</SHORT-NAME>
        <LONG-NAME>BV1</LONG-NAME>
        <BASE-VARIANTS>
            <BASE-VARIANT ID="BV_BV1">
                <SHORT-NAME>BV_BV1</SHORT-NAME>
```

```
                    <LONG-NAME>BV1</LONG-NAME>
                    <DIAG-DATA-DICTIONARY-SPEC>
                        <DATA-OBJECT-PROPS>
                            <DATA-OBJECT-PROP
ID="BV_BV1.DOP_TexttKeyDOP.DATA- OBJECT-PROP">
                                <SHORT-NAME>DOP_TexttKeyDOP</SHORT-NAME>
                                <LONG-NAME>Texttable_KeyDOP</LONG-NAME>
                                <COMPU-METHOD>
                                    <CATEGORY>TEXTTABLE</CATEGORY>
                                    <COMPU-INTERNAL-TO-PHYS>
                                        <COMPU-SCALES>
                                            <COMPU-SCALE>
                                                <LOWER-LIMIT>1</LOWER-LIMIT>
                                                <COMPU-CONST>
                                                    <VT>ONE</VT>
                                                </COMPU-CONST>
                                            </COMPU-SCALE>
                                            <COMPU-SCALE>
                                                <LOWER-LIMIT>2</LOWER-LIMIT>
                                                <COMPU-CONST>
                                                    <VT>TWO</VT>
                                                </COMPU-CONST>
                                            </COMPU-SCALE>
                                            <COMPU-SCALE>
                                                <LOWER-LIMIT>3</LOWER-LIMIT>
                                                <COMPU-CONST>
                                                    <VT>THREE</VT>
                                                </COMPU-CONST>
                                            </COMPU-SCALE>
                                            <COMPU-SCALE>
                                                <LOWER-LIMIT>4</LOWER-LIMIT>
                                                <COMPU-CONST>
                                                    <VT>FOUR</VT>
                                                </COMPU-CONST>
                                            </COMPU-SCALE>
                                            <COMPU-SCALE>
                                                <LOWER-LIMIT>5</LOWER-LIMIT>
                                                <COMPU-CONST>
                                                    <VT>FIVE</VT>
                                                </COMPU-CONST>
                                            </COMPU-SCALE>
                                        </COMPU-SCALES>
                                    </COMPU-INTERNAL-TO-PHYS>
                                </COMPU-METHOD>
                                <DIAG-CODED-TYPE     BASE-DATA-TYPE="A_UINT32"
xsi:type="STANDARD-LENGTH-TYPE">
                                    <BIT-LENGTH>8</BIT-LENGTH>
                                </DIAG-CODED-TYPE>
                                <PHYSICAL-TYPE                      BASE-DATA-
TYPE="A_UNICODE2STRING"/>
                            </DATA-OBJECT-PROP>
                            <DATA-OBJECT-PROP
ID="BV_BV1.DOP_IdentUInt32.DATA-OBJECT-PROP">
                                <SHORT-NAME>DOP_IdentUInt32</SHORT-NAME>
```

```
                              <LONG-NAME>Identical_UInt32</LONG-NAME>
                              <COMPU-METHOD>
                                  <CATEGORY>IDENTICAL</CATEGORY>
                              </COMPU-METHOD>
                              <DIAG-CODED-TYPE      BASE-DATA-TYPE="A_UINT32"
xsi:type="STANDARD-LENGTH-TYPE">
                                  <BIT-LENGTH>32</BIT-LENGTH>
                              </DIAG-CODED-TYPE>
                              <PHYSICAL-TYPE BASE-DATA-TYPE="A_UINT32"/>
                          </DATA-OBJECT-PROP>
                          <DATA-OBJECT-PROP   ID="BV_BV1.DOP_IdentInt32.DATA-
OBJECT-PROP">
                              <SHORT-NAME>DOP_IdentInt32</SHORT-NAME>
                              <LONG-NAME>Identical_Int32</LONG-NAME>
                              <COMPU-METHOD>
                                  <CATEGORY>IDENTICAL</CATEGORY>
                              </COMPU-METHOD>
                              <DIAG-CODED-TYPE      BASE-DATA-TYPE="A_INT32"
xsi:type="STANDARD-LENGTH-TYPE">
                                  <BIT-LENGTH>32</BIT-LENGTH>
                              </DIAG-CODED-TYPE>
                              <PHYSICAL-TYPE BASE-DATA-TYPE="A_INT32"/>
                          </DATA-OBJECT-PROP>
                          <DATA-OBJECT-PROP        ID="BV_BV1.DOP_NAME.DATA-
OBJECT-PROP">
                              <SHORT-NAME>DOP_NAME</SHORT-NAME>
                              <LONG-NAME>Identical_Float32</LONG-NAME>
                              <COMPU-METHOD>
                                  <CATEGORY>IDENTICAL</CATEGORY>
                              </COMPU-METHOD>
                              <DIAG-CODED-TYPE    BASE-DATA-TYPE="A_FLOAT32"
xsi:type="STANDARD-LENGTH-TYPE">
                                  <BIT-LENGTH>32</BIT-LENGTH>
                              </DIAG-CODED-TYPE>
                              <PHYSICAL-TYPE BASE-DATA-TYPE="A_FLOAT32"/>
                          </DATA-OBJECT-PROP>
                      </DATA-OBJECT-PROPS>
                      <STRUCTURES>
                          <STRUCTURE
ID="BV_BV1.Example_STRUCT_A.STRUCTURE">
                              <SHORT-NAME>Example_STRUCT_A</SHORT-NAME>
                              <LONG-NAME>Example_STRUCT_A</LONG-NAME>
                              <PARAMS>
                                  <PARAM xsi:type="VALUE">
                                      <SHORT-NAME>Temperature</SHORT-NAME>
                                      <LONG-NAME>Temperature</LONG-NAME>
                                      <BYTE-POSITION>0</BYTE-POSITION>
                                      <BIT-POSITION>0</BIT-POSITION>
                                      <DOP-SNREF                    SHORT-
NAME="DOP_IdentUInt32"/>
                                  </PARAM>
                              </PARAMS>
                          </STRUCTURE>
                          <STRUCTURE ID="BV_BV1.STR_Env.STRUCTURE">
                              <SHORT-NAME>STR_Env</SHORT-NAME>
```

```
                            <LONG-NAME>STR_Env</LONG-NAME>
                            <PARAMS>
                                <PARAM xsi:type="VALUE">
                                    <SHORT-NAME>Temperature</SHORT-NAME>
                                    <LONG-NAME>Temperature</LONG-NAME>
                                    <BYTE-POSITION>0</BYTE-POSITION>
                                    <BIT-POSITION>0</BIT-POSITION>
                                    <DOP-SNREF                          SHORT-
NAME="DOP_IdentUInt32"/>
                                </PARAM>
                                <PARAM xsi:type="VALUE">
                                    <SHORT-NAME>Speed</SHORT-NAME>
                                    <LONG-NAME>Speed</LONG-NAME>
                                    <BYTE-POSITION>4</BYTE-POSITION>
                                    <BIT-POSITION>0</BIT-POSITION>
                                    <DOP-SNREF                          SHORT-
NAME="DOP_IdentInt32"/>
                                </PARAM>
                            </PARAMS>
                        </STRUCTURE>
                        <STRUCTURE
ID="BV_BV1.Example_STRUCT_B.STRUCTURE">
                            <SHORT-NAME>Example_STRUCT_B</SHORT-NAME>
                            <LONG-NAME>Example_STRUCT_B</LONG-NAME>
                            <PARAMS>
                                <PARAM xsi:type="VALUE">
                                    <SHORT-NAME>Env_Sequence</SHORT-NAME>
                                    <LONG-NAME>Env_Sequence</LONG-NAME>
                                    <BYTE-POSITION>0</BYTE-POSITION>
                                    <BIT-POSITION>0</BIT-POSITION>
                                    <DOP-SNREF SHORT-NAME="EOPF_Field"/>
                                </PARAM>
                            </PARAMS>
                        </STRUCTURE>
                        <STRUCTURE
ID="BV_BV1.Example_STRUCT_C.STRUCTURE">
                            <SHORT-NAME>Example_STRUCT_C</SHORT-NAME>
                            <LONG-NAME>Example_STRUCT_C</LONG-NAME>
                            <PARAMS>
                                <PARAM xsi:type="VALUE">
                                    <SHORT-NAME>Env</SHORT-NAME>
                                    <LONG-NAME>Env</LONG-NAME>
                                    <BYTE-POSITION>0</BYTE-POSITION>
                                    <BIT-POSITION>0</BIT-POSITION>
                                    <DOP-SNREF SHORT-NAME="STR_Env"/>
                                </PARAM>
                            </PARAMS>
                        </STRUCTURE>
                    </STRUCTURES>
                    <END-OF-PDU-FIELDS>
                        <END-OF-PDU-FIELD    ID="BV_BV1.EOPF_Field.END-OF-
PDU-FIELD">
                            <SHORT-NAME>EOPF_Field</SHORT-NAME>
                            <LONG-NAME>Field</LONG-NAME>
```

```
                              <BASIC-STRUCTURE-REF                    ID-
REF="BV_BV1.STR_Env.STRUCTURE"/>
                        </END-OF-PDU-FIELD>
                  </END-OF-PDU-FIELDS>
                  <TABLES>
                        <TABLE ID="BV_BV1.TAB_Table.TABLE">
                              <SHORT-NAME>TAB_Table</SHORT-NAME>
                              <LONG-NAME>Table1</LONG-NAME>
                              <KEY-DOP-REF                            ID-
REF="BV_BV1.DOP_TexttKeyDOP.DATA-OBJECT-PROP"/>
                              <TABLE-ROW    ID="BV_BV1.TAB_Table.ROW_I.TABLE-
ROW">
                                    <SHORT-NAME>ROW_I</SHORT-NAME>
                                    <LONG-NAME>ONE</LONG-NAME>
                                    <KEY>ONE</KEY>
                                    <STRUCTURE-REF                    ID-
REF="BV_BV1.Example_STRUCT_A.STRUCTURE"/>
                              </TABLE-ROW>
                              <TABLE-ROW    ID="BV_BV1.TAB_Table.ROW_II.TABLE-
ROW">
                                    <SHORT-NAME>ROW_II</SHORT-NAME>
                                    <LONG-NAME>TWO</LONG-NAME>
                                    <KEY>TWO</KEY>
                                    <STRUCTURE-REF                    ID-
REF="BV_BV1.Example_STRUCT_B.STRUCTURE"/>
                              </TABLE-ROW>
                              <TABLE-ROW
ID="BV_BV1.TAB_Table.ROW_III.TABLE-ROW">
                                    <SHORT-NAME>ROW_III</SHORT-NAME>
                                    <LONG-NAME>THREE</LONG-NAME>
                                    <KEY>THREE</KEY>
                                    <STRUCTURE-REF                    ID-
REF="BV_BV1.Example_STRUCT_C.STRUCTURE"/>
                              </TABLE-ROW>
                              <TABLE-ROW    ID="BV_BV1.TAB_Table.ROW_IV.TABLE-
ROW">
                                    <SHORT-NAME>ROW_IV</SHORT-NAME>
                                    <LONG-NAME>FOUR</LONG-NAME>
                                    <KEY>FOUR</KEY>
                                    <DATA-OBJECT-PROP-REF             ID-
REF="BV_BV1.DOP_IdentUINT32.DATA-OBJECT-PROP"/>
                              </TABLE-ROW>
                              <TABLE-ROW    ID="BV_BV1.TAB_Table.ROW_V.TABLE-
ROW">
                                    <SHORT-NAME>ROW_V</SHORT-NAME>
                                    <LONG-NAME>FIVE</LONG-NAME>
                                    <KEY>FIVE</KEY>
                              </TABLE-ROW>
                        </TABLE>
                  </TABLES>
            </DIAG-DATA-DICTIONARY-SPEC>
            <DIAG-COMMS>
                  <DIAG-SERVICE ID="BV_BV1.DS_ExampServi.DIAG-SERVICE">
                        <SHORT-NAME>DS_ExampServi</SHORT-NAME>
                        <LONG-NAME>Example_Service</LONG-NAME>
```

```
                        <AUDIENCE/>
                        <REQUEST-REF                            ID-
REF="BV_BV1.REQ_ExampServi.REQUEST"/>
                        <POS-RESPONSE-REFS>
                            <POS-RESPONSE-REF                   ID-
REF="BV_BV1.PRE_ExampServi.POS-RESPONSE"/>
                        </POS-RESPONSE-REFS>
                    </DIAG-SERVICE>
                </DIAG-COMMS>
                <REQUESTS>
                    <REQUEST ID="BV_BV1.REQ_ExampServi.REQUEST">
                        <SHORT-NAME>REQ_ExampServi</SHORT-NAME>
                        <LONG-NAME>Example_Service</LONG-NAME>
                        <PARAMS>
                            <PARAM                  xsi:type="CODED-CONST"
SEMANTIC="SERVICE-ID">
                                <SHORT-NAME>ServiID</SHORT-NAME>
                                <LONG-NAME>Service ID</LONG-NAME>
                                <BYTE-POSITION>0</BYTE-POSITION>
                                <BIT-POSITION>0</BIT-POSITION>
                                <CODED-VALUE>25</CODED-VALUE>
                                <DIAG-CODED-TYPE       xsi:type="STANDARD-
LENGTH-TYPE" BASE-DATA-TYPE="A_UINT32" IS-HIGHLOW-BYTE-ORDER="true" BASE-
TYPE-ENCODING="NONE">
                                    <BIT-LENGTH>8</BIT-LENGTH>
                                </DIAG-CODED-TYPE>
                            </PARAM>
                            <PARAM
ID="BV_BV1.REQ_ExampServi.Example_KEY.PARAM" xsi:type="TABLE-KEY">
                                <SHORT-NAME>Example_KEY</SHORT-NAME>
                                <LONG-NAME>Table_Key</LONG-NAME>
                                <BYTE-POSITION>1</BYTE-POSITION>
                                <BIT-POSITION>0</BIT-POSITION>
                                <TABLE-REF                          ID-
REF="BV_BV1.TAB_Table.TABLE"/>
                            </PARAM>
                            <PARAM xsi:type="TABLE-STRUCT">
                                <SHORT-NAME>Param_TABLE_STRUCT</SHORT-
NAME>
                                <LONG-NAME>Table_Struct</LONG-NAME>
                                <BYTE-POSITION>2</BYTE-POSITION>
                                <BIT-POSITION>0</BIT-POSITION>
                                <TABLE-KEY-REF                      ID-
REF="BV_BV1.REQ_ExampServi.Example_KEY.PARAM"/>
                            </PARAM>
                        </PARAMS>
                    </REQUEST>
                </REQUESTS>
                <POS-RESPONSES>
                    <POS-RESPONSE          ID="BV_BV1.PRE_ExampServi.POS-
RESPONSE">
                        <SHORT-NAME>PRE_ExampServi</SHORT-NAME>
                        <LONG-NAME>Example_Service</LONG-NAME>
                        <PARAMS>
```

```
                             <PARAM                     xsi:type="CODED-CONST"
SEMANTIC="SERVICE-ID">
                                 <SHORT-NAME>ServiID</SHORT-NAME>
                                 <LONG-NAME>Service ID</LONG-NAME>
                                 <BYTE-POSITION>0</BYTE-POSITION>
                                 <BIT-POSITION>0</BIT-POSITION>
                                 <CODED-VALUE>89</CODED-VALUE>
                                 <DIAG-CODED-TYPE      xsi:type="STANDARD-
LENGTH-TYPE" BASE-DATA-TYPE="A_UINT32" IS-HIGHLOW-BYTE-ORDER="true" BASE-
TYPE-ENCODING="NONE">
                                     <BIT-LENGTH>8</BIT-LENGTH>
                                 </DIAG-CODED-TYPE>
                             </PARAM>
                             <PARAM
ID="BV_BV1.PRE_ExampServi.Example_KEY.PARAM" xsi:type="TABLE-KEY">
                                 <SHORT-NAME>Example_KEY</SHORT-NAME>
                                 <LONG-NAME>Table_Key</LONG-NAME>
                                 <BYTE-POSITION>1</BYTE-POSITION>
                                 <BIT-POSITION>0</BIT-POSITION>
                                 <TABLE-REF                               ID-
REF="BV_BV1.TAB_Table.TABLE" DOCREF="BV_BV1" DOCTYPE="LAYER"/>
                             </PARAM>
                             <PARAM xsi:type="TABLE-STRUCT">
                                 <SHORT-NAME>Param_TABLE_STRUCT</SHORT-
NAME>
                                 <LONG-NAME>Table_Struct</LONG-NAME>
                                 <BYTE-POSITION>2</BYTE-POSITION>
                                 <BIT-POSITION>0</BIT-POSITION>
                                 <TABLE-KEY-REF                          ID-
REF="BV_BV1.PRE_ExampServi.Example_KEY.PARAM"/>
                             </PARAM>
                         </PARAMS>
                     </POS-RESPONSE>
                 </POS-RESPONSES>
             </BASE-VARIANT>
         </BASE-VARIANTS>
     </DIAG-LAYER-CONTAINER>
</ODX>
```

In ODX a table is composed of a non-empty set of TABLE-ROWs. Each TABLE-ROW references a STRUCTURE or a simple DOP and can be uniquely identified by a key. All table-rows of a `MCDDbTable` will be delivered by `MCDDbTable::getDbTableRows()`.

In diagnostic server the interface `MCDDbTableParameter` is used to represent a table-row of an ODX table or a parameter in the structure referenced from a table-row, respectively. That is, it represents the ODX elements TABLE-ROW and STRUCTURE where the STRUCTURE is referenced from the TABLE-ROW.

In case of representing a TABLE-ROW, the `MCDDbTableParameter` is of data type eTABLE_ROW. Calling the method `MCDDbTableParameter::getKey()` delivers the value of the key associated with the `MCDDbTableParameter` in the corresponding `MCDDbTable`. In all other cases (representing the referenced STRUCTURE or one of its elements), the general diagnostic server mapping of ODX-Elements to `MCDDataTypes` is applied.

8.12.2 Usage of tables within DiagComPrimitives

For the usage of tables within requests and responses of DiagComPrimitives the parameter types TABLE-STRUCT, TABLE-KEY and TABLE-ENTRY are defined in ODX.

The principle is similar to a multiplexer. The content of a TABLE-STRUCT parameter depends on a certain switch value. This switch value is given by the TABLE-KEY parameter referenced by the TABLE-STRUCT parameter.[1]

More detailed, a parameter of type TABLE-KEY is the part of the PDU where the KEY (data identifier) has to be read in the case of a response or written in the case of a request. A TABLE-KEY parameter selects a TABLE-ROW of a TABLE. The selection can be done by referencing a TABLE-ROW directly (static parameter definition) or by referencing a TABLE and selecting a TABLE-ROW at runtime by using the current value of the TABLE-KEY to match it against the unique KEYs of that TABLE (dynamic parameter definition). The content of a TABLE-STRUCT parameter is the content of the STRUCTURE or simple DOP referenced by the selected TABLE-ROW.

If a TABLE-ROW is selected, which is not executable the Diag Service is also not executable. This does not change the visibility of the Diag Service. Audiences have no influence on visibility or the possibility of execution.

Inside a Job IsExecutable has no influence.

TABLE-KEY, TABLE-STRUCT and TABLE-ENTRY parameters are represented within diagnostic server as `MCD(Db)Parameter` objects of parameter type `eTABLE_KEY`, `eTABLE_STRUCT` and `eTABLE_ENTRY`.

To query all keys that can be read (`MCDResponseParameter`) or written (`MCDRequestParameter`) at runtime by a parameter of type `eTABLE_KEY` the method `MCDDbParamter::getKeys()` can be used at the corresponding `MCDDbParameter` object. Please note that in case of static parameter definition the returned collection contains exactly one key – the key of the statically referenced TABLE-ROW.

If a client needs to know which `eTABLE_STRUCT` parameters depend on a certain `eTABLE_KEY` parameter within one request or response, it can call `MCDDbParameter::getDbTableStructParams()` for the corresponding `eTABLE_KEY` parameter. Vice versa, calling `MCDDbParameter::getDbTableKeyParam()` on a `eTABLE_STRUCT` parameter, will deliver the referenced `eTABLE_KEY` parameter.

As stated before, the decomposition of a `eTABLE_STRUCT` parameter into further parameters represents exactly one table row of the table referenced by the corresponding `eTABLE_KEY` parameter. To get the `MCDDbTable` the parameter structure of an `eTABLE_STRUCT` parameter taken from `MCDDbParameter::getDbTable()` can be used. Please note that in case of a static parameter definition, the decomposition can already be obtained at a `eTABLE_STRUCT` parameter by calling the method `MCDDbParameter::getDbParameters()`. In case of a dynamic parameter definition, the collection returned by the method `MCDDbParameter::getDbParameters()` is empty.

If a `MCDDbTableParameter` with datatype `eTABLE_ROW` references a simple DOP in ODX, the resulting `MCDDbParameters` collection delivered by `MCDDbTableParameter::getDbParameters()` contains exactly one `MCDDbTableParameter`. The `MCDDbTableParameter`'s parameter type is `eGENERATED`. Its data type and ShortName are obtained from the referenced simple DOP

The structure of a `MCDDbParameter` with parameter type `eTABLE_ENTRY` depends on the value of the TARGET attribute of the TABLE-ENTRY parameter in ODX:

---

[1] In ODX, it is allowed that several parameters of type TABLE-STRUCT reference the same parameter of type TABLE-KEY. The only restriction is that these parameters need to be located on the same level in the parameter hierarchy.

— TARGET = KEY: The `MCDDbParameter` has the datatype of the KEY-DOP of the TABLE-ROW that is referenced from the TABLE-ENTRY parameter.

— TARGET = STRUCT: The parameter has the datatype `eSTRUCTURE`. The containing `MCDDbParameters`, delivered by method `MCDDbTableParameter::getDbParameters()`, are the parameters that are contained at the referenced TABLE-ROW structure of the TABLE-ENTRY parameter.
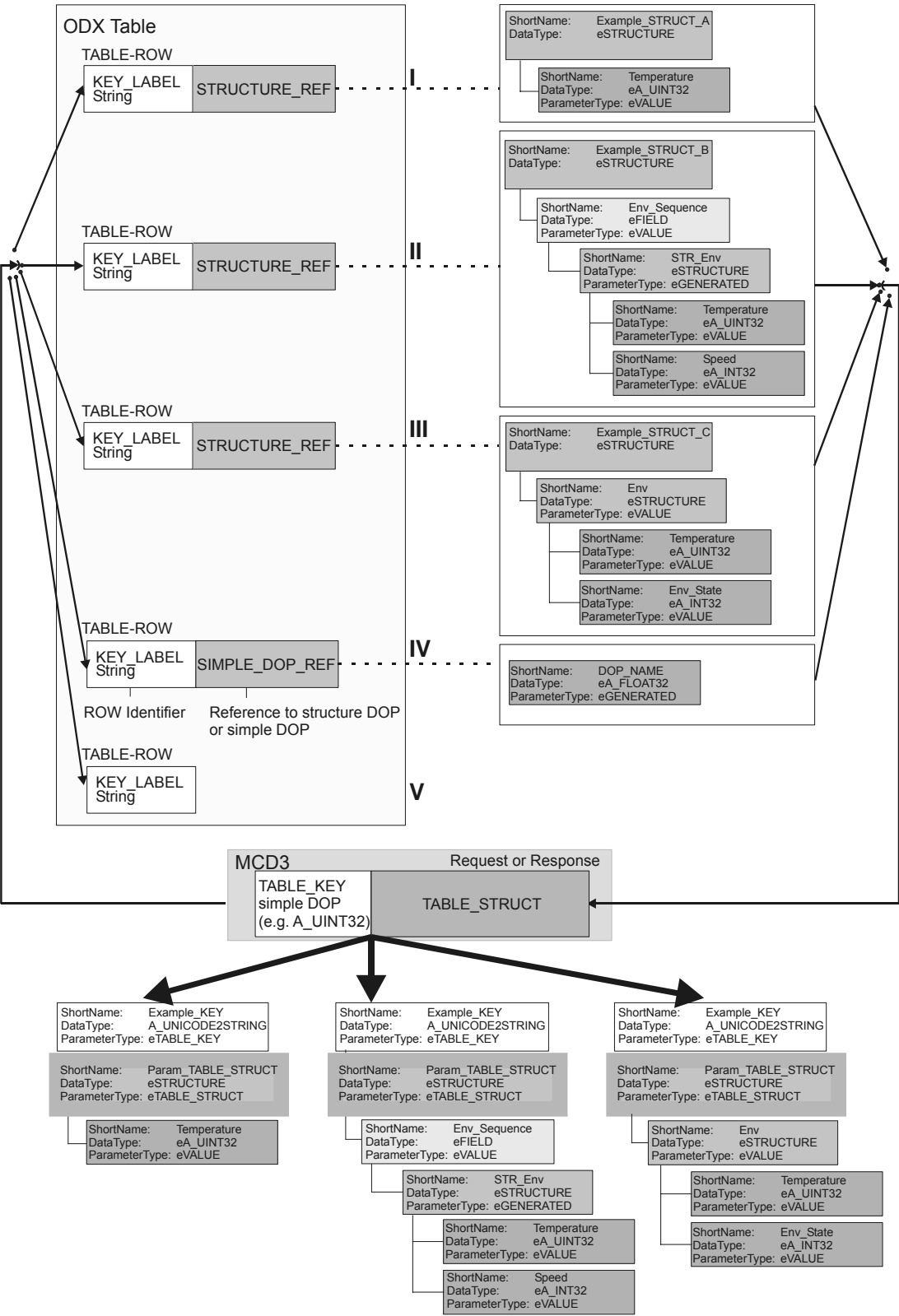
Figure 97 depicts the example of a table ROW I, ROW II, ROW III.

Figure 97 — Example of a table ROW I, ROW II, ROW III

Figure 98 depicts the usage of ODX Tables.

Figure 98 — Usage of ODX Tables

8.13 Dynamically Defined Identifiers (DynId)

8.13.1 General

Various diagnostic application layer protocols (e.g. UDS or KWP2000) incorporate the concept of dynamically defined diagnostic services, i.e. services where the user can define the response's contents dynamically at runtime. This chapter describes how the concept of dynamically defined services is handled in the MVCI diagnostic server API. Support for dynamically defined identifiers is an optional part of this standard.

8.13.2 DYNID principle and requirements

To use a dynamically defined identifier, a diagnostic application first has to declare the identifier definition to an ECU. This means that a service will be sent to the ECU, which associates a certain dynamic identifier with a set of values that should be returned on future requests with that identifier. After definition of a dynamic identifier, the diagnostic application can then use a read-dynamically-defined-identifier service to retrieve the set of values that was associated with the dynamic identifier. When a diagnostic application does not need a dynamically defined identifier anymore, a dynamic identifier definition can be revoked by sending a clear-dynamically-defined-identifier service, which tells the ECU to delete a dynamic service definition with the identifier. In ODX, these three services are marked by a specific diagnostic class:

— `DYN-DEF-MESSAGE` for definition of new dynamic services

— `READ-DYN-DEF-MESSAGE` for reading the contents of a previously defined dynamic service and

— `CLEAR-DYN-DEF-MESSAGE` for clearing a previously defined dynamic service definition

Although the usage principle is the same in all cases, in the context of ODX data it is distinguished between fully dynamic, semi dynamic and static DynID. These three cases differ only in how certain parameters concerning the DYN-DEF-MESSAGE service are set up in ODX:

— For the fully dynamic case, the application is entirely free to chose the contents of a dynamic service.

— In the semi dynamic case, a part of the dynamic service's response signature is pre-defined by `CODED-CONST` parameters in ODX. However, the application is allowed to add more parameters to that dynamic service. Here, the client application can use the `READ-DYN-DEF-MESSAGE` service like a normal diagnostic service, without first having to explicitly define the contents of the specific dynamic service id. However, the client application still has to create the appropriate DynID definition service for that service id, and execute it using its default parameters.

— In the static case, all response parameters for a certain dynamic service definition are pre-set by `CODED-CONST` parameters in ODX. The application is not allowed to add more parameters to a dynamic service definition. Here, the client application uses the `READ-DYN-DEF-MESSAGE` service like a normal diagnostic service, it is not allowed to change the contents of the specific dynamic service id beforehand. However, the client application still has to create the appropriate DynID definition service for that service id, and execute it using its default parameters.

The steps a diagnostic application has to follow when using dynamically defined services are outlined below:

a) Creation of a dynamic ID

— Use a DiagComPrimitive for DynID definition (`DYN-DEF-MESSAGE in ODX`)

— Get supported and available dynamic identifiers (`MCDDbLocation::getSupportedDynIds`)

— Get a set of parameters for parameterisation of a dynamic service, set relevant parts of that parameter structure

— Execute the DynID definition service

b) Reading by dynamic ID

— Use a DiagComPrimitive to read by DynID (`READ-DYN-DEF-MESSAGE` in ODX)

— The MVCI diagnostic server knows through previous DynID-definition how to interpret DynID reading-service results

c) Deletion of a dynamic ID

— Use a DiagComPrimitive for DynID deletion (`CLEAR-DYN-DEF-MESSAGE` in ODX)

8.13.3 Lifecycle

8.13.3.1 General

The three steps that have to be executed during a diagnostic session that uses dynamically defined identifiers are outlined in more detail in this chapter.

8.13.3.2 Creation of dynamically defined identifier

Before a dynamic service can be used by an application (`MCDDynIdReadComPrimitive`), it has to be defined first by using the `MCDDynIdDefineComPrimitive`.

In ODX, a specific location (EcuBaseVariant or EcuVariant) defining a `DYN-DEFINED-SPEC` can contain an arbitrary number of DiagComPrimitives with a diagnostic class of `CLEAR-DYN-DEF-MESSAGE`, `READ-DYN-DEF-MESSAGE`, or `DYN-DEF-MESSAGE`. That means that these ComPrimitives are not unique within a location. However, each location may only contain one `DYN-DEFINED-SPEC` with a certain definition mode (ODX: DEF-MODE), and for each definition mode there can only be one associated `CLEAR-DYN-DEF-MESSAGE`, `READ-DYN-DEF-MESSAGE`, or `DYN-DEF-MESSAGE` service. This means that the combination of a location and a `DYN-DEFINED-SPEC` with a certain definition mode will result in a unique set of DiagComPrimitives for defining, reading and clearing of a dynamically defined identifier.

To create the default `MCDDynIdDefineComPrimitive`, `MCDDynIdReadComPrimitive` and `MCDDynIdClearComPrimitive` for a selected `DYN-DEFINED-SPEC`, the method `MCDLogicalLink::createDynIdComPrimitiveByTypeAndDefinitionMode(MCDObjectType type, A_ASCIISTRING definitionMode)` shall be used, using one of the unique `MCDObjectType` values

— `eMCDDYNIDDEFINECOMPRIMITIVE`,

— `eMCDDYNIDREADCOMPRIMITIVE` or

— `eMCDDYNIDCLEARCOMPRIMITIVE`,

in combination with a definition mode (`DEF-MODE`) like

— `DATA-ID`

— `COMMON-ID`,

— `LOCAL-ID` or

— `ADDRESS`.

Please note that the list of valid definition modes can be extended in ODX. Only the values DATA-ID, COMMON-ID, LOCAL-ID, and ADDRESS are predefined. The list of definition modes available for a certain DynID DiagComPrimitive can be obtained by using the method `MCDDbDynIdxxxComPrimitive::getDefinitionModes()`.

As described above, only the combination of a definition mode and a DynID-related MCDObjectType uniquely identifies a DiagComPrimitive for fully dynamic identifier within a DYN-DEFINED-SPEC in ODX.

Please note that if methods `execueSync()` or `executeAsync()` are called for a `MCDDynIdxxxComPrimitive` with no definition mode or if no valid DynID is set (see below) a `MCDParameterizationException` is thrown with error code `ePAR_INCOMPLETE_PARAMETERIZATION`.

`MCDDbLocation::getSupportedDynIds(A_ASCIISTRING definitionMode)` returns the list of DynIDs supported at this location for a given definition mode. This list contains all supported DynIDs (e. g. 0xF0 … 0xF9 in ISO 14230-3 KWP 2000) regardless of the fact whether a DynID has already been assigned to a dynamic service or not.

NOTE   Supported DynIDs are dependent on location and definition mode.

For every DynID that is actually to be defined at runtime, a `MCDDynIdDefineComPrimitive` has to be executed. The actual DynID to be used is set at the different DiagComPrimitives (`MCDDynIdDefineComPrimitive`, `MCDDynIdReadComPrimitive`, `MCDDynIdClearComPrimitive`) by using the method `setDynId`. The `setDynId`-methods are used to parameterize a `MCDDynId...ComPrimitive` with a specific Id (e.g., 0xF0 … 0xF9 in ISO 14230-3 KWP 2000).
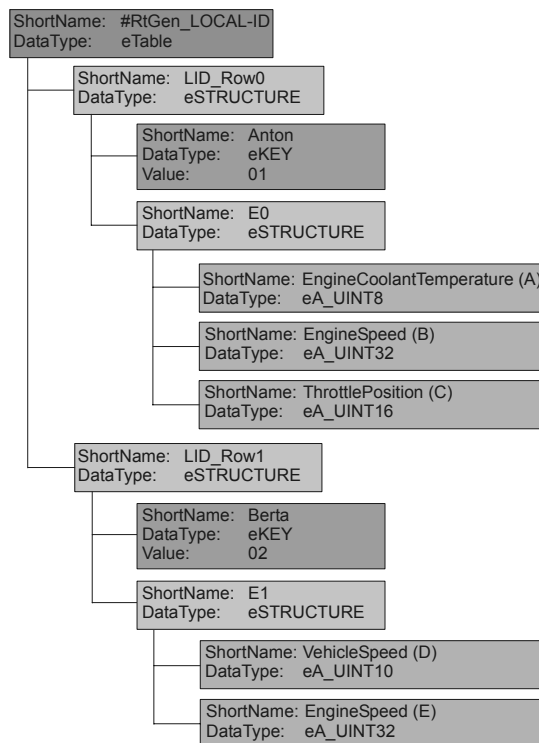
The method `MCDDynIdDefineComPrimitive::setDynIdParams(MCDDatatypeAsciiStrings paramNames)` is used to add a collection of parameters which should be part of the newly defined dynamic service. The parameters are selected from the respective ODX tables by using positive filter expressions (see `setDynIdParams()` below).

In case of a fully dynamic `MCDDynIdDefineComPrimitive`, all parameters have to be set by using this method. In case of a semi-dynamic `MCDDynIdDefineComPrimitive`, only the dynamic part of the parameter set can be defined by this method, i.e. the parameters are appended to the static part of the dynamic service. In case of a static `MCDDynIdDefineComPrimitive`, a `MCDParameterizationException` with error code `ePAR_INCONSISTENT_VALUE_LIST` is thrown when the client application tries to set DynID parameters. All DynID parameters used in a static, semi-dynamic or dynamic DynIdComPrimitive have to be located in the tables obtainable by `MCDDbLocation::getDbTableByDefinitionMode(A_ASCIISTRING definitionMode)`.

The returned table is generated with the name "#RtGen_<definitionMode>". As in ODX there may be multiple Tables for the same definition mode the rows of these tables are merged. To be unique, which is required inside one table, the short name is extended with the table short name. The naming rule is "<table shortname>_<row shortname>".

Figure 99 depicts the example for definition Mode LOCAL-ID.

Figure 99 — Example for definition Mode LOCAL-ID

When a `MCDDynIdDefineComPrimitive` service is first created, the dynamic part of the request template is empty. It is filled by means of the method `setDynIdParams(MCDDatatypeAsciiStrings paramNames)`. An empty collection object of type MCDDatatypeAsciiStrings can be obtained by executing the method `MCDDataPrimitive::getNewAsciiStrings():MCDDatatypeAsciiStrings`.

Every entry in the collection of filter keys has the format

< Table Name> |

< Row Name > |

<eSTRUCTURE Shortname>|

<ShortName>|…|<ShortName> (path to the element).

If the `MCDDynIdDefineComPrimitive` has no valid definition mode set, calling method `setDynIdParams(MCDDatatypeAsciiStrings paramNames)` will throw a MCDParameterizationException with error code `ePAR_INCOMPLETE_PARAMETERIZATION`.

In addition, this method should throw a `MCDProgramViolationException` with error code `eRT_NO_UNIQUE_ELEMENT` if there are duplicate filter keys in the set of filter keys supplied as parameter to the method. Please note that is not necessary to name the <eKEY> parameter shortname, because eKey Shortname and eSTRUCTURE Shortname are a unit for each row.
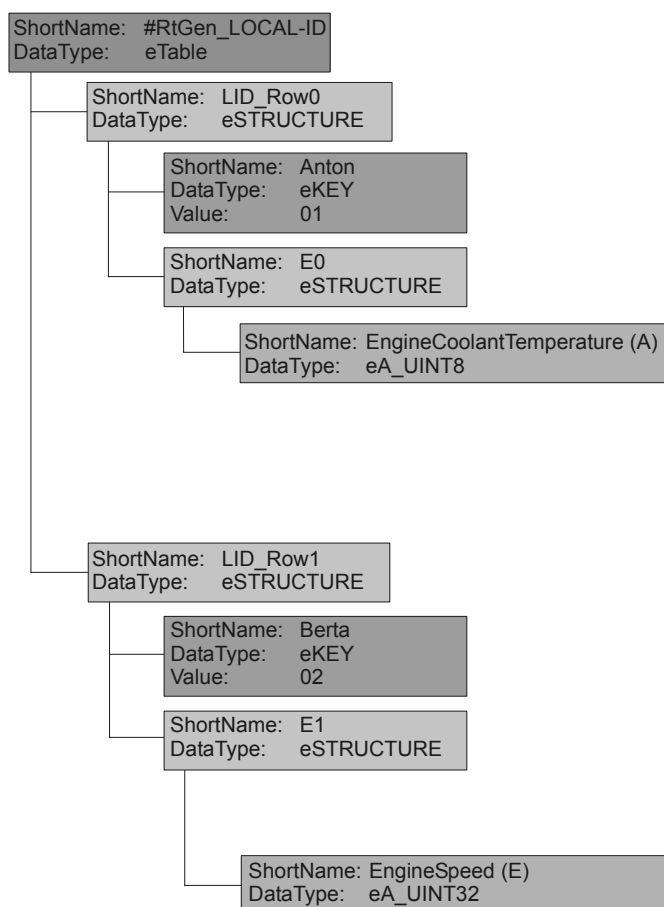
This shall be demonstrated with help of the Database Template Figure 99.

The filter keys are declared as unstructured Collection of elements. At this, one element is given as a sequence of DbShortNames each separated by a | (ASCII 124), starting from the root. Simplifications are achieved as in a positive case all elements contained below the Parameter are taken over automatically. The main structure has to be retained in any case! This means, that in case of naming E1 all elements of this structure (D + E) are used.

Filter Keys are created by giving the elements to be used. It is done in the way, that all Response Parameter to be used are stated completely. On the left side you see the filter keys which should be used and on the right side the Collection of sequences of DbShortnames, which is used to create the filter keys from the original Database template. Every picture shows a new example.
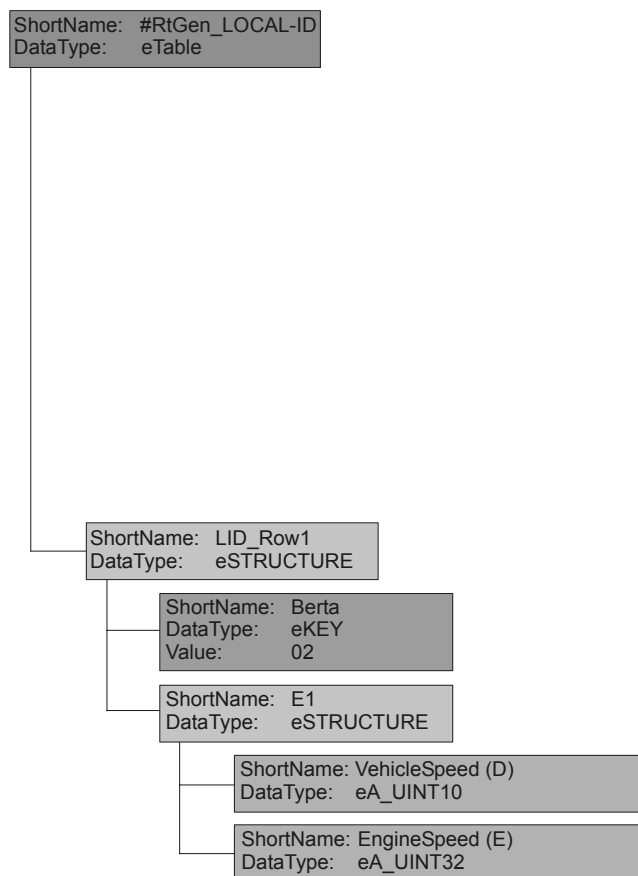
Figure 100 depicts the example, which uses the following multiple filter keys in the collection:

— #RtGen_LOCAL-ID | LID_Row0 | E0 | EngineCoolantTempearture (A)

— #RtGen_LOCAL-ID | LID_Row1 | E1 | EngineSpeed (E)

Figure 100 — Multiple filter key

Figure 101 depicts the example, which uses the following filter key in the collection:
#RtGen_LOCAL-ID | LID_Row1 | E1

Figure 101 — Single filter key

In ODX the `DYN-DEFINED-SPEC` references all Tables which can be used for creation of new DynID records. As well as the DynID DiagComPrimitives the Tables are grouped by the definition mode. For each filter key the MVCI diagnostic server tries to find the TableRow that fits to the <KEY> given through the filter. Therefore the MVCI diagnostic server looks up only Tables selected by the current definition mode of the `MCDDynIdDefineComPrimitive`. In case the MVCI diagnostic server finds a fitting TableRow it uses the further part of the filter key to select a certain parameter within the eSTRUCTURE referenced by that TableRow. The runtime system has the protocol specific knowledge to prepare the request structure at runtime for the `MCDDynIdDefineComPrimitive` using the parameters selected by `setDynIdParams(paramNames)`. This structure is also used to create the response structure at runtime for the corresponding `MCDDynIdReadComPrimitive`. The sequence of parameters will have the same order as the parameters added by this method. For each Logical Link, the MVCI diagnostic server should internally maintain a list which dynamic Ids are defined and which response structure is bound to a certain dynamic service Id.

If a value (`MCDValue`) is passed to a `setDynId` method which is not in the list of supported dynamic Ids, which are obtainable by calling the method `MCDLogicalLink::getDefinableDynIds()`, a `MCDParameterizationException` exception with error code `ePAR_VALUE_OUT_OF_RANGE` is thrown.

The method `setDynId` is used to assign a DynID to a DynIdComPrimitve, regardless of the fact whether a static, a semi-dynamic, or a dynamic DynId service is to be used. If a DynID parameter that the client application tries to change is defined as `CODED-CONST` in ODX, this method should throw a `MCDProgramViolationException` with error code `ePAR_INVALID_VALUE`.

When executing a `MCDDynIdDefineComPrimitve`, the DynId defined in the `DYN-ID` parameter of this DiagComPrimitve is added to the list of used DynIds at the logical link. If this DynId is already occupied, a `MCDProgramViolationException` with error code `eRT_DYNID_ALREADY_USED` is thrown.

It is not allowed to execute a service for DynID definition repeatedly, therefore if a `MCDDynIdDefineComPrimitive` is called with method `startRepetition` a `MCDProgramViolationException` with error code `eRT_SERVICE_REPEATED_NOT_ALLOWED` is thrown.

8.13.3.3 Reading by dynamically defined identifier

A dynamically defined LID can be read from the ECU by executing a `MCDDynIdReadComPrimitive`.

The D-Server will fill the dynamic part of the response template at runtime depending on the request parameters of a previously executed `MCDDynIdDefineComPrimitive` with the same definition mode as the `MCDDynIdReadComPrimitive`.

8.13.3.4 Deletion of dynamically defined identifier

Dynamic services exist until the end of a diagnostic session (execution of `MCDStopCommunication`, or until the ECU falls out of diagnostics by itself (see Figure 91)).

DynIDs can be deleted by executing a `MCDDynIdClearComPrimitive`.

— A DynId will be removed from the list that contains the assignable DynIds when the execution of a `MCDDynIdDefineComPrimitive` returns with a positive result. To avoid that a DynId is assigned multiple times by different `DynIdDefineComPrimitives` executed simultaneously, the DynId used in this service needs to be locked temporarily during execution of a `MCDDynIdDefineComPrimitive`.

— A DynId will be re-added to the definable DynId list (the list that contains the assignable DynIds) when a `MCDDynIdClearComPrimitive` executed with that DynId as parameter returns with a positive response from the ECU.

— If the ECU transits from the state `eCOMMUNICATION` to `eONLINE`, all DynIDs will become available again.

If a DynID list changes, the event `onLinkDefinableDynIdListChanged(MCDValues, MCDLogicalLink)` will be created by the logical link and sent to all registered event handlers. It transports the actual list of available (not defined) DynIds, as well as the corresponding logical link.

8.13.3.5 DB-Templates for requests and responses regarding dynamically defined identifier

As the DynID concept is an intrinsically dynamic concept of diagnostics at runtime, no complete database template for read DynID-services can be obtained from the ODX database. The part of the response that will be filled dynamically at runtime can be identified by a `MCDDbParameter` with parameter type eDYNAMIC. Calling method `MCDDbParameter::getDbParameters()` for this parameter will always deliver an empty collection. The client application is responsible for assembling a DynID ComPrimitive by selecting physical values from ODX tables and adding them as parameters to a DynIdComPrimitive by using the `MCDDynIdDefineComPrimitive`.

Please note that the current specification of the DynID ComPrimitives as shown in the examples in the ODX specification requires protocol dependent knowledge in the MVCI diagnostic server. The protocol dependence is manifest in the fact that number, type, and meaning of the different elements in a DynID item structure and DynID content parameter is purely protocol dependent and requires protocol-specific knowledge in the MVCI diagnostic server. To overcome this specification gap, ODX data that aims to be useable in a protocol-independent manner has to define specific semantic values for all the parameters that are used for DynID-related services.

Table 27 defines a value for the semantic attribute of each relevant parameter that will be recognized by the MVCI diagnostic server to allow protocol-independent handling of DynId data.

Table 27 — DYNID parameter semantic attribute definitions

Parameter name	Semantic value	Description
DYNID	DYN-ID	The parameter that contains the DynId-value to be assigned/read/deleted.
STRUCT_DYNIDItem	DYNID-DEF-STRUCT	The structure definition that contains the parameters necessary for DynID definition at runtime.
definitionMode	DYNID-DEF-MODE	Definition mode for a certain DynID parameter, protocol specific in ODX.
positionInDYNID	DYNID-POS	Position of the parameter in the response returned by a DynID read service, protocol specific in ODX.
memorySize	DYNID-MEMORY-SIZE	The size of the parameter that is to be included in a DynID service, protocol specific in ODX.
LID	DYNID-LID	The LOCAL-ID, COMMON-ID, or ADDRESS that is the source of the DynID parameter, protocol specific in ODX.
positionInRecordLID	DYNID-LID-POS	The position of the DynID parameter in its source LID, protocol specific in ODX.
DYNID	DYN-ID	The parameter that contains the DynId-value to be assigned/read/deleted.

Please note that the statements above do not have any implication on the MVCI diagnostic server API. Rather, they affect the MVCI diagnostic server internal realization of DynID.

ODX data that is supposed to allow a MVCI diagnostic server to support DynID in a protocol-independent way has to adhere to the parameter semantic definitions outlined in Table 27.

8.13.3.6 Procedure description

The execution of a `MCDDynIdDefineComPrimitive`, a `MCDDynIdReadComPrimitive` or a `MCDDynIDClearComPrimitive` is only allowed in the logical link states `eONLINE` and `eCOMMUNICATION`.

After successful execution of the `MCDDynIdDefineComPrimitive`, a database template for the corresponding `MCDDynIdReadComPrimitive` is generated internally within the MVCI diagnostic server. The database template is stored for one specific DynID. If a `MCDDynIdReadComPrimitive` is executed on this DynID, the stored database template is used to interpret the `MCDDynIdReadComPrimitive`'s response.

A stored database template for a DynID is deleted internally in the following cases:

The ECU changes state from `eCOMMUNICATION` to `eONLINE`.

⎯ The `MCDDynIdClearComPrimitive` is called (the DynID is deleted within the ECU).

Whenever the DiagComPrimitives `MCDDynIdDefineComPrimitive`, `MCDDynIdReadComPrimitive`, and `MCDDynIdClearComPrimitive` are called in the wrong sequence, e.g. a read is executed before the corresponding DynID has been defined, a `MCDProgramViolationException` with error code `eRT_WRONG_SEQUENCE` is thrown.

Whenever one of the ComPrimitives `MCDDynIdDefineComPrimitive`, `MCDDynIdReadComPrimitive`, and `MCDDynIdClearComPrimitive` is called that has not been parameterised completely, e.g. the DynID has not been set before execution, a `MCDParameterizationException` with error code `ePAR_INCOMPLETE_PARAMETERIZATION` is thrown.

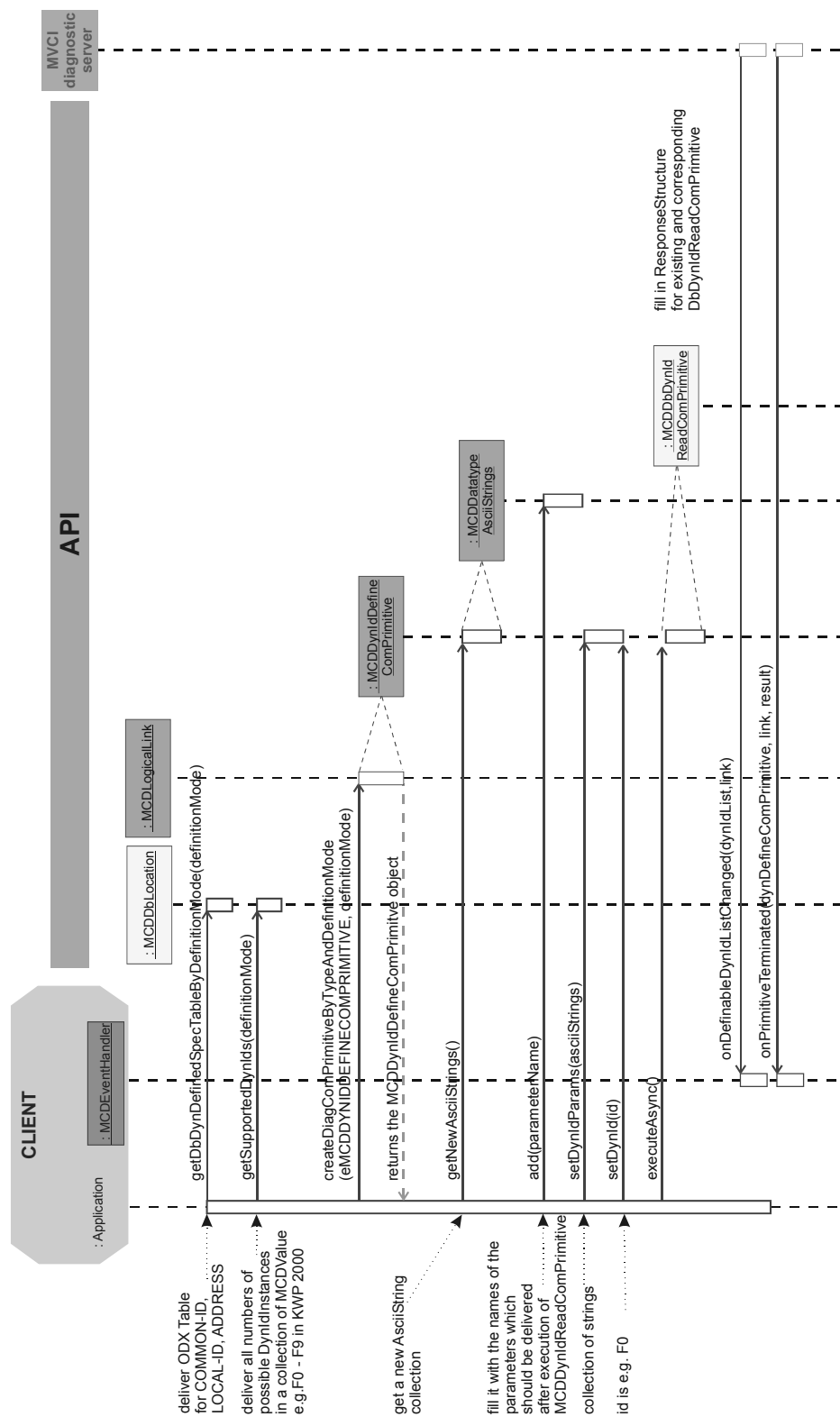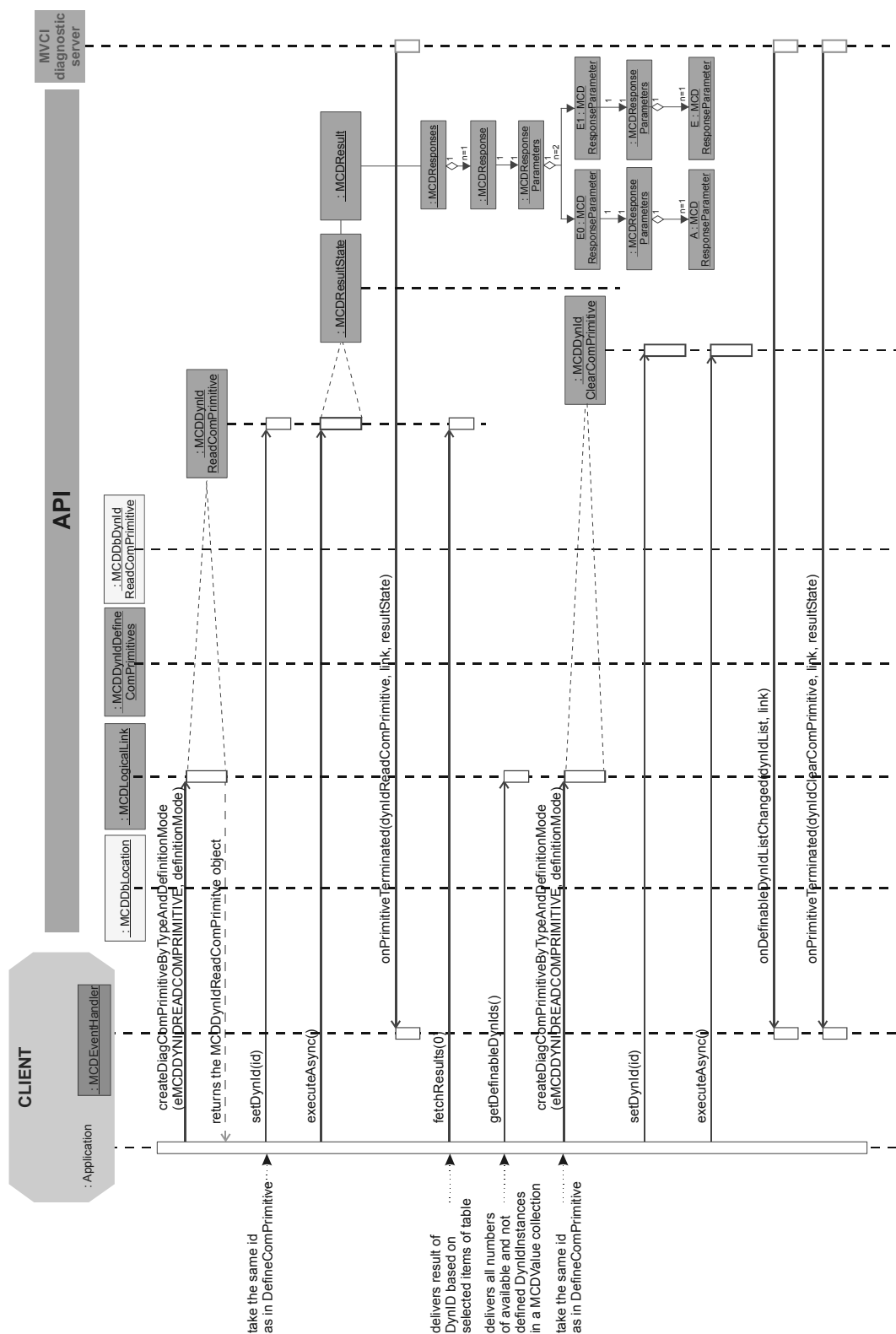Figure 102 depicts the usage of dynamically defined identifier (Part 1).

Figure 102 — Usage of dynamically defined identifier (Part 1)

Figure 103 depicts the usage of dynamically defined identifier (Part 2).

Figure 103 — Usage of dynamically defined identifier (Part 2)

8.14 Internationalization

8.14.1 Multi language support

Every database object has beside of the short name additional long name and description. To support country specific settings for messages (e.g. long names) and descriptions a string ID is used. With the string ID every string can mapped in the local language by the Client. To get the string ID each database object has two additional methods: `getLongNameID()` and `getDescriptionID()`.

For MCDParameter and MCDResponse the values for translation can be received by the corresponding database objects via the method getDbObject.

8.14.2 Units

All units in the diagnostic server are given in relation to SI units. But in many client applications country specific settings are needed. Therefore the database provides units and unit groups. The unit group (of type "COUNTRY") can be set for each Logical Link separately. (It makes no sense to set a single unit.) If the unit group "DEFAULT" is set, the diagnostic server sets the unit group back to the original settings given by data base.
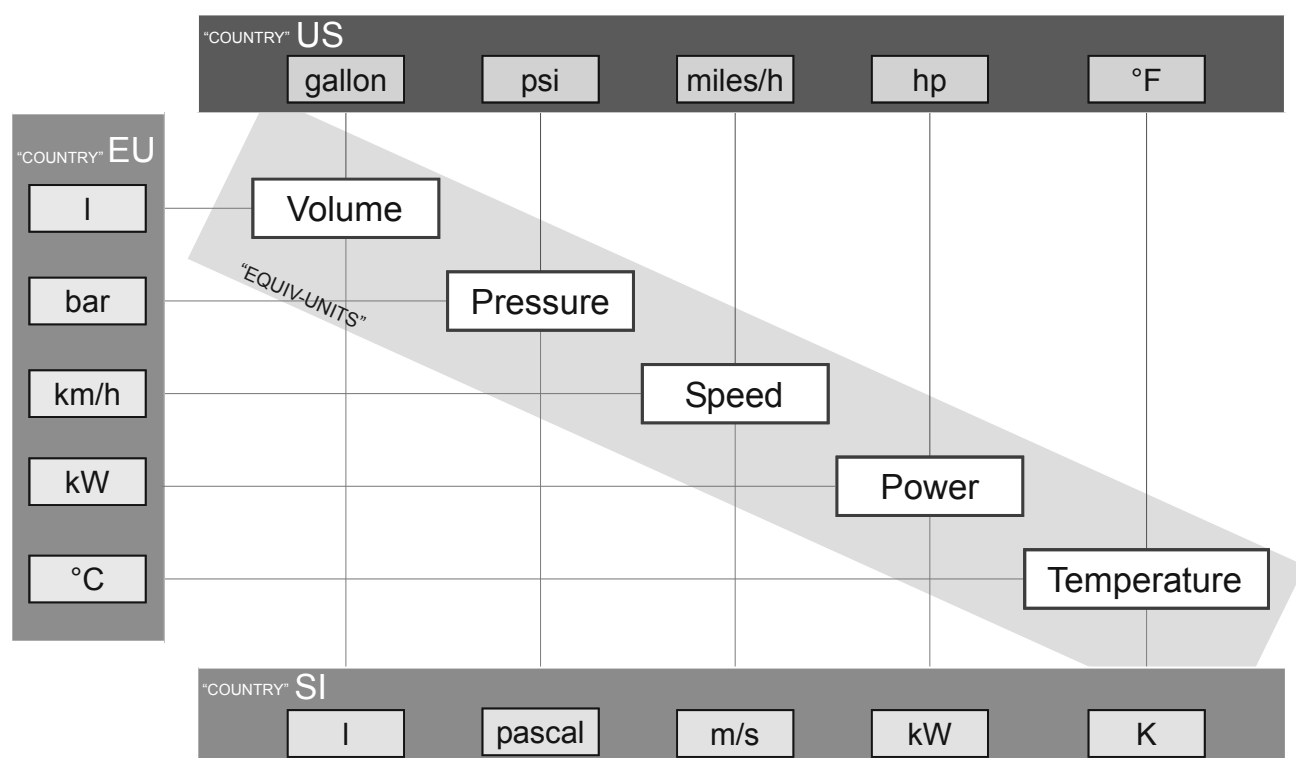
Figure 104 depicts the Unit groups.

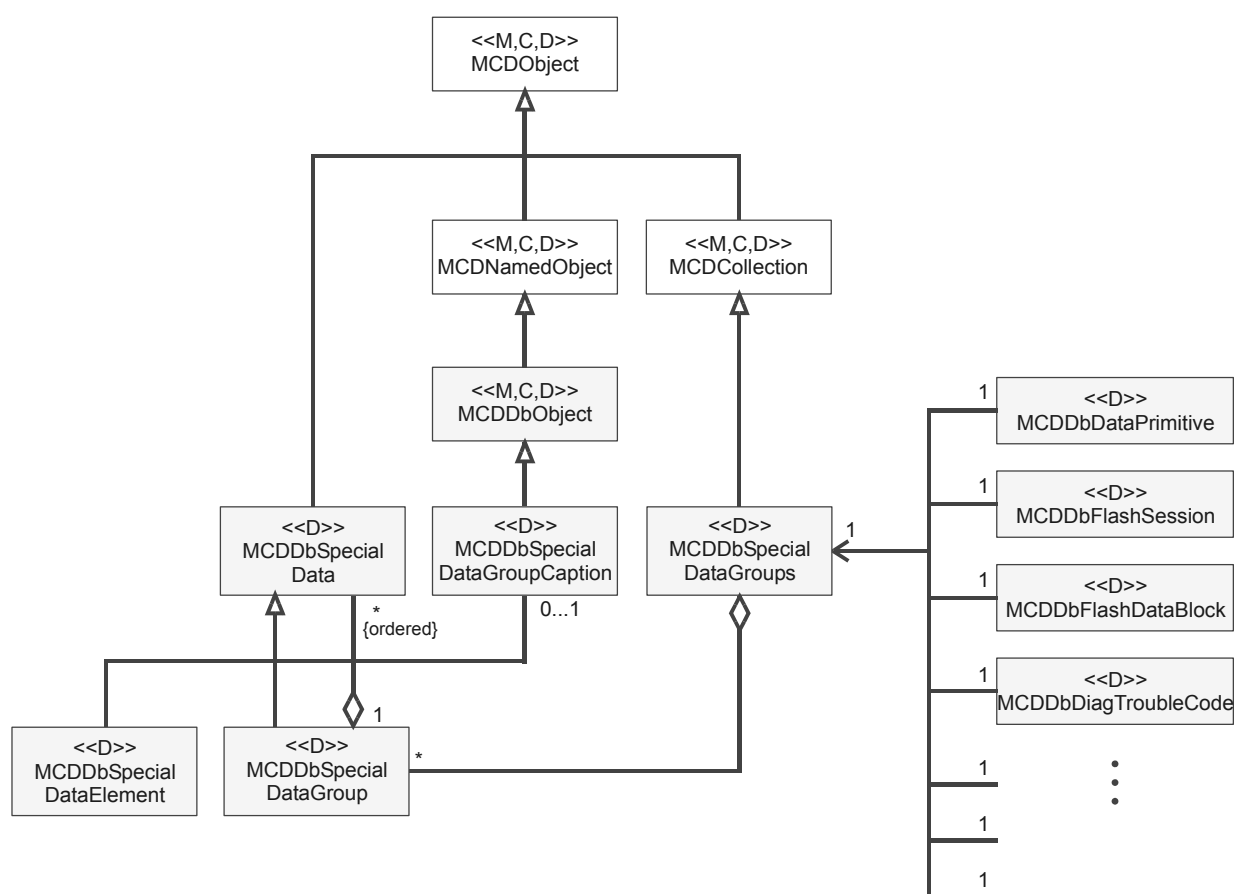**Figure 104 — Unit groups**

8.15 Special Data Groups

The generic Special Data Groups (SDGs) structure was introduced to allow the definition of company specific data structures necessary for arbitrary use cases. This can be so highly company specific purposes like ECU programming or flash processes.

Special Data Groups (SDGs) are used to add additional information to specific ODX elements, e.g. DIAG-COMM (MCDDbDataPrimitive), DTCs, or FlashSessions. SDGs have been introduced in ODX to be able to capture information that is not covered by general ODX mechanisms. For example, SDGs can be used to attach error set conditions to DTCs or to provide error trees. However, SDGs have been introduced to be able to provide additional information but not calculation relevant information, i.e. SDGs do not have semantics.

An SDG contains an optional MCDDbSpecialDataGroupCaption to describe the SDG content, and a list of MCDDbSpecialDataGroup and MCDDbSpecialDataElement objects that contain the special data. This list can contain an arbitrary number of MCDDbSpecialDataGroup and MCDDbSpecialDataElement, and the ordering of these MCDDbSpecialDataGroup and MCDDbSpecialDataElement is not restricted in any way. Note that SDGs can be nested recursively; that way, very complex data structures may be defined as SDGs. The MCDDbSpecialDataElement is used to add semantic information to the appropriate object.

SDGs can belong to data primitives like diag services and jobs, flash sessions, flash data blocks or diag trouble codes. Figure 105 depicts the Special Data Groups.

Figure 105 — Special Data Groups

Table 28 provides an overview about the related ODX element, where the SDG information can be taken from.

Table 28 — Special Data Group relation

ODX Element	MVCI API Class
CONFIG-DATA	MCDDbConfigurationData
CONFIG-ITEM, OPTION-ITEM	MCDDbConfigurationItem
CONFIG-RECORD	MCDDbConfigurationRecord
DATABLOCK	MCDDbFlashDataBlock
DATA-RECORD	MCDDbDataRecord
DIAG-COMM	MCDDbDataPrimitive
DIAG-LAYER	MCDDbLocation
DTC	MCDDbDiagTroubleCode
ECU-MEM-CONNECTOR	MCDDbECUMem
ITEM-VALUE	MCDDbItemValue
MULTIPLE-ECU-JOB	MCDDbDataPrimitive
PARAM	MCDDbParameter
REQUEST	MCDDbRequest
RESPONSE	MCDDbResponse
SESSION-DESC, SESSION	MCDDbFlashSession
TABLE	MCDDbTable
TABLE-ROW	MCDDbParameter
FUNCTION-DICTIONARY	MCDDbFuntionDictionary
BASE-FUNCTION-NODE	MCDDbFunctionNode

8.16 ECU (re-) programming

8.16.1 Goal

The design of the ECU (re-) programming is based on the following items:

— The DB part of the object model has to provide access to all fields within the ASAM ECU-MEM.

— The MVCI diagnostic server must be able to list ECU MEMs in dependence of the selected MCDDbLocation.

— It must be possible to write a generic flash job, that can be programmed independently from the data and where the session which has to flash can be parameterised by runtime.

— All protocol dependent activities have to be done in a job, or inside the protocol processor (for ISO underneath D-PDU API)

— Upload will in this version not be supported.

— Offsets will be handled by the MVCI diagnostic server internally to calculate start- and end-addresses.

8.16.2 Structuring of the function block flash

8.16.2.1 Database part

The root element of DB part of the flash processor is the `MCDDbLocaton` object. This object can list all sessions that are defined for the actual `MCDDbLocation`. It also gives access to a collection of `MCDDbFlashSessionClasses` to give the application the possibility to structure the FlashSessions. From this root element you have access to all information that is stored within the ECU-MEM and the FlashSession data.
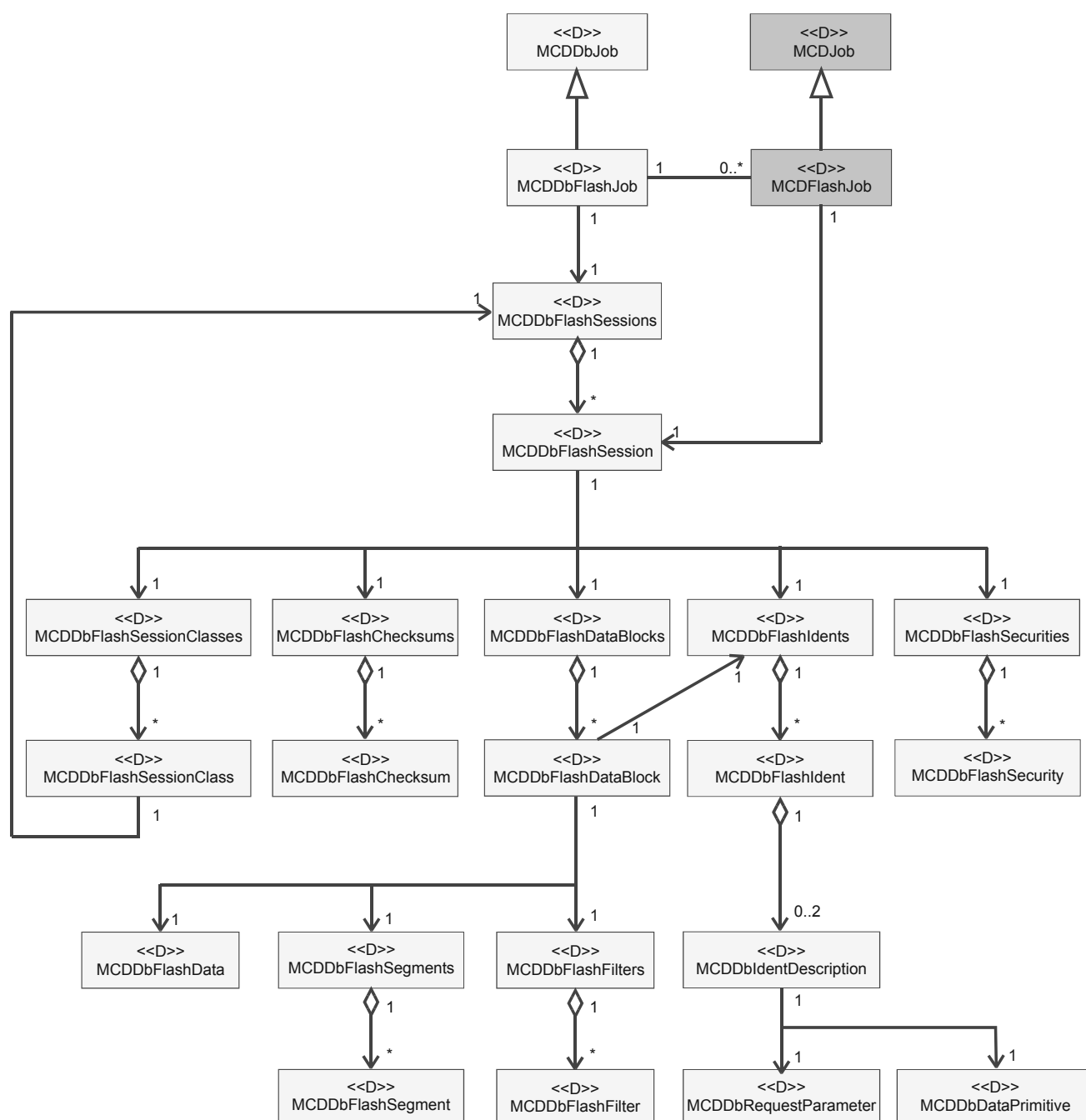
The methods `MCDDbFlashSession::getLongName()`, `MCDDbFlashSession::getShortName()`, and `MCDDbFlashSession::getDescription()` return the corresponding values of a SESSION-DESCRIPTION element in ODX.

The data returned by `MCDDbFlashSession::getDescriptionID`, `MCDDbFlashSession::getLongNameID` and `MCDDbFlashSession::getUniqueObjectIdentifier` is taken over from the a SESSION-DESCRIPTION element in ODX.

The next ERD shows the associations between the database objects for a single FlashJob in UML notation.

Figure 106 depicts the Flash associations.

Figure 106 — Flash associations

A `MCDDbFlashFilter` is used to enable handling of segments of binary or hexadecimal files, that is, to extract a part from of such a file. (Binary data does not include a start address.)

Use case 1: *Binary data file*

A binary data file does not include the start address. Therefore the flash filter delivers this as start address. This start address is delivered by `MCDDbFlashSegment::getSourceStartAddress():A_UINT32`

Use case 2: *Hex data file* **(e.g. motorola srecord, intel hex)**

A hex data file includes the start address and the size. In this case the filter delivers start and end address of an area in the file, e.g. code / data / boot.

When no SEGMENTs are specified in ODX, the `MCDDbFlashSegments` will be constructed from the inline data or the referenced file. Each `MCDDbFlashSegment` will be given the ShortName: "Segment_"<SourceAddressStart>"_"<SourceAddressEnd>

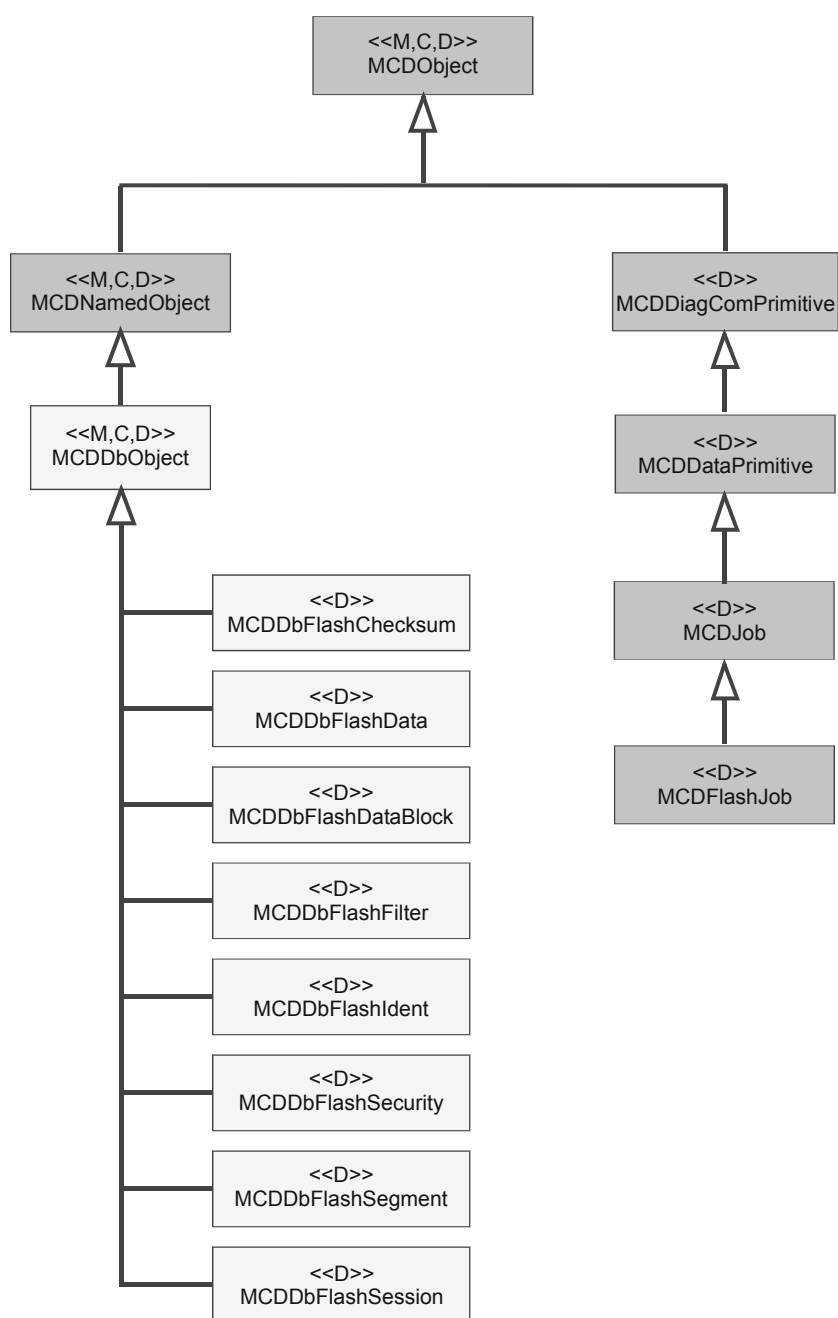EXAMPLE   Example: "Segment_0x00000000_0x000000FF"

No namespace conflicts will arise, because no other Segments are available in the segments collection and the segments may not overlap.

When no SEGMENTs are specified in ODX, the SourceStartAddress and the SourceEndAddress will be read from the inline data or the referenced file respectively. This applies only for the formats Motorola-S and IntelHex.

If the data is stored in binary representation and no SEGMENTS are defined in ODX, the data cannot be flashed. Therefore no `MCDDbFlashSegment` object will be defined and the Method `MCDDbFlashDataBlock::getDbFlashSegments():MCDDbFlashSegments` returns an empty collection.

If the flash data is in binary format, reprogramming is only possible if there are also FlashSegments defined. Otherwise, reprogramming is not possible. Therefore, the definition of FlashSegements is mandatory here. On the other hand, the definition of FlashSegments is optional in case of hex data. As a result, segments for hex data have to be determined internally if they cannot be read from the ODX data base.

Figure 107 shows the inheritance of the single Interfaces for all flash objects.

Figure 107 — Flash interfaces

8.16.2.2 Online part

To execute ECU (re-) programming it is possible to get the correct flash job for a session from the MCDDbSession object. For this MCDFlashJob object you can call the method `setSession()` to choose the session to be flashed. It is only possible to set one session at a time.

To start the download of that session call the `executeSync() or executeAsync()` method.

If you have to flash more than one session, the above described sequence have to be done for every session. If you need to sort the sessions, this has to be done by the application. The application can get the priority for every session from the projects data basis via the MVCI diagnostic server.

Within the job the whole flash sequence is described. e.g.

— security access,

— start diagnostic session

— download of every data block within the selected FlashSession

— checksum prove

— ECU reset

This job has all information to generate all telegrams needed to send to the ECU. (For KW2000 the job can create all "0x34", "0x36", "0x37" sequences for all segments). The binary data for every segment can be accessed by the method `getBinaryData` at the MCDDbFlashSegment interface. If the data is in any other format than binary (e.g. MotorolaS or IntelHex) the MVCI diagnostic server has to convert the data to provide it as binary for the job.

For the reason of simplification and reuse it is possible to create sub jobs that will flash one whole data block, or a job that will flash a single segment. The MVCI diagnostic server API provides all mechanisms that this creation of sub jobs is possible. All these jobs used for ECU (re-) programming shouldn't be executed at the API directly. For this reason they have to be signed as not executable. The DbPart still lists these jobs, but an creation of an online object of these jobs outside the MCDDbFlashJob will be denied by the MVCI diagnostic server (`eRT_SERVICE_EXECUTION_FAILED`).

8.16.2.3 Progress information

A general mechanism for progress in jobs is:

If flash job is executed synchronously or asynchronously, the application can retrieve progress by calling method `MCDJob::getProgress()` and JobInfo by method `MCDJob::getJobInfo()`.

If flash job is executed asynchronously, MVCI diagnostic server will issue events of type `onPrimitiveProgressInfo` according to the flash job's `setProgress` method calls. The progress event shall provide the progress. The progress is expressed in values between 0 and 100.
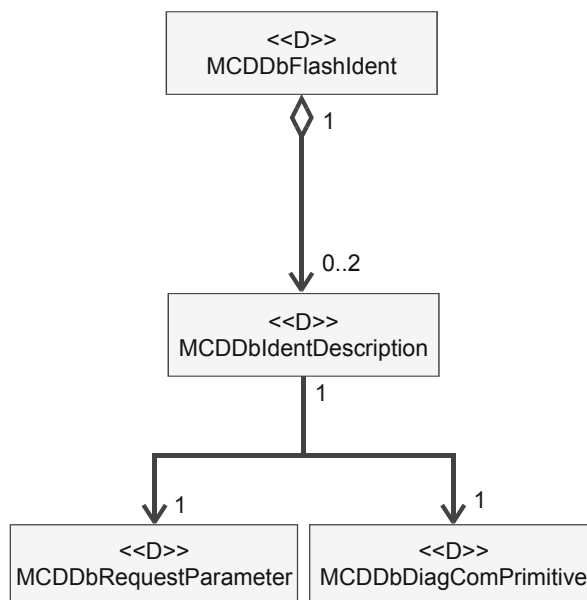
If flash job is executed asynchronously, MVCI diagnostic server will issue events of type `onPrimitiveJobInfo` according to the flash job's `setJobInfo` method calls.

8.16.2.4 Ident mechanism

ASAM ECU-MEM provides expected identification values for every FlashSession and own identification values for every data block. This information is stored in the ECU-MEM and must simply be accessed by the MVCI diagnostic server. After reading the FlashIdents from the ECU, they can be compared with the

corresponding values in the ECU-MEM. The object model provides a reference to a service that reads / writes this id out of the ECU for every FlashIdent object. This link between the ECU-MEM and the diagnostic data is definitively necessary. Idents may be linked with a Read- and a Write-Service. The information which parameter holds the ident value is reachable via the `MCDDbIdentDescription`. The IDENT-DESCs shall not have parameters with COMPLEX-DOPs.

Figure 108 depicts the FlashIdent.

Figure 108 — FlashIdent

8.16.3 ECU-MEM

In ODX an ECU-MEM is connected with the belonging ECU-VARIANTs and BASE-VARIANTs through an ECU-MEM-CONNECTOR. The methods `MCDDbEcuMem::getLongName()`, `MCDDbEcuMem::getShortName()`, and `MCDDbEcuMem::getDescription()` return the corresponding values of an ECU-MEM-CONNECTOR in ODX.

It should be possible to load ECU-Descriptions and ECU-MEMs for a project. The state of the `MCDSystem` must be `eDBPROJECT_CONFIGURATION`.

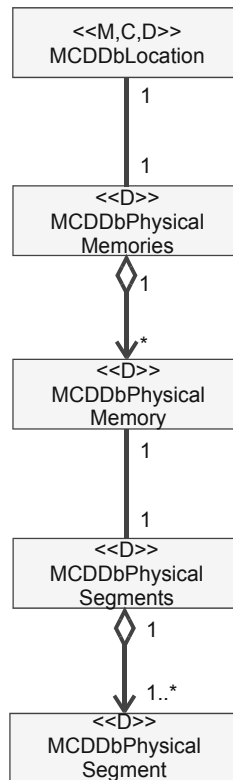ECU-Descriptions and ECU-MEMs should be loaded to a MCDDbProject or permanently added to a project configuration.

It is possible to list all ECU-MEMs at MCDDbProject using the method `MCDDbProject:getDbECUMems`.

It is possible to list ECU-MEMs at MCDDbProject, using the method `MCDDbProjectConfiguration.getAdditionalECUMEMNames() : A_ASCIISTRINGs`, which are not included in the project. It will be searched within the paths of the System. This is the current list of ECU-MEMs. Two calls can deliver different lists.

By means of the method `MCDDbProject:loadNewECUMEM(MCDDatatypeShortName ecumemName, bool permanent=false)` an ECU_MEM, which does not exist within in the project, will be loaded in the project. The parameter permanent controls if this ECU MEM is added permanently or not.

The method `MCDDbProject::loadNewEcuMem()` throws an exception of type `ePAR_SHORTNAME_INVALID` in case that a file for the MCDDatatypeShortname value supplied as parameter to this method was not found in the data base.

Figure 109 depicts the Physical memory.

Figure 109 — Physical memory

The MVCI diagnostic server needs no API-method to provide an address-offset. Offsets will be handles by the MVCI diagnostic server internally to calculate start- and end-addresses. Addresses are represented as byte addresses.

8.17 Handling binary flash data

8.17.1 Late-bound data files

In the context of data files for ECU programming, the term late-bound denotes that the corresponding data file will be loaded by a diagnostic server as late as possible. That is, a late-bound data file is loaded the latest

— when one of the methods
  `MCDDbFlashSegment::getBinaryData()`,
  `MCDFlashSegmentIterator::getFirstBinaryDataChunk()`,
  `MCDFlashSegmentIterator::hasNextBinaryDataChunk()`,
  `MCDFlashSegmentIterator::getNextBinaryDataChunk()`
  is called for one of the `MCDDbFlashSegments` contained in a `MCDDbFlashDataBlock`
  or

— when the method `getFlashSegments()` is called for the first time at an `MCDDbFlashDataBlock` in case of a Motorola-S or an Intel-Hex data file.

The binary data associated with a `MCDDbFlashDataBlock` can only be marked late-bound if this binary data is located in an external file referenced from the ODX data. That is, the binary data is not embedded into the ODX data.

If the value returned by `MCDDbFlashData::isLateBound()` is 'false' at a reference to an external resource file (e.g., job code, flash data, coding data), this is considered a guarantee that the content of this resource file will not change while a diagnostic server is running. More precisely, the content of the external resource file shall be static for the time between `MCDSystem::selectProjectXXX()` and `MCDSystem::deselectProject()` for the same project.

Please note that exchanging external resource files may lead to non-deterministic behavior of a diagnostic server. In particular, this statement holds in case the late-bound property is set to 'true' for some ODX element.

8.17.2 Wildcards in data file names

The reference to external binary data from an element of type `MCDDbFlashData` attached to a `MCDDbFlashDataBlock` can either point to a specific external data file or it can be unspecific. In the first case (specific data file), the value of type A_ASCIISTRING returned by the method `MCDDbFlashData::getDataFileName()` does not contain any wildcards, that is, none of the characters '?' or '*' is contained in the filename. Here, the external data file can be accessed at any time, e.g. to read the binary data or to calculate `MCDDbFlashSegments`.

In case of an unspecific data file, the filename returned by `MCDDbFlashData::getDataFileName()` contains one or more wildcard characters '?' or '*'. These wildcards need to be resolved at runtime in order to identify matching data files. Please note that wildcards are only allowed in a filename if the flash data is also marked late-bound, that is, if the method `MCDDbFlashData::isLateBound()` returns true for `MCDDbFlashData` element. Here, the binary data file to be used for a corresponding `MCDDbFlashDataBlock` needs to be determined at runtime. For this purpose, the following steps need to take place:

— The client application calls the method `MCDDbFlashDataBlock::getMatchingFileNames()`. Now, the diagnostic server tries to identify all files which match the pattern returned by the method `getDataFileName()` at the `MCDDbFlashData` in the database part of the current `FlashDataBlock`. The result of the calculation is a collection of `MCDDatatypeAsciiString` objects where each `MCDDatatypeAsciiString` represents the filename of a matching data file. Note that the search scope of the diagnostic server depends on vendor-specific definitions and settings. That is, each implementation of a diagnostic server may have a different definition of the directories, which are searched for matching data files. Please refer to the documentation of the specific diagnostic server implementation for more details.

— The client application calls the method `MCDDbFlashData::setActiveFileName` to select the flash file to be used.

— The client application calls the method `MCDDbFlashDataBlock::getFlashSegments()`. The diagnostic server now creates the `MCDDbFlashSegments` for the current `MCDDbFlashDataBlock`.

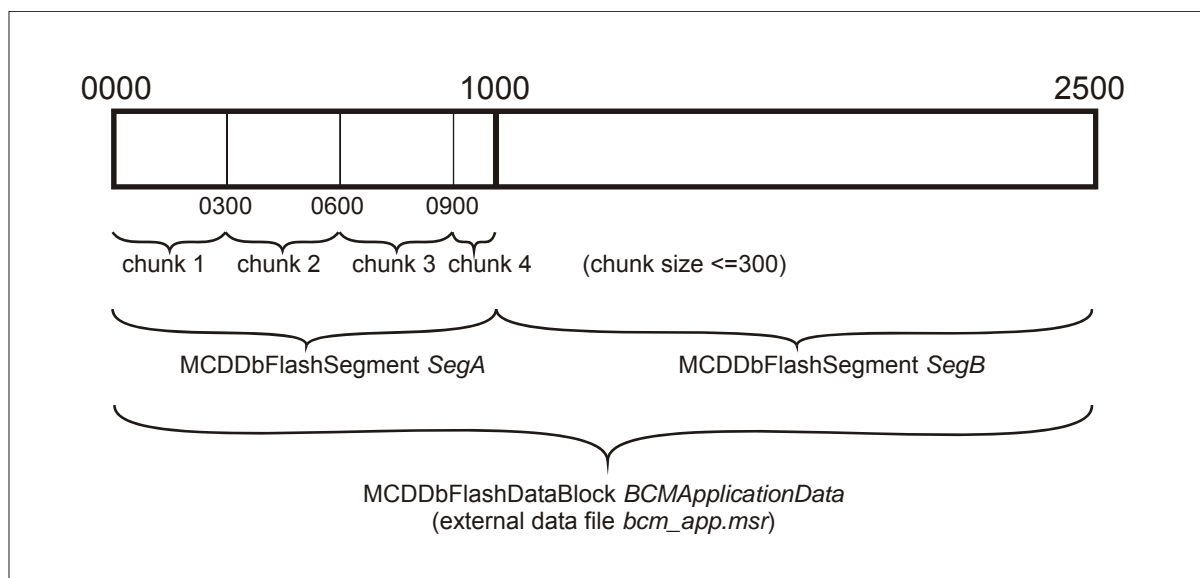### 8.17.3 Flash segment iterator

Flash data files can become quite large (several tens of megabytes). As a consequence, keeping data files in the main memory of a tester consumes huge amounts of memory. To also support thin tester applications with less main memory capacity, the concept of FlashSegmentIterators has been introduced. Such a FlashSegmentIterator allows reading the data, which represents the binary data of a `MCDDbFlashSegment` in an external data file, in small chunks. As a result, large data files can be handled by tester applications with less memory consumption.

In this standard, a FlashSegmentIterator is represented by the interface `MCDFlashSegmentIterator`. A new iterator for a `MCDDbFlashSegment` can be obtained by calling the method `MCDDbFlashSegment::createFlashSegmentIterator(size : A_UINT32)`. The parameter `size` of this method defines the maximum size of the chunks that will be delivered by the newly created iterator in bytes. Note that the last chunk delivered by the diagnostic server is potentially smaller than defined by `size`. By means of the method `MCDFlashSegmentIterator::getFirstBinaryDataChunk()`, the first piece of binary data is obtained from the external data file – if possible. In addition, the FlashSegmentIterator's internal pointer is set to the next piece of binary data.

By means of the method `MCDFlashSegmentIterator::hasNextBinaryDataChunk()`, a client application can check whether there are more binary data chunks available in the iterator. If this is the case, the method `MCDFlashSegmentIterator::getNextBinaryDataChunk()` can be used to obtain this next piece of binary data.

The MCDFlashSegmentIterator shall be removed by the client if it is not needed anymore by calling the function `MCDDbFlashSegment::RemoveFlashSegmentIterator()`.

Figure 110 shows an example of a DbFlashDataBlock *BCMApplicationData* which references an external data file *bcm_app.msr* via a corresponding element of type `MCDDbFlashData`. This FlashDataBlock is decomposed into two FlashSegments *SegA* and *SegB*. In the example, the client application has created a new FlashSegmentIterator with a size of 300 bytes for Flash Segment *SegA*. When iterating through the Flash Segments binary data, the FlashSegmentIterator returns three chunks of 300 bytes and a forth chunk of 100 bytes. After the fourth chunk has been obtained by the client application, the end of Flash Segment SegA has been reached. Therefore, no further chunks are available. That is, the method `MCDFlashSegmentIterator::hasNextBinaryDataChunk()` returns false after the forth chunk has been obtained.

Figure 110 — Example of data chunks obtained from a FlashSegment SegA

8.18 Library

Library elements are used within ODX to specify additional program code which is used ('included' in Java terms) by the executable code referenced by the CODE-FILE attribute of a PROG-CODE instance. A data element of type PROG-CODE is used by ODX to specify Java program code which is executable by the diagnostic server. Libraries are defined by LIBRARY elements in ODX and can be referenced by one or multiple PROG-CODE elements to extend the classpath that is associated with that PROG-CODE. A PROG-CODE must be executed in the classpath environment defined by the program code (class file or JAR) referenced by the CODE-FILE attribute at PROG-CODE, as well as the program code referenced by CODE-FILE attribute of the referenced LIBRARY elements. This also applies if the PROG-CODE execution is embedded within another program code (Java job) execution. In such a case, the calling job and the called job need to be executed within different classloader environments if a different class context is defined by the associated ODX data.
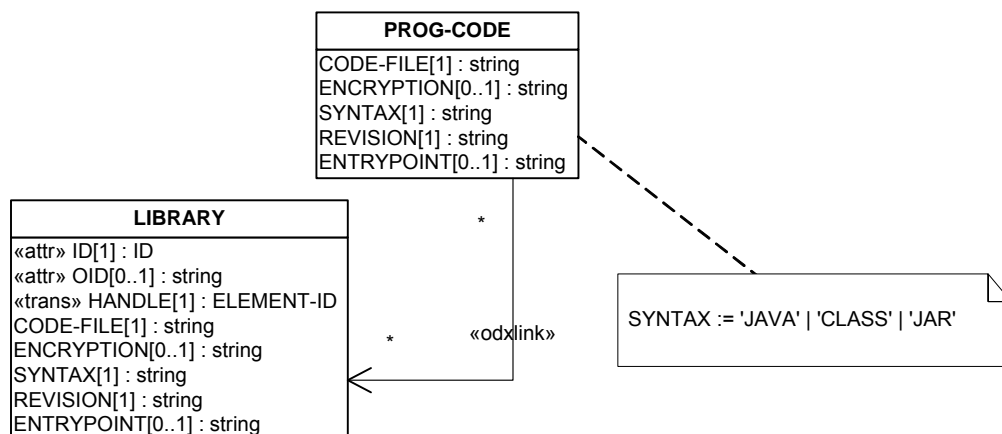
In ODX, LIBRARY definitions are associated with a DIAG-LAYER and can be referenced by `MCDDbCodeInformation` (ODX: PROG-CODE) instances at `MCDDbSingleEcuJob` (ODX: SINGLE-ECU-JOB), `MCDDbMultipleEcuJob` (ODX: MULTIPLE-ECU-JOB) and COMPU-METHOD of category COMPUCODE. Please note that ODX also allows library elements to point to arbitrary other types of data (e.g. it is possible to reference a dynamic system library (*so* or *dll*) from a LIBRARY element. These cases cannot be covered in the scope of this standard, and diagnostic server behaviour has to be defined in a customer- and project-specific way.

The following resources of program code have to be included into the Java classpath environment for a PROG-CODE (MCDDbCodeInformation and DOPs referencing Java code):

— Resources referenced by the PROG-CODE itself.

— Resources referenced by any of the LIBRARY elements that are referenced by PROG-CODE.

— Permitted packages of the standard JRE.

— MVCI diagnostic server API (except for COMPUCODE)

For performance improvements a classloader may be reused by any PROG-CODE which defines the same Java class environment.

Figure 111 depicts the relation between Library and Prog-code.

Figure 111 — Relation between Library and Prog-code

Please note that in contrast to "CLASS" and "JAR", the support of Java source code is optional for a diagnostic server. Resources referenced by PROG-CODE or LIBRARY with SYNTAX="JAVA" would either have to be compiled at runtime, or during a pre-processing step transforming the ODX resources to a proprietary runtime format.
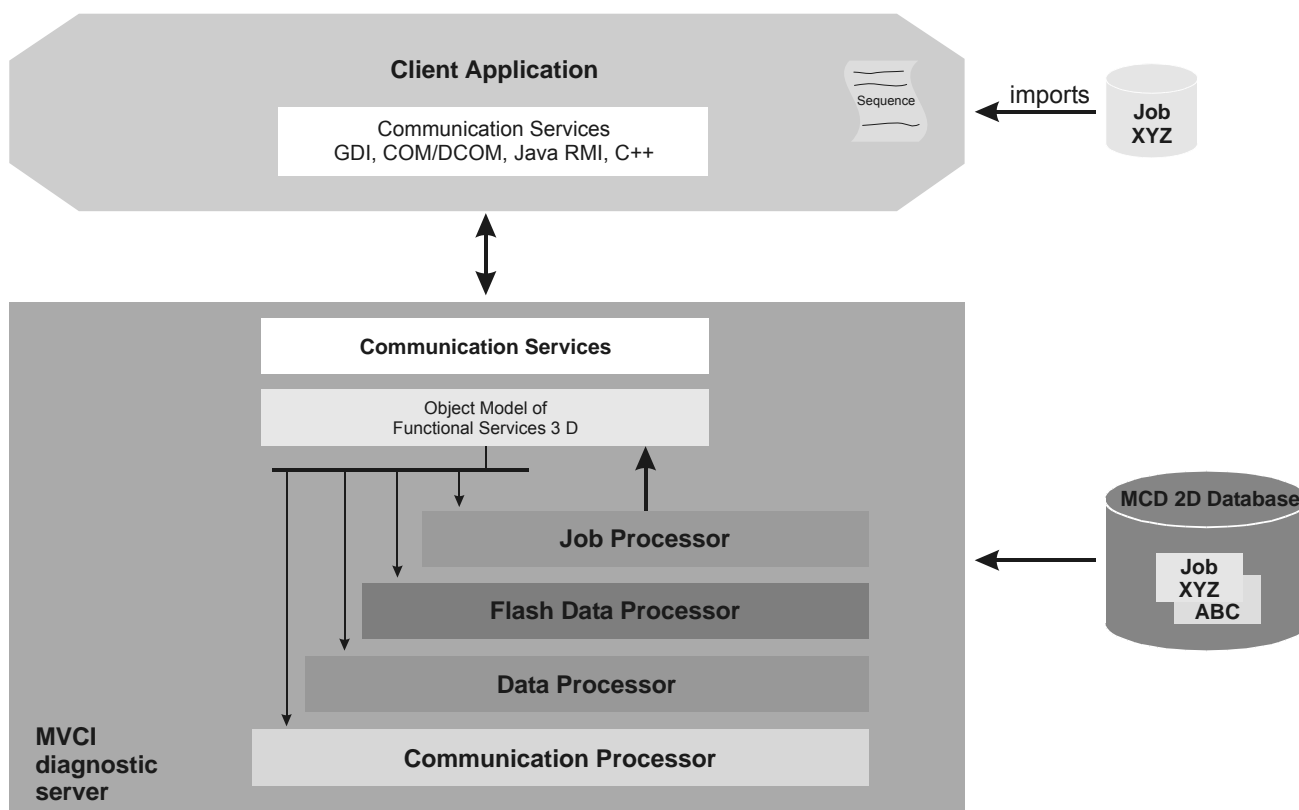
8.19 Jobs

8.19.1 General

The source code of Jobs is always identical, no matter if it is executed in the MVCI diagnostic server or in the Client.

Job   Diagnostic Sequence executed by Job Processor (Java) of
                                MVCI diagnostic server

Sequence   Diagnostic Sequence executed by Client of MVCI diagnostic
                                server

Figure 112 depicts the execution of jobs as a job in MVCI diagnostic server or as a sequence in the client.

Figure 112 — Executing jobs as a job in MVCI diagnostic server or as a sequence in the client

Generally, Jobs are handled like diagnostic services. However, they may be called only non cyclic and single. A Job has Meta information about the input and output parameters and the services used within the Job. Anyway, a Job may return 0..n intermediate results and only one final result. After the final result no further intermediate result can be delivered.

Threads or processes are not allowed inside a job. Parallel execution of the same SingleECUJob must be possible on different Logical Links, this means the SingleECUJob must be re-entrant. Only exceptions defined in the ASAM MVCI diagnostic server API may be thrown, other exceptions e.g. of Java SDK shall not be thrown.

The Job exchange format is Java source code. Allowed are interfaces and methods of API and some special Java libraries (see 8.19.8). Inheritance is allowed for all classes within the classpath of the MVCI diagnostic Server. Further information to classpath definition and classloader handling with Jobs can be found in 8.18.

It is allowed to use `MCDControlPrimitives` in jobs. The restriction to do so is, that the jobs shall be defined in the base variant and shall not be overwritten or excluded by any variant. The application is responsible to take care about potential parallel running threads, because the execution may cause logical link state changes and so on.

Communication Parameters can although be modified. If a Java-Job needs to alter the currently valid protocol parameters of the Logical Link, it should use and execute a `MCDProtocolParameterSet` from within its code. Please note that all changes to protocol parameters caused by a `MCDProtocolParameterSet` executed within a Java-Job will be persistent after this job has terminated – just as if a client application would have issued the changes. So, a clean Job implementation has to restore the protocol parameters at the end of the Job execution. Please note that the usage of `MCDProtocolParameterSet` in a Java-Job is considered harmful, as it could cause undocumented and therefore unexpected changes to the protocol parameters of a logical link at runtime.

8.19.2 Input and output parameters

Job input and output parameters (including results) can use complex and simple DOP. There may not be a difference between results of jobs and diag services.

For a diag job, the definition of input and output parameters is always static. This means the number of elements may be increased for a field, whereas the field type may not be changed into any other data type at run-time.

With respect to ODX, a Single ECU Job or a Multiple ECU Job , hereafter referred to as "Jobs", has a possibly empty set of input parameters, a possibly empty set of positive output parameters, and a possibly empty set of negative output parameters. These three sets of parameters are represented by three database objects - a request, a positive response, and a negative response - that can be obtained from the database object of a Job. With this in mind, the diagnostic server needs to generate the required request and response templates (DB request and DB response). The Shortname of the generated request object should be "#RtGen_Request". The Shortname of the generated positive response should be "#RtGen_Positive_Response". The Shortname of the generated negative response should be "#RtGen_Local_Neg_Response".
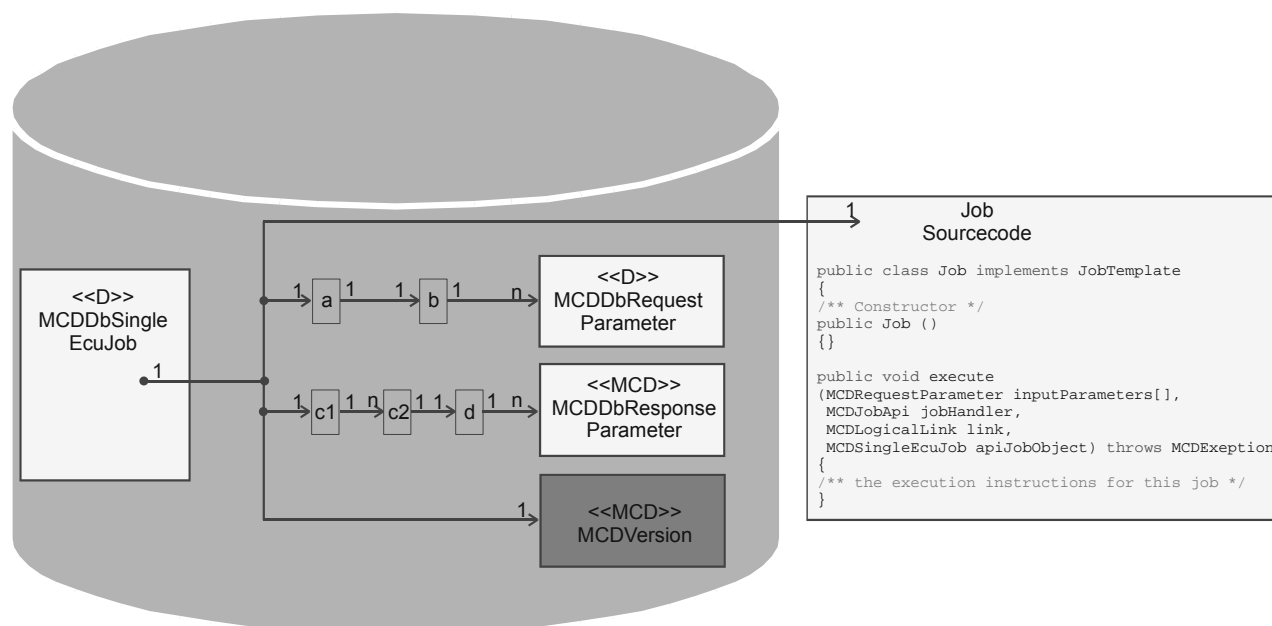
The type of a response – positive response or negative response – can be obtained by `MCDDbResponse::getResponseType()`. For a positive reponse the value is "ePOSITIVE_RESPONSE". For a negative reponse, the value is "eLOCAL_NEG_RESPONSE".

As all sets of parameters of a Job -input parameters, positive output parameters, and negative output parameters - can be empty, the request and response templates of a Job can be empty, that is, they do not contain any request or response parameters, respectively.

As the set of request parameters, positive response parameters, and negative response parameters are optional (cardinality 0..n) in ODX (see ISO 22901-1), the parameter collections of the corresponding request and response objects can be empty. However, the database objects for request, positive reponse, and negative response of a Job are nevertheless generated by the diagnostic server.

At MVCI diagnostic server API level, there is always the same static access to the input and output parameters, like in case of diagnostic services.

Figure 113 depicts the separation between database and job source code.

Key

a   MCDDbRequest

b   MCDDbRequestParameters

c1   MCDDbResponses

c2   MCDDbResponse

d   MCDDbResponseParameters

Figure 113 — Separation between database and job source code

8.19.3 Job result

The result of the Job is freely designed by the Job writer. Because of this it is optional, if elements of the intermediate results are included within the result of the Job or not. Every result (intermediate or final) must correspond to the output parameters described in the Meta information.

The Job writer defines, what has to be returned as intermediate or final result of the job. Thus, it is also not defined, that an intermediate result has to be identical to a complete result set of a diagnostic service used within the Job. It is up to the Job writer to decide if elements of the intermediate results shall be put out within the final result again. Anyway, a Job may return 0..n intermediate results and only one final result. After the final result no further intermediate result can be delivered.

Intermediate and final results shall use the same database template.

Jobs with dynamically reduced results (i.e. returning only subsets of the results defined in the DB template) are be allowed under the following conditions:

— to enable dynamic results the function enableReducedResults must be called,

— If a job marked with the ODX flag `IS-REDUCED-RESULT-ENABLED` is to be executed while dynamic results are disabled, the error `eRT_NO_DYNAMIC_MODE` shall be reported,

— If a job supports dynamic results can be asked with the method `isReducedResultEnabled`.

Events for intermediate results are initiated by the diagnostic job. The MVCI diagnostic server (not the Job itself) passes the event towards the application. Whether the application uses the intermediate results or not is left up to the application itself. The application may just ignore the event signal and leave the result data in the ring buffer until the final result shows up. Depending on the ring buffer's maximum count of elements, the application will then have access to zero, several or all intermediate results besides the final result.

If the initiation of intermediate result events shall be activated depending on the user's needs, the job must provide an appropriate input parameter.

Also, if the contents of intermediate and final results are of different types, the differentiation shall be done within a single and static result structure (template), e.g. this can be handled by using a multiplexer entry at the template's root.

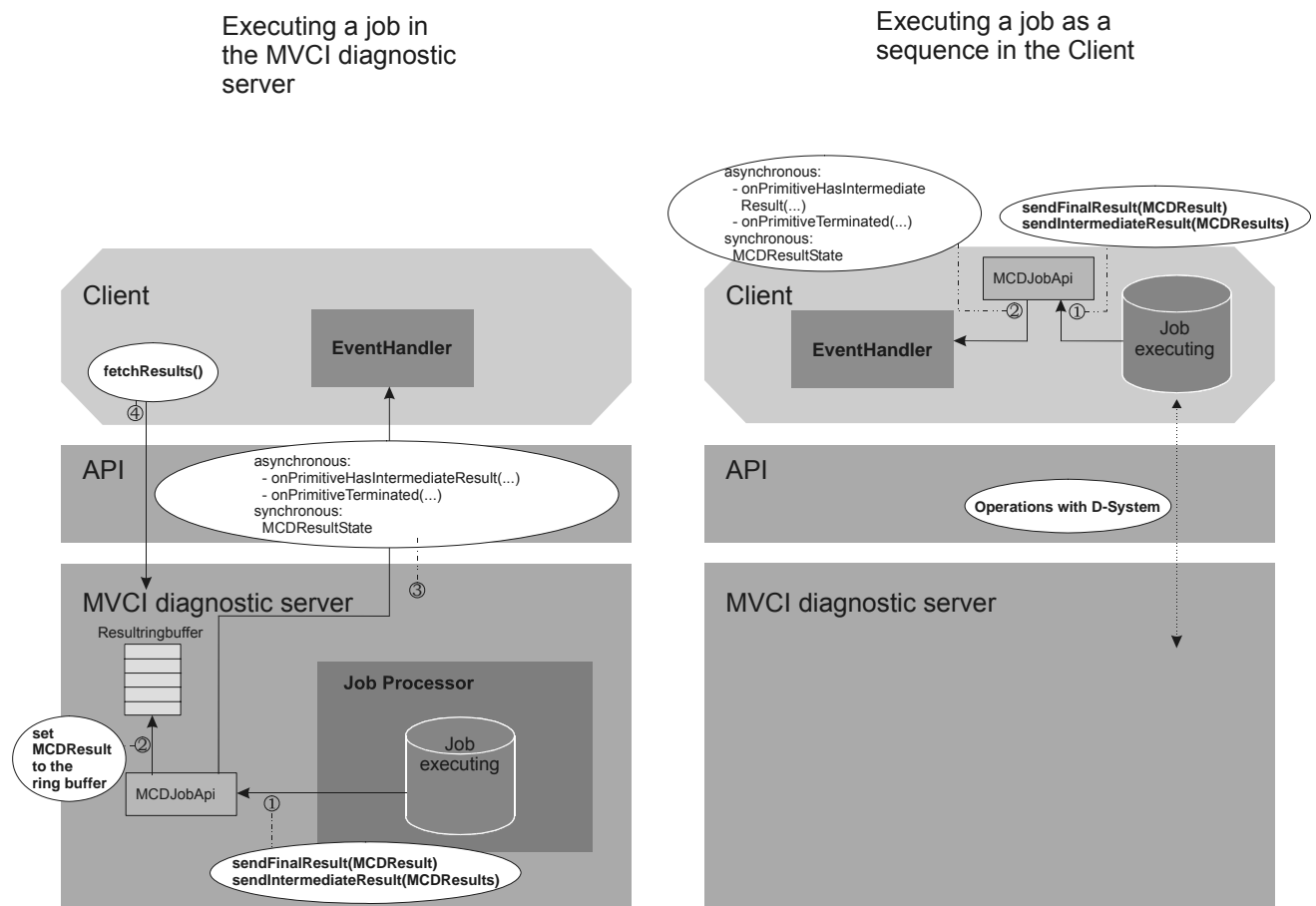Figure 114 depicts the principle of single execution of jobs.

Figure 114 — Principle of single execution of jobs

8.19.4 Single ECU jobs

SingleECUJobs have to be assigned to one location at run-time, that is, functional addressing is not possible. As result this kind of Jobs can only be executed in levels of Protocol, Base-Variant, and Variant. Inside the job code no reference to the accessed location will be necessary because the system already knows the accessed location. For SingleECUJobs the mechanism of inheritance, overwriting and elimination is supported (see ISO 22901-1).

In a SingleECUJob no Logical Link can be created. The location on which the Job is executed is a input parameter for the `execute` method. Thereby SingleECUJobs can be executed on different locations which have the same service names (and the services have the same result structures), e.g. Base Variant or Variant.

8.19.5 FlashJobs

A FashJob is derived from Job and behaves like a Single ECU Job witch is used to start flash sessions within the MVCI diagnostic server. This information is provided by the data bases. At the runtime object it is possible to set the Session that have to be flashed by this service. Only one session can be set for one job. The application can access the priority defined in the database for every flash session and can then sort the Sessions according to this priority.
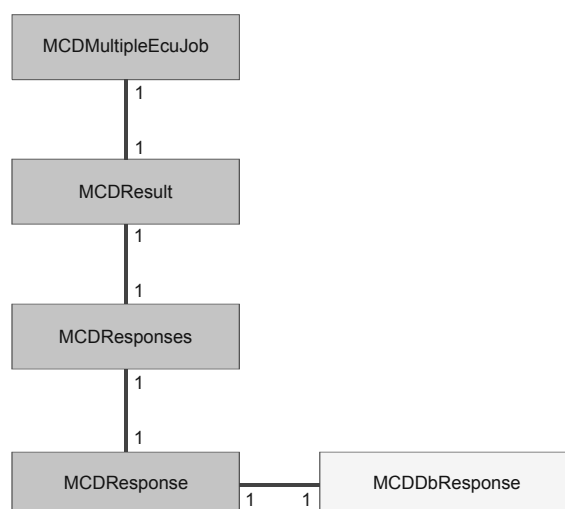
8.19.6 Multiple ECU jobs

For multiple ECU jobs the location [MultipleEcuJob] is used. This location does only contain jobs, no services are allowed.

The MVCI diagnostic server will provide the mechanism to list the locations (Protocols, Functional Groups, ECU Base Variant, ECU Variant) which are referred in such a job.

In a MultipleECUJob various Logical Links can be created. Thereby all these locations (e.g. different ECUs) can be used inside this Job. The Job writer must handle the creation and destruction of the location objects similarly as the used services. The Logical Links select the access paths.

When using the method createLogicalLinkByAccessKey for a MultipleECUJob the parameter for ShortNamePhysicalVehicleLink shall be ignored.

Figure 115 depicts the relation between MCDMultipleECUJob to MCDResponse (MCDDbResponse).
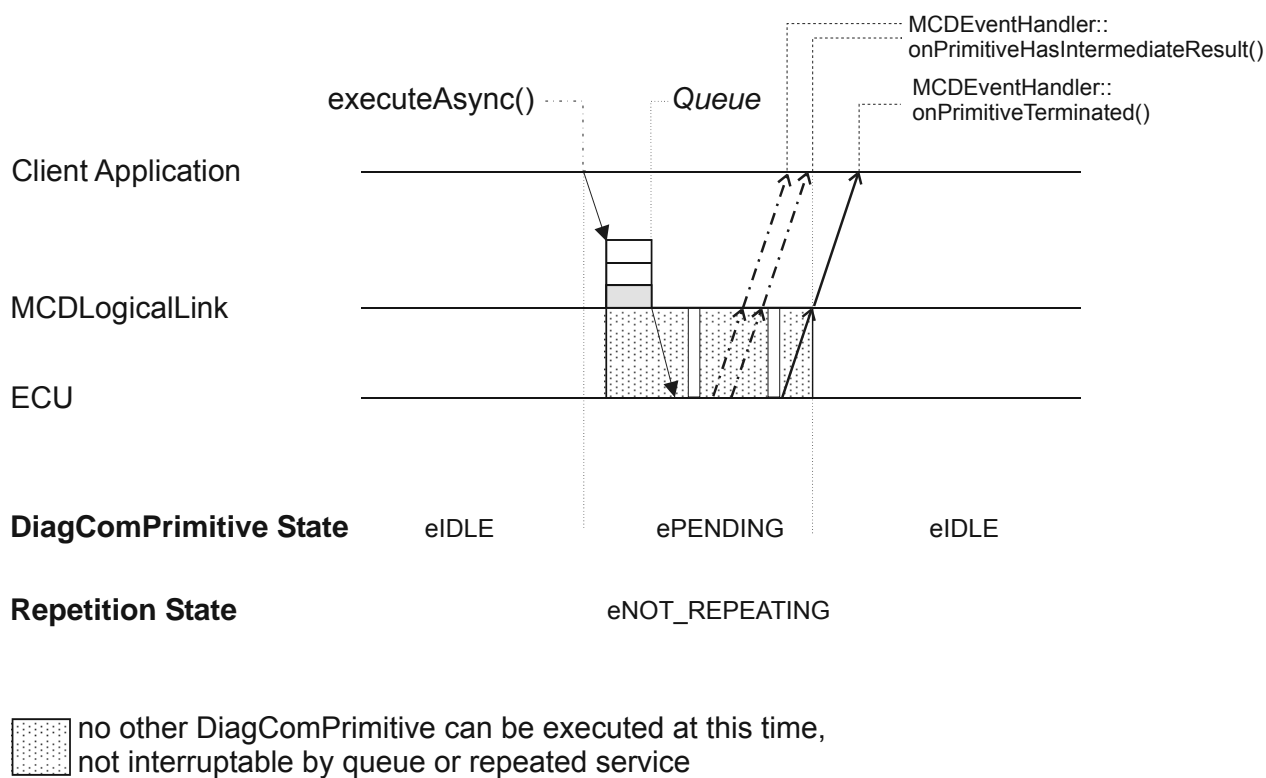

Figure 115 — Relation between MCDMultipleECUJob to MCDResponse (MCDDbResponse)

A MultipleEcuJob delivers one runtime Response in the Result for the one and only existing DbResponse.
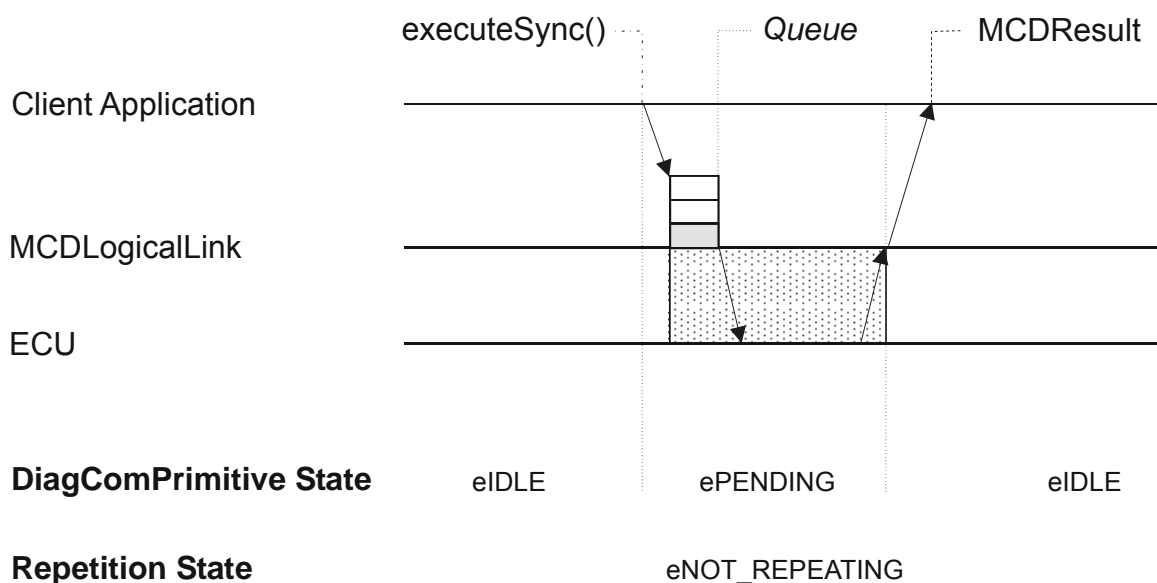
8.19.7 Job execution

8.19.7.1   Single execution of a Job

Figure 116 depicts the Job execution asynchronous.

Figure 116 — Job execution asynchronous

Figure 117 depicts the Job execution synchronous.

executeSync()   *Queue*   MCDResult

Client Application

MCDLogicalLink

ECU

DiagComPrimitive State   eIDLE   ePENDING   eIDLE

Repetition State   eNOT_REPEATING

no other DiagComPrimitive can be executed at this time,
not interruptable by queue or repeated service

Figure 117 — Job execution synchronous

Sample: normal execution of a Single ECU Job

DiagComPrimitive method description

executeAsync()   asynchronous start of Job execution

executeSync()   synchronous start of Job execution

cancel()   quit Job execution as fast as possible or remove it from execution queue

Remark:

It is allowed to execute a job itself in a synchronous way, but in this way no intermediate results were delivered.

Intermediate results can only be delivered if the job is executed in asynchronous way.

All diagnostic services or jobs which will be executed <u>inside</u> a job had to be started synchronous (`executeSync()`).

States:

The repetition state of the Job execution is `eNOT_REPEATING`.

The DiagComPrimitive state changes from eIDLE (initially; state before starting Job execution) to `ePENDING` (state while execution) back to `eIDLE` (state after execution). The states of the DiagComPrimitive are set by the MVCI diagnostic server.

<u>Results:</u>

There can be only 0 or 1 FINAL result and several (0..n) INTERMEDIATE results.

All results (intermediate and final) are stored in a ring buffer.

Intermediate results can be complete result sets of the used diagnostic services. They are independent from the final job result.

Requesting the result state or the result itself is allowed after getting the event `onPrimitiveHasIntermediateResult()`.

Inside a job only non cyclic single diagnostic services and jobs can be executed and these must be started synchronous (see Figure 49).

Figure 118 depicts the service execution inside job(Part1).

Figure 118 — Service execution inside job(Part1)

Figure 119 depicts the service execution inside job(Part2).

Figure 119 — Service execution inside job(Part2)

Every service or job inside a job has to be executed synchronously, because there is no possibility to get the event announcing the termination of the work of the Service or job and transporting the result state. The synchronous execution returns the result state as the return value, so that it can be evaluated. Mostly, the results specially the values from the results of the executed services will be used to create a job result (intermediate or final).

The next sequence diagram shows the execution of a job in a job. The job started by the Client will be called OuterJob. The job started by the OuterJob will be called InnerJob. The InnerJob is handled by the OuterJob like a service. The execution of the source code of the InnerJob is like any job execution, but no intermediate result will be evaluated by the OuterJob because of synchronous execution of the InnerJob.
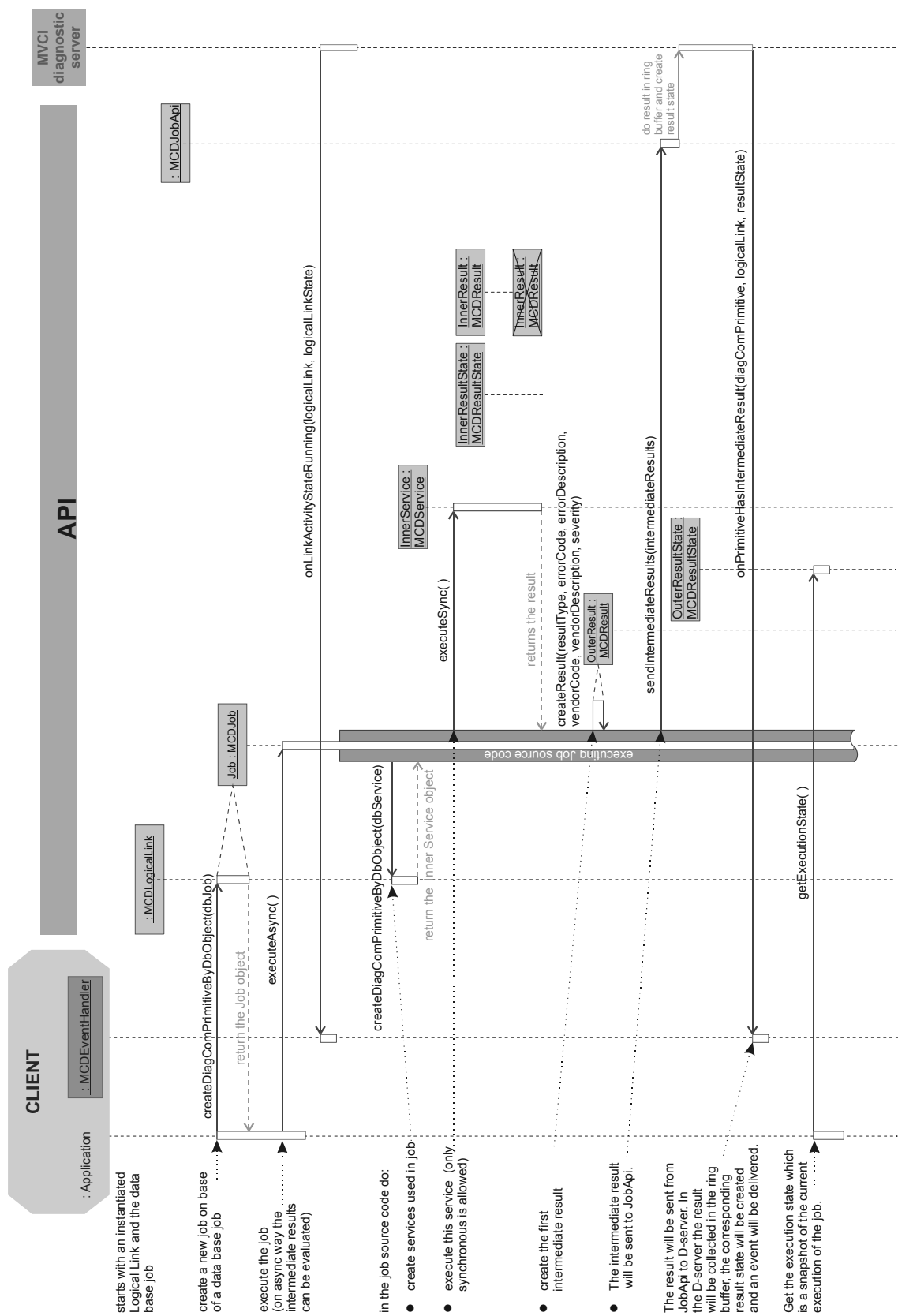
Figure 120 depicts the Job execution inside job(Part1).

Figure 120 — Job execution inside job(Part1)

Figure 121 depicts the Job execution inside job(Part2).

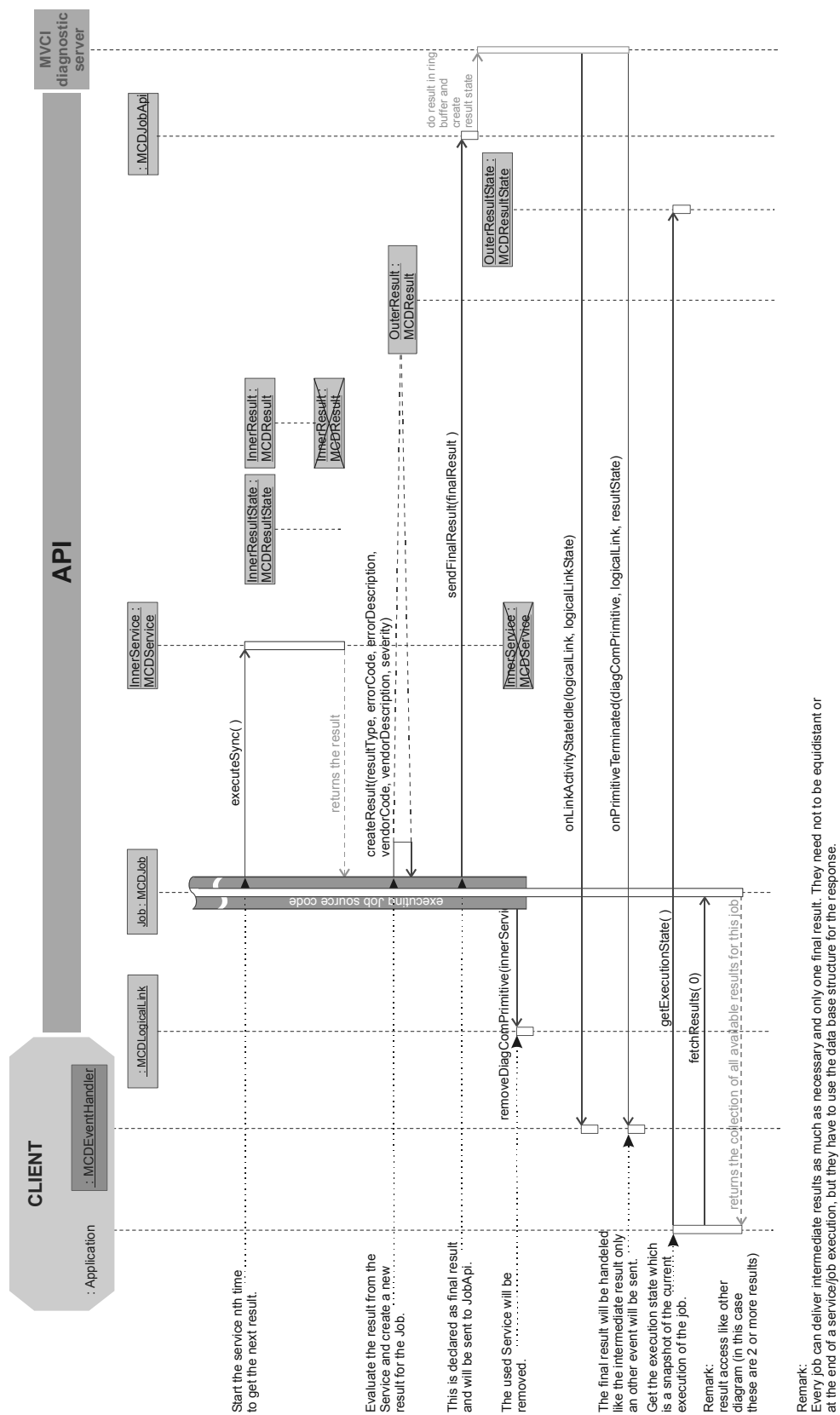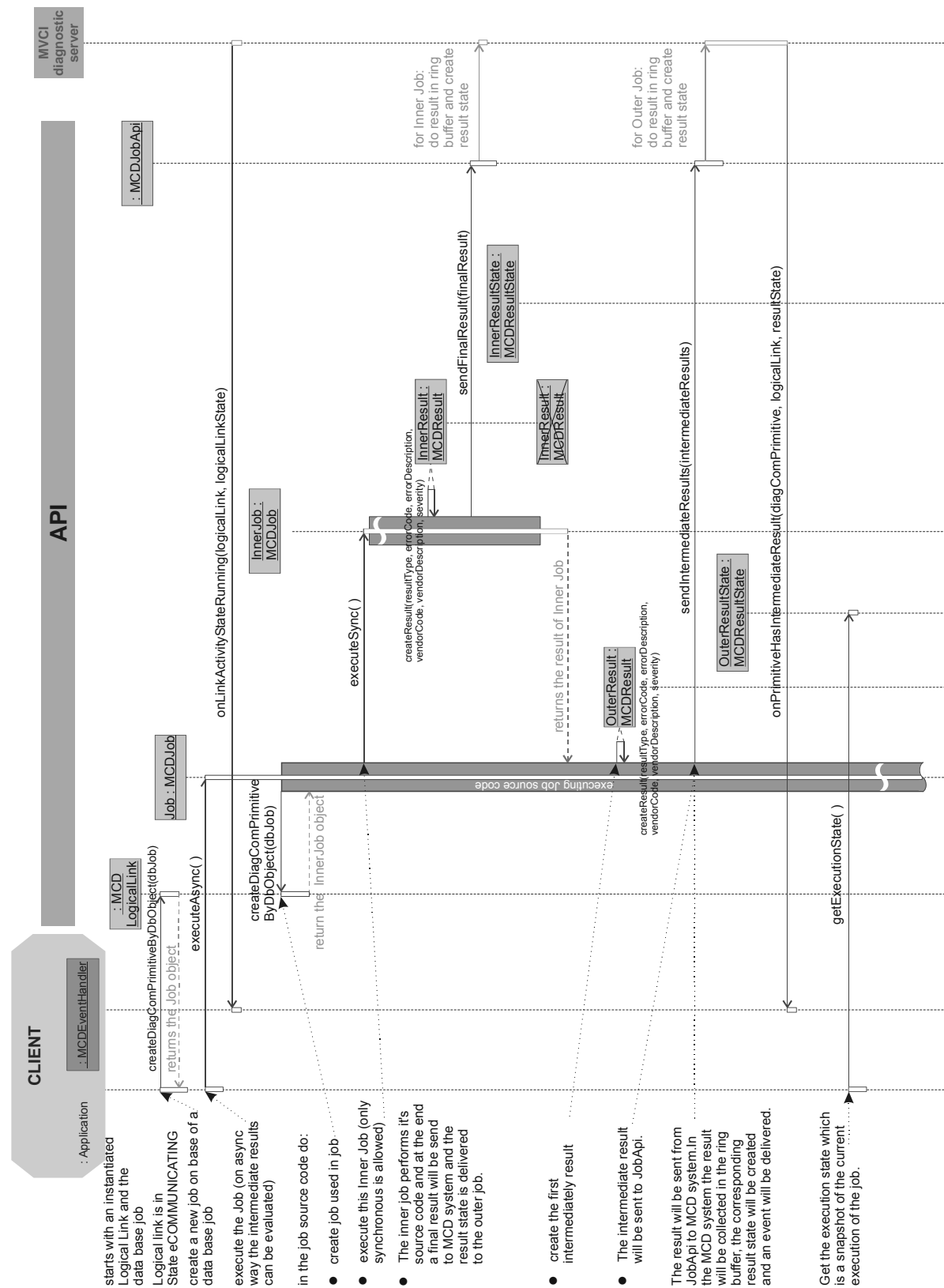**Figure 121 — Job execution inside job(Part2)**

8.19.7.2   Repeated execution of Job

Jobs can be executed repeated, so that a job is repeated multiple times without having a loop statement in the job. That is, the job does not "know" that it is executed multiple times in a loop. As a result, every execution delivers one single result (as an final result). Jobs and Services act identically in case of repeated execution.

Figure 122 depicts the Job execution repeated.

Figure 122 — Job execution repeated

Description: Repeated execution of a Job

The time between two repeated executions will be set by Client Application and is not stored in data base.

DiagComPrimitive method description:

startRepetition()	start of DiagComPrimitive execution, after passing the queue, the job will live in a loop and start action (not through the queue)
stopRepetition()	quit DiagComPrimitive execution
cancel()	quit DiagComPrimitive execution as fast as possible
updateRepetitionParameter()	in the state eIDLE the job parameter can be changed, this method goes not through the queue

States:

The repetition state changes from `eNOT_REPEATING` (before `startRepetition()`) to `eREPEATING` (after `startRepetition()` and back to `eNOT_REPEATING` (after `stopRepetition()` or `cancel()`).

The DiagComPrimitive state changes from `eIDLE` to `ePENDING` is made every time the Client Application starts a method, that goes through the queue till end of this method (e.g. `startRepetition()`), not for repeated execution, `updateRepetionParameters` and `stopRepetition`.

Results:

There can be one or more results, stored in the ring buffer. For each completed Job execution exactly one result is returned by the Job processor. The result of a Job is based on a DB template, which can be either a positive or (in future) a negative template. In case of repeated execution, every execution cycle can result in a different result because each execution of a Job is independent of previous executions. This result is passed on to the JobAPI using `sendFinalResult`. After the result has been entered to the result ring buffer by the MVCI diagnostic server, the sending of the event `onPrimitiveHasResult` takes place. Additionally, there might occur intermediate results at the Job execution. These results are reported to the JobAPI by means of `sendIntermediateResults` and after being entered to the result buffer, these results will be indicated to the application by sending the event `onPrimitiveHasIntermediateResult`. The sending of intermediate results is Job specific and is an addition to the repeated execution of DiagServices. All results (intermediate and final results) have to correspond to the Response Database Template of the Job. Requesting the result state, the number of results or the result(s) is allowed after getting one of the events `onPrimitiveHasResult` or `onPrimitiveRepetitionStopped`.

8.19.8 Allowed java libraries

A Java job usually is executed within the runtime environment of the MVCI diagnostic server. Therefore, any lock-ups, memory leaks, exception conditions or performance issues caused by job code can potentially degrade the MVCI diagnostic server performance or even render it completely useless. For this reason, it is strongly suggested that a Java job (including the associated libraries defined in the ODX data) only uses the external libraries listed in Table 29.

Table 29 — Allowed Java Libraries

Package	Allowed Classes	Description
Java.lang	Boolean, Byte, Character, Character.Subset, Character.UnicodeBlock, Class, Double, Float, Integer, Long, Math, Number, Object, Short, StrictMath, String, StringBuffer, Throwable, Void, ArithmeticException, ArrayIndexOutOfBoundsException, ArrayStoreException, ClassCastException, ClassNotFoundException, CloneNotSupportedException, Exception, IllegalAccessException, IllegalArgumentException, IllegalMonitorStateException, IllegalStateException, IllegalThreadStateException, IndexOutOfBoundsException, InstantiationException, InterruptedException, NegativeArraySizeException, NoSuchFieldException, NoSuchMethodException, NullPointerException, NumberFormatException, RuntimeException, SecurityException, StringIndexOutOfBoundsException, UnsupportedOperationException, AbstractMethodError, AssertionError, ClassCircularityError, ClassFormatError, Error, ExceptionInInitializerError, IllegalAccessError, IncompatibleClassChangeError, InstantiationError, InternalError, LinkageError, NoClassDefFoundError, NoSuchFieldError, NoSuchMethodError, OutOfMemoryError, StackOverflowError, ThreadDeath, UnknownError, UnsatisfiedLinkError, UnsupportedClassVersionError, VerifyError, VirtualMachineError	Provides classes that are fundamental to the design of the Java programming language.
Java.math	All classes allowed	Provides classes for performing arbitrary-precision integer arithmetic (BigInteger) and arbitrary-precision decimal arithmetic (BigDecimal).
Java.text	All classes allowed	Provides Classes and interfaces for handling text, dates, numbers, and messages in a manner independent of natural languages.
Java.util	ArrayList, Arrays, BitSet, Collections, Date (no deprecated methods), GregorianCalendar, HashMap, HashSet, Hashtable, LinkedList, Random, SimpleTimeZone, Stack, StringTokenizer, TimeZone, TreeMap, TreeSet, Vector, WeakHashMap, ConcurrentModificationException, EmptyStackException, MissingResourceException, NoSuchElementException, TooManyListenersException	Contains the collections framework, legacy collection classes, event model, date and time facilities, internationalisation, and miscellaneous utility classes (a string tokeniser, a random-number generator, and a bit array).
asam.mcd	All classes allowed	Provides interfaces for interacting with the MVCI diagnostic server
asam.d	All classes allowed	Provides interfaces for interacting with the MVCI diagnostic server
asam.job	All classes allowed	Provides interfaces for interacting with the MVCI diagnostic server

NOTE   the usage of external libraries is potentially harmful, as the MVCI diagnostic server is not able to control or restrict what is being done by job code. Therefore, server vendors are not to be held liable for any damage caused by a Java job using external libraries.

Java jobs are allowed to use standard Java class inheritance mechanisms. The general rules and practices of Java programming and class loading apply, e.g. it is recommended to use packages to avoid naming conflicts. For further information on Java programming guidelines and relevant style guides, please refer to the Sun Java documentation pages.

To be able to use packages with Java Jobs, classes have to be given full-qualified in the source code, Java code has to be given including subdirectories (path of packages), and for jar-files the entry point has to be given full-qualified (+ jar-file including path).

8.19.9 Naming conventions

Job files use the name: MCD3_*jobname.java*. The name is built with the MCD3 prefix and the job name. The major version, minor version and revision must be placed in the @version attribute of the Java source code.

Shortname of the Job object at the API and the name of the Job file (source code) can differ. If more than one source code is assigned to a Job these source codes needs to be semantically equivalent. As this ist not decidalble by any algorithm, the data engeneer has to take care that this rule is fulfilled.

In the diagnostic server, the base directory of Java Jobs should be configurable to be able to extract the correct package structure from a Java code's path description.

8.19.10 Job Communication Parameter handling

If a Java job needs to alter the currently valid communication parameters, it should use and execute a `MCDProtocolParameterSet` primitive within its code. Please note that all changes to communication parameters caused by a `MCDProtocolParameterSet` executed within a Java job will be persistent after the job has terminated – just as if the application would have issued the same change. However, the usage of MCDProtocolParameterSets in a Java job's code is considered harmful as this can cause undocumented and therefore unexpected changes to the communication parameters of a logical link at runtime.

Please note that in contrast to `DIAG-COMMs`, `SINGLE-ECU-JOBs` and `MULTIPLE-ECU-JOBs` cannot have local communication parameters. As a result, the MVCI diagnostic server is not required to handle local and overwritten communication parameters for Java jobs.

8.19.11 Job Result Generation

The construction of a job's result object structure is achieved by a set of methods provided by the MCDJobApi class. These methods and their usage are described in this section. Because a job has one set of positive response parameters and one set of negative response parameters, it has to be decided within the job's source code whether a positive or a negative response is to be created before calling `MCDResponses::add(MCDDbLocation dbLocation, boolean isPositive = true)`. By setting the isPositive flag to either true or false, it is possible to create a negative as well as a positive response within job source code.

The starting point for a job result is its database template. Based upon this, a corresponding result structure is created the same way as for ‗normal' diagnostic services. In case of dynamic result elements (fields/arrays, multiplexers and environment data), the job code has additional possibilities when creating the result structure:

⎯ Fields let the job create an arbitrary number of elements of one given sub element.

⎯ A multiplexer (MUX) lets the job choose between one of many branches.

⎯ The same applies for env-data elements, which are a part of DTC handling.

These dynamic elements can occur at any level of the result structure.

If a dynamic element has to be created, the job has to choose which and how many of the selectable sub elements are to be included in the response structure. In case of an field (array), an element is added using the `MCDResponseParameters::addElement()` and `MCDResponseParameters::addElementWithContent(…)` method. In case of a dynamic element of the type multiplexer, the respective branch is added using the

`MCDResponseParameters::addMuxBranch*(…)` method. On the level of an element of type `eENVDATADESC`, the method `MCDResponseParameters::addEnvDataByDTC(A_UNIT32)` is used to insert an `eENVDATA` block.

Other than that, complex (structured) elements of type `eENVDATADESC` and `eSTRUCTURE` are handled like any other simple element and are inserted into the response structure in the same way.

Table 30 defines the methods for result construction in Jobs.

Table 30 — Methods for result construction in Jobs

DOP Type	MCDResponseParameter Methods	Used by DataType
Complex DOP	addElement	for Fields
	addElementWithContent	
	addEnvDataByDTC	at EnvDataDesc
	addMuxBranch	for MuxBranch
	addMuxBranchByIndex	
	addMuxBranchByIndexWithContent	
	addMuxBranchByMuxValue	
	addMuxBranchWithContent	
	setParameterWithName	for simple DOPs
Simple DOP	setvalue	for simple DOPs

A job's result structure has to be constructed anew for each job run. However, it is possible to reuse result objects for intermediate results by temporarily storing and copying the result structure(s) within the job code.

When creating, adding or selecting a multiplexer branch to a result structure, the MVCI diagnostic server verifies the relevant data using the job's database template. That way result structure integrity is tested at each step of response construction, and it is guaranteed that the output format corresponds to the database template.

In case of some of the complex DOPs (eFIELD, eMULTIPLEXER, eSTRUCTURE, eENVDATA), the internal values of these elements (which the client application retrieves by calling the `MCDResponseParameter.getValue()` method of the elements that correspond to these complex DOPs) are updated internally by the MVCI diagnostic server's job processor; They are not to be updated within the job source code.

An `MCDResponseParameter` used as a parameter for an `addXXX(…)` method needs to comply to the corresponding `MCDDbResponseParameter` definition of the database response object (structure, types, ranges). Otherwise, the `addXXX(…)` method throws an `MCDException`. If the passed parameter value does not comply with the database template of the parameter, no information is copied.

Rules for `MCDResponseParameters::addElementWithContent(…)` - Content is copied for:

⎯ MCDError and Error Availability,

⎯ MCDResponseParameters,

⎯ MCDValue;

Rules for `MCDResponseParameters::addMuxBranchByIndexWithContent(…)`:

— The target `MCDResponseParameter` is of type `eMULTIPLEXER`. Otherwise an `MCDException` will be thrown.

— The parameter index is the index of the target MUX-branch. It selects the MUX-branch in the DB-template of the target `MCDResponseParameter` which is to be used for validity checks.

— The elements of the `MCDResponseParameters` collection used as the content parameter represent the (complex) values of the top-level elements of the target MUX branch, in the order (by index) they are placed in the collection. That is, the values of the first element inside the target MUX branch are copied from the first element in the source collection.

— For copying the content of every element in the source collection to the corresponding element in the target MUX branch, the same rules apply as for `MCDResponseParameters::addElementWithContent(…)`.

Rules for `MCDResponseParameters::addMuxBranchWithContent(…)`:

— The target is of type eMULTIPLEXER.

— The branch parameter identifies the branch of the target MUX-branch. That is, it selects the MUX-branch in the DB-template of the target `MCDResponseParameter` which is to be used for validity checks.

— The elements of the MCDResponseParameters collection used as the content parameter represent the (complex) values of the top-level elements of the target MUX branch, in the order (by index) they are placed in the collection. That is, the values of the first element inside the target MUX branch are copied from the first element in the source collection.

— For copying the content of every element in the source collection to the corresponding element in the target MUX branch, the same rules apply as for `MCDResponseParameters::addElementWithContent(…)`.

Rules for `MCDResponseParameters::addEnvDataByDTC(…):parameters`:

— Adds a single structured response parameter of type `eENVDATA` to the collection of response parameters.

— Parent element of this response parameter needs to be of type `eENVDATADESC`. Otherwise, an MCDException will be thrown

— Similarly to `addMuxBranch()`, an ‚empty' response parameter structure is added up to the first dynamic element. Empty means that no parameter values have been filled in.

— Returns the response parameter of type `eENVDATA` which was added to the collection of response parameters. Parameters with parameter type `eTABLE_KEY` are handled like other simple parameters. For parameters of type `eTABLE_STRUCT`, it is distinguished between the following two cases:

  — The corresponding `eTABLE_KEY` parameter has a valid `MCDValue` see a)

    In this case, an `eTABLE_STRUCT` parameter is handled like a static complex parameter (data type `eSTRUCTURE`). Thus, all elements will be automatically inserted in the created result structure up to the first dynamic result element.

  — The corresponding eTABLE_KEY parameter has no valid MCDValue see b)

    In this case, an eTABLE_STRUCT parameter is handled like a dynamic complex parameter. That means, its sub-elements will not be automatically inserted in the result structure. Thus, a call to getParameters()

for this eTABLE_STRUCT parameter will deliver an empty collection. As soon as a valid MCDValue is set for the corresponding eTABLE_KEY parameter, the sub-elements are filled in and the result structure and can be retrieved by calling the getParameters() method.

a) A parameter of type `eTABLE_KEY` will have a valid MCDValue in case of static parameter definition or in case of dynamic parameter definition where either a default value is defined in the database or a value is already set through the job at runtime.

b) An `eTABLE_KEY` parameter will have a not initialized MCDValue in case of dynamic parameter definition and neither a default value is defined in the database nor a value was set by the job at runtime.

8.19.12   Job template SingleEcuJob

This template is in the package called 'asam.job'.

```
/*
   * SingleEcuJobTemplate.java
   * SingleEcuJob
   * Created March 2005
   */

/**
   *
   * @author MVCI diagnostic server standardization group
   * @version 3.00.00
   */

package asam.job;

import asam.d.*;

public interface SingleEcuJobTemplate
{
   /** executes this SingleEcuJob */
   /** and sets the input parameters for this SingleEcuJob */
   public void execute
      (MCDRequestParameters inputParameters,
      MCDJobApi jobHandler,
      MCDLogicalLink link,
      MCDSingleEcuJob apiJobObject) throws MCDException;
}
```

8.19.13   Job template MultipleEcuJob

This template is in the package called 'asam.job'.

```
/*
   * MultipleEcuJobTemplate.java
   * MultipleEcuJob
   * Created March 2005
   */
/**
   *
   * @author MVCI diagnostic server standardization group
   * @version 3.00.00
```

```
    */
package asam.job;
import asam.d.*;

public interface MultipleEcuJobTemplate
{
    /** executes this MultipleEcuJob */
    /** sets the input parameters for this MultipleEcuJob */
    public void execute
        (MCDRequestParameters inputParameters,
        MCDJobApi jobHandler,
        MCDLogicalLink link,
        MCDMultipleEcuJob apiJobObject,
        MCDProject project) throws MCDException;
}
```

8.19.14   Job template FlashJob

This template is in the package called 'asam.job'.

```
/*
    * FlashJobTemplate.java
    * FlashJob
    * Created March 2005
    */
/**
    *
    * @author MVCI diagnostic server standardization group
    * @version 3.00.00
    */

package asam.job;
import asam.d.*;

public interface FlashJobTemplate
{
    /** executes this FlashJob */
    /** sets the input parameters for this FlashJob */
    public void execute
        (MCDRequestParameters inputParameters,
        MCDJobApi jobHandler,
        MCDLogicalLink link,
        MCDFlashJob apiJobObject,
        MCDDbFlashSession session) throws MCDException;
}
```

8.20 ECU configuration

8.20.1 Introduction

ECU configuration, also known as variant coding, describes the data elements and the process of configuring an ECU – either in the vehicle or in a test bench. The feature of ECU configuration is optional in a diagnostic

server. That means that a diagnostic server implementation may not support ECU configuration and still be considered standard conformant.

In ECU configuration, an ECU is integrated into its vehicle environment (functional environment as well as electrical environment). For this purpose, two different types of configuration information can be distinguished – functional configuration and non-functional configuration information. By means of functional configuration information, setup information on the electrical and digital network within a vehicle is written to ECUs. For example, the presence of a GPS antenna system can be notified to an infotainment ECU. Furthermore, software features can be enabled and disabled by means of functional configuration information. In contrast, non-functional configuration information comprises all kinds of additional information which does not change a vehicles behaviour or functionality. For example, the vehicle colour or the interior colour can be written to an ECU by means of non-functional configuration information. This non-functional information can be used to, e.g. identify the correct replacement part in the service workshops.

Both, functional and non-functional configuration information are passed to or read from an ECU by means of configuration strings where each configuration string typically contains more than one piece of configuration information. That is, a single configuration string contains information on a set of configuration items where each configuration item either represents a functional or a non-functional configuration information. Indeed, the difference between functional and non-functional configuration information is not visible in a configuration string any more. Technically, a configuration string is represented by a byte array.

8.20.2 ECU Configuration database part

In this section, the database part of the ECU configuration functionality of the diagnostic server API is described. As this database part is closely related to ODX structures, the ODX elements which correspond to a certain type of MVCI diagnostic server object are given in parentheses. The central terms and structures of the ECU configuration database part are illustrated in Figure 123. For more detail on the ODX representation of ECU configuration data, see ODX specification ISO 22901-1.

For describing the structure and the content of a certain type of configuration string, each configuration string is associated with a database pattern. Such a database pattern is represented by an object of type `MCDDbConfigurationRecord` (CONFIG-RECORD). Multiple `MCDDbConfigurationRecord` objects can be contained in a configuration data container of type `MCDDbConfigurationData`. A `MCDDbConfigurationRecord` can have a unique identifier associated, the so-called configuration identifier (CONFIG-ID). The configuration ID allows to directly address a specific `MCDDbConfigurationRecord` in the scope of a `MCDDbConfigurationData`.

Every `MCDDbConfigurationRecord` can be composed of objects of type `MCDDbConfigurationItem` (CONFIG-ITEM). A `MCDDbConfigurationItem` represents a single piece of configuration information.

Figure 123 depicts the Terms and Structure ECU Configuration Database Part.

Figure 123 — Terms and Structure ECU Configuration Database Part

The following subclasses of `MCDDbConfigurationItem` exist, to be able to distinguish different types of configuration items:

— `MCDDbConfigurationIdItem` (CONFIG-ID-ITEM) – defines the position in a configuration string and the bytes that will be occupied by the configuration identifier of a `MCDDbConfigurationRecord` at runtime.

— `MCDDbDataIdItem` (DATA-ID-ITEM) – defines the position in a configuration string and the bytes that will be occupied by the data identifier of a `MCDDbDataRecord` that has been used to define a configuration string at runtime.

— `MCDDbSystemItem` (SYSTEM-ITEM) – defines the position and the bytes in a configuration string that will be filled with the value of the system parameter referenced by this element.

— `MCDDbOptionItem` (OPTION-ITEM) – defines the position and the bytes in a configuration string that represent a certain configuration option. Option items represent functional or non-functional configuration information which can be altered by the user. Similarly to request or response parameters, a value domain is associated with an option item.

In contrast to any other kind of `MCDDbConfigurationItem`, `MCDDbOptionItems` provide a physical default value and optionally a set of `MCDDbItemValue` objects. Similarly to a text table in case of a request or response parameter, this set of type `MCDDbItemValues` defines an enumeration of possible values of a `MCDDbOptionItem`. Every `MCDDbItemValue` has a constant physical value and optionally a meaning and a description. The physical value is the value to be placed in a configuration string at runtime, if the corresponding item value is selected. The meaning provides a human readable phrase which illustrates this option item's value. For example, the meaning 'available' could be assigned to a `MCDDbItemValue` which is associated with a `MCDDbOptionItem` named 'FogLights'.

The description of a `MCDDbItemValue` gives more elaborate information on the value's meaning and can be used to describe the effect caused in the ECU when setting this value. Both, meaning and description can be internationalised as already known from LONGNAMEs and DESCRIPTIONs in case of other MVCI diagnostic server objects. That is, an ID can be defined in the ODX data which is to be resolved against an external data space for the internationalised string.

Furthermore, a `MCDDbItemValue` can optionally have a key and a rule assigned. The key is a unique identifier of a `MCDDbItemValue` within its superior option item. It can be used to obtain a certain `MCDDbItemValue` without knowing its ShortName.

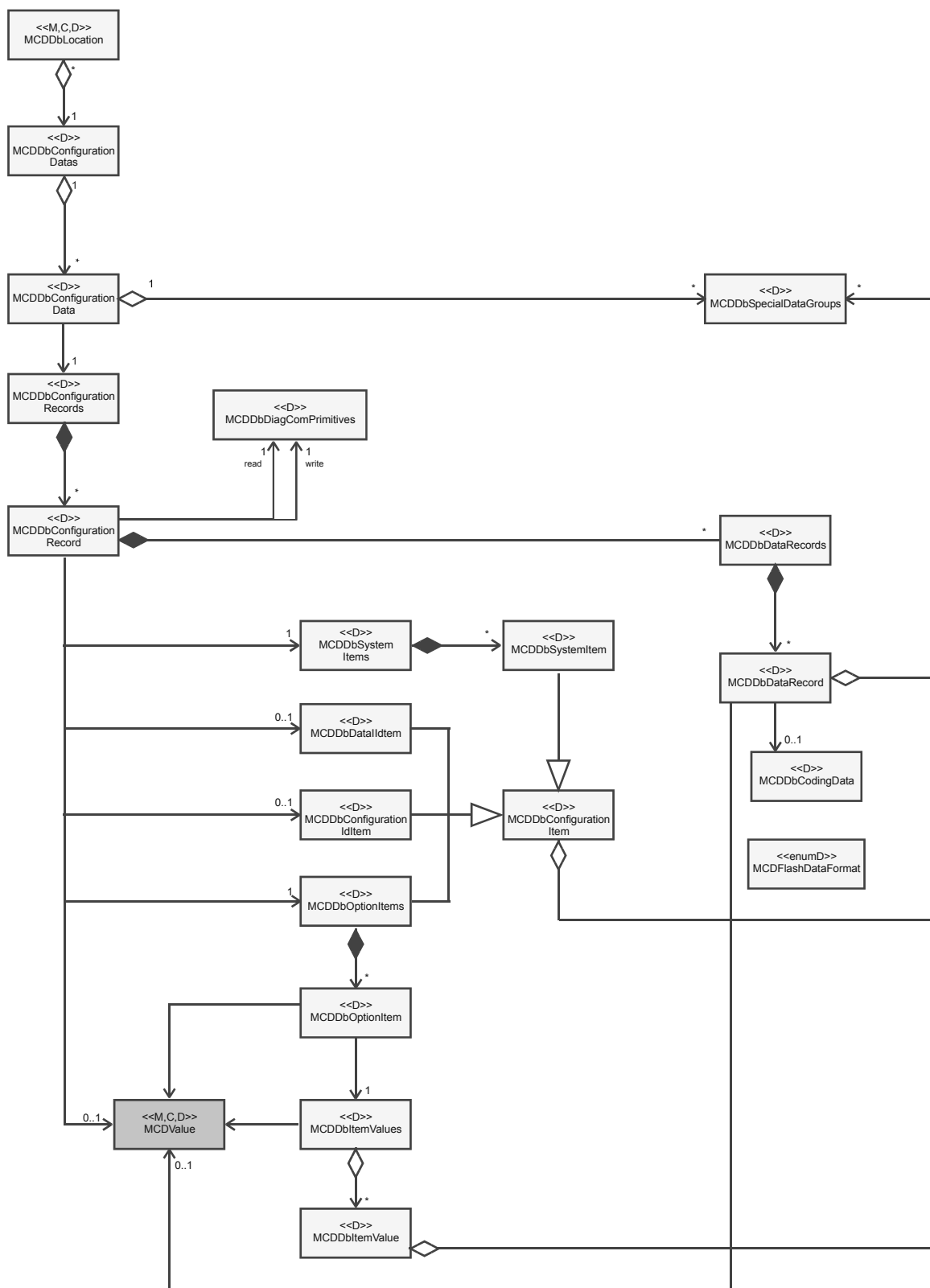Figure 124 depicts the ECU Configuration Model – Database part.

Figure 124 — ECU Configuration Model – Database part

While Figure 124 shows the classes that provide the database part of the ECU Configuration feature, Figure 125 indicates audience-related meta-data as regards these elements.
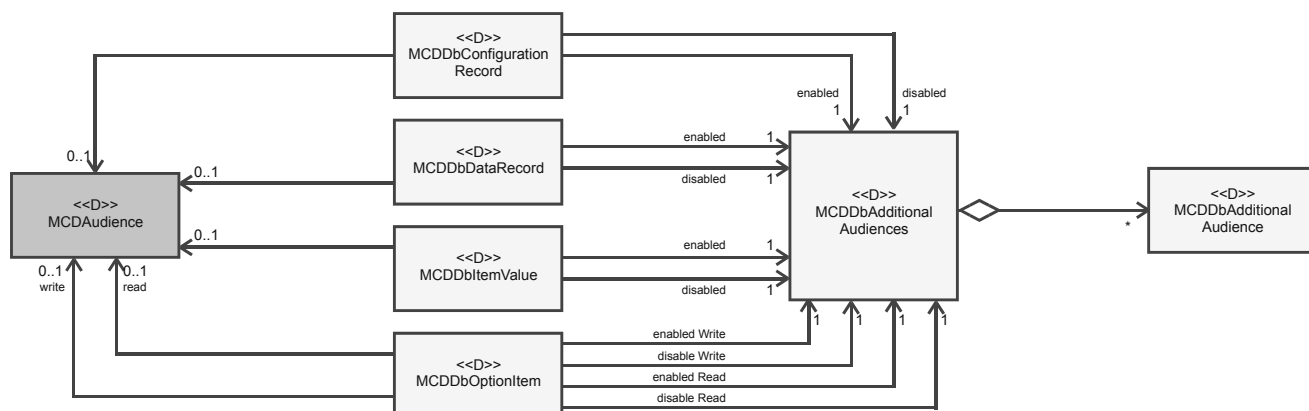

Figure 125 — MCDAudience

Default configuration strings which comply to a `MCDDbConfigurationRecord` are represented by objects of type `MCDDbDataRecord` (DATA-RECORD). That is, for every CONFIG-RECORD, the ODX data can contain zero or more default values, each represented by its own DATA-RECORD. To be able to uniquely identify and address a certain default configuration string, a `MCDDbDataRecord` is associated with a so-called data identifier (DATA-ID). Alternatively, the key assigned to a `MCDDbDataRecord` can be used.

Inside a `MCDDbDataRecord`, the configuration string is stored as binary data. This binary data can either be directly contained in the ODX data or it can be placed in an external file. In case of an external file, which is represented by an element of type `MCDDbCodingData`, the filename can be marked as 'late-bound'. Similarly to flash data, a late-bound file, is loaded by a diagnostic server as late as possible. That is, it needs to be loaded latest when the binary data is accessed for the first time. Furthermore, the filename of a late-bound data file can contain wildcards. The wildcards which can be used in late-bound data files should conform to basic regular expressions (BRE) as supported by IEEE Std 1003.1-2001 [5]. In this case, the actual filename needs to be resolved at runtime. The filenames for data files in ODX should be given as Uniform Resource Identifiers (URIs). See RFC 3305 [5] and related documents for more details. In this case, the method `MCDDbDataRecord::getBinaryData()` does not return any data. For late-bound data files with wildcards in the filename, only the runtime part of ECU configuration is able to resolve the name.

The binary data contained in a `MCDDbDataRecord` or referenced by a `MCDDbCodingData`, respectively, can be formatted. The following data formats are supported (see flash programming also):

— Intel-Hex Records – formatted binary data including segment (address) information.

— Motorola-S Records – formatted binary data including segment (address) information.

— Plain binary data – binary large object without any segment or address information.

Segmented binary data referenced from a `MCDDbDataRecord` (intern or external) shall define values for at least all addresses from zero to size of configuration record minus 1. For each of these addresses, exactly one value shall be given, i.e. no gaps and no overlapping segments are allowed.

The size of a configuration record (configuration string) in bytes is the maximum of the sums of the SOURCE-START-ADDRESS and the UNCOMPRESSED-SIZE from its DIAG-COMM-DATA-CONNECTORs given by the ODX data.

If the value returned by `MCDDbCodingData::isLateBound()` is 'false' at a reference to an external resource file (e.g., job code, flash data, coding data) is considered a guarantee that the content of this resource file will not change while a diagnostic server is running. More precisely, the content of the external resource file shall be static for the time between `MCDSystem::selectProjectXXX()` and `MCDSystem::deselectProject()` for the same project.

Please note that exchanging external resource files may lead to non-deterministic behavior of a diagnostic server. In particular, this statement holds in case of the LATEBOUND-DATAFILE attribute is set to 'true'.

8.20.3 ECU Configuration Runtime Part

The database part of ECU configuration merely describes the structure and the possible values of configuration strings or configuration items associated with an ECU. The ECU configuration runtime part provides the classes to create, modify, read, and write configuration strings. The corresponding classes are shown in Figure 127. An example illustrating the basics is shown in Figure 126.

Figure 126 — Terms and Structure ECU Configuration runtime part

At runtime, a single configuration string is represented by the value of an object of type `MCDConfigurationRecord`. As a configuration string is specific to a `MCDDbLocation` of a certain ECU Base Variant or ECU Variant, every `MCDConfigurationRecord` is stored in a collection of type `MCDConfigurationRecords` which is a member of a `MCDLogicalLink`. New configuration strings are created by adding a new `MCDConfigurationRecord` to a `MCDConfigurationRecords` collection. Existing configuration strings can be discarded by removing the corresponding `MCDConfigurationRecord` from the collection which contains this record. On removing a `MCDConfigurationRecord` from the collection, contained DiagComPrimitives will also be removed.

Figure 127 depicts the ECU Configuration Model – Runtime part.

Figure 127 — ECU Configuration Model – Runtime part

Similarly to the database part, a `MCDConfigurationRecord` can be composed of objects of type `MCDConfigurationItem`. Again, different types of `MCDConfigurationRecords` exist:

— `MCDConfigurationIdItem` – defines the position in a configuration string and the bytes that will be occupied by the configuration identifier of the `MCDDbConfigurationRecord` a MCDConfigurationRecord is based on. The value of this element cannot be altered by a client application. Instead, the value is defined in the ODX data and will be automatically inserted into a configuration string by the diagnostic server at runtime. In case of an upload of configuration information from an ECU, the value of a configuration ID item is read from the ECU.

— `MCDDataIdItem` – defines the position in a configuration string and the bytes that will be occupied by the data identifier of a `MCDDbDataRecord` that has been used to define a configuration string at runtime.

The value of this element can only be indirectly altered by a client application. The value is defined in the ODX data and will be automatically inserted into a configuration string by the diagnostic server at runtime whenever a `MCDDbDataRecord` is used to set the value of a `MCDConfigurationRecord` (see below). In case of an upload of configuration information from an ECU, the value of a data ID item is read from the ECU.

⎯ `MCDSystemItem` – defines the position and the bytes in a configuration string that will be filled with the value of the system parameter referenced by this element. The value of this element cannot be altered by a client application. Instead, the value is calculated and will be inserted into a configuration string by the diagnostic server at runtime. In case of an upload of configuration information from an ECU, the value of a system item is read from the ECU.

⎯ `MCDOptionItem` – defines the position and the bytes in a configuration string that represent a certain configuration option. Option items represent functional or non-functional configuration information which can be altered by a client application. Similarly to request or response parameters of a diagnostic service, a value domain is associated with an option item. In case of an upload of configuration information from an ECU, the value of an option item is read from the ECU.

The value of a `MCDConfigurationRecord` can be set in different ways

⎯ By setting the value free-form (`setConfigurationRecord (A_BYTEFIELD configRecordValue)`);

⎯ By setting the value by means of a selected `MCDDbDataRecord` (`setConfigurationRecordByDbObject(MCDDbDataRecord dbDataRecord)`);

⎯ By setting appropriate values at the `MCDOptionItem` objects referenced from a `MCDConfigurationRecord`;

In case of the former two options – entering a value free-form or overwriting the value with an `MCDDbDataRecord` –, the diagnostic server needs to re-calculate the values of all `MCDConfigurationItem` objects contained in the current `MCDConfigurationRecord`, by decomposing and translating the value of the `MCDConfigurationRecord` appropriately. Whenever the value of a `MCDOptionItem` is changed (see below), the value of the containing `MCDConfigurationRecord` needs to be re-calculated by the diagnostic server from the values of all contained `MCDConfigurationItem` objects.

The value of a `MCDOptionItem` can be changed either by entering a new value free-form (`MCDOptionItem::setItemValue(MCDValue newValue)`) or by assigning the value of a `MCDDbItemValue` (`MCDOptionItem::setItemValueByDbObject(MCDDbItemValue dbItemValue)`). In case a new `MCDValue` object is written to a `MCDOptionItem`, the value represented by this `MCDValue` object needs to match the value range of the `MCDOptionItem`. That is, the method `MCDOptionItem::setItemValue(MCDValue newValue)` can fail because of the following reasons:

⎯ The data type of the value represented by the `MCDValue` object does not match the data type of the `MCDOptionItem`

⎯ The value represented by the `MCDValue` object is not within the value range (interval) defined for the `MCDOptionItem`. This value range can be obtained via `MCDOptionItem::getDbObject()::getInterval()`.

⎯ The value represented by the `MCDValue` object cannot be converted into a corresponding coded value. That is, the conversion associated with the corresponding `MCDDbOptionItem` in ODX failed.

— The value represented by the `MCDValue` object cannot be translated into a corresponding `MCDDbItemValue`. This translation does only take place in case the collection `MCDDbItemValues` is not empty at the corresponding `MCDDbOptionItem`. In this case, the method `MCDOptionItem::getMatchingDbItemValue()` throws an exception (`MCDProgramViolationException`, `eRT_ELEMENT_NOT_AVAILABLE`).

In case the value of a `MCDOptionItem` is changed by assigning a `MCDDbItemValue`, the value of the `MCDOptionItem` is defined as the physical default value of this `MCDDbItemValue`. The method `MCDOptionItem::setItemValueByDbObject(MCDDbItemValue dbItemValue)` fails, if the `MCDDbItemValue` supplied as parameter is not defined for the `MCDDbOptionItem` which corresponds to the current `MCDOptionItem`. However, if an `MCDOptionItem`'s value can be set by using an `MCDDbItemValue`, the method `MCDOptionItem::getMatchingDbItemValue()` does deliver the `MCDDbItemValue` which corresponds to the current value of the `MCDOptionItem`.

8.20.4 Error Handling

On initialization or when setting a ConfigurationRecord, various errors can occur. Regarding to the `MCDResponse`, the method `MCDConfigurationRecord::hasErrors()` returns 'true', if the ConfigurationRecord is faulty. The containing `MCDErrors` can be requested via method `MCDConfigurationRecord::getErrors()`.

In case of a ConfigurationRecord initialization or uploading a configuration string from an ECU to a ConfigurationRecord that contains ConfigurationItems, e.g. OptionItems, internal to physical value conversion errors can occur. Such an error will be added to the corresponding ConfigurationItem. Regarding to the `MCDResponseParameter`, the `MCDConfigurationItem` offers the methods `hasError()` and `getError()`.

If one of the ConfigurationItems of the ConfigurationRecord has an error or the ConfigurationRecord could not be initialized with its DataRecord, at least one error with the error code eRT_CONFIGRECORD_INVALID is contained in the ConfigurationRecord. The error will be reset when the configuration string becomes valid, regarding to the ODX template.

If the execution of a ConfigurationRecord's Write-/ReadDiagComPrimitive contains errors, these errors are also adopted to the ConfigurationRecord. The Write-/ReadDiagComPrimitives of the ConfigurationRecord need to be executed in the order, given from ODX data. If the Write-/ReadDiagComPrimitives are executed in the wrong order, an error with the error code eRT_WRONG_SERVICE_EXECUTION_ORDER is added to the ConfigurationRecord. If one but not all ReadDiagComPrimitives are executed, the error with the error code eRT_CONFIGRECORD_INCOMPLETE is added to the ConfigurationRecord. Note that the incomplete execution of WriteDiagComPrimitives does not modify the ConfigRecord. In this case, the error code eRT_CONFIGRECORD_INCOMPLETE is not added to the ConfigRecord.

The errors of a ConfigurationRecord and its ConfigurationItems will be reset when one of the following methods from a `MCDConfigurationRecord` object is called: `setConfigurationRecord (A_BYTEFIELD configRecordValue)`, `loadCodingData(A_ASCIISTRING fileName)`, `setConfigurationRecordByDbObject(MCDDbDataRecord dbDataRecord)`, `getReadDiagComPrimitives()`, `getWriteDiagComPrimitives()`.

An error of a ConfigurationItem will be reset when the ConfigurationItem is set with a new value. This is the case when the configuration string of a ConfigurationRecord is set, e.g. via `MCDConfigurationRecord::setConfigurationRecord (A_BYTEFIELD configRecordValue)`. In case of an OptionItem, the new value can also be set via the `MCDOptionItem` methods `setItemValue(MCDValue value)` and `setItemValueByDbObject(MCDDbItemValue dbItemValue)`.

8.20.5 Initialising an MCDConfigurationRecord

Whenever a new `MCDConfigurationRecord` is created at a Logical Link, the diagnostic server needs to initialise the value of the configuration record such that at first valid configuration string is created within this configuration record. The initialisation of such a `MCDConfigurationRecord` shall be performed as defined by the following rules (only one rule can apply to a `MCDConfigurationRecord`):

— If no `MCDDbDataRecord` objects are defined for the `MCDDbConfigurationRecord` the `MCDConfigurationRecord` is based on and if no `MCDDbOptionItems` and no `MCDDbItemValues` are given in the ODX data for this `MCDConfigurationRecord`, the `MCDConfigurationRecord`'s value (`MCDValue` of type A_BYTEFIELD) will not be initialized with a value. As a result, the method "`hasError`" returns 'true' for this bytefield.

— If no `MCDDbDataRecords` and no `MCDDbItemValues` are given for the current `MCDConfigurationRecord` in the ODX data but if this `MCDConfigurationRecord` contains `MCDOptionItems`, then the `MCDConfigurationRecord`'s value will be initialized with the physical default values of all these option items. These physical default values are mandatory for every `MCDDbOptionItem`.

— If no `MCDDbDataRecords` are given in the ODX data for the `MCDDbConfigurationRecord` the current `MCDConfigurationRecord` is based on but if option items and item values are given, the `MCDConfigurationRecord`'s value will be initialized with the physical default values of all option items. The physical default value shall match a `MCDDbItemValue` of the corresponding `MCDDbOptionItem`.

— If exactly one `MCDDbDataRecord` is referenced from the `MCDDbConfigurationRecord` the current `MCDConfigurationRecord` is based on and if no option items or item values are given, then this `MCDDbDataRecord` will be used to initialise the `MCDConfigurationRecord`'s value. In this case it is not relevant if a DEFAULT-DATA-RECORD (`MCDDbConfigurationRecord::getDbDefaultDataRecord()`) references a `MCDDbDataRecord` or not.

— If more than one `MCDDbDataRecord` is referenced from the `MCDDbConfigurationRecord` the current `MCDConfigurationRecord` is based on, a DEFAULT-DATA-RECORD `MCDDbConfigurationRecord::getDbDefaultDataRecord()` references a `MCDDbDataRecord` and if no option items or item values are given, then this `MCDConfigurationRecord`'s value will not be initialised with this bytefield. As a result, the method "`hasError`" returns 'true' for this bytefield.

— If exactly one `MCDDbDataRecord`, `MCDDbOptionItems`, and `MCDDbItemValues` are given for the `MCDDbConfigurationRecord` the current `MCDConfigurationRecord` is based on, then the `MCDConfigurationRecord`'s value will be initialised with the `MCDDbDataRecord`. In this case it is not relevant if a DEFAULT-DATA-RECORD `MCDDbConfigurationRecord::getDbDefaultDataRecord()` references a `MCDDbDataRecord` or not.

— If more than one `MCDDbDataRecord`, `MCDDbOptionItems`, and `MCDDbItemValues` are given for the `MCDDbConfigurationRecord` the current `MCDConfigurationRecord` is based on, then the `MCDConfigurationRecord`'s value will be initialised with the physical default values of all option items. In this case it is not relevant if a DEFAULT-DATA-RECORD `MCDDbConfigurationRecord::getDbDefaultDataRecord()` references a `MCDDbDataRecord` or not.

8.20.6 Offline versus Online Configuration

Two different use cases have been defined in the context of ECU configuration – offline configuration and online configuration. For offline configuration, it is required to be able to define new configuration strings or to modify existing configuration strings without communicating with the corresponding ECU, e.g. because it is currently not available in a vehicle or a test bench. In terms of MVCI diagnostic server, offline configuration means that it shall be possible to create or modify a `MCDConfigurationRecord`, e.g. by assigning a new value to this `MCDConfigurationRecord` or one of its `MCDConfigurationItem` objects. To support offline configuration, it is possible to add new `MCDConfigurationRecord` objects to the `MCDConfigurationRecords` collection associated with a `LogicalLink` while this link is in state `eCREATED`. In the same state, it is also possible to modify a `MCDConfigurationRecord` contained in the collection. In logical link state `eCREATED`, no communication to an ECU is running.

For online configuration, it is required to be able to initialise the value of a `MCDConfigurationRecord` object with the current configuration string as stored in the corresponding ECU. Online configuration is supported in a diagnostic server by being able to use the same `MCDConfigurationRecord` to read a configuration string from an ECU (upload, see 8.20.7.4) and to write this configuration string back to the ECU after modification (download, see 8.20.7.3). Furthermore, it is possible to initialise a new `MCDConfigurationRecord` with the value of a default configuration string, that is, with the value of a `MCDDbDataRecord` (see 8.20.5). As a result, the following scenarios in online configuration can be realised in a client application:

⎯ upload of current configuration data from an ECU,

⎯ upload of current configuration data from an ECU and modification of this configuration data,

⎯ modification of a configuration record before download,

⎯ upload, modification and download of a configuration record,

⎯ initialisation of a configuration record with a selected data record,

⎯ initialisation of a configuration record with a selected data record followed by a download,

⎯ … .

8.20.7 Uploading and Downloading Configuration Strings

8.20.7.1   Basics

Configuration strings can be written to an ECU (download) and can be read from an ECU (upload). This section describes how the upload and download of configuration strings can be performed with a diagnostic server.

In general, a configuration string can exceed the maximum size of a D-PDU which can be transferred to and from an ECU via a LogicalLink and the diagnostic protocol the communication via this link is based on. Based on the ODX schema, a configuration string can be transferred to and from an ECU in several fragments.

8.20.7.2   Decomposing a Configuration String for Transfer

Every configuration string is represented by a `MCDConfigurationRecord` in the diagnostic server. A configuration string is transferred to and from an ECU by means of the DiagComPrimitives referenced from such a `MCDConfigurationRecord`. For each of the directions upload and download, a separate set of DiagComPrimitives is referenced (see Figure 124 and Figure 127) – a set of WriteDiagComPrimitives (download) and a set of ReadDiagComPrimitives (upload).

If a configuration string exceeds the maximum size that can be transferred with a single DiagComPrimitive, each of the sets of ReadDiagComPrimitives and WriteDiagComPrimitives contains one DiagComPrimitive for every piece the configuration strings needs to be decomposed into. The information on which piece is to be transferred by which DiagComPrimitive and the size of these pieces is described by means of so-called DIAG-COMM-DATA-CONNECTORs in the ODX data (see Figure 168 "UML representation of ECU configuration: DIAG-COMM-DATA-CONNECTOR" in the ISO 22901-1). These DIAG-COMM-DATA-CONNECTORs are only used internally within a diagnostic server, they are not exposed at the server's API.

Every DIAG-COMM-DATA-CONNECTOR refers to a piece of configuration string, a DiagComPrimitive, request and/or response parameters, and optionally elements of type READ-PARAM-VALUE or WRITE-PARAM-VALUE. The piece of the configuration string is identified by a start address in the configuration string (starting at zero) and an uncompressed size of the piece in bytes (see Figure 128 for illustration). The DiagComPrimitive (read or write) to be used as well as the request parameter to place the piece of configuration string in (WriteDiagComPrimitive) or the response parameter to read the piece of configuration string from (ReadDiagComPrimitive) are referenced directly. READ-PARAM-VALUEs can be used to obtain additional information on the current piece of configuration string, e.g. its number, from the response of a ReadDiagComPrimitive. WRITE-PARAM-VALUEs can be used to put additional information on the current piece of configuration string, e.g. its number, in the request of a WriteDiagComPrimitive.

Figure 128 — Example of a configuration string which is decomposed for transfer

8.20.7.3 Downloading configuration records to an ECU

A single configuration string is written to an ECU, i.e. downloaded to this ECU as follows. First the collection of `MCDWriteDiagComPrimitives` needs to be obtained from the corresponding `MCDConfigurationRecord` by means of the method `getWriteDiagComPrimitives()`. The content of this collection is defined as the DiagComPrimitives referenced from the corresponding WRITE-DIAG-COMM-CONNECTORs in the ODX data.

The elements in the `MCDWriteDiagComPrimitives` collection of DiagComPrimitives are ordered, regarding to the order in the ODX data. If this collection contains more than one element, this indicates that

the configuration string represented by the current `MCDConfigurationRecord` is too large to be transferred to the ECU by means of a single DiagComPrimitive. That is, the size of the configuration string exceeds the maximum size of a DiagComPrimitive's request in the current diagnostic protocol. In this case, the configuration string needs to be decomposed into pieces as described in section 8.20.7.2.

In the next step, all WriteDiagComPrimitives referenced by this configuration record need to be executed in the order defined in the `MCDWriteDiagComPrimitives` collection obtained in the first step. Therefore, the request parameters of every WriteDiagComPrimitive in this collection need to be filled with data as described by the corresponding DIAG-COMM-DATA-CONNECTOR. All request parameters in the WriteDiagComPrimitive which are not explicitly filled with a value from the information in the corresponding DIAG-COMM-DATA-CONNECTOR shall set to default values by the diagnostic server. If a WriteDiagComPrimitive cannot be executed after all request parameters have been set as described above, e.g. because at least one request parameter does not have a default value, the execution of a WriteDiagComPrimitive fails (MCDParameterizationException, ePAR_INCOMPLETE_PARAMTERIZATION) and, as a consequence, the download of the configuration string fails.

WriteDiagComPrimitives can be executed either synchronously or asynchronously. However, the diagnostic server does not execute any WriteDiagComPrimitives automatically. Instead, the client application is responsible for executing the WriteDiagComPrimitives in the correct order. Only the preparation of the request parameters is performed by the diagnostic server.

A sample workflow for writing a single configuration string to an ECU is shown in Figure 129. In this example, a `MCDDbDataRecord` is obtained from the corresponding `MCDDbConfigurationRecord` first. Then, a new ConfigurationRecord is created at the runtime LogicalLink. Next, the value of this ConfigurationRecord is set by using the `MCDDbDataRecord` obtained in the first step. Finally, the value of the ConfigurationRecord, that is, the configuration string, is written to the ECU. For this purpose, all WriteDiagComPrimitives referenced at the ConfigurationRecord are executed in the correct order.
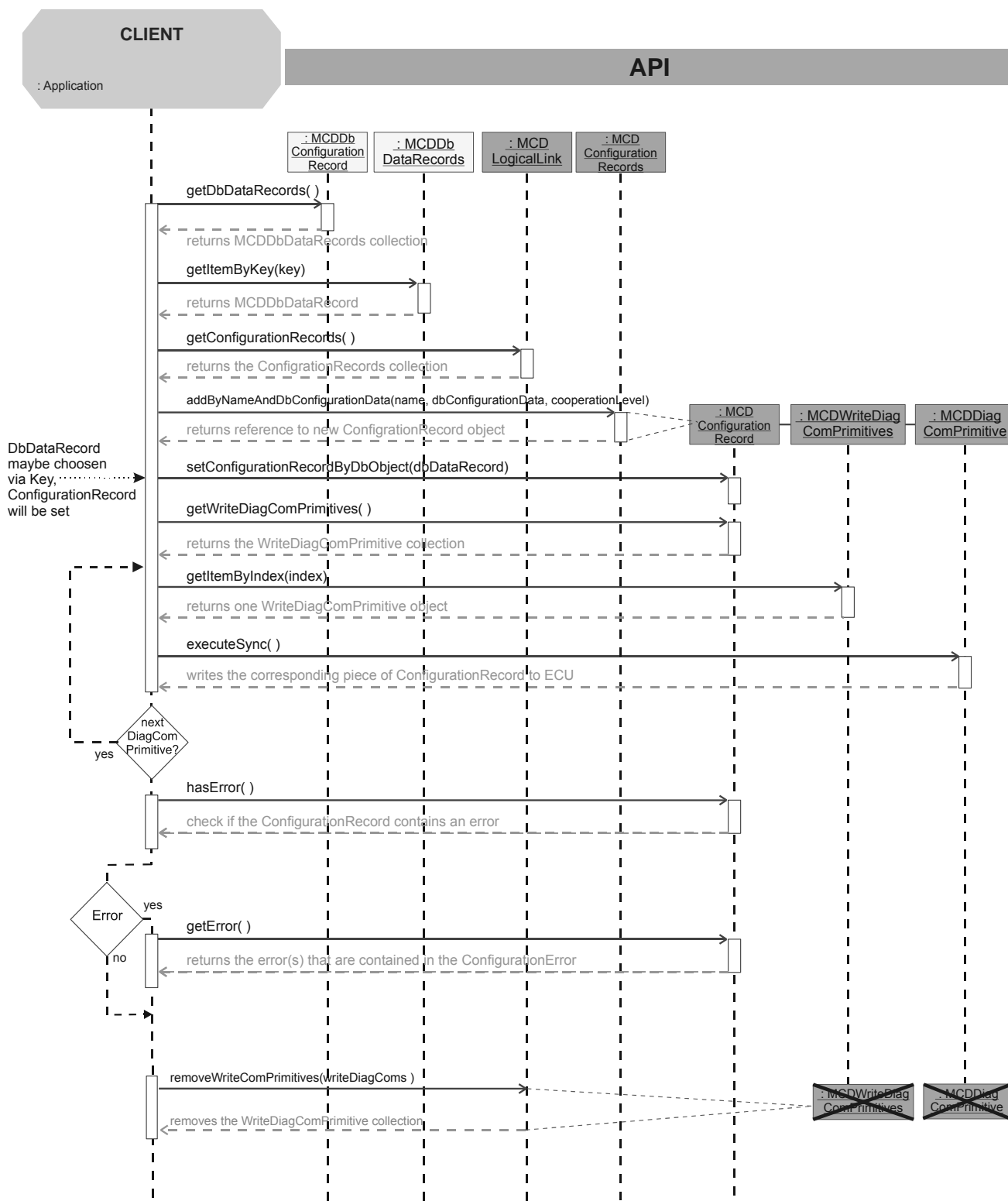
Figure 129 — Example workflow for writing a configuration string to an ECU

8.20.7.4 Uploading configuration records from an ECU

A single configuration string is read from an ECU, i.e. uploaded from this ECU as follows. First the collection of `MCDReadDiagComPrimitives` needs to be obtained from the corresponding `MCDConfigurationRecord` by means of the method `getReadDiagComPrimitives()`. The content of this collection is defined as the DiagComPrimitives referenced from the corresponding READ-DIAG-COMM-CONNECTORs in the ODX data.

The elements in the `MCDReadDiagComPrimitives` collection of DiagComPrimitives are ordered, regarding to the order in the ODX data. If this collection contains more than one element, this indicates that the configuration string represented by the current `MCDConfigurationRecord` is too large to be transferred from the ECU by means of a single DiagComPrimitive. That is, the size of the configuration string exceeds the maximum size of a DiagComPrimitive's response in the current diagnostic protocol. In this case, the configuration string needs to be composed from the different pieces of configuration information read from the ECU.

In the next step, all ReadDiagComPrimitives referenced by this configuration record need to be executed in the order defined `MCDReadDiagComPrimitives` collection obtained in the first step. Therefore, the request parameters of every ReadDiagComPrimitive in this collection need to be filled with data as described by the corresponding DIAG-COMM-DATA-CONNECTOR. All request parameters in the ReadDiagComPrimitive which are not explicitly filled with a value from the information in the corresponding DIAG-COMM-DATA-CONNECTOR shall set to default values by the diagnostic server. If a ReadDiagComPrimitive cannot be executed after all request parameters have been set as described above, e.g. because at least one request parameter does not have a default value, the execution of a ReadDiagComPrimitive fails (`MCDParameterizationException`, `ePAR_INCOMPLETE_PARAMTERIZATION`) and, as a consequence, the upload of the configuration string fails.

ReadDiagComPrimitives can be executed either synchronously or asynchronously. However, the diagnostic server does not execute any ReadDiagComPrimitive automatically. Instead, the client application is responsible for executing the ReadDiagComPrimitives in the correct order. Only the preparation of the request parameters is performed by the diagnostic server.

In the final step, the diagnostic server needs to write the configuration information which has been read from the ECU into the current `MCDConfigurationRecord`. For this purpose, the method `setConfigurationRecord (A_BYTEFIELD configRecordValue)` can be used by a diagnostic server implementation internally. Prior to setting the value of the `MCDConfigurationRecord`, the value needs to be created by concatenating the values of all those response parameters which are referenced from the ReadDiagComPrimitives in the `MCDReadDiagComPrimitives` collection. The concatenation needs to take place in the order of execution as defined by the `MCDReadDiagComPrimitives` collection. That is, the return value of the second ReadDiagComPrimitive is appended to the return value of the first ReadDiagComPrimitive, the third value is appended to the second, and so forth.

A sample workflow for reading a single configuration string from an ECU is shown in the following figure. In this example, a new ConfigurationRecord is created at a LogicalLink first. Then, the value of this ConfigurationRecord is read from the ECU by means of the ReadDiagComPrimitives referenced from this ConfigurationRecord. Finally, the (byte) value of the configuration string is read from this ConfigurationRecord as well as the value and the meaning of every single OptionItem which is contained in the ConfigurationRecord.

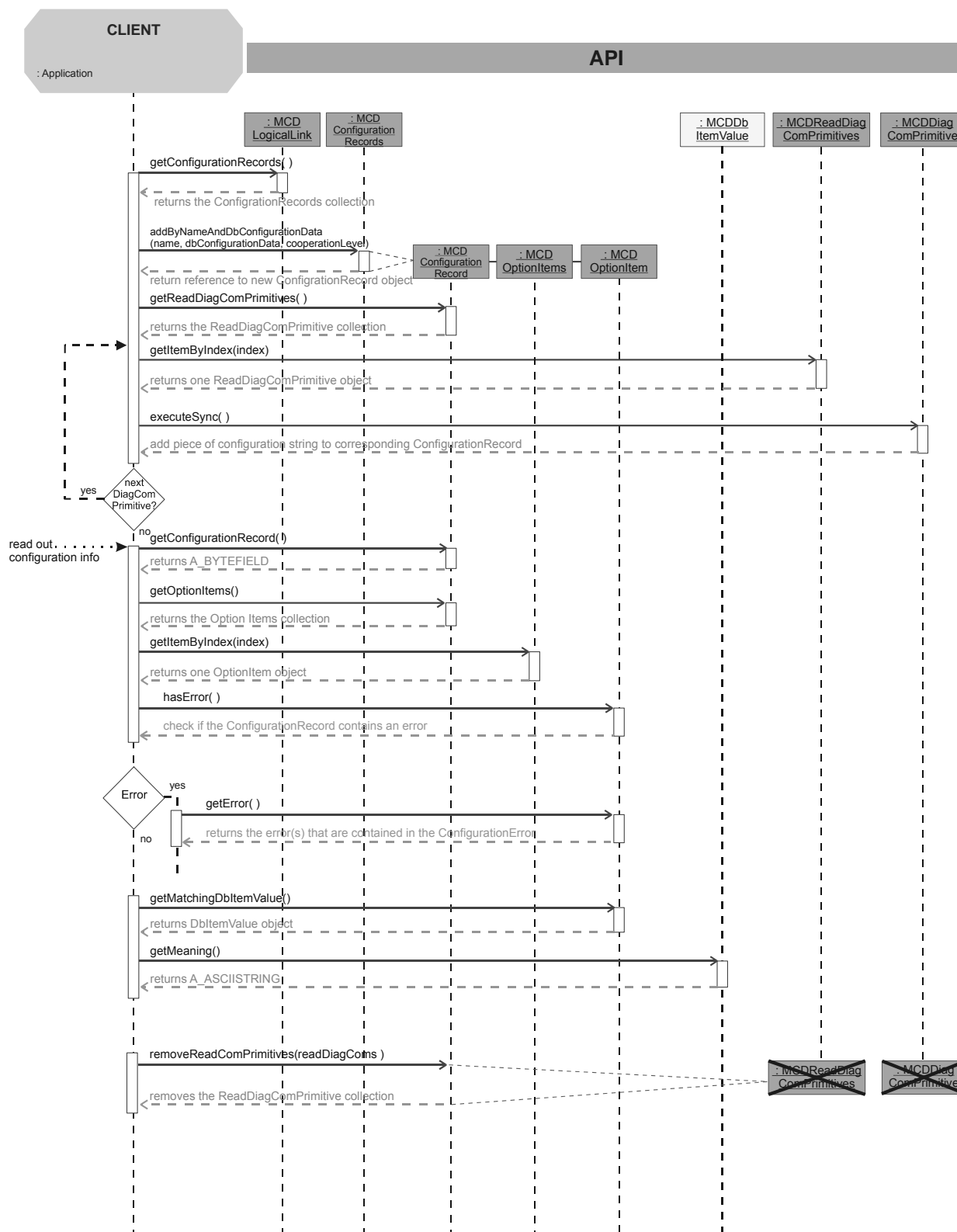Figure 130 depicts an example workflow for reading a configuration string from an ECU.

Figure 130 — Example workflow for reading a configuration string from an ECU

8.21 Audiences and additional audiences

8.21.1 General

Audiences and additional audiences make it possible to supply information about access restrictions for MCD/D objects with respect to predefined target groups of users. For example, audience definitions can be used to declare that MVCI diagnostic server database objects should only be available to a certain class of clients (e.g. service testers). The roles that can be handled by the standard audience filter are `AfterMarket`, `AfterSales`, `Development`, *Manufacturing* and *Supplier*. The feature of additional audiences (see below) can be used to define additional roles to extend the standard audience definitions.

Filtering of diagnostic server database objects according to audience and additional audience settings shall always be done by the client application. The reason for this is that it is not possible for the diagnostic server to handle all cases where audience definitions can be applied to ODX data in a meaningful way. For example, server-side filtering of database objects according to their audience settings could lead to the following scenarios:

— Diagnostic services, which have been available at engineering time, might disappear at runtime (caused by the diagnostic server filtering by audience settings). As a result, ECUs might be damaged, for example when the filtered diagnostic service is necessary to finish an ECU reprogramming session.

— With respect to related services, audience attributes at DataPrimitives are potentially harmful, because a DataPrimitive that is defined as a precondition to another DiagComPrimitive might disappear at runtime (when it gets filtered out as a result of its audience settings).

— Also, creating ODX data which contains audience information that is to be used for variant identification or base variant identification could result in undesirable runtime behaviour: In case the diagnostic server executes one of these services, and in case these services in turn subsume DataPrimitives that have an associated audience attribute which disallows them in the current diagnostic server instance, the (base) variant identification service will fail.

— Even more subtle complications could result by the diagnostic server filtering out specific datablocks from a flash session, or coding fragments from an ECU configuration data set.

Out of these considerations, it has been decided that audience settings will be accessible at the diagnostic server API, but that any kind of filtering or program logic depending on audience settings will have to be implemented by the client application. Audience settings can be defined for the following MCD/D objects (ODX element is given in parentheses):

— `MCDDbDataPrimitive (DIAG-COMM, MULTIPLE-ECU-JOB)`

— `MCDDbConfigurationRecord (CONFIG-RECORD)`

— `MCDDbItemValue (ITEM-VALUE)`

— `MCDDbOptionItem (OPTION-ITEM)`

— `MCDDbFlashDataBlock (FLASH-DATABLOCK)`

— `MCDDbDataRecord (DATA-RECORD)`

— `MCDDbFunctionNodeGroup (FUNCTION-NODE-GROUP)`

— `MCDDbBaseFunctionNode (FUNCTION-NODE)`

— `MCDDbTableParameter (TABLE-ROW)`

8.21.2 Audiences

Audiences reflect a predefined set of user groups for diagnostic data. The following user groups have been defined in ODX:

— Supplier,

— Development,

— Manufacturing,

— AfterSales,

— AfterMarket;

In the MVCI diagnostic server, the current access status with respect to these audiences is represented by an object of type `MCDAudience` which is returned by the method `getAudienceState()` available at the database objects listed above. Within the class `MCDAudience`, the access status with respect to each of the user groups is represented by a boolean attribute which can be queried by means of a corresponding method:

— `isSupplier(),`

— `isDevelopment(),`

— `isManufacturing(),`

— `isAfterSales(),`

— `isAfterMarket();`

Each of the attribute values can be set to "true" or "false" in ODX. If no information on the audience access status of an element is available in ODX, then this element's access status defaults is "true" for all five user groups. That means that the corresponding MCDAudience object is generated by the diagnostic server, and delivers "true" for all of its status methods.

8.21.3 Additional Audiences

In addition to the predefined audiences, so-called additional audiences can be referenced from the MCD/D objects listed in the introduction of this section. These additional audiences allow to expand or to redefine the list of user groups that are subject to audience restrictions. The additional audiences in MVCI diagnostic server are represented by objects of type `MCDDbAdditionalAudience`. Every object of this type represents one ODX element ADDITIONAL-AUDIENCE which is contained in a DIAGLAYER. Additional audiences define individual lists of users or user groups which can be subject to accessibility constraints on the corresponding diagnostic elements. A diagnostic element (e.g. a `MCDDataPrimitive`) may contain references to either enabled or disabled ADDITIONAL-AUDIENCE elements. By means of the method `getDbAdditionalAudiences()`, the additional audiences that are associated with one of the following elements can be listed and evaluated by a client application:

— `MCDDbLocation` (introduced at DIAG-LAYER in ODX and valid for DIAG-COMMs and MULTIPLE-ECU-JOBs)

— `MCDDbConfigurationData` (introduced at ECU-CONFIG in ODX and valid for CONFIG-RECORD, DATA-RECORD, ITEM-VALUE and OPTION-ITEM)

— `MCDDbEcuMem` (introduced at FLASH in ODX and valid for SESSION-DESC and DATA-BLOCK)

— `MCDDbFunctionDictionary` (introduced at FUNCTION-DICTIONARY in ODX and valid for FUNCTION-NODE-GROUP and FUNCTION-NODE)

At a specific MCD/D object (e.g. a `MCDDataPrimitive`), either enabled additional audiences or disabled additional audiences can be defined. That means that the group of users with access rights is either extended or restricted by additional audience definitions. In ODX, a corresponding element provides a so-called ENABLED-AUDIENCE-REF or a DISABLED-AUDIENCE-REF:

— The ENABLED-AUDIENCE-REF – represented by the method `MCDDbDataPrimitive.getDbEnabledAdditionalAudiences()` – means that this element, e.g. a DIAG-COMM, is only suitable for the referenced ADDITIONAL-AUDIENCEs.

— The "DISABLED-AUDIENCE-REF" – represented by the method `MCDDbDataPrimitive.getDbDisabledAdditionalAudiences()` – means that this element is not appropriate for the referenced ADDITIONAL-AUDIENCEs, but available for all other listed ADDITIONAL-AUDIENCEs.

As with the predefined audiences, additional audiences have to be evaluated by the client application. That is, the diagnostic server will not perform any access control on any MCD/D object depending on additional audience settings.

8.22 ECU states

ECUs in vehicles are stateful electronic components. They usually implement state machines for different purposes. The state changes in these state machines are triggered, e.g. by sensor data, by diagnostic requests or by internal functionality. Those parts of the state machines which are relevant in vehicle diagnostics can be modelled by means of state charts in ODX. For example, the possible session changes and the security states can be modelled as state charts in ODX. As a result of this focus, the session and security sub model in ODX mainly covers two aspects:

— The first is to describe possible state transitions resulting from the execution of a DiagComPrimitive.

— The second is to describe preconditions for the execution of a DiagComPrimitive.

— Figure 131 depicts the UML Class diagram of the ECU state chart sub model.

Figure 131 — UML Class diagram of the ECU state chart sub model

Both the session handling and the security handling are modelled within one generic state machine model. The element `MCDDbEcuStateChart` stores the possible states of an ECU (see Figure 131). The collection of all state charts valid at a certain DbLocation can be fetched using the method `MCDDbLocation::getDbEcuStateCharts()`. Every state within a state chart is represented by an own `MCDDbEcuState` object. The attribute 'semantic' associated with a `MCDDbEcuStateChart` defines the type of the state chart. The value of this attribute can be obtained by means of the method

`MCDDbEcuStateChart::getSemantic()`. The values "SESSION" and "SECURITY" are predefined for this semantic in ODX. However, the set of values of the attribute 'semantic' is extendible by the user.

Each state chart references all ECU states belonging to this state chart. One of these ECU states is the start state. This start state of a state chart can be obtained by means of the method `MCDDbEcuStateChart::getDbStartState()`. There is exactly one start state per state chart.

A state transition within one state machine is modelled by an object of type `MCDDbStateTransition`. A state transition references exactly one source state and exactly one target state. Self transitions can be described by state transitions with identical SOURCE and TARGET states. The state transitions belonging to an ECU state chart are also referenced from the corresponding `MCDDbEcuStateChart` object. Please note that ECU state charts need to be disjoint with respect to their collections of ECU states and state transitions. That is, ECU states and state transitions cannot be shared between two different ECU state charts.

State transitions are fired as a result, e.g. to

— ECU-internal logic,

— processing a request of a DiagComPrimitive,

— reception of specific sensor signals.

In the first case, the corresponding `MCDDbEcuStateTransition` object does neither provide a so-called external access method nor does it reference any state transition event in terms of an object of type `MCDDbStateTransitionAction`. In the second case, the state transition refers to at least one `MCDDbStateTransitionAction`. In the third case, the `MCDDbEcuStateTransition` provides information on an external access method.

External access methods are represented by objects of type `MCDDbExternalAccessMethod`. An external access method allows specifying an OEM-specific method necessary to perform the corresponding state transition. The textual information returned by the method `MCDDbExternalAccessMethod::getMethod()` needs to be interpreted by an OEM-specific extension of a diagnostic server. The method information can be used, e.g., in case of security-critical applications to link to protected functionality. Another example is to simulate a certain sensor signal in case of a test bench.

If the collection of `MCDDbEcuStateTransitionAction` objects which can be obtained using the method `getDbEcuStateTransitionActions()` at a `MCDDbEcuStateTransition` is non-empty, this has the following semantic: If the DiagComPrimitive referenced from one of the `MCDDbEcuStateTransitionActions` is successfully executed, the state of the ECU changes from the specified source state to the specified target state. In case that the `MCDDbEcuStateTransitionAction` refers to a request parameter and a value, the DiagComPrimitive needs to have this value set at the request parameter to fire the state transition after successful execution.

Sometimes, the execution of a DiagComPrimitive is only possible or successful if the target ECU is in a certain state, e.g. a certain session or a certain security state. In ODX, such preconditions can be modelled by means of PRE-CONDITION-STATE-REFs at a DIAG-COMM. PRE-CONDITION-STATE-REF can be used to define the allowed states for the execution of the DIAG-COMM in cases where the execution of the DIAG-COMM does not result in a state-transition of the ECU but its execution is bound to the condition that the ECU already is in a certain state. In the DB part of the diagnostic server API, a PRE-CONDITION-STATE-REF is represented by an object of type `MCDDbPreconditionDefinition`. Similarly to `MCDDbStateTransitionAction`, a `MCDDbPreconditionDefinition` can refer to a request parameter and a value. If both are set, the precondition definition is bound to this configuration of the referenced DiagComPrimitive. The DiagComPrimitives which are restricted by a certain ECU state can be obtained by means the method `MCDDbEcuState::getDbRestrictedDiagComPrimitives()`. This method returns a collection of MCDDbPreconditionDefinition objects each referencing a DiagComPrimitive restricted by the current ECU state.

If a STATE-TRANSITION-REF is used with a DIAG-COMM in ODX, there optionally may be a PRE-CONDITION-STATE-REF for the source states of the referenced transitions. If STATE-TRANSITION-REFs and/or PRE-CONDITION-STATE-REFs are used, the DIAG-COMM is executable in the SOURCE states of the referenced STATE-TRANSITIONs and in the referenced preconditions' STATEs but no other states. This means that the collection of MCDDbEcuState objects returned by the methods `MCDDbDiagComPrimitive::getDbPreConditionStatesByDbObject(…)` and `MCDDbDiagComPrimitive::getDbPreConditionStatesBySemantic(…)` are the union of the source states of the state transitions referencing this DiagComPrimitive with those ECU states which reference a `MCDDbPreconditionDefinition` to this DiagComPrimitive - both with respect to the same ECU state chart.

If a DiagComPrimitive is neither referenced from a `MCDDbEcuStateTransition` nor from any `MCDDbPreconditionDefinition`, this DiagComPrimitive is executable in all states and the execution does not result in a state transition. In this case, the collection of precondition states at this DiagComPrimitive is empty for all ECU state charts at the same DbLocation. Otherwise, this DiagComPrimitive is only executable successfully in one of the ECU states within the collection of precondition states at this DiagComPrimitive.

Please note that the entire ECU states and state charts model is only available at the DB part of the diagnostic server API. The diagnostic server will not provide any support for active tracking of the current ECU states. Moreover, the diagnostic server will not prevent the client application to execute restricted DiagComPrimitives in ECU states not supported for these DiagComPrimitives. Hence, the client application is responsible for processing and interpreting the information in ECU state charts.

8.23 Function dictionary

8.23.1 General

In a wide range a communication-oriented view on an ECU's diagnostic functionality is provided by MVCI diagnostic server and ODX. However, this does not always meet today's way of designing vehicles, because many functions of a vehicle are distributed across several ECUs.

Function oriented diagnostics becomes more and more important since the functional point of view is much closer to the customer experienced symptoms a malfunction might cause.

This is covered with the structures described in this section based on the aspects of communication oriented diagnostics definitions and requirements.

8.23.2 Functions and funtion groups in ODX

A function represents a vehicle subsystem or functionality (e.g. Indicator lights) considered from the point of view of diagnostics. A function may divide into one or more system components/sub functions that each may consist of several parts again and so on.

A function in `MCDDbFunctionDictionary` refers to the set of diagnostic information implemented in one or several ECUs that is related to this function including diagnostic services, DTCs, environment data and parameters. It is not intended (and therefore not possible) to (re-)define any information that is already available.

For example the function "Indicator Light Left" as a sub function of "Indicator Lights" is implemented across different ECUs and related to the according fault memories, input/output controls, measurement values and so on. Furthermore the function "Blinking left" may be valid for several model lines and model years. From a functional point of view it is interesting to know which ECUs and diagnostic elements of an ECU are part of a function.

Finally, a function may have input and output parameters to influence the behavior of a function respectively indicating the correct function behavior.

Functions are defined in hierarchical structures (by recursion) to allow expressing different functional granularities:

— Indicator Lights,

    — Indicator Light Left,

    — Indicator Light Right,

— Warning Indicator Lights,

— … .

A `MCDDbFunctionNodeGroup` is a container for already defined `MCDDbFunctionNodes` regardless to their hierarchical context. The `MCDDbFunctionNodeGroup` "Lights" could for example contain the `MCDDbFunctionNodes` "Indicator Lights" and "Head Lights".

— Lights,

    — Head Lights,

        — … ,

    — Indicator Lights,

        — Indicator Light Left,

        — Indicator Light Right,

        — Warning Indicator Lights,

        — … .

A `MCDDbFunctionNodeGroup` may also contain other `MCDDbFunctionNodeGroups`. For a `MCDDbFunctionNodeGroup` the same requirements apply as for a `MCDDbFunctionNode` (relation to ECUs, services, DTCs, parameters…). In the example above it is use case dependent either to choose a `MCDDbFunctionNode` or a `MCDDbFunctionNodeGroup`.

8.23.3 Function dictionary data model description

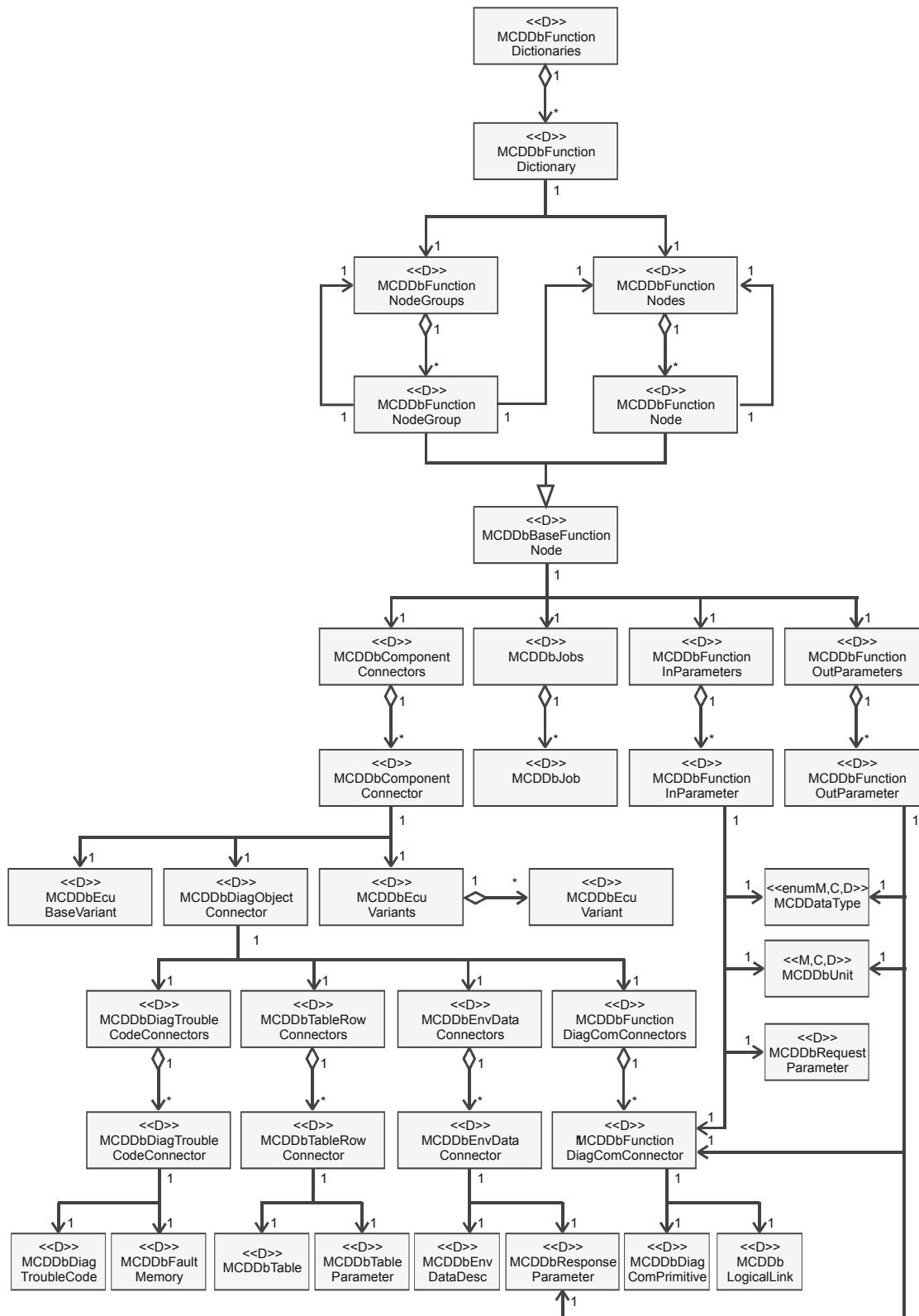Figure 132 depicts the `MCDDbFunctionDictionaries` data model.

Figure 132 — MCDDbFunctionDictionaries data model

A `MCDDbBaseFunctionNode` aggregates all common features of a `MCDDbFunctionNode` / `MCDDbFunctionNodeGroup` (see previous section) and therefore is implemented by the according subclasses.

The hierarchy of `MCDDbFunctionNodes` / `MCDDbFunctionNodeGroups` can be considered as a function catalog representing the functional layout of the vehicle. It might be useful to generate the function catalog out of the same system where the vehicle's functional layout is described.

The element `MCDDbFunctionNode` represents a function as part of a function hierarchy. The requirement of grouping functions is implemented by a `MCDDbFunctionNodeGroup`. A `MCDDbFunctionNode` or `MCDDbFunctionNodeGroup` may only be relevant for or even restricted to certain departments or data customers. To reflect this, it is allowed to specify (ADDITIONAL-) AUDIENCEs, which may be used by a diagnostic application or as a data export/ conversion filter criteria by appropriate tools.

As mentioned above, a function often is distributed across ECUs and other components of the vehicle's network and therefore not only implemented in one single control unit. The layout of a function may vary from model line to model line. From that perspective it makes sense at first to describe which components (references to BASE- / ECU-VARIANTS) are contributing to a certain function and also to describe what is the right layout for a function.

A component's contribution to a function from diagnostic point of view may include DTCs (`MCDDbDiagTroubleCodes` and its corresponding `MCDDbFaultMemories`), ENVDATAs (`MCDDbResponseParameters` and its corresponding `MCDDbEnvDataDescs`), DIAG-COMMs (`MCDDbDiagComPrimitives` via `MCDDbFunctionDiagComConnector`) and TABLE-ROWS (`MCDDbTableRows` and its corresponding `MCDDbTables`). The `MCDDbDiagObjectConnector` aggregates these objects. The uniqueness is guaranteed by the `MCDDbLocations` that are referenced by the super ordinate `MCDDbComponentConnectors`.

The `MCDDbComponentConnetors` controls the validity for following objects:

— BASE-FUNCTION-NODE (`MCDDbFunctionNode`, `MCDDbFunctionNodeGroup`)

— DIAG-OBJECT-CONNECTOR and its sub-objects DTC-DOPs, TABLEs and ENV-DATA-DESCs.

Case 1: Only a BASE-VARIANT is specified.

The BASE-FUNCTION-NODE/DIAG-OBJECT-CONNECTOR (DTC-DOPs, TABLEs, ENV-DATA-DESCs) applies to this BASE-VARIANT and all of its ECU-VARIANTs.

Case 2: Only one (or more) ECU-VARIANTs are specified.

All specified ECU-VARIANTs shall inherit from the same BASE-VARIANT. The BASE-FUNCTION-NODE/DIAG-OBJECT-CONNECTOR (DTC-DOPs, TABLEs, ENV-DATA-DESCs) applies to these ECU-VARIANTs, not to the common BASE-VARIANT.

Case 3: A BASE-VARIANT and one (or more) ECU-VARIANTs are specified.

All specified ECU-VARIANTs shall inherit from the same BASE-VARIANT. The BASE-FUNCTION-NODE/DIAG-OBJECT-CONNECTOR (DTC-DOPs, TABLEs, ENV-DATA-DESCs) applies to these ECU-VARIANTs and the common BASE-VARIANT.

Case 4: Neither a BASE-VARIANT nor an ECU-VARIANT is specified. This case is forbidden.

`MCDDbFunctionInParameters` and `MCDDbFunctionOutParameters` cover the following use cases:

— Documentation of high level function input and output parameters without any technical relation, that means e.g. no relevance for an MVCI diagnostic server.

— Documentation of Vehicle Message Matrix (VMM) input and output signals related to a function.

— Mapping of VMM-Signals to diagnostic content

For the higher level description of a function, higher level input and output parameter descriptions can be useful, therefore the diagnostic description of the input and output parameters is optional. The higher level input and output parameter description could be useful, if a `MCDDbMultipeEcuJob` is used for diagnosing the function.

For example, the value of an output signal of one function is described as a parameter of a measurement value, while the value of an input signal of a function may be covered by a service parameter of a routine. In the example the output signal and the service parameter of the routine could both be accessed by diagnostic services, but for a higher level description only input "Current in V" and output "Temperature in C" is necessary.

Both `MCDDbFunctionInParameters` and `MCDDbFunctionOutParameters` have a `MCDDataType` (PHYSICAL-TYPE) and a `MCDDbUnit` to reflect this information without the necessity to resolve the (optionally) attached parameters. Since the optional `MCDDbRequestParameters` / `MCDDbResponseParameters` may only be referenced by SHORT-NAME, it is necessary to specify the service scope, for which the SHORT-NAME of the parameters has to be unique. This scope is given by the `MCDDbDiagDomPrimitive` referenced by the `MCDDbFunctionInParameters` and `MCDDbFunctionOutParameters` element via `MCDDbFunctionDiagComConnector`. A corresponding `MCDDbLogicalLink` may be available in the database where the service can be generated and executed.

NOTE   The reference to the `MCDDbDiagComPrimitive` is optional. In the case of having no `MCDDbDiagComPrimitive` referenced it is not allowed to have a `MCDDbRequestParameters` / `MCDDbResponseParameters` referenced by SHORT-NAME specified and vice versa.

8.23.4 Uniqueness of MVCI diagnostic server function dictionary data resolution

8.23.4.1   MVCI server resolution

The current state of the ODX Function Dictionary data model allows several instances of ambiguous definitions that need additional information to be uniquely resolved by a MVCI diagnostic server.

8.23.4.2   Example 1

There exists a BaseVariant BV which inherits from two Protocol layers P1 and P2. There are two logical links, LL1 pointing to BV inheriting from P1 and LL2 pointing to BV inheriting from P2.

The COMPONENT-CONNECTOR of a BASE-FUNCTION-NODE contains a reference to BV, the DIAG-OBJECT-CONNECTOR of this COMPONENT-CONNECTOR points to a TABLE of the BV and a TABLE-ROW from this TABLE. The STRUCTURE of the TABLE-ROW contains a parameter using a DOP-SNREF, where the DOP is defined both in P1 and P2 but differently in each case (e.g. the DOP is a STRUCTURE with three parameters in P1, while it is a STRUCTURE with four parameters in P2).

When this TABLE-ROW is accessed through the MCDDbComponentConnector at runtime, the MVCI diagnostic server needs to know the context (through a MCDDbLocation) to be able to resolve the correct parameter STRUCTURE in a unique and deterministic way.

8.23.4.3   Example 2

There exists a BaseVariant BV which is inherited by two ECU Variant layers EV1 and EV2. There exists a logical link LL pointing to BV.

The COMPONENT-CONNECTOR of a BASE-FUNCTION-NODE contains a reference to EV1 and EV2. The DIAG-OBJECT-CONNECTOR of this COMPONENT-CONNECTOR references an ENV-DATA-DESC of the BV and an ENV-DATA element of this ENV-DATA-DESC. The ENV-DATA definition contains one parameter – not neccessarily on top level – with a DOP-SNREF, where the DOP is only defined in EV1 but not in EV2.

When this ENV-DATA is accessed through the MCDDbComponentConnector at runtime, the MVCI diagnostic server needs to know the context (through a MCDDbLocation) to be able to resolve the parameter definition in a unique and deterministic way. It is also possible that a chosen MCDDbLocation context is not sufficient to perform complete resolution of the parameter structure (in case the MCDDbLocation representing EV2 is chosen in the above example).

8.23.4.4   Example 3

There exists a BaseVariant BV which inherits from two Protocol layers P1 and P2. There are two logical links, LL1 pointing to BV inheriting from P1 and LL2 pointing to BV inheriting from P2.

The COMPONENT-CONNECTOR of a BASE-FUNCTION-NODE contains a reference to BV, the DIAG-OBJECT-CONNECTOR of this COMPONENT-CONNECTOR points to a TABLE of BV1. The same DIAG-OBJECT-CONNECTOR references a ENV-DATA-DESC from P2.

When this TABLE is accessed through the MCDDbComponentConnector at runtime, the MVCI diagnostic server needs to know the context (through a MCDDbLocation) to be able to resolve the correct TABLE-ROW in a unique and deterministic way. The only valid option is LL1, while for resolving the correct ENV-DATA-DESC the only valid option would be LL2. A MCDDbDiagObjectConnector allows the referencing of elements which are valid on different MCDDbLocations.

8.23.4.5   Example 4

There exists a BaseVariant BV which defines a diagnostic service DC.

A BASE-FUNCTION-NODE contains a reference to a FUNCTION-IN-PARAM that in turn references a FUNCTION-DIAG-COM-CONNECTOR. This CONNECTOR points to the diag service DC but does not reference a logical link. Also there is no COMPONENT-CONNECTOR to the BASE-FUNCTION-NODE.

Whenever the runtime system is accessing the MCDDbRequestParameter of the MCDDbFunctionInParameter it has to be able to resolve the context of a MCDDbLocation to be able to uniquely identify the parameter based on its name. In the example this is not possible due to the lack of a COMPONENT-CONNECTOR and a LOGICAL-LINK; therefore the access will fail even though it could potentially be succeeding in special constellations.

8.23.4.6   Example 5

There exists a BaseVariant BV which is inherited by two ECU Variant layers EV1 and EV2. All three layers define a diagnostic service DC. In addition there exists a logical link LL pointing to BV.

The COMPONENT-CONNECTOR of a BASE-FUNCTION-NODE contains a reference to variants EV1 and EV2. This means that at runtime only these two ECU variant layers comprise the valid set of referenced objects but not the BV layer. The BASE-FUNCTION-NODE references a FUNCTION-IN-PARAM which in turn references a FUNCTION-DIAG-COM-CONNECTOR.

The FUNCTION-DIAG-COM-CONNECTOR has a reference to a diag service DC and a logical link LL. The DC reference clearly points to the DC definition in EV1. Therefore a LL pointing to the base variant BV cannot be used for providing the context for resolving object references, it is practically useless.

If the DC reference of the FUNCTION-DIAG-COM-CONNECTOR points to a service on the BV the LL can be used for context resolution, but this would contradict the information in the COMPONENT-CONNECTOR.

Such a situation could be resolved by an implementation that favours resolution of FUNCTION-DIAG-COM-CONNECTOR references based on the LL instead of information in the COMPONENT-CONNECTOR.

These examples only illustrate a subset of the problems that could theoretically be inherent in ODX FD data. To be able to deal with these and similar situations, the MVCI diagnostic server API incorporates an additional parameter at methods that are provided for database browsing of FUNCTION-IN-PARAM and FUNCTION-OUT-PARAM elements, as well as DIAG-OBJECT-CONNECTOR information. This parameter provides the MCDDbLocation access key that is to be used as a context for object resolution. In the case where the MCDDbLocation provided by the calling application contradicts information that is provided by the ODX FD data, the location provided by the API function takes precedence. This can e.g. be the case when a FUNCTION-IN/OUT-PARAM is accessed by the application where the FUNCTION-DIAG-COM-CONNECTOR that is accessible from the PARAM provides a logical link reference (which in turn also points to a location).

In case the MVCI diagnostic server is unable to unambiguosly resolve a FD data reference due to a situation as described above, the affected database browsing method shall throw a MCDDatabaseException with error code eDB_INCONSISTENT_DATABASE.

8.23.5 Function dictionary usage scenario

Figure 133 depicts the example for a vehicle with four doors, each having an ECU in it.

Figure 133 — Example four Door ECUs in a vehicle network

The four ECUs each have their own functionality (e.g. window up and down, lock/ unlock door), but may also be addressed by a common functionality, e.g. central lock activating all locks or heat extraction opening all windows simultaneously.

From the diagnostics point of view, these functions may be reflected in 3 ways:

a)   The activation of the window or lock actuator is performed by an IO-Control service.

b)   Any problems with the actuators are expressed by error codes.

c)   The current position of the window or the locker of a door is requested by a measurement service.

Table 31 defines the Individual and Common functionality of Door ECUs.

Table 31 — Individual and Common functionality of Door ECUs

	Door ECU Back Left	Door ECU Back Right	Door ECU Front Left	Door ECU Front Right
Individual Functionaltity	Window Back Left up / down	Window Back Right up / down	Window Front Left up / down	Window Front Right up / down
	Door Lock Back Right open / close	Door Lock Back Left open / close	Door Lock Front Right open / close	Door Lock Front Left open / close
Common Functionality	Central locking			
	Heat Extraction			

Additional Comments:

— In function hierarchies it is intentionally not defined how MCDDbFunctionNodes and their subordinates relate regarding their diagnostic functionality. That means, the author (or the diagnostic application) decides whether the content related to a subordinate function automatically is relevant for its superior function and is automatically considered in its context.

— Function parameters are considered mainly to be used for documentation purposes (e.g. Vehicle Message Matrix / VMM signals). As mentioned above, if there are any MCDDbRequestParameters / MCDDbResponseParameters referenced, their SHORT-NAMEs are to be resolved in the scope of the related services referenced by the MCDDbFunctionDiagComConnector aggregated to the MCDDbFunctionInParameters / MCDDbFunctionOutParameters.

— There is no relation defined between DTCs and ENV-DATAS referenced by a MCDDbDiagObjectConnector.

Again, the author or the diagnostic application decides about whether or not making this relation and how to define such a relation.

NOTE   Multiple MCDDbFunctionDictionaries are allowed. But the SHORT-NAMEs of all MCDDbFunctionNodeGroups / MCDDbFunctionNodes shall be globally unique.

8.24 Sub-Component data model description

8.24.1 Sub-Component data model

Figure 134 depicts the MCDDbSubComponents data model.

Figure 134 — MCDDbSubComponents data model

A `MCDDbSubComponent`, defined at the `MCDDbLocation`, is considered to be a functional unit in- or outside of an ECU that covers certain additional diagnostics relevant functionality either physically (e.g. a LIN slave) or logically. To point out the use case the SEMANTIC attribute may be used. Two SEMANTICS are predefined:

⎯ SLAVE, if the MCDDbSubComponent describes a physical function unit,

⎯ FUNCTION, if the MCDDbSubComponent describes a logical function unit.

The latter is interesting in context with `MCDDbFunctionDictionary`, when a counterpart to a `MCDDbFunctionNode`/`MCDDbFunctionNodeGroup` shall be defined.

In opposite to a `MCDDbFunctionNode` or a `MCDDbFunctionNodeGroup` a `MCDDbSubComponent` is always related to one explicit ECU (or even ECU-VARIANT) and can be considered as an additional layer below `MCDDbLocation`.

The difference is that no new data (DTC, ENV-DATA, and TABLE-ROW) is defined but only reused (referenced) from other layers. Therefore, the `MCDDbDiagTroubleCodeConnector`, `MCDDbTableRowConnector` and `MCDDbEnvDataConnector` elements aggregated by a `MCDDbSubComponent` shall always only point to elements that are part of the `MCDDbLocation` that the `MCDDbSubComponent` belongs to.

A `MCDDbSubComponentParamConnector` allows referring to an `MCDDbDiagComPrimitive` and optionally to one or more `MCDDbRequestParameters` and `MCDDbResponseParameters`. The related `MCDDbDiagComPrimitive` is the SHORT-NAME boundary for these parameters.

Note: The `MCDDbDiagComPrimitive` may be referenced without any `MCDDbRequestParameter`/ `MCDDbResponseParameter`. In this case, the whole `MCDDbDiagComPrimitive` is relevant for the referencing `MCDDbSubComponent`.

`MCDDbMatchingPatterns` (`MCDDbSubComponent::getDbSubComponentPatterns()`) may be used to specify how the presence of a `MCDDbSubComponent` can be determined at runtime. In opposite to an ECU-VARIANT's VARIANT-PATTERN (`MCDDbEcuVariant::getDbVariantPatterns()`) this is not intended to be performed automatically but only for documentation purposes. However, after a `MCDDbSubComponent` has been selected or "identified" by a diagnostic application, the diagnostic application may provide according functionality. For example it can filter out any content that is not relevant for this `MCDDbSubComponent`.

NOTE   There is no inheritance given for SUB-COMPONENTs. SUB-COMPONENTs should be defined for each BASE-/ECU-VARIANT if necessary.

8.24.2 Sub-Component usage scenario

Consider a multi purpose ECU with two Seat LIN slave controllers attached.

Figure 135 depicts the multi purpose ECU with 2 LIN slaves.

Front Seat
Left LIN-ECU

Multiple
Purpose
Front ECU

Front Seat
Right LIN-ECU

Figure 135 — Multi purpose ECU with 2 LIN slaves

Since only the multi purpose ECU (master) is attached to the CAN network, all diagnostic messages that have an influence on the behavior or request the status of the seat ECU, will be sent to the master. The master then decides how to handle those diagnostic messages. On the other hand, if an error occurs inside one of the LIN slaves, this is probably indicated by a DTC activated in the master's fault memory since the slaves may not have their own fault memory.

This means that the diagnostic data description of the master ECU also has to consider the diagnostic functionality of the Slaves. This relationship can be expressed by defining a `MCDDbSubComponent` for each LINslave that:

— Reuses the content of the master ECU and therefore avoids redundancy.

— Resides inside the diagnostic description of the master ECU, to keep the LIN relevant parts of the ECU self contained within the MCDDbSubComponent.

— May be checked for validity or presence by using its MCDDbSubComponentPatterns to filter out any information that is not relevant in the current `MCDDbSubComponents` context.

Additional Comments:

— The MCDDbSubComponent addresses primarily documentation use cases and offers another approach to the diagnostic content of the master. In other scenarios it might be more useful to have a separate DIAGLAYER for the LIN slaves.

— There may be the use case to have a relationship between a function defined in MCDDbFunctionDictonary and a MCDDbSubComponent that e.g. covers this functionality in the ECU. This is not explicitly modeled but still can be covered by a diagnostic application mapping to SEMANTICS or naming conventions.

8.25 Monitoring vehicle bus traffic

A common use case for a diagnostic server is the need to monitor traffic on a vehicle communication bus. To provide simple bus monitoring capability, this chapter defines how bus monitoring shall be implemented in the MVCI diagnostic server. The concepts presented in this chapter are based on standardized features of the D-PDU API layer. If the system is using any other VCI, the concept is the same and should act in a similar way as far as the VCI supports it.

The monitoring link that is introduced in this chapter is an entirely passive entity regarding the monitored physical resource. That means that a monitoring link offers no way to alter any protocol parameters that are associated with the monitored bus resource. As all information pertaining to protocol parameters is part of the diagnostic database, this information is not available before a database is selected. Therefore, a monitoring link will always use the already existing settings of the physical resource that is being monitored – if a logical link to that resource already exists in the MVCI diagnostic server, a monitoring link to that resource will be using the link settings that have previously been configured by the logical link. In case no logical link to a resource exists when a monitoring link to that resource is created, the monitoring link will be using that resource's default settings as implemented in the communication/protocol layer. As the D-PDU API allows any link to a physical resource to modify protocol parameters at any time, it is always possible to create a logical link to a resource that already has an associated monitoring link, and modify that resource's configuration by setting protocol parameters at the logical link. A monitoring link could be implemented by using corresponding D-PDU API protocols such as ISO_11898_RAW, which forwards all bus communication without applying any application layer logic to the monitored messages.

In the MVCI diagnostic server API, the class `MCDMonitoringLink` is provided for performing bus monitoring. Instances of this class can be created based on `MCDInterfaceResources` to monitor on one of the communication channels available to the MVCI diagnostic server. `MCDMonitoringLink` instances are created using the method `MCDProject::createMonitoringLink(MCDInterfaceResource IfResource)`. Please note that the method `MCDSystem::prepareVCIAccessLayer()` has to be called before doing bus monitoring, so that the D-PDU API layer can be set up by the MVCI diagnostic server. In case the VCI access layer has not been prepared beforehand, a call to `createMonitoringLink()` will result in a MCDProgramViolationException with error code `eRT_VCI_ACCESS_LAYER_NOT_PREPARED` being thrown. In case the VCI does not support monitoring, a call to `createMonitoringLink()` will result in a MCDSystemException with error code `eSYSTEM_MONITORING_NOT_SUPPORTED`. The interface resource contains additional information like the protocol, which is not taken into account for generating the monitoring link. The kernel may return the same monitoring link object for the same bus, but other protocol resource interface. It is strongly recommended to the application to use only one monitoring link for one physical bus interface.

The ShortName of a `MCDMonitoringLink` will be generated by the MVCI diagnostic server and it will include the ShortName of the `MCDInterface` as well as the ShortName of the `MCDInterfaceResource` that is being monitored using the following rule:

```
#RtGen_Monitor_<ShortnameOfMCDInterface>_<ShortnameOfMCDInterfaceResource
>
```
Monitoring can be switched on or off on a per-link basis by calling the `MCDMonitoringLink::start()` and `MCDMonitoringLink::stop()` methods. Monitored data will be contained in the `MCDDatatypeAsciiStrings` collection returned by the `MCDMonitoringLink::fetchMonitoringFrames(A_INT32 numReq)` method. Each monitored message will be contained in one `MCDDatatypeAsciiString` within that collection.

A client application can either directly poll a `MCDMonitoringLink` object for available monitoring results, or can use an event handler to be notified by the MVCI diagnostic server when monitoring frames are available. The used event name which is fired is `onMonitoringFramesReady(MCDMonitoringLink monLink)`.

To lessen the event load on the client application, the 'number of samples before firing event' field can be used to define how many samples should be collected before an event is raised by the MVCI diagnostic

server. This number of samples can be set by calling the method
`MCDMonitoringLink::setNoOfSampleToFireEvent(A_UINT16 noOfSamples).`

In addition, the client application can use the `MCDMessageFilter` class, which is available through the `MCDMessageFilters` collection from a MCDMonitoringLink object, to specify filters for the incoming bus messages. These filter definitions are analogous to the IOCTL filter data structure specifications from the D-PDU API standard. For detailed semantic descriptions please refer to the relevant chapters of the D-PDU API specification ISO 22900-2. If the VCI is not a D-PDU API device it may not be possible to activate filtering.

Please note that, despite message filtering and other performance enhancing features of the kernel, the monitoring solution proposed here is not supposed to be used (or expected to scale) for monitoring of high-speed, high-volume bus traffic. Due to internal restrictions, e.g. in the coupling between MVCI diagnostic server and D-PDU API implementation, bus monitoring using a MVCI diagnostic server is a very resource-intensive task, and as such probably will not perform adequately for high-load situations until a more suitable solution can be implemented based on upcoming versions of the relevant standards.

Currently, the D-PDU API will send an event to the MVCI diagnostic server for every single received frame, which potentially results in the MVCI diagnostic server being completely overloaded by D-PDU API events in high-traffic situations. It is of course up to the implementation of the MVCI diagnostic server and the D-PDU API to circumvent these problems; however, such optimizations cannot be considered in this standard.

The format of the monitored messages as returned by the method `MCDMonitoringLink::fetchMonitoringFrames(A_INT32 numReq)` cannot be defined in sufficient detail and comprehensiveness to cover all bus architectures and VCI driver implementations that might have to be covered by this specification.

8.26 Support of VCI module selection and other VCI module features according to D-PDU API Standard

8.26.1 Introduction

For being able to physically connect to a vehicle or a set of ECUs, respectively, a MVCI diagnostic server uses a vehicle communication interface (VCI). This VCI can either be a proprietary interface or it can be compliant to the D-PDU API standard ISO 22900-2. The integration of a D-PDU API compliant MVCI access is described in this chapter. If different definitions are required for integrating a proprietary VCI, these are stated in the corresponding places in addition.

The support of a D-PDU API is an optional feature for a MVCI diagnostic server. Even if the standard was mainly developed to work with a D-PDU API, any other proprietary or standardised way to do vehicle diagnostics may be integrated into a MVCI diagnostic server.

There are two different approaches how a client can work with VCIs on a MCD3 3D-API. The first approach was introduced with the MCD3 V2.0.2 specification. It uses automatic VCI selection from the D-Server. This static VCI selection approach is selected while calling the `MCDSystem::prepareInterface()` method. This prepares the VCI for use, but it does not explicitly state which VCI to use, in case that more than one VCI is available for diagnostics. The client creates logical links with one of the 4 methods on MCDProject: `createLogicalLink()`, `createLogicalLinkByAccessKey()`, `createLogicalLinkByName()` or `createLogicalLinkByVariant()`.

To handle more than one VCI and to select a specific VCI to use with a given logical link, a second approach is introduced. This approach allows selection of a given VCI and even of a given resource (e.g. one of 4 CAN controllers on a VCI) to be used with a logical link. A call to `MCDSystem::prepareVCIAccessLayer()` enables this approach within the MVCI diagnostic server. The client creates the logical links with one of the 8 remaining methods on MCDProject: createLogicalLink… with suffix: "AndInterface" or "AndInterfaceResource".

The two VCI selection approaches are mutual exclusive. A client can call `prepareInterface` or `prepareVCIAccessLayer`. Unless the corresponding unprepare method is not called, no call to one of those methods is permitted anymore.

8.26.2 Definitions

The MVCI D-PDU API supports the selection of a specific VCI module to be used for diagnostic communication by a client application, e.g. a MVCI diagnostic server. VCI module is the D-PDU API term for a vehicle communication interface. From the D-PDU API's point of view several of these VCI modules can be managed below the D-PDU API at the same time. However, in general only one of these VCI modules is actively used by the MVCI diagnostic server. Furthermore, the D-PDU API provides access to the properties of the VCI modules.

A physical vehicle link is the physical connection between a vehicle's diagnostic connector and the ECU. The physical vehicle links available in a vehicle and the properties of these physical vehicle links are defined in the ODX data used to describe a vehicle's electrical configuration.

A physical interface link is the physical connection between the VCI connector of a VCI and the interface connector. Technically speaking, the VCI connector and the interface connector are the connectors at the two opposite ends of the cable connecting a VCI with a vehicle. In the VCI, a VCI connector is driven by a VCI module (also called interface in the following sections). The VCI modules contained in a VCI and their properties are accessed by the MVCI diagnostic server via the D-PDU API.

The interface connector (called DLC connector in the D-PDU API) at the vehicle's end of the *interface cable* is plugged into the vehicle connector. Therefore, both connectors shall match mechanically and have identical pin layouts (1:1 match). At the other end, the interface cable is plugged into the VCI module's VCI connector. The cable description file (CDF) shipped with the D-PDU API lists the supported interface cables.

The set of types of possibly available physical interface links and their properties are defined inside a so-called module description file (MDF) and the so-called cable description file (CDF). Both files are shipped together with the D-PDU API to be used by the MVCI diagnostic server.

A physical link is the combination of a physical vehicle link connected to a physical interface link.

The following subclauses describe the behaviour of a MVCI diagnostic server supporting VCI modules according to the D-PDU API standard. Subclause 8.26.13 describes the behaviour of a MVCI diagnostic server not supporting VCI modules according to the D-PDU API standard.

General behaviour of D-PDU API related MVCI diagnostic server methods

Many of the MVCI diagnostic server methods described in the following sections include a call to a D-PDU API function. In these cases, the name of the called D-PDU API function is mentioned.

In general, each call of a D-PDU API function may fail returning a D-PDU API error code, which is described in the D-PDU API standard.

8.26.3 Overview of VCI module related classes

The class diagram in Figure 136 shows an overview of the classes required to represent VCI modules in the MVCI diagnostic server API and to access the corresponding D-PDU API resources.

Figure 136 — Class diagram of VCI module related classes and their properties

8.26.4 VCI module selection

For some diagnostic applications based on a MVCI diagnostic server, it may be necessary to dynamically select or re-select the VCI module to be used for vehicle diagnostics at runtime. For example, a service tester application in a workshop with several VCI modules (indirectly) hooked up to different vehicles needs to be able to connect to each of these VCI modules on demand.

The available VCI modules are detected by the MVCI D-PDU API internally. The application running on top of the MVCI diagnostic server does not provide any connection parameters itself. The MVCI diagnostic server provides the list of currently available VCI modules to the diagnostic application. In this list, every VCI module

is identified by its unique ShortName which is provided by the MVCI diagnostic server. Now, the diagnostic application can choose the VCI module to be used for diagnostic communication from the list, connect to this module or disconnect from this module to be able to connect to a different module in the next step.

It is also possible to connect to more than one VCI module at the same time, e.g. when there are two VCI modules connected to the same vehicle via a Y-cable with a shared interface connector.

8.26.5 MCDInterface

The class `MCDInterface` represents a single VCI module at the MVCI diagnostic server API and provides access to the features of this VCI module. The class `MCDInterface` is derived from `MCDNamedObject`.

To retrieve the named collection of `MCDInterfaces` currently available for a MVCI diagnostic server, the method `MCDSystem::getCurrentInterfaces():MCDInterfaces` has been introduced. Every `MCDInterface` object can be identified by its unique ShortName. This ShortName is generated by the MVCI diagnostic server with the following pattern: #RtGen_MCDInterface_<Number>. The number in this generated ShortName is increased for every available or newly available VCI module. The generated ShortName does not contain the denoted brackets. For the first instance, the number is zero. This pattern is also used if only a single VCI module, represented by the MCDInterface object, is part of the MCDInterfaces collection.

The LongName of the MCDInterface object is composed of the structure element that is e.g. delivered by the D-PDU API via method PDUGetModuleIds. The PDU_MODULE_ITEM list contains the PDU_MODULE_DATA structures. Each PDU_MODULE_DATA structure contains an optional vendor specific module identification name <pVendorModuleName> and an optional vendor specific additional information <pVendorAdditionalInfo>. These two optional strings are composed to the LongName of the corresponding MCDInterface by the MVCI diagnostic server separated by a space character: <pVendorModuleName> <pVendorAdditionalInfo>. The LongName does not contain the denoted brackets.

Before it is possible to obtain the list of available `MCDInterfaces` by calling the method `MCDSystem::getCurrentInterfaces():MCDInterfaces` (calls the D-PDU API function `PDUGetModulIds`), the method `MCDSystem::prepareVciAccessLayer()` shall be called.

NOTE   In this case, the client must not call `MCDSystem::prepareInterface`.

A call to `prepareVciAccessLayer()` triggers the initialization of the VCI access layer (in the D-PDU API calls the D-PDU API function PDUConstruct). This includes the identification of all available VCI modules. Once the VCI access layer has been initialized, the D-PDU API continuously tracks which VCI modules are currently available. This includes detection of new VCI modules in the visibility scope and detection of VCI modules which left the visibility scope. The method to release the D-PDU APIs VCI access layer is `MCDSystem::unprepareVciAccessLayer()` (in case of static VCIs the method unprepareInterface should be called, calls the D-PDU API function PDUDestruct). The method `prepareVCIAccessLayer()` shall only be called once, unless `unprepareVCIAccessLayer()` is called.

The diagnostic application on top of the MVCI diagnostic server shall decide which of the available interfaces to use for diagnostic communication to a specific vehicle. Otherwise, no diagnostic communication to an ECU or a vehicle, respectively, can be established. This decision is usually based upon the unique ShortName (delivered by the D-PDU API function `PDUGetModuleIds`) of an interface known to be physically connected to the vehicle. To select an interface for diagnostic communication, the application calls `MCDInterface::connect():void` at the corresponding interface object, which calls the D-PDU API function `PDUModuleConnect`. Note that calling `connect()` for at least one interface is mandatory to run any diagnostic communication in the MVCI diagnostic server, if the MVCI diagnostic server supports the D-PDU API.

If the diagnostic application has finished its diagnostic communication via a selected interface, it needs to call the method `MCDInterface::disconnect():void` to disconnect from this interface, which calls the D-PDU API function `PDUModuleDisconnect`.

To find out which interfaces are currently connected, the diagnostic application can fetch the list of connected interfaces by means of the method `MCDSystem::getConnectedInterfaces(): MCDInterfaces` (calls the D-PDU API function `PDUGetStatus` for all VCI modules to find out which modules are connected). This method never throws an exception, at least an empty collection is returned.

Before connecting to an interface, the diagnostic application should check if this interface is available for connection by checking its status via `MCDInterface::getStatus(): MCDInterfaceStatus` (calls the D-PDU API function   for the related VCI module). Connection is only possible if the status `MCDInterfaceStatus::eAVAILABLE` is returned.

As a Logical Link requires a physical link to the vehicle to be available, at least one VCI module which is connected and in state `MCDInterfaceStatus::eREADY` needs to be present, if the MVCI diagnostic server supports the D-PDU API. Of course, this VCI module needs to provide the physical resources and protocol resources defined for the logical link. The combination of physical resource and protocol resource is called a `MCDInterfaceResource`. In principle a single interface could provide more than one `MCDInterfaceResource` matching the requirements of a Logical Link. In this case, the interface resource which will be used for a specific Logical Link is either selected automatically by the MVCI diagnostic server or it is actively selected by the diagnostic application (see 8.26.9). The same applies if multiple interfaces are connected to the same vehicle at the same time and if some of these interfaces provide matching interface resources for the Logical Link.

8.26.6   VCI module selection sequence

Table 32 shows the sequence of methods to call at the MVCI D-Server API for VCI module selection together with the corresponding calls of D-PDU API methods from within the MVCI diagnostic server logic.

Table 32 — Method for VCI module selection - MVCI D-Server API and D-PDU API

Methods called at MVCI D-Server API	Methods called at D-PDU API
MCDSystem::prepareVciAccessLayer()	PDUConstruct() will be executed for all available D-PDU APIs
interfaces = MCDSystem::getCurrentInterfaces()	PDUGetModuleIds(): delivers list of handles hMod1, hMod2, … of available or connected VCI modules
Decide to which interface(s) to connect… interface1.connect() [interface2.connect() …]	 PDUModuleConnect(hMod1) [PDUModuleConnect(hMod2) …]
create and open Logical Links, execute DiagComPrimitives, …. close and remove Logical Links	…all method calls carry the selected VCI module handle hModx as input parameter…
interface1.disconnect() [interface2.disconnect() …] => Only allowed in LogicalLink state eCREATED!	PDUModuleDisconnect(hMod1) PDUModuleDisconnect(hMod2)
MCDSystem::unprepareVciAccessLayer()	PDUDestruct() will be executed for all available D-PDU APIs

8.26.7 Interface status events

Status changes of a VCI module are notified by the MVCI diagnostic server to a diagnostic application by means of different events.

The event `MCDEventHandler::onInterfaceStatusChanged(MCDInterface interface, MCDInterfaceStatus status)` indicates that the status of the VCI module given as first parameter has changed to the state given as the second parameter. This event is e.g. triggered by a D-PDU API module callback with a module status event item.

The event `MCDEventHandler::onInterfacesModified(void)` indicates that the list of VCI modules available for a MVCI diagnostic server has changed – either because a new VCI module has become available or because a VCI module is not available any more. This event is sent when the MVCI diagnostic server or the D-PDU API, respectively, has automatically detected a new VCI module or when communication to a VCI module has been lost (see paragraph below for details). This event is e.g. triggered by a D-PDU API system callback with the information event PDU_INFO_MODULE_LIST_CHG. The diagnostic application can use the method `MCDSystem::getCurrentInterfaces()` to retrieve an updated list of currently available VCI modules in this case.

The event `MCDEventHandler::onInterfaceError (MCDInterface interface, MCDError error)` indicates that the MVCI diagnostic server or the D-PDU API, respectively, has automatically detected that communication to the VCI module given as first parameter has been lost or a hardware fault occurred indicated by the `MCDErrorCode eCOM_LOST_COMM_TO_VCI` or `eCOM_VCI_HARDWARE_FAULT`. These events are only possible, if the affected `MCDInterface` object has been successfully connected via the method `MCDInterface::connect()`, and has not yet been disconnected via the method `MCDInterface::disconnect()`. These events are triggered e.g. by a D-PDU API module callback with a module error event PDU_ERR_EVT_LOST_COMM_TO_VCI or PDU_ERR_EVT_VCI_HARDWARE_FAULT. In case of the event `onInterfaceError()` the affected `MCDInterface` object including all its MCDLogicalLink objects are invalid. In this case the MCDLogicalLink objects can only be closed or reset, respectively. The MCDLogicalLink objects cannot be reused for further communication. The MCDInterface object is also invalid and can only be removed from the `MCDInterfaces` collection. All calls to one of the MCDInterface specific object methods lead to a `MCDProgramViolationException` with the error code eRT_PDU_API_CALL_FAILED, except a final call to the method `MCDInterface::disconnect()`. This final call has to be performed by the application, after the application has received the event `MCDEventHandler::onInterfaceError`.

8.26.8 MCDInterfaceResource

The class `MCDInterfaceResource` can be used to obtain information about the resources available at an interface (VCI module). In addition, this class allows selecting a specific interface resource to be used with a Logical Link. If the D-PDU API is supported, an interface resource is equivalent to a "Resource" object as defined in the D-PDU API. Hence, an interface resource is characterized by the attributes

— communication protocol

— physical interface link type (called "Bustype" in D-PDU API), and

— pins on the interface connector (called "DLC" connector in D-PDU API).

To retrieve the named collection of `MCDInterfaceResources` which are provided by a VCI module, the method `MCDInterface::getInterfaceResources():MCDInterfaceResources` is used. Every `MCDInterfaceResource` is a named object which can be identified by its ShortName. All information needed for this method including the ShortName of a `MCDInterfaceResource` is provided e.g. by the D-PDU API's MDF file.

The method `MCDInterfaceResource::getInterface():MCDInterface` allows to navigate back to the interface the current resource belongs to.

8.26.9 Selection of an interface resource

To open a Logical Link, it is necessary to select an interface resource which matches the requirements of this Logical Link, i.e. it shall match the communication protocol type required for the Logical Link, match the physical vehicle link type, and the pins on the vehicle connector. For a Logical Link, the requirements can be obtained from the ODX data.

This means in detail:

— The resource's physical interface link type, obtainable via the method `MCDInterfaceResource::getDbPhysicalInterfaceLink():` `MCDDbPhysicalInterfaceLink::getType():A_ASCIISTRING`, shall match the Logical Link's physical vehicle link type, obtainable via `MCDDbDLogicalLink::getDbPhysicalVehicleLinkOrInterface():MCDDbPhysical` `VehicleLinkOrInterface::getType():A_ASCIISTRING`.

— The resource's interface connector pins, obtainable via the method `MCDDbPhysicalVehicleLinkOrInterface::getDbInterfaceConnectorPins():` `MCDDbInterfaceConnectorPins`, shall match the Logical Link's vehicle connector pins, obtainable via the method `MCDDbPhysicalVehicleLinkOrInterface::getDbVehicleConnectorPins():` `MCDDbVehicleConnectorPins`, where pin number and pin type MCDConnectorPinType shall match for each connector pin.

— The resource's communication protocol type, obtainable via the method `MCDInterfaceResource::getProtocolType():A_ASCIISTRING`, shall match the Logical Link's communication protocol type, obtainable via `MCDDbLogicalLink::getProtocolType():A_ASCIISTRING`.

A matching interface resource can be selected for a Logical Link in the following two ways:

Case a)

— Automatic selection by the MVCI diagnostic server:

— Before creating a Logical Link, the application decides which interface has to be used. The logical link creation is done by calling `MCDProject::createLogicalLink…ByInterface`. When opening a Logical Link, the MVCI diagnostic server automatically selects a matching interface resource, e.g. supported by the D-PDU API function `PDUGetResourceIds`. In this case, the application using the MVCI diagnostic server does not have to care about this process, no additional method calls are necessary. During the automatic selection, the MVCI diagnostic server is free to choose which of the matching and available interface resource to use for opening the LogicalLink. This selection may differ between different MVCI diagnostic server implementations and is even not reproducible on multiple executions of the same MVCI diagnostic server implementation.

Case b)

— Selection of the interface resource by the application:

— Before creating a Logical Link, the application decides which interface resource has to be used. The logical link creation is done by calling `MCDProject::createLogicalLink…ByInterfaceResource`. Interface resources are retrieved by calling an `MCDInterface::getInterfaceResources()`.

The class `MCDLogicalLink` – which represents a runtime Logical Link in the MVCI diagnostic server – provides the following method to retrieve the selected interface resource:

— `MCDLogicalLink::getInterfaceResource():MCDInterfaceResource`
returns the interface resource assigned to this LogicalLink. Throws an exception of type `MCDProgramViolationException` with error code `eRT_PDU_API_NOT_SUPPORTED` if the logical link does not support the optional D-PDU API.

NOTE   The method `MCDInterfaceResource::isInUse():A_BOOLEAN` returns true if this resource is currently in use. Even if the resource is already in use, it may still be available - depending on the characteristics of this resource. For example, if a resource for a CAN protocol on a certain CAN bus is already in use by one Logical Link, it may also be used multiple times by other Logical Links, depending on the capabilities of the VCI module's CAN protocol driver.

The status values for `isAvailable()` and `isInUse()` are retrieved e.g. via the D-PDU API function `PDUGetResourceStatus`.

8.26.10 Send Break Signal

The method `MCDLogicalLink::sendBreak():void` allows to send a break signal on the Logical Link if a D-PDU API is used, calls the function PDUIoCtl with command PDU_IOCTL_SEND_BREAK). A break signal is a feature of certain physical layers and can only be sent on these physical layers (e.g. SAE J1850 VPW physical links and UART physical links - see ISO 22900-2, subclause PDU_IOCTL_SEND_BREAK, for more details). Throws an exception of type `MCDProgramViolationException` with error code `eRT_NOT_ALLOWED_IN_LL_STATE_CREATED` if the Logical Link is in state eCREATED, or with error eRT_PDU_API_CALL_FAILED if the corresponding function call of PDUIoCtl() at the D-PDU API failed.

8.26.11 MCDDbInterfaceCable

A VCI module is connected to a vehicle using an interface cable which has the interface connector (to be plugged into the vehicle connector) attached to one end of the cable and the VCI connector (to be plugged into the VCI) attached to the other end. To connect a VCI to a vehicle with a different type of vehicle connector, e.g. an OEM specific connector instead of an OBD connector, a VCI is usually shipped with several different interface cables which have different types of connectors.

The class `MCDInterface` offers methods to access information about the different interface cables available for a certain VCI module. However, this information is not required for any runtime operations. It is information only and can be used in a diagnostic application to guide the user to connect the correct cable to the VCI module.

The method `MCDInterface::getDbInterfaceCables():MCDDbInterfaceCables` returns the collection of all possible interface cables specified for the VCI module. These cables are defined in the MVCI's cable description file (CDF). The MVCI diagnostic server shall parse the CDF file to create the result of this method.

The method `MCDInterface::getCurrentDbInterfaceCable():MCDDbInterfaceCable` returns the interface cable which is currently connected to the VCI module. A VCI module may use an automatic cable detection or internal configuration to provide this information. The method calls e.g. the D-PDU API function `PDUIoCtl` with the command PDU_IOCTL_GET_CABLE_ID, and shall parse the CDF file in addition to retrieve the properties of the cable with the cable ID delivered by `PDUIoCtl`.

A `MCDDbInterfaceCable` object provides information about an interface cable. This information is provided by the MVCI's cable description file (CDF):

The method `MCDDbInterfaceCable::getInterfaceConnectorType():A_ASCIISTRING` returns the type of the interface connector of the interface cable. The connector type is provided in the CDF. Note: there is no predefined list of possible return values defined in the MVCI diagnostic server or D-DPU API standards.

The method `MCDDbInterfaceCable::getDbInterfaceConnectorPins():` `MCDDbInterfaceConnectorPins` returns the interface connector pins of the interface cable.

A `MCDDbInterfaceConnectorPin` object contains the mapping of the interface connector pin, returned by `getPinNumber():A_UINT32`, to a pin on the VCI connector of the interface cable, returned by `getPinNumberOnVci():A_UINT32`.

NOTE   The pin number on the VCI connector is only additional information which is not required for any runtime operation. Information about the VCI connector pin number is provided in the CDF.

8.26.12 Accessing VCI module features

Several features of VCI modules are accessible by the following methods of `MCDInterface`:

The method `MCDInterface::reset():void` resets the VCI module (if a D-PDU API is used, calls the function PDUIoCtl with the command PDU_IOCTL_RESET). The reset command will cancel all activities currently being executed by the VCI module (without proper termination). All existing Logical Links will go in the state eOFFLINE and associated ComPrimitives are cancelled. All hardware properties of the MVCI Protocol Module (e.g. programming voltage) will be reset to the default settings. After the completion of the reset command, the VCI module shall be treated as if it were a newly connected VCI module.

The method
`MCDInterface::setProgrammingVoltage(pinOnInterfaceConnector:A_UINT32, voltage:A_FLOAT64):void` sets the programming voltage (in Volts) on the specified pin of the interface connector of the VCI module (if a D-PDU API is used, calls the function PDUIoCtl with the command PDU_IOCTL_SET_PROG_VOLTAGE).

The method
`MCDInterface::getProgrammingVoltage(pinOnInterfaceConnector:A_UINT32):A_FLOAT64` returns the programming voltage (in Volts) on the specified pin of the interface connector of the VCI module. This method is used to read the feedback of the programming voltage from the voltage source, which is set by the method setProgrammingVoltage (if a D-PDU API is used, calls the function PDUIoCtl with the command PDU_IOCTL_READ_PROG_VOLTAGE).

The method `MCDInterface::getBatteryVoltage():A_FLOAT64` reads the battery voltage (in Volts) on the VCI module's VCI connector (if a D-PDU API is used, calls the function `PDUIoCtl` with the command PDU_IOCTL_READ_VBATT).

The method `MCDInterface::getClampState(pinOnInterfaceConnector:A_UINT32, clampName:A_ASCIISTRING):MCDValue` returns the current state of the clamp specified by the pin number on the interface connector (first parameter). In general, the specified clamp name (second parameter) is tool manufacturer specific, and shall be supplied e.g. by an equivalent manufacturer specific IO_CTRL-command to be carried out with the D-PDU API function `PDUIoCtl`, where the clamp name is expected to match the ShortName of the IO_CTRL-command. The clamp state is returned as a `MCDValue`. The data type of this value and the interpretation of the value (e.g. resolution, unit and range) depend on the tool manufacturer specific definition for the specific clamp name.

The following clamp name is expected to be supplied by any MVCI diagnostic server supporting a D-PDU API:

"IgnitionClamp": clamp to retrieve the ignition state of the vehicle. Data type of the returned value is A_BOOLEAN, where value "true" means "ignition on", "false" means "ignition off". In case of PDU_API PDU_IOCTL_READ_IGNITION_SENSE_STATE shall be used.

Tool manufacturer specific clamp names shall start with the domain name (similar to getProperty()). For a german vendor A a clamp name would look like "de.VendorA.NameOfTheClamp".

Other `MCDInterface` methods like `getVendorName()`, `getPDUApiSoftwareName()`, `getPDUApiSoftwareVersion()`, `getHardwareSerialNumber()` provide information about the VCI vendor and several different version numbers of the VCI module. This information is retrieved from the D-

PDU API by the function `PDUGetVersion`. A D-Server not supporting D-PDU API might return a valid value, if it is availbable via the proprietary VCI-API. Otherwise an exception is thrown.

8.26.13 Behaviour of a MVCI diagnostic server not using the VCI Module API

The MCD VCI Module API consists of MCDInterface, MCDInterfaceResource, MCDInterfaceResources, MCDDbPhysicalVehicleLink, MCDDbPhysicalVehicleLinkOrInterface, MCDDbInterfaceConnectorPins, MCDDbInterfaceConnectorPin, MCDInterfaceCables, and MCDInterfaceCable. If the methods `MCDSystem::prepareInterface` and `MCDSystem::unprepareInterface` have been used to prepare the interface, the VCI Module API cannot be used and thus it is not possible to select a specific VCI module from the application, and also it is not possible to access any VCI module features as defined in the classes `MCDInterface and MCDInterfaceResource`.

In this case the methods `getConnectedInterfaces()` and `getCurrentInterfaces()` of `MCDSystem` always return an empty collection. The method `getInterfaceResource()`of `MCDLogicalLink` throws an `MCDProgramViolationException` with error code `eRT_PDU_API_NOT_SUPPORTED`.

The application can open a Logical Link without knowledge about any VCI module, and without having to select a specific VCI module. Handling and selection of a VCI module is done completely inside the MVCI diagnostic server in this case.

To open a logical link with unused VCI Module API, the methods:

— MCDProject::createLogicalLink,

— MCDProject::createLogicalLinkByAccessKey,

— MCDProject::createLogicalLinkByName,

— MCDProject::createLogicalLinkByVariant,

shall be used.

8.27 Handling DoIP entities

8.27.1 General

The following subclauses describe the detection and selection of DoIP entities using the MCDInterface API.

The description relies on D-PDU API support by the MVCI D-Server. If the D-PDU API is not supported the D-Server has to provide the same functionality using the corresponding features of the alternatively integrated MVCI protocol module access layer. In the following this is not explicitly mentioned but has to be considered in any case.

8.27.2 Detection of DoIP entities

8.27.2.1 Basics

Depending on the specific implementation of the MVCI protocol module access, e.g. a D-PDU API implementation, and usage of specific MVCI devices, the application may execute the detection of DoIP entities at system level, e.g. using the Ethernet adaptor of the PC system, or at specific MVCI devices.

8.27.2.2   Detection of DoIP entities connected to the system

The detection of DoIP entities connected to the sytem can be done as follows:

— The application calls `MCDSystem::prepareVciAccessLayer()`.

— The application calls `MCDSystem::detectInterfaces(optionString)`. If the D-PDU API is supported the D-Server will internally call PDUIoCtl(PDU_IOCTL_VEHICLE_ID_REQUEST) at D-PDU API system level; in this case the D-PDU API module handle is PDU_HANDLE_UNDEF. The D-PDU API executes a DoIP vehicle identification request to detect all available DoIP entities. The MVCI diagnostic server provides the detected DoIP modules as additional MCDInterface objects, indicated by an event onInterfacesModified. See below for the format of the parameter optionString.

— The application calls `MCDSystem::getCurrentInterfaces()`. This method delivers a collection of available MVCI modules, including all MVCI devices (hardware modules) and DoIP modules, detected by the D-PDU API.

— The application identifies a DoIP module by evaluating the LongName of the corresponding MCDInterface object. This is the concatenation of pVendorModuleName and pVendorAdditionalInfo from D-PDU API separated by a space, e.g. "Type='MVCI_ISO_13400_DoIP_Entity' EID='0A:1B:2C:3D:4E:5F LA='4096' IP='192.168.1.255' VIN='ABCDEFG1234567XYZ' GroupID=''12".

Option string format:

The format of the input parameter "optionString" is described with the following grammar:

OptStringFmt	::=	**PDU_IOCTL='PDU_IOCTL_VEHICLE_ID_REQUEST'**
		<sep><PSM><sep><PSV><sep><CM><sep><VDT><sep> [<DAC><sep><DAList>]
PSM	::=	**PreselectionMode**=<PreselectionMode>
PreselectionMode	::=	**'None' | 'VIN' | 'EID'**
PSV	::=	**PreselectionValue**=<PreselectionValue>
PreselectionValue		'*string*'
		/* *containing either an optional VIN or an EID to preselect detection of DoIP entities, depending on parameter "PreselectionMode". This string is left empty if no preselection is desired (PreselectionMode='None'). */*
CM	::=	**CombinationMode**=<CombinationMode>
CombinationMode	::=	**'DoIP-Entity' | 'DoIP-Vehicle' | 'DoIP-Group' | 'DoIP-Collection'**
VDT	::=	**VehicleDiscoveryTime**=<VehicleDiscoveryTime>
VehicleDiscoveryTime	::=	'*unsigned short*'
		/* *Time-out to wait for vehicle identification responses. 0=return immediately, or time in milliseconds. */*
DAC	::=	**DestinationAddressCount**=<DestinationAddressCount>
DestinationAddressCount	::=	'*unsigned short*'
		/* *No. of broadcast/multicast addresses in the destination address list DAList. In case of an empty list, the sequence* <DAC><sep><DAList> *is omitted. */*
DAList	::=	(**DA**<n>='<IPv4>' | '<IPv6>')+
n	::=	/* *current number starting with 1, maximum value according to*

*DestinationAddressCount */*

IPv4	::=	*/* dottet decimal notation, e.g. 192.168.1.255 */*
IPv6	::=	*/* hexadecimal, colon separated according to RFC4291, section 2.2 */*
sep	::=	*blank*

EXAMPLE

```
"PDU_IOCTL='PDU_IOCTL_VEHICLE_ID_REQUEST' PreselectionMode='VIN'
PreselectionValue='ABCDEFG1234567XYZ' CombinationMode='DoIP-Entity'
VehicleDiscoveryTime='5000' DestinationAddressCount='1' DA1='192.168.1.255'"
```

An MCDInterface object representing a DoIP module has limited capabilities:

— `MCDInterface::getInterfaceResources()` delivers exactly one item; this MCDInterfaceResource object is used as input parameter when calling one of the methods `MCDProject::createLogicalLinkBy…InterfaceResource`.

8.27.2.3 Detection of DoIP Entities connected to a specific MVCI device

After initial calls of `MCDSystem::prepareVciAccessLayer` and `MCDSystem::getCurrentInterfaces`, the detection of DoIP entities connected to a specific MVCI device can be done as follows:

— The application calls `MCDInterface::detectInterfaces (optionString)`. This method is only supported by an MCDInterface object representing an MVCI device that supports a DoIP vehicle identification request. The MVCI diagnostic server will internally call PDUIoCtl() at the corresponding MVCI module to execute a vehicle identification request on the network connected to the MVCI device. The MVCI diagnostic server provides the detected DoIP modules as additional MCDInterface objects, indicated by an event onInterfacesModified. The definition of the parameter optionString is identical to the according description for the method MCDSystem::detectInterfaces.

— The application calls `MCDSystem::getCurrentInterfaces()` again to retrieve all available MVCI modules, including the newly detected DoIP modules.

8.27.3 Selection of DoIP Entities

The application selects the DoIP module(s) for communication, usually to a specific vehicle, by evaluating the LongName of the corresponding MCDInterface, and calls `MCDInterface::connect()` at the corresponding MCDInterface object(s).

The application creates and opens LogicalLinks, using one of the following alternatives:

— The application selects a connected MCDInterface for the LogicalLink. The D-Server assigns the MCDLogicalLink to the selected DoIP module, using the single available MCDInterfaceResource of this module. For this purpose the application creates the logical link by calling one of the methods

— MCDProject::createLogicalLinkByInterface,

— MCDProject::createLogicalLinkByAccessKeyAndInterface,

— MCDProject::createLogicalLinkByNameAndInterface,

— MCDProject::createLogicalLinkByVariantAndInterface.

 Matching of a DoIP entity's LogicalGatewayAddress with the value of CP_LogicalGatewayAddress at the LogicalLink is handled in the D-PDU API internally.

 The application selects an MCDInterfaceResource of a connected MCDInterface for the LogicalLink. The D-Server assigns the MCDLogicalLink to the corresponding MCDInterface owning this single available MCDInterfaceResource. For this purpose the application creates the logical link by calling one of the methods

  MCDProject::createLogicalLinkByInterfaceResource,

  MCDProject::createLogicalLinkByAccessKeyAndInterfaceResource,

  MCDProject::createLogicalLinkByNameAndInterfaceResource,

  MCDProject::createLogicalLinkByVariantAndInterfaceResource.

Matching of a DoIP entity's LogicalGatewayAddress with the value of CP_LogicalGatewayAddress at the LogicalLink is handled in the D-PDU API internally.

When all LogicalLinks to a DoIP module are in state eCREATED or already removed, the application calls `MCDInterface::disconnect()` for the related DoIP module.

Finally, the application disconnects all MVCI modules and unprepares the D-PDU API by `MCDSystem::unprepareVciAccessLayer()`.

8.28 Mapping of D-PDU API methods

8.28.1 Introduction

The following sections describe the internal behaviour of an MVCI diagnostic server implementation using a D-PDU interface according to D-PDU API standard ISO 22900-2.

For such an MVCI diagnostic server implementation, the mapping of MVCI diagnostic server methods to D-PDU API methods as described in this section shall be used.

8.28.2 Initialization and Selection of VCI Modules

See Table 32.

8.28.3 Communication on a Logical Link

Table 33 shows the sequence of method calls from a client application at the MVCI diagnostic server API and the corresponding method calls from the MVCI diagnostic server implementation at the D-PDU API used for communication on a Logical Link.

Table 33 — Methods for Communication on a Logical Link – MVCI D-Server API and D-PDU API

Methods called at MVCI D-server API	Methods called at D-PDU API
MCDProject::createLogicalLink_XYZ() where XYZ refers to all variants of create logical link methods. Initial link state is eCREATED	(No counterpart)

Table 32 — *(continued)*

Methods called at MVCI D-Server API	Methods called at D-PDU API
MCDLogicalLink::open() Link state changes to eOFFLINE	Access to D-PDU API resources: PDUGetModuleIds, PDUGetObjectId or by parsing the Module Description File (MDF) Optional: the availability of the resources can be checked by PDUGetResourceIds(), PDUGetResourceStatus(), PDUGetConflictingResources() Allocate the resource and create the D-PDU Logical Link: PDUCreateComLogicalLink()
Setting the communication parameters: No counterpart. MVCI diagnostic server internally reads the communication parameters from ODX.	Get default values from the D-PDU API: PDUGetComParam(), PDUGetUniqueRespIdTable() Set values from ODX to the D-PDU API: PDUSetComParam(),PDUSetUniqueRespIdTable()
MCDLogicalLink::gotoOnline() Link state changes to eONLINE	PDUConnect() Communication parameters become active during PDUConnect()
MCDLogicalLink::CreateDiagComPrimitive …XYZ() refers to all variants of creation diagcomprimitives.	(No counterpart)
MCDLogicalLink::suspend()	PDUIoctl(PDU_IOCTL_SUSPEND_TX_QUEUE)
MCDLogicalLink::resume()	PDUIoctl(PDU_IOCTL_RESUME_TX_QUEUE)
MCDLogicalLink::clearQueue()	1. PDUIoctl (PDU_IOCTL_SUSPEND_TX_QUEUE) 2. PDUIoctl (PDU_IOCTL_CLEAR_TX_QUEUE) 3. PDUIoctl (PDU_IOCTL_RESUME_TX_QUEUE)
MCDLogicalLink::sendBreak()	PDUIoctl(PDU_IOCTL_SEND_BREAK)
MCDLogicalLink::reset()	1. PDUIoctl (PDU_IOCTL_SUSPEND_TX_QUEUE) 2. PDUIoctl (PDU_IOCTL_CLEAR_TXQUEUE) 3. PDUIoctl (PDU_IOCTL_RESUME_TX_QUEUE) 4. PDUDisconnect()/PDUDestroyComLogicalLink()
MCDDiagComPrimitive::executeSync()	PDUStartComPrimitive () The server has to suspend the calling thread until the interface has completely processed the passed comprimitive.
MCDDataPrimitive::executeAsync()	PDUStartComPrimitive () The server stores the comprimitive only in the related queue and returns immediately.
MCDDataPrimitive::startRepetition()	PDUStartComPrimitive ()
MCDDiagComPrimitive::cancel()	PDUCancelComPrimitive()
MCDLogicalLink::close() Or MCDLogicalLink::reset() Link state changes to eCREATED	Only in Link State eONLINE or eCOMMUNICATION: PDUDisconnect() In all cases: PDUDestroyComLogicalLink()
MCDLogicalLink::remove…() Refers to all variants of remove methods in the MCDLogicalLinks collection	(No counterpart)

Figure 137 shows the mapping of those MVCI D-Server API and D-PDU API methods which are used for handling a Logical Link.

Key

1   Only the first call of `MCDProject::createLogicalLink…` with the given settings leads to a state `eCREATED`. If a reference on an already existing logical link is returned, the initial state of the returned logical link is the one of the referenced logical link.

2   PDUCreateLogicalLink

3   PDUDestroyComLogicalLink

4   PDUConnect

5   PDUDisconnect, PDUDestroyComLogicalLink

6   PDUStartComPrimitive(PDU_COPT_STARTCOMM)

7   PDUStartComPrimitive(PDU_COPT_STOPCOMM)

Figure 137 — Methods for Logical Link handling (mapping between MVCI diagnostic server API and D-PDU API)

8.28.4 Handling of Communication Parameters

8.28.4.1 Changing communication parameters from the client application

A client application may want to set one or more communication parameters at runtime to values differing from the values defined in the ODX data. Table 34 shows the sequence of method calls in the MVCI diagnostic server API and the D-PDU API required for changing the communication parameters of a Logical Link.

Table 34 — Changing communication parameters from the client application (MVCI diagnostic server API and D-PDU API)

Methods called at MVCI diagnostic server API	Methods called at D-PDU API
create a Control Primitive of type MCDProtocolParameterSet at the Logical Link	Get default values of all communication parameters from the DataBase: (no counterpart)
change value of one ore more Request Parameters of the MCDProtocolParameterSet Control Primitive (the Request Parameters of this Control Primitive correspond to the communication parameters of the logical link)	(no counterpart)
MCDProtocolParameterSet::executeSync()	Set values changed by the client application to the D-PDU API: PDUSetComParam() or PDUSetUniqueRespIdTable() (depends on type of ComParam) for each modified ComParam PDUStartComPrimitive() with ComPrimitive type PDU_COPT_UPDATEPARAM: the changed ComParams become active
MCDProtocolParameterSet::fetchValueFromInterface() MCDProtocolParameterSet::fetchValuesFromInterface()	PDUGetComParam() shall be called to retrieve current settings from interface. (either the specified one with the given shortName or all protocolParameter defined for the related DbLocation)
create a Control Primitive of type MCDProtocolParameterSet at the Logical Link	Get default values of all communication parameters from the DataBase: (no counterpart)

8.28.4.2 Setting temporary communication parameters for a DiagComPrimitive

Several communication parameter values may be overwritten at a DIAG-SERVICE in the ODX data. These values shall only be temporarily valid for the duration of the execution of this service.

Table 35 shows the sequence of method calls for this case.

Table 35 — Setting temporary communication parameters for a DiagComPrimitive (MVCI D-Server API and D-PDU API)

Methods called at MVCI D-Server API	Methods called at D-PDU API
execute a DiagComPrimitive with temporary ComParams	Set values overwritten at the DIAG-SERVICE in the ODX data to the D-PDU API: PDUSetComParam() for each overwritten ComParam PDUStartComPrimitive() with flag TempParamUpdate set to „true" (1)

8.28.4.3 Changing UNIQUE_ID Communication Parameters

The MVCI diagnostic server allows to change the values of all communication parameters (protocol parameters) of a runtime logical link by using the control primitive MCDProtocolParameterSet. As there is no restriction with respect to which kinds of protocol parameters can be changed, it is also possible to

change the addressing information represented by the protocol parameters of parameter class `eUNIQUE_ID`. The result of overwriting the values of the `UNIQUE_ID` protocol parameters is that the handle calculated from the `UNIQUE_ID` information can differ between runtime LogicalLink and DbLogicalLink in case the values of the corresponding protocol parameters have been changed at the runtime LogicalLink. This may lead to problems in the MVCI diagnostic server as it might fail in resolving responses etc. Hence, changing the value of `UNIQUE_ID` protocol parameters is considered harmful.

8.28.5 MCDStartCommunication and MCDStopCommunication

The execution of Control Primitives of type MCDStartCommunication and MCDStopCommunication is mapped to D-PDU API method calls as shown in Table 36.

Table 36 — MCDStartCommunication and MCDStopCommunication - MVCI D-Server API and D-PDU API

Methods called at MVCI D-Server API	Methods called at D-PDU API
MCDStartCommunication::executeSync (Note that there can be a definition for the StartCom service in the ODX data. In this case, this service's request and possibly overwritten communication parameters need to be passed to the D-PDU API.)	PDUStartComPrimitive() with ComPrimitive type PDU_COPT_STARTCOMM (May include request and overwritten communication parameters if supplied by the MVCI diagnostic server, handling as described in Table 35)
MCDStopCommunication::executeSync	PDUStartComPrimitive() with ComPrimitive type PDU_COPT_STOPCOMM

8.28.6 D-PDU API IO-Control support

The D-PDU API uses IO-Controls to execute functions or set values related to an MVCI protocol module. Client applications often need to use some of the IO-Controls of a D-PDU API. An IO-Control is executed in context of a VCI or in context of a logical link. Their execution is always synchronous.

For the client to use IO-Controls on the MCD3 API some requirements are necessary. The D-PDU API function PDUIoCtl has a pInputData and pOutputData parameter. Both are C-pointers to a structure containing IO-Control specific data. The layout of the structure is dependent on the IO-Control and can only be known by the server for the IO-Controls specified within the ISO 22900-2 standard. The D-PDU API allows customer specific IO-controls. The memory layout of those structures cannot be determined at runtime from a 3D-Server. Hence it cannot build a database object and access the data members in a symbolic way. To use IO-Controls a memory based approach is used. Each C-Structure is finally reduced to some consecutive bytes in memory.

An MCDValue object is used to define the input parameter. Only the datatypes A_INT8, A_INT16, A_INT32, A_INT64, A_UINT8, A_UINT16, A_UINT32, A_UINT64 and A_BYTEFIELD can be used as input parameter, an output parameter is always of datatype A_BYTEFIELD. If the datatype is one of the integer values, the value is interpreted as a bytefield of the corresponding length filled with the value. Integer support is only for client's convenience on input parameters, the use of bytefields is recommended.

9 Error Codes

9.1 Principle

This chapter describes the errors which may occur. Working with the API, errors may occur during or after a method call or generally within the MVCI diagnostic server. The MVCI diagnostic server returns the resulting error objects (`MCDError`) via the API to the Client using different ways.

— If an error crops up during the method execution, that means before the method returns, this error is passed on by means of an Exception. In case of serious errors e.g. for `DiagComPrimitives` this is independent from asynchronous or synchronous execution.

— If any error occurs independent from a method call, for example a system wide error, this error is returned by means of an Event. The error object will either be located directly within the Event (`onSystemError`, `onLinkError`) and can be polled using `getError`, or the Event (`onPrimitiveError`) transports a `MCDResultState`, within which the error object is located.

— In case of a synchronous execution of a `DiagComPrimitive` and the cropping up of non serious errors, the error is handed over as return value in `MCDResultState` .

— In case of asynchronous execution of `DiagComPrimitives`, the error is returned within a `MCDResultState` object via an Event (`onPrimitiveError`).

Most of the methods in this object model can throw an exception, an `MCDException` or one of the from `MCDException` derived exceptions. These exceptions only transport the error object.

Types of exception:

— `MCDParameterizationException`
(inadmissible or inconsistent parameterisation for the execution of a method)

— `MCDProgramViolationException`
(Problem at program flow)

— `MCDDatabaseException`
(Problem at database access)

— `MCDSystemException`
(system wide problem)

— `MCDCommunicationException`
(Problem in Communication between MVCI diagnostic server and ECU)

— `MCDShareException`
(Problem at the handling of shared objects)

An error object consists of:

— Code,

— Code description,

— Severity,

— Vendor Code,

⎯ Vendor Code description

⎯ The **Code** is represented by a A_UINT16 and is defined in the following tables as a general definition of the errors. The code 0x0000 means error free.

The **Code description** of the error is represented by a A_ASCIISTRING and is defined in the following tables as a general definition of the errors.

The **Vendor Code** is represented by a A_UINT16 and intended as manufacturer specific error supplement (Information). The code is not standardized. The vendor code is be useable in all cases of an not empty code, that means a common standardized error must be used in code.

The **Vendor Code description** is represented by a A_ASCIISTRING and is provided by the manufacturer of the MVCI diagnostic server. These are the description of the manufacturer specific vendor error code.

The **Severity** is used for the assessment of the error and can be subdivided as defined in Table 37.

Table 37 — Severity

Severity	Short cut	Description
eMESSAGE	M	This is information with importance for the user. This does not change the execution path in the software.
eWARNING	W	A problem occurred and was successfully solved by the MVCI diagnostic server.
eERROR	E	A problem occurred and must be handled by the client. The object still exists and could react normally after the problem is solved by the respective clients.
eFATAL_ERROR	F	An unsolvable problem occurred at an object and the complete operation could not be performed or done work is lost. The object still exists, but cannot act normally and usually is not accessible. The problem cannot be completely handled or solved in the software.
eTERMINATE	T	An unsolvable problem occurred, the MVCI diagnostic server is in an unsafe state and shuts down immediately (the server / MVCI diagnostic server is not accessible / the object cannot be accessed anymore).

To unify the error handling with normal runtime response, the two Severity's eMESSAGE and eWARNING are used. An eMESSAGE or an eWARNING will never be reported by an exception, but could use all logging functionality of the error handler.

The main difference between eERROR and eFATAL_ERROR is, that after an eERROR the application could continue if it handles the error. eFATAL_ERROR means that something principle is not working correct. Even if the application handles the error, some damage was done or it's not possible to resolve the problem.

EXAMPLE

⎯ If the application could not open a data base-file, this is an eERROR, because no damage was done and the application could open a different database file.

⎯ If an ECU is disconnected, this is an eFATAL_ERROR, because the application itself could not reconnect the hardware and the operations could not be completed.

9.2   Description of the errors

9.2.1   Error free behaviour

If no error has occurred, but an error object is handed over within the Event or `MCDResultState` (this only in diagnostic part), the ErrorCode of the `MCDError` object is 0000. The error codes defined in Table 38 and Table 39 can be used by all method calls.

9.2.2   Parameterisation errors

This errors can be carried by `MCDParameterizationException`, every event or `MCDResultState`.

An error occurred on the basis of a wrongly set method parameter. The parameter has e.g. the wrong type or has a wrong value.

The exception can be solved by adjusting the parameter, so that it fits the defined constraints of the method.

Table 38 defines the MCD parameterisation errors.

Table 38 — MCD parameterisation errors

Error code	Stereo type	Severity	Enum identifier	Error text
0xC013	MCD	E	`ePAR_INVALID_DB_OBJECT`	the given database object has an invalid type
0xC011	MCD	E	`ePAR_INVALID_OBJECTTYPE`	invalid MCDObjectType
0xC014	MCD	E	`ePAR_VALUE_OUT_OF_RANGE`	parameter out of range
0xC010	MCD	E	`ePAR_INVALID_TYPE`	invalid type of input parameter

9.2.3   RunTime / ProgramViolation errors

This errors can be carried by `MCDCommunicationException`, `MCDShareException`, every event or `MCDResultState`.

ProgramViolation Errors are errors, where the client can do something or has provoked the error itself (usage error), e.g. false LogicalLink state, VI in false LL state (`eCREATED`).

Table 39 defines the MCD RunTime / Program Violation Errors.

Table 39 — MCD RunTime / Program Violation Errors

Error code	Stereo type	Severity	Enum identifier	Error-Text
0xD043	MCD	E	`eRT_INTERNAL_ERROR`	An internal error occurred in the MVCI diagnostic server. See vendor specific code for details.

9.2.4   Database errors

Database errors occur if an access to the ODX database has failed. An error occurred, while the MVCI diagnostic server tried to access a database entry. The database access can e.g. fail, if an entry is not filled with data. The exception can be solved by adjusting the data base.

This errors can be carried by `MCDDatabaseException`, every event or `MCDResultState`.

9.2.5 System errors

This errors can be carried by `MCDSystemException` or `onSystemError`.

SystemErrors are critical errors, where a client can do nothing (the error can be in OS or in MVCI diagnostic server), e.g memory overflow, division by zero.

9.2.6 Communication errors

This errors can be carried by `MCDCommunicationException` or `onSystemError`.

A problem occurred during the communication between the MVCI diagnostic server and the connected ECU (hardware). (e.g. ECU was removed from the MVCI diagnostic server, ECU does not respond, wrong ECU is attached.).

The exception can be solved by adjusting the hardware environment.

9.2.7 Share error

Problem at the handling of shared objects like locking problems. This ErrorObjects can be carried by `onSystemError` or `onLinkError`.

The error occurred, because a shared object could not be accessed. The object is e.g. used by another client or another thread. The exception is automatically solved after the shared object is released from the other binding.

Annex A
(informative)

Flash programming according to ISO 14230-3 KWP 2000

A.1 Content

This annex specifies the implementation of the MVCI diagnostic server methods `startDownloadDataBlock` and `startDownloadSegment` for ISO 14230-3 KWP2000.

A.2 Flash programming microsequence inside the MVCI diagnostic server

A.2.1 General

ISO 14230-3 KWP2000 protocol provides two possibilities to realize the download process. These are dependent on the diagnostic services used for the flash programming process:

— WriteMemoryByAddress: This service needs the number of bytes to be written as input parameter. Therefore this service cannot be used for the MVCI diagnostic server API methods, because the number of bytes cannot be set. There exits no mechanism to be able to parameterise the download methods.

— RequestDownload with TransferData: The number of bytes to be written is returned as response from the service RequestDownload. Therefore no parameterisation is necessary for this flash process.

As defined above only the second possibility can be used for the MVCI diagnostic server flash methods, because it does not need to be parameterised. The sequence for the download is described in the following:

A.2.2 RequestDownload

The RequestDownload service is used by the MVCI diagnostic server to initiate a data transfer from the MVCI diagnostic server to the ECU. After the ECU has received the RequestDownload request message the ECU shall take all necessary actions to receive data before it sends a positive response message.

The transferRequestParameter is used by the requestDownload request message. This parameter consists of several other parameters which are required by the ECU to prepare for the data transfer from the MVCI diagnostic server to the ECU. Each parameter is specified in detail:

— The parameter **memoryAddress** identifies the start address in the ECU's memory. If the ECU supports a 16 bit wide address range the high byte of the memoryAddress shall be used as a memoryIdentifier. The memoryIdentifier shall be vehicle manufacturer/system supplier/project specific. This information is extracted from the ECU MEM during the execution of the method `startDownloadDataBlock / startDownloadSequence` in the MVCI diagnostic server. The FlashJob does not have to set it.

— The parameter **dataFormatIdentifier** is a one byte identifier. Each nibble shall be encoded separately. The high nibble specifies the compressionMethod and the low nibble specifies the encrypting Method. The use of a data compression method saves overall data transfer time if large amount of data shall be transferred between MVCI diagnostic server and ECU. Using an encoding method improves tamper protection. For the MVCI diagnostic server methods `startDownloadDataBlock / startDownloadSegment` this parameter is set to the default values:

   — Hex 0x which means uncompressed

— Hex x0 which means unEncrypted.

— The parameter **uncompressedMemorySize** is a three byte value. This parameter shall be used by the ECU to compare the uncompressed memory size with the total amount of data transferred until the requestTransferExit service. This increases the programming security. This information is extracted from the ECU MEM during the execution of the method `startDownloadDataBlock / startDownloadSequence` in the MVCI diagnostic server. The FlashJob does not have to set it.

The positive response message is sent, if the ECU is ready to receive data from the MVCI diagnostic server. It used the transferResponseParameter "maxNumberOfBlockLength" to inform the MVCI diagnostic server how many data bytes shall be included in each transferData request message from the MVCI diagnostic server. This parameter allows the MVCI diagnostic server to adapt to the received buffer size of the ECU before it starts transferring data to the ECU.

After the MVCI diagnostic server has received the positive response message, the next service TransferData can be sent. If the MVCI diagnostic server receives a negative response message, the flash process is canceled an exception is thrown.

A.2.3 TransferData

The TransferData service is used by the MVCI diagnostic server to transfer data either from the MVCI diagnostic server to the ECU (download) or from the ECU to the MVCI diagnostic server (upload). The data transfer direction is specified by the preceding RequestDownload or RequestUpload service. If the MVCI diagnostic server initiates a RequestDownload the data, to be downloaded, is included in the parameter(s) transferRequestParameter. The transferData service shall only be terminated by the requestTransferExit service.

If the ECU returns a TransferData Positive Response Message, the next bytes can be written into the ECU by calling transferData again. If the ECU returns a negative Response Message the flash process is cancelled and an exception is thrown.

The TransferData service is repeatedly called until all data was written to the ECU. Afterwards the requestTransferExit service is called.

A.2.4 RequestTransferExit

This service is used by the MVCI diagnostic server to terminate a data transfer between MVCI diagnostic server and ECU. The user's optional parameters transferRequestParameter and transferResponseParameter shall be of any type and length specified within ISO 14230-3 KWP 2000.

If the ECU returns a Positive Response Message the methods startDownloadDataBlock / startDownloadSequence is terminated normally. If the ECU returns a Negative Response Message a exception is thrown and the flash process cancelled.

In Figure A.1 the described sequence is illustrated for the positive case.

Figure A.1 — RequestDownload, TransferData, RequestTransferExit

Annex B
(normative)

Value reading and setting by string

B.1 Datatype conversion into Unicode2 string

The following rules for the method `MCDValue:getValueAsString` are defined Table B.1.

Table B.1 — Data type conversion

ASAM Data type	Unicode2 String
eA_ASCIISTRING	position by position, character by character
eA_BITFIELD	position by position, bit by bit (other chars than 0 and 1 are ignored)
eA_BYTEFIELD	position by position (excluding non-hex characters, including A-Fa-f)
eA_FLOAT64	normalized form: 0.0123 = 0.123 E-1 (cf. e.g. IEEE Standard 754 Floating Point Numbers)
eA_FLOAT32	normalized form: 0.0123 = 0.123 E-1 (cf. e.g. IEEE Standard 754 Floating Point Numbers)
eA_INT8	position by position, digit by digit, with leading sign if negative
eA_INT16	position by position, digit by digit, with leading sign if negative
eA_INT32	position by position, digit by digit, with leading sign if negative
eA_INT64	position by position, digit by digit, with leading sign if negative
eA_UNICODE2STRING	nothing to do
eA_UINT8	position by position, digit by digit
eA_UINT16	position by position, digit by digit
eA_UINT32	position by position, digit by digit
eA_UINT64	position by position, digit by digit

B.2 Representation floating numbers

The string argument of setValueAsString() must have the following form:

[white space] [sign] [digits] [.digits] [{d | D | e | E }[sign]digits]

If no digits appear before the decimal point, at least one must appear after the decimal point.

white space consists of space and/or tab characters, which are ignored;

sign is either plus (+) or minus (−);

The internal format of floating values in `MCDDValue` objects is a normalized floating number. The output string delivered by `getValueAsString()` is also in normalized format.

B.3 Normalized floating-point numbers

sign * mantissa * radix exponent

For each floating-point number there is one representation that is said to be normalized. A floating-point number is normalized if its mantissa is within the range defined by the following relation:

1/radix <= mantissa < 1

A normalized radix 10 floating-point number has its decimal point just to the left of the first non-zero digit in the mantissa. The normalized floating-point representation of -5 is -1 * 0.5 * 10 1. In other words, a normalized floating-point number's mantissa has no non-zero digits to the left of the decimal point and a non-zero digit just to the right of the decimal point. Any floating-point number that doesn't fit into this category is said to be demoralized. Note that the number zero has no normalized representation, because it has no non-zero digit to put just to the right of the decimal point.

Annex C
(normative)

System parameter

C.1 Overview

The base of all time dependent system parameter is the local time.

Represented time information is based on ISO 8601 standard "Data elements and interchange formats – Information interchange – Representation of dates and times" (see ISO 8601).

Table C.1 defines the System parameter.

Table C.1 — System parameter

ASCIISTRING for system parameter	data type (physical)	coding	example
TIMEZONE	A_INT32	Count of minutes to UTC	+60 (Berlin)
YEAR	A_UINT32	YYYY	2004
MONTH	A_UINT32	MM	03 (march)
DAY	A_UINT32	DD	01 (first day of month)
HOUR	A_UINT32	hh	22 (10 pm)
MINUTE	A_UINT32	mm	00 (full hour)
SECOND	A_UINT32	ss	00 (full minute)
TESTERID	A_BYTEFIELD		"00F056"
USERID	A_BYTEFIELD		"043FF0"
CENTURY	A_UINT32	CC	20 (in year 2004)
WEEK	A_UINT32		04 (fourth week of the year)

C.2 Description of the system parameters

C.2.1 TIMEZONE

The timezone is coded in an A_INT32 value. The range is between –720 and +780.

Table C.2 defines examples for timezones.

Table C.2 — Examples for timezones

Minutes	UTC time offset	Related Towns
- 720	UTC – 12h	Eniwetok, Kwajalein
- 660	UTC – 11h	Midway islands, Samoa
- 600	UTC – 10h	Hawaii
…		
- 480	UTC – 8h	Los Angeles, Seattle, Vancouver
…		
- 300	UTC – 5h	New York, Atlanta, Detroit, Toronto
…		
-60	UTC – 1h	Azores
0	UTC	London
+60	UTC + 1h	Berlin, Rome
+120	UTC + 2h	Athens, Istanbul
+180	UTC + 3h	Moscow, Nairobi
+210	UTC + 3,5h	Teheran
+240	UTC + 4h	Abu Dhabi
…		
+720	UTC + 12h	Aukland, Wellington
+780	UTC + 13h	Nuku'alofa

C.2.2 YEAR

The year is coded in an A_UINT32 value with four digits.

C.2.3 MONTH

The year is coded in an A_UINT32 value with two digits. It is starting with January as 01 up to December with 12.

C.2.4 DAY

The day is coded in an A_UINT32 value with two digits. It is starting with the first day of the month as 01 up to the last day of the month.

C.2.5 HOUR

The hour is coded in an A_UINT32 value with two digits in a 24 hours rhythm. It is starting with 12am as 00 (via 1am as 01) up to 11pm as 23.

C.2.6 MINUTE

The minute is coded in an A_UINT32 value with two digits. It is starting with 00 (full hour) up to 59.

C.2.7 SECOND

The second is coded in an A_UINT32 value with two digits. It is starting with 00 (full minute) up to 59.

C.2.8 TESTERID

The tester ID is coded in an A_BYTEFIELD.

C.2.9 USERID

The user ID is coded in an A_BYTEFIELD.

C.2.10 CENTURY

The century is coded in an A_UINT32 value with two digits. This value contains the first two digits of the four-digit year (unlike language usage).

C.2.11 WEEK

The century is coded in an A_UINT32 value with two digits.

The first week of a year is the first week which includes at least four days of the new year. Alternatively, the first week of a year is the week which includes the first Thursday of January and January 4. As a result, week 01 of a year can contain days of the previous year and week 53 can contain days of the following year. For example, 2004-01-01 is a Thursday. Hence, 2004-W01 does comprise the days 2003-12-29 to 2004-01-04. Furthermore, 2005-01-01 is a Saturday. As a result, 2004-W53 comprises the days 2004-12-27 to 2005-01-02. Week 2005-W01 starts on 2005-01-03.

Annex D
(normative)

Deprecated methods and classes

Table D.1 defines the deprecated methods, classes and enum values.

Table D.1 — Deprecated methods, classes and enum values

Deprecated	New
MCDDbFlashSession :: <<D>> getDbFlashJob	MCDDbFlashSession :: <<D>> getDbFlashJobByLocation
MCDDbIdentDescription :: <<D>> getDbDataPrimitive	MCDDbIdentDescription :: <<D>> getDbDataPrimitiveByLocation
MCDDiagComPrimitives	---
MCDDiagComPrimitives :: <<D>> getItemByIndex	---
MCDResponse :: <<M,C,D>> getResponseMessage	MCDResponse :: <<M,C,D>> getContainedResponseMessage
MCDDbParameter ::<<D>> getBytePos ::<<D>> getParameterType ::<<D>> getInterval	MCDDbParameter ::<<D>> getODXBytePos ::<<D>> getMCDParameterType ::<<D>> getValidInternalIntervals
MCDParameter ::<<D>> getType	MCDParameter ::<<D>> getDataType
Enum Values of MCDDataType eKEY eLENGTHKEY eTABLE	---
MCDDbLocation ::<<D>> getDbServicesBySemanticAttribute ::<<D>> getDbEnvDataByTroubleCode	MCDDbLocation ::<<D>> getDbDiagComPrimitivesBySemanticAttribute ::<<D>> getDbEnvDataDescs
MCDDbLocation ::<<D>> getDbDynDefinedSpecTableByDefinitionMode ::<<D>> getDbDynDefinedSpecTableByName ::<<D>> getDbDynDefinedSpecTables	MCDDbLocation ::<<D>> getDbTableByDefinitionMode
MCDJobAPI ::<D> startDownloadDataBlock ::<D> startDownloadSegment	---
MCDLogicalLink ::<D> setUnitGroup ::<D> getUnitGroup	---
MCDDbDiagTroubleCode ::<D> getDTCText	MCDDbDiagTroubleCode ::<D> getUnicodeDTCText

Table D.1 *(continued)*

Deprecated	New

MCDDbFlashSession
   :: <<D>> getDbFlashJob
	MCDDbFlashSession
	   :: <<D>> getDbFlashJobByLocation

MCDDbIdentDescription
   :: <<D>> getDbDataPrimitive
	MCDDbIdentDescription
	   :: <<D>> getDbDataPrimitiveByLocation

MCDDiagComPrimitives	---
MCDDiagComPrimitives :: <<D>> getItemByIndex	---

MCDResponse
   :: <<M,C,D>> getResponseMessage
	MCDResponse
	   :: <<M,C,D>> getContainedResponseMessage

MCDDbParameter
  ::<<D>> getBytePos
  ::<<D>> getParameterType
  ::<<D>> getInterval
	MCDDbParameter
	  ::<<D>> getODXBytePos
	  ::<<D>> getMCDParameterType
	  ::<<D>> getValidInternalIntervals

MCDParameter
  ::<<D>> getType
	MCDParameter
	  ::<<D>> getDataType

Enum Values of MCDDataType
    eKEY
    eLENGTHKEY
    eTABLE
	--- 	

MCDDbLocation
  ::<<D>> getDbServicesBySemanticAttribute
 	
  ::<<D>> getDbEnvDataByTroubleCode
	MCDDbLocation
	::<<D>> getDbDiagComPrimitivesBySemanticAttribute
	::<<D>> getDbEnvDataDescs

An exception will be thrown, if the DEPRECATED method is not supported (eSYSTEM_METHOD_NOT_SUPPORTED, MCDSystemException).

Annex E
(normative)

Overview optional functionalities

Table E.1 defines the optional methods and classes.

Table E.1 — Optional methods and classes

Optional functionality	Optional methods	Optional classes
MCDDbProject-configuration	MCDSystem ::getDbProjectConfiguration()	MCDDbProjectConfiguration

ECU Configuration (Variantcoding)	MCDLogicalLink
	
	    ::getConfigurationRecords
	
	MCDDbLocation
	
	    ::getDbConfigurationDatas
		MCDConfigurationRecords
		MCDConfigurationRecord
		MCDConfigurationItem
		MCDConfigurationIdItem
		MCDDataIdItem
		MCDDbConfigurationData
		MCDDbConfigurationDatas
		MCDDbConfigurationItem
		CDDbConfigurationRecord
		MCDDbConfigurationRecords
		MCDDbDataRecord
		MCDDbDataRecords
		MCDDbSystemItems
		MCDDbSystem
		MCDDbCodingData
		MCDDbDataIdItem
		MCDDbConfigurationIdItem
		MCDDbOptionItems
		MCDDbOptionItem
		MCDDbItemValues
		MCDDbItemValue
		MCDSystemItems
		MCDSystemItem
		MCDOptionItems
		MCDOptionItem

Optional functionality	Optional methods	Optional classes
ECU RE Programming (Flashing)	MCDDbProject ::getDbEcuMems MCDDbProject ::loadNewEcuMem MCDDbLocation ::getDbFlashSessionClasses ::getDbFlashSessions ::getDbPhysicalMemories	MCDDbFlashChecksum MCDDbFlashChecksums MCDDbFlashData MCDDbFlashDataBlock MCDDbFlashDataBlocks MCDDbFlashFilter MCDDbFlashFilters MCDDbFlashIdent MCDDbFlashIdents MCDDbFlashJob MCDDbFlashSecurities MCDDbFlashSecurity MCDDbFlashSegment MCDDbFlashSegments MCDDbFlashSession MCDDbFlashSessionClass MCDDbFlashSessionClasses MCDDbFlashSessions MCDFlashJob MCDFlashSegmentIterator MCDDbPhysicalMemories MCDDbPhysicalMemory MCDDbPhysicalSegment MCDDbPhysicalSegments MCDDbEcuMem MCDDbEcuMems MCDDbIdentDescription
DynID	MCDDbLocation ::getSupportedDynIds MCDLogicalLink ::createDynIdComPrimitiveByTypeAndDefinitionMode ::getDefinableDynIds MCDDbLocation ::getDbTableByDefinitionMode	MCDDbDynIdDefineComPrimitive MCDDynIdDefineComPrimitive MCDDbDynIdClearComPrimitive MCDDynIdClearComPrimitive MCDDbDynIdReadComPrimitive MCDDynIdReadComPrimitive
Monitoring	MCDProject ::createMonitoringLink	MCDMonitoringLink MCDMessageFilter MCDMessageFilters MCDMessageFilterValues MCDMessageFilterTypes
System Properties	MCDSystem ::getProperty ::getPropertyNames ::resetProperty ::setProperty	

Table E.1 — *(continued)*

Optional functionality	Optional methods	Optional classes
Function Dictionary	MCDDbProject ::getDbFunctionDictionaries	MCDDbFunctionDictionaries MCDDbFunctionDictionary MCDDbFunctionNodes MCDDbFunctionNode MCDDbBaseFunctionNode MCDDbFunctionNodeGroups MCDDbFunctionNodeGroup MCDDbComponentConnectors MCDDbComponentConnector MCDDbFunctionInParameters MCDDbFunctionInParameter MCDDbFunctionOutParameters MCDDbFunctionOutParameter MCDDbDiagObjectConnector MCDDbDiagTroubleCodeConnectors MCDDbDiagTroubleCodeConnector MCDDbEnvDataConnectors MCDDbEnvDataConnector MCDDbFunctionDiagComConnectors MCDDbFunctionDiagComConnector MCDDbTableRowConnectors MCDDbTableRowConnector
SubComponents	MCDDbLocation ::getDbSubComponents	MCDDbSubComponents MCDDbSubComponent MCDDbDiagTroubleCodeConnectors MCDDbDiagTroubleCodeConnector MCDDbEnvDataConnectors MCDDbEnvDataConnector MCDDbTableRowConnectors MCDDbTableRowConnector MCDDbSubComponentParamConnectors MCDDbSubComponentParamConnector

Table E.1 — *(continued)*

Optional functionality	Optional methods	Optional classes
Audiences	MCDDbItemValue ::getAudienceState ::getDbDisabledAdditionalAudiences ::getDbEnabledAdditionalAudiences MCDDbFlashDataBlock ::getAudienceState ::getDbDisabledAdditionalAudiences ::getDbEnabledAdditionalAudiences MCDDbDataRecord ::getAudienceState ::getDbDisabledAdditionalAudiences ::getDbEnabledAdditionalAudiences MCDDbConfigurationRecord ::getAudienceState ::getDbDisabledAdditionalAudiences ::getDbEnabledAdditionalAudiences MCDDbDataPrimitive ::getAudienceState ::getDbDisabledAdditionalAudiences ::getDbEnabledAdditionalAudiences MCDDbFunctionNode ::getDbDisabledAdditionalAudiences ::getDbEnabledAdditionalAudiences MCDDbFunctionNodeGroup ::getDbDisabledAdditionalAudiences ::getDbEnabledAdditionalAudiences MCDDbTableParameter ::getAudienceState ::getDbDisabledAdditionalAudiences ::getDbEnabledAdditionalAudiences MCDDbOptionItem ::getDbDisabledReadAdditionalAudiences ::getDbEnabledReadAdditionalAudiences ::getDbDisabledWriteAdditionalAudiences ::getDbEnabledWriteAdditionalAudiences	MCDDbAdditionalAudience MCDAudience

Table E.1 — *(continued)*

Optional functionality	Optional methods	Optional classes
Audiences	MCDDbECUMEM 　::getDbAdditionalAudiences MCDDbLocation 　::getDbAdditionalAudiences MCDDbConfigurationData 　::getDbAdditionalAudiences MCDDbBaseFunctionNode 　::getAudienceState MCDDbFunctionDictionary 　::getDbAdditionalAudiences MCDAudience 　::isAfterMarket 　::isAfterSales 　::isDevelopment 　::isManufacturing 　::isSupplier	MCDDbAdditionalAudience MCDAudience
ECU State	MCDDbLocation 　::getDbECUStateCharts MCDDbDiagComPrimitive 　::getDbECUStateTransitionsByDbObject 　::getDbECUStateTransitionsBySemantic 　::getDbPreConditionStatesByDbObject 　::getDbPreConditionStateBySemantic	MCDDbEcuState MCDDbEcuStateChart MCDDbEcuStateCharts MCDDbEcuStates MCDDbEcuStateTransition MCDDbEcuStateTransitionAction MCDDbEcuStateTransitionActions MCDDbEcuStateTransitions
Multiple ECU Jobs	MCDAccessKey 　:getMultipleEcuJob MCDDbProject 　:getDbMultipleEcuJobLocation MCDDbBaseFunctionNode 　:getDbMultipleEcuJobs	MCDDbMultipleEcuJob MCDMultipleEcuJob
PDU TimeStamps	MCDResponse 　::getStartTime 　::getEndTime MCDResult 　::getRequestEndTime	
Library	MCDDbJob 　::getDbCodeInformations	MCDDbCodeInformations MCDDbCodeInformation

Table E.1 — *(continued)*

Optional functionality	Optional methods	Optional classes

DoIP	MCDSystem
	    ::detectInterfaces
	MCDInterface
	    ::detectInterfaces	

PDU API	MCDInterface
	    ::getMVCIVersionPart1StandardVersion
	    ::getMVCIVersionPart2StandardVersion
	    ::getPDUApiSoftwareName
	    ::getPDUApiSoftwareVersion
	    ::getVendorName
	    ::execIOCtrl
	MCDDbInterfaceConnectorPin
	    ::getPinNumberOnVCI
	MCDLogicalLink
	    ::execIOCtrl
	MCDProject
	    ::execIOCtrl	

Annex F
(informative)

Monitoring message format

F.1 General

Also, the definitions in this chapter should be taken as an implementation guideline – depending e.g. on specifics of the protocol driver implementation layer of an actual system, message type flags or other implementation details can differ from the definitions in this appendix. Other protocol types might require different/additional message format definitions.

F.2 CAN Format

A format for monitored messages as returned by the method `MCDMonitoringLink::fetchMonitoringFrames()` is defined in Table F.1.

Table F.1 — Format of monitored message CAN

Type	Time stamp	Address	Length	Data
RX	00000000000011700068	0000012D	8	0F 0F 00 00 30 CD 85 AC
TX	00000000000011746938	000003F6	5	03 21 02 0A 00

Message type:

The message type definitions are dependent on the actual protocol driver implementation used by the system. For example, when using a standard D-PDU API driver and the standard link types available to such a driver (e.g. an ISO_11898_RAW protocol link), it might not even be possible to discern between RX (receive) and TX (transmit) messages, as the monitoring link is in a receiver-only role and doesn't transmit any messages of its own. In case the actual implementation lends itself to a more detailed discrimination of messages types, the following type definitions should be used. Additional types may be defined by the kernel.

In any case the message type shall be of two characters to ensure constant positioning.

— RX (Receive)

— TX (Transmit)

Examples for CAN-specific message types:

— ES (ERROR_STUFF)   Bit stuffing error, more than 5 consecutive bits of equal polarity

— EF (ERROR_FORM)   Form error, e.g. violation of end of frame (EOF) format

— EA (ERROR_ACK)   ACK error, transmitting node receives no dominant acknowledgement bit

— EC (ERROR_CRC)   CRC error, received CRC code does not match calculated code

Timestamp:

20 digit decimal (microseconds), e.g. `00000001234560089768`

The timestamp is only useful for putting messages into chronological relation to each other. As multiple controllers are potentially involved in the timestamp generation for logged messages (CAN-controller, VCI hardware controller, the controller that is running the MVCI diagnostic server, the controller that is running the client application, etc.), it is not feasible to provide an exhaustive definition in the scope of this standard. A MVCI diagnostic server uses the timestamp from the message delivered by the underlying VCI access layer. Otherwise the timestamp is generated by the kernel, which may lead to inaccurate values. The maximum value shall be fit into a 64 bit integer.

Communication node address:

8 digits hexadecimal, padded with leading zeros if necessary. In front of the address there will be put a "." and in case of 11 bit length addressing and a "*" in case of 29 bit addressing mode. In case there is no address information available, the address column shall consist of 8 dots ('……..').

Data length:

Decimal value.

Data stream:

Two-digit hexadecimal, e.g. `47 61 6C 65 6F 20 05 67` (separation by blanks, no leading 0x)

General rules:

⎯ Data blocks (message type, timestamp, address etc.) within a message are separated by blanks (' ')

In case an error (e.g. a buffer overrun) occurred in the communications hardware, the following message is to be inserted into the monitoring data: `"### <Error description>"`

F.3 K-Line Format

A format for monitored messages as returned by the method `MCDMonitoringLink::fetchMonitoringFrames()` is defined in Table F.2.

Table F.2 — Format of monitored message K-Line

Type	Time stamp	Data
RX	00000000000011700068	3C

Message type:

⎯ RX (Receive)

⎯ TX (Transmit)

⎯ GI (GPD-Insert)

⎯ KL (KLINE-LOW)

⎯ KH (KLINE-HIGH)

— FL (Flush-Later)

— FS (Flush)

Timestamp:

See CAN.

F.4 DoIP Format

A format for monitored messages as returned by the method `MCDMonitoringLink::fetchMonitoringFrames()` is defined in Table F.3.

Table F.3 — Format of monitored message DoIP

Type	Time stamp	Data
TT	00000000000011700068	6E 0A2D 012D 12 0F 0F 00 00 30 CD 85 AC 30 CD 85 AC

Message type:

— TT (TCPSend)

— TR (TCPRecv)

— UT (UDPSend)

— UR (UDPRecv)

— UB (UDPBroadcast)

Timestamp:

See CAN.

Data:

Frame according ISO 13400-2.

Bibliography

[1] ISO°4092:1988/Cor.1:1991, *Road vehicles - Testers for motor vehicles - Vocabulary Technical Corrigendum 1*

[2] ISO/IEC°7498-1:1984 *Information processing systems - Open systems interconnection - Basic reference model*

[3] ISO/IEC°10731*, Information technology — Open Systems Interconnection — Basic Reference Model — Conventions for the definition of OSI services*

[4] ASAM-Data, *ASAM Data Types V 2.0*
source: http://www.asam.net

[5] ANSI/IEEE Std 754-1985, *Binary Floating-Point Arithmetic for microprocessor systems*
source: http://ieeexplore.ieee.org/

[6] IEEE Std 1003.1-2004, *Portable Operating System Interface*
source: http://ieeexplore.ieee.org/

[7] RFC 3305 *Uniform Resource Identifiers (URIs), URLs, and Uniform Resource Names (URNs),*
source: http://www.ietf.org/

[8] ASAM Technology Reference COM-IDL, *COM-IDL Technology Reference Mapping Rules,*
source: http://www.asam.net

[9] ASAM Technology Reference C++, *C++ Technology Reference Mapping Rules,*
source: http://www.asam.net

[10] ASAM Technology Reference Java, *Java Technology Reference Mapping Rules,*
source: http://www.asam.net