



## INRIA INNOVATION LAB

### CERTIVIBE

V1.0

### MODULES DETAILED DESIGN SPECIFICATIONS

Document Approval			
	Name	Function	Date
Originated by	Charles Garraud	Development team	12/01/2015
Reviewed by	Cédric RIOU	Development team	21/09/2017
Approved by	Benoît Perrin	Project Manager	16/02/2018

## HISTORY

Version	Author	Date	Comments
01	CG	12/01/2015	Creation of document
02	CRIO	21/09/2017	Add some missing plugins in Plugins Components List section

# TABLE OF CONTENTS

1.	Document Information.....	5
1.1	Document Roadmap.....	5
1.1.1	Objectives .....	5
1.1.2	Document Overview .....	5
1.1.3	Document User Guide .....	6
1.1.4	References .....	7
1.1.5	Definitions .....	7
1.1.6	Abbreviations .....	7
1.2	Architectural View Documentation.....	8
1.2.1	View Description .....	8
1.2.2	Views Listing .....	8
2.	System Data Structures .....	9
2.1	C++ Object Data Structures .....	9
2.1.1	Matrix .....	9
2.1.2	Stimulation Set .....	11
2.1.3	Memory Buffer .....	11
2.2	Stream Structures.....	11
2.2.1	Definition .....	11
2.2.2	Stream Structure Specification.....	13
2.2.3	Stream Hierarchy .....	14
2.2.4	Stream Encoding/Decoding.....	15
2.3	Structures Identification.....	19
3.	System Logical Units .....	23
3.1	Plugin Mechanism .....	23
3.1.1	Plugin API.....	23
3.1.2	Plugin Callbacks .....	26
3.1.3	Plugin Management .....	26
3.2	Algorithm.....	27
3.2.1	Algorithm Prototype.....	27

3.2.2	Algorithm Core .....	27
3.3	Box Algorithm .....	29
3.3.1	Box Algorithm Prototype .....	29
3.3.2	Box Algorithm Core .....	30
3.4	Box Listener .....	32
4.	Kernel Management .....	33
5.	Algorithm Management .....	36
6.	Scenario Management .....	40
6.1	Scenario Creation .....	41
6.2	Scenario Loading/Saving .....	46
7.	Scenario Playback .....	47
7.1	Data Acquisition .....	47
7.2	Execution Workflow .....	48
7.3	Timing .....	57
7.3.1	Time Model .....	57
7.3.2	Time Representation .....	59
7.3.3	System Clock .....	59
8.	Configuration Management .....	62
8.1	Configuration Token .....	62
8.2	Box Settings Customization .....	64
9.	Log and Error Management .....	66
9.1	System Logging .....	66
10.	Appendix .....	69
10.1	Stream Structure Specifications .....	70
10.2	Plugins Components List .....	80
10.3	Standard Configuration Tokens .....	84
10.4	Error Management Codes .....	<b>Erreur ! Signet non défini.</b>

## 1. Document Information

### 1.1 Document Roadmap

#### 1.1.1 Objectives

The purpose of the document is to provide a detailed description of the software system.

As it is not possible to describe every single architecture decision, emphasis is put on:

- Architectural decision that are directly driven from SRS,
- Parts of the system that are difficult to apprehend.

If the automatically generated documentation and/or the code are clear enough to understand underlying design choices, it might not be detailed in this document.

#### 1.1.2 Document Overview

The document structure results from the functional overview presented in SDD.



The document is divided into subsections:

CertiViBE - v1.0 Modules Detailed Design Specifications	CERT-01 MSD-01	Page 5 / 86
--	----------------	-------------

All rights reserved. Passing on or copying of this document, use and communication of its contents not permitted without written authorization

- provides information about the purpose and use of the document;
- Other sections present the software system from different perspectives.

In each section, some architectural views are presented as defined in SDD. Refer to §1.2.2 Views Listing for the list of descriptive views provided in this document. Moreover, some paragraphs are emphasized with identifiable icons:

	Information section used to present an essential information.
	Focus section used to focus on a particular aspect of the software.

### 1.1.3 Document User Guide

This document can be explored in different ways according to the stakeholder expectations:

- Software developers or scientist can browse all sections to get better understanding on how the system works;

QA team members can explore

to inspect the scope of the document check requirements are handled at detailed design level.

Different references can be found in the document:

- SRS-XXX identifiers are used to reference requirements (see SRS);
- RSK-XXX identifiers are used to reference risks;
- SDD-XXX identifiers are used to reference architectural views presented in SDD;
- MSD-XXX identifiers are used to reference architectural views presented in this document.

#### 1.1.4 References

DOCUMENT #	TITLE
93/42EEC	Medical Device Directive
EN ISO 13485	Quality systems – Medical devices – System requirements for regulatory purposes
EN ISO 14971	Medical Devices - Application of Risk Management to Medical Devices
EN ISO 62304	Medical Device Software - Software Life Cycle Processes
MEDDEV 2.1/6 January 2012	Qualification and Classification of standalone software
SRS	Software Requirement Specifications
SDD	Software Design Description

#### 1.1.5 Definitions

See SDD.

#### 1.1.6 Abbreviations

See SDD.

#### 1.1.7 Tools

UML diagrams in this document were generated with plantUML that generates diagram from simple text files.

## 1.2 Architectural View Documentation

### 1.2.1 View Description

See SDD.

### 1.2.2 Views Listing

Views are given a unique incremental identifier (MSD-XXX) to be used as reference in other QA documents.

ID	View Reference	Description
<i>MSD-XXX</i>		<i>overview</i>
...		

## 2. System Data Structures

This chapter describes some transversal concepts related to data structures used across the system.

This two first sections deals with the description of the different kind of data structures that are manipulated within the system. Two types of structures are currently involved:

- Data structures implemented as C++ classes are described in §2.1;
- Stream structures containing bit stream data are described §2.2. The information that is transmitted from a box to another box in the pipeline is packaged into unit of data whose structure complies with data layout formatting requirements (stream structures).

§2.3 Structures Identification deals with the identification process used to distinguish entities within the framework. This concept is essential to the recognition and manipulation of any type of data structures.

### 2.1 C++ Object Data Structures

This section focuses on three fundamental objects that are manipulated across the system. These objects are all part of the base framework as defined in SDD:

- Matrices, instances of `CMatrix` class and manipulated through `IMatrix` interface;
- Stimulation sets, instances of `CStimulationSet` class and manipulated through `IStimulationSet` interface;
- Memory buffers, instances of `CMemoryBuffer` class and manipulated through `IMemoryBuffer` interface.

#### 2.1.1 Matrix

Matrices are multi-purposes data containers (e.g. EEG signal data container, feature vector data container) that can be used within boxes as input/temporary/output data container.

Matrices are represented as a tensor with arbitrary number of dimensions, from 0 to up any n.



Matrix as input data container: boxes typically expect matrices with definite properties as input data (e.g. dimensionality). These properties have to be checked programmatically within each box.

Matrix can be stored into a file. The following view describes the storage format.

CertiViBE - v1.0 Modules Detailed Design Specifications	CERT-01 MSD-01	Page 9 / 86
--	----------------	-------------

All rights reserved. Passing on or copying of this document, use and communication of its contents not permitted without written authorization

## IV-FileFormat-MATRIX

### Primary Presentation

```
# Comments
# header
[
    # example: dimension 1 of size 2
    ["label 11" "label 12"]
    # example: dimension 2 of size 3
    ["label 21" "label 22" "label 23"]
    ...
    # dimension n
    ["label n1" "label n2"]
]
# buffer
[ // -> dimension 1, label11
    [ // -> dimension 2, labe21
        ... // -> other dimensions
        [val1 val2] // dimension n
    ]
    [ // -> dimension 2, labe21
        ...
        [val1 val2]
    ]
    [ // -> dimension 2, labe23
        ...
        [val1 val2]
    ]
]
[// -> dimension 1, label12
    ...
]
# end of buffer
```

### Element Catalog / Description

CertiViBE - v1.0	CERT-01 MSD-01	Page 10 / 86
Modules Detailed Design Specifications		

All rights reserved. Passing on or copying of this document, use and communication of its contents not permitted without written authorization

Matrix files are divided into a header and a buffer section.

The header part is delimited by opening and closing brackets and contains one [ ] section per dimension:

- Each [ ] section contain the dimension labels;
- The number of label must match the dimension size (empty labels are allowed).

The buffer part is built recursively on n dimensions contained in [ ] sections with innermost dimension being linearized.

### 2.1.2 Stimulation Set

Stimulation sets are more specific than matrices. They are specially used to contain a collection of OpenViBE stimulations, each stimulation being represented by:

- An identifier (see §2.3 Structures Identification);
- A date;
- A duration.



OpenViBE stimulations were meant to represent sensory excitation used as trigger in EEG brain signal experiments (e.g. light stimuli, beep). But its use was extended to represent any event (e.g. keyboard press, labeling event, experiment management event).

### 2.1.3 Memory Buffer

Memory buffers are used as raw data bits container that can be manipulated with no specific care on memory allocation. They are especially used to contain bit stream data transmitted between boxes (see §2.2 Stream).

## 2.2 Stream Structures

### 2.2.1 Definition

As described in §3.3 Box Algorithm, boxes potentially receive input data, process them and send produced output data to the next box in the pipeline.

A stream can be seen as a virtual pipe between two boxes with bit data transiting through the pipe into fragmented chunks (i.e. packet). In the system, a stream is identified by a unique identifier and is defined by the structure of data chunks allowed in the pipe (**stream structure specification**). The data structure is defined using EBML (see [EBML Specifications](#)) and represents the formal description of the ordering and meaning of bytes within a chunk (chunk data layout).

CertiViBE - v1.0 Modules Detailed Design Specifications	CERT-01 MSD-01	Page 11 / 86
--	----------------	--------------



Streams are no C++ object.

Stream types just lead to structural specifications. This leads to significant architectural consequences on the manipulation of input and output data by boxes (see §2.2.4 Stream Encoding/Decoding).

This design choice allows the transmission of data over the network for remote/scattered multiprocessing within a pipeline. Currently, the player does not take advantage of computation distribution. However, the stream concept makes such distribution possible and easier because box algorithms do not share information directly.



The transmission of data along the pipeline is a set of disconnected segments and not a flow as a box can expect a given stream type as input and produces another stream type as output.

The following table describes the different stream types available in the system and the information it aims at conveying.

Stream Type	Data description
EBML	Convey information about the stream type and version (not used directly).
Streamed Matrix	Convey information represented as Matrix.
Channel Localization	Electrodes Cartesian coordinates information.
Channel Units	Convey information about channel measurement units.
Feature Vector	Convey feature vectors for classification purposes.
Spectrum	Convey spectrum analysis results.
Signal	Convey EEG signal data on multiple channels.
Stimulation	Convey stimulation data.
Experiment Information	Convey information on the experiment being conducted.
Acquisition	Multiplexed stream conveying information of a Signal stream, a Channel Localization stream, an Experiment Information stream, a Channel Units stream and a Stimulation stream. It is intended to be used by an acquisition module to convert raw data to data usable in the processing pipeline.

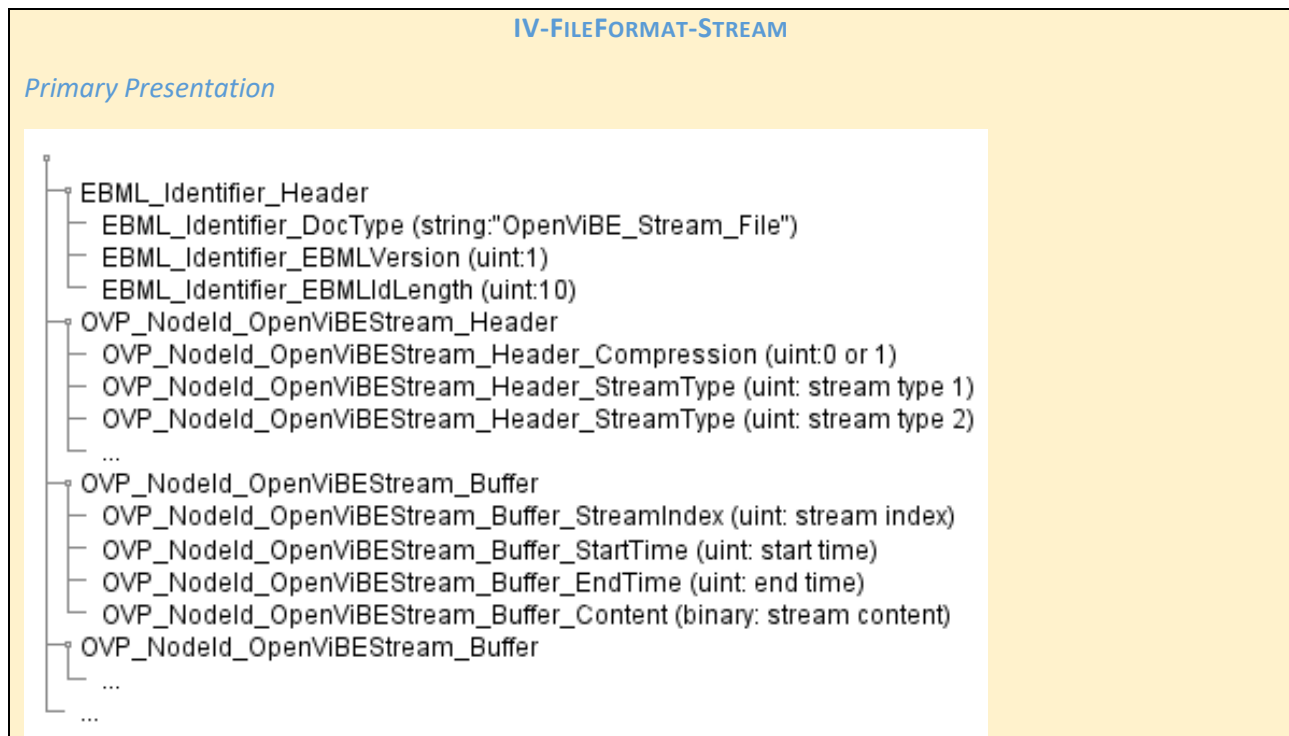
## 2.2.2 Stream Structure Specification

The stream structure specifications for all stream types are available in §10.1 Stream Structure Specifications.

A stream structure specification consists of 3 sections:

- The HEADER section describes the content of head chunks. Head chunks contain the necessary runtime parameters needed to interpret payload data chunks. When a scenario is played, head chunks are the first chunks to be propagated in the pipeline. Each box receives a head chunk with the right structure (i.e. structure following the HEADER section of the stream structure specification related to the expected stream type on this input) on each input, interprets it and sends an output head chunk with the right structure on each output.
- The BUFFER section describes the content of payload chunks. Payload chunks contain data that are interpreted thanks to header chunks. As long as a scenario is running, payload chunks are propagated in the pipeline.
- The END section describes the content of tail chunks. Tails chunks are the last chunks propagated in the pipeline.

Stream data can be saved in a file. The following view presents the file format used to record stream data.



### Element Catalog / Description

The node structure corresponds to the following DTD-like representation:

```
declare header {
  DocType := "OpenViBE_Stream_File";
  EBMLVersion := 1;
  EBMLMaxIDLength := 10;
}
define elements {
  Header := 0x0040F59505AB3684C8D8 container [ card:1; ] {
    Compression := 0x00C0358769166380D1 uint;
    Stream := 0x00F32EC1D1FE904087 uint [ card:*; ];
  }

  Buffer := 0x00AE60AD1887A29BDF container [ card:*; ] {
    StreamIndex := 0x00B0A56D8AB9C12238 uint [ card:1; ];
    StartTime := 0x00893E6A0AC5A9467B uint [ card:1; ];
    EndTime := 0x00408B5CCCD9C5024F29 uint [ card:1; ];
    Content := 0x00408D4B0BE87051265C binary [ card:1; ];
  }
}
```

#### 2.2.3 Stream Hierarchy

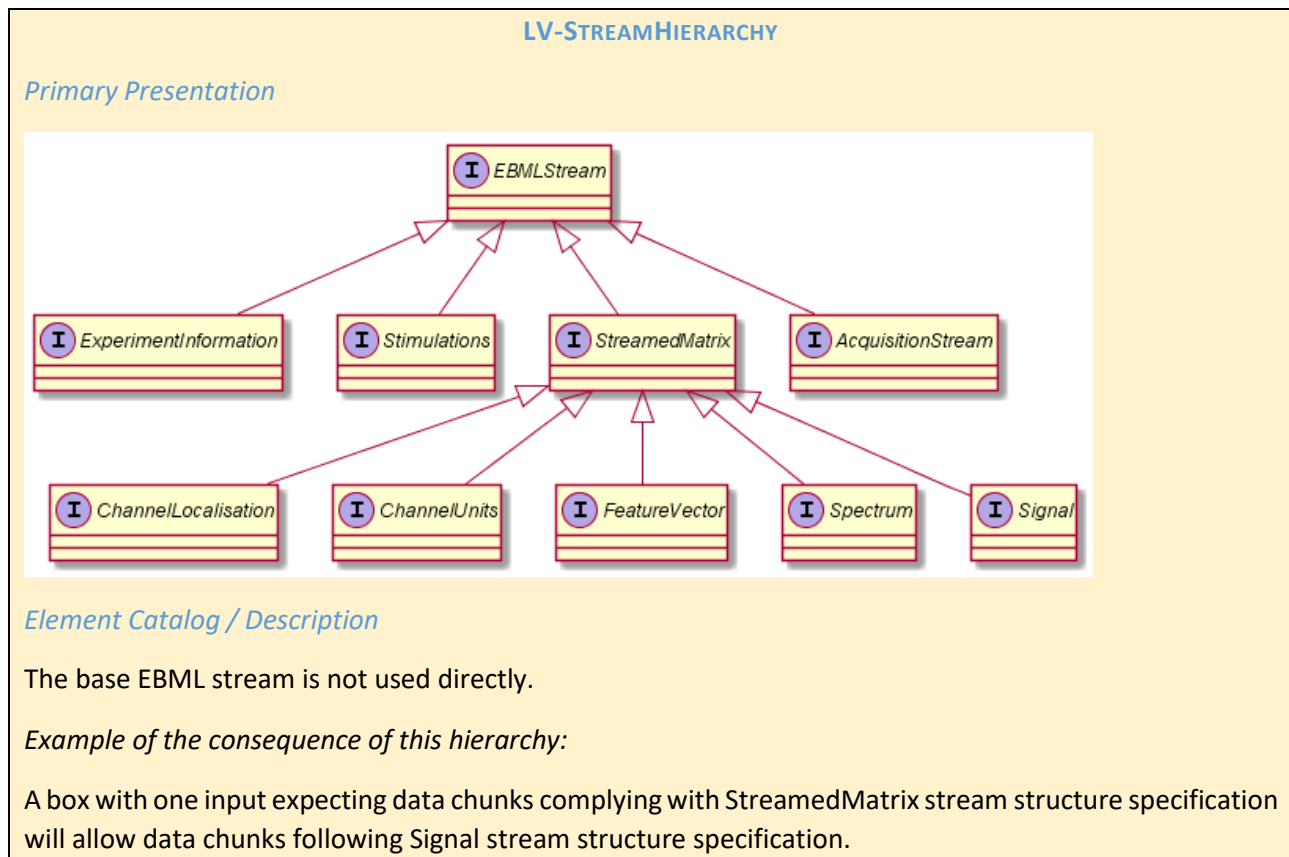
**Streams are organized in a hierarchical manner with attributes from a parent stream being inherited by its children streams. As a result, a stream expecting data with a structure complying with the specification related to a parent stream type will also accept data with a structure complying with the specification related to children stream types. This is an essential feature of streams as it affects the connection rules during pipeline creation (see §0**

CertiViBE - v1.0 Modules Detailed Design Specifications	CERT-01 MSD-01	Page 14 / 86
--	----------------	--------------

All rights reserved. Passing on or copying of this document, use and communication of its contents not permitted without written authorization

Scenario Management).

The following view presents the hierarchical organization of stream types.



As streams are no C++ objects, the stream hierarchy is not a class hierarchy in an object-oriented sense. The stream hierarchy is reflected at the codec level (see 2.2.4 Stream Encoding/Decoding).

## 2.2.4 Stream Encoding/Decoding

As stated in the previous section, data is transmitted between boxes as chunks of raw bits with each chunk layout following structural specification. However, boxes do not manipulate raw bits internally but C++ objects. Thus, object data produced by boxes have to be converted into raw memory buffers before transmission to the next box (encoding) and incoming raw memory buffers have to be converted into object-data (decoding).

These coding/decoding tasks are performed through a set of codec algorithms (see §3.2 Algorithm for details on algorithms) that are responsible for the implementation of stream structure specifications. For

CertiViBE - v1.0 Modules Detailed Design Specifications	CERT-01 MSD-01	Page 15 / 86
--	----------------	--------------

All rights reserved. Passing on or copying of this document, use and communication of its contents not permitted without written authorization

each stream type, there is a corresponding encoder and decoder class so that the encoder/decoder class hierarchy mirrors the one defined in §2.2.3 Stream Hierarchy.

Typical encoder properties are:

- **Input:** Header section data + buffer section data (e.g. Matrix, Stimulation Set)
- **Output:** Memory buffer

Typical decoder properties are:

- **Input:** Memory buffer
- **Output:** Header section data + buffer section data (e.g. Matrix, Stimulation Set)



The implementation of codec algorithms are part of the `openvibe-plugins-stream-codecs` component.



## EBML Module

As stated in previous sections, stream types are defined using EBML. Codec algorithms rely on `openvibe-ebml-module` component to parse/serialize stream data. This module uses a strategy-like pattern to implement the following callback mechanism:

- `IReader/IWriter` are the context interfaces and `CReader/CWriter` the context implementations whose behaviors vary according to the parsing/serializing strategy;
- `IReaderCallback/IWriterCallback` are the strategy interfaces supplied to the context to perform tasks specific to the EBML structure to parse/serialize.

Codec algorithms use the module as follows:

- Either the codec algorithm is the strategy (by inheriting strategy interface) or instantiates an object (`TReaderCallbackProxy/TWriterCallbackProxy`) that acts like the strategy and keeps a reference on the codec algorithm;
- The codec algorithm instantiates a new reader/writer context with itself or the proxy as strategy;
- The codec algorithm forwards parsing/serializing to the reader/writer context.

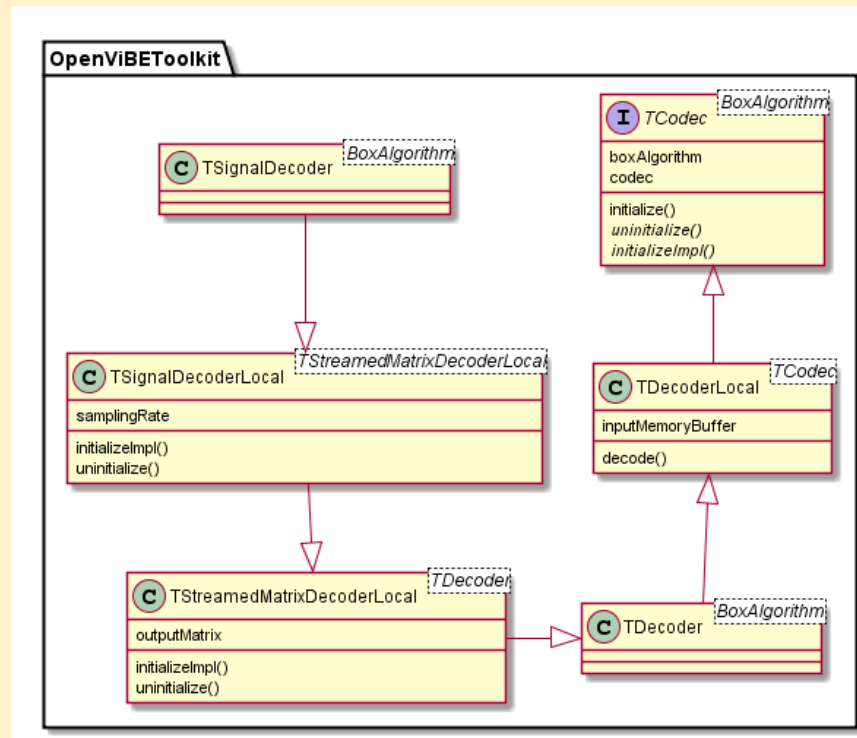
Encoding/decoding is not performed at kernel level and must be performed programmatically within each box. As the use of codec algorithms can be tedious, `openvibe-toolkit` component provides a wrapper API that eases the use of these algorithms from boxes.

The wrapper API uses mixin/parametrized inheritance (see SDD) to implement a hierarchy of wrappers matching the codec hierarchy. T

The following view shows the inheritance hierarchy for a signal stream decoder `openvibe-toolkit` module. Note that the behavior is similar for the encoding process and for other stream types.

CertiViBE - v1.0 Modules Detailed Design Specifications	CERT-01 MSD-01	Page 17 / 86
--	----------------	--------------

## Primary Presentation



## Element Catalog / Description

Every class is templated with its superclass as parameter starting with the most basic class: **TCodec**. A box algorithm uses toolkit by declaring an object of type: **TSignalDecoder<BoxAlgorithm>**. Feeding codec algorithms with the client box is necessary to enable access to the box context by the codec API.

**TCodec**: Base abstract class for encoder and decoder. This class contains a reference to the client box algorithm and a reference to the codec algorithm that does the effective work.

**TDecoderLocal**: This class contains the partial implementation of **TCodec** interface common to all decoders (storage for the input memory buffer and high level methods `decodeHeader`, `decodeBuffer` and `decodeEnd` implementation called by boxes to decode input chunks). The decoding step involves filling the input memory buffer with incoming chunks, decoding it and marking input as deprecated.

**TSignalDecoderLocal/TStreamedMatrixDecoderLocal**: This class contains the implementation of **TCodec** interface specific to the decoded stream type (storage for specific output parameters). The right codec algorithm is selected at this level.

CertiViBE - v1.0	CERT-01 MSD-01	Page 18 / 86
Modules Detailed Design Specifications		


## 2.3 Structures Identification


The system uses 64-bits identifiers (`CIdentifier` class) to identify any entities in the framework. The following table presents typical uses of identifiers within the framework. Identifiers can be hardcoded or randomly generated. Each time an identifier is generated, the generator prevents from duplicated identifiers in the scope of their use. Duplicated identifiers prevention or assertion are managed inside each use scope.

Role	Use	Definition Location (relative to OpenViBE root unless otherwise stated)
Box class identifiers used for box types (see §3.3.1 Box Algorithm Prototype) <i>Ex:</i> <code>OVP_ClassId_BoxClass</code>	Used internally in boxes implementation.  Used indirectly by the plugin manager to create the right box instance.	Either in <code>ovp_defines.h</code> of the right plugin module or in the box algorithm declaration header file.
Stream type identifiers used for streams (see §2.2 Stream Structures) <i>Ex:</i> <code>OV_TypeId_Signal</code> <code>OV_TypeId_EBMLStream</code>	Used in box input/output type identification.  Used for box connection compatibility check.  Used in stream castability check (see §2.2.3).  Registered in type manager (see focus below).	<b>Definition:</b> <code>./openvibe/include/openvibe/ov_defines.h</code> <b>Registration:</b> <code>./kernel/src/kernel/ovtkCKernelContext.cpp</code>
Box settings global identifiers <i>Ex:</i> <code>OV_TypeId_Boolean</code> (simple type) <code>OV_TypeId_Stimulation</code> (enum type) <code>OV_TypeId_LogLevel</code> (enum type)	Used to identify the type of string-based box settings (see §3.3.1).  For enum type, there is an additional identifier for each enum value.	<b>Definition:</b> <code>./openvibe/include/openvibe/ov_defines.h</code> <b>Registration:</b> <code>./kernel/src/kernel/ovtkCKernelContext.cpp</code>

	Registered in type manager (see focus below).	
Box settings specific identifiers	Same as above but only used internally in plugins to identify specific string-based settings.	<b>Definition:</b> <code>./plugins/xxx/src/ovp_defines.h</code> <b>Registration:</b> <code>./plugins/xxx/src/ovp_main.cpp</code>
Stimulation type identifiers <i>Ex:</i> <code>OVTk_Stimulation_Id_Beep</code>	Used to define stimulation types in set (see §2.1.2) and as possible enum values for settings of type <code>OV_TypeId_Stimulation</code> (see Box settings global identifiers).  Registered in type manager (see focus below).	<b>Definition:</b> <code>./toolkit/include/toolkit/ovtk_defines.h</code> <b>Registration:</b> <code>./toolkit/src/ovtk_main.cpp</code>
Algorithm class identifiers used for algorithm types §3.2 Algorithm) <i>Ex:</i> <code>OVP_GD_ClassId_Algorithm_X</code> <code>OVP_ClassId_Y</code>	Used internally in algorithms implementation.  Used by external code (code from another plugin, application) to query the algorithm manager (see §0) for algorithm creation.	2 preprocessor definitions refers to the same identifier:  <b>One for internal use:</b> Either in <code>ovp_defines.h</code> of the plugin or in the algorithm declaration header file.  <b>One for external use:</b> <code>./common/include/ovp_global_defines.h</code>
Algorithm parameters identifiers	Used internally in the algorithm implementation.  Used by external code (code from another plugin, application) to query the algorithm parameters from an algorithm instance.	2 preprocessor definitions refers to the same identifier:  <b>One for internal use:</b> Either in <code>ovp_defines.h</code> of the plugin or in the algorithm declaration header file.  <b>One for external use:</b> <code>./common/include/toolkit/ovp_global_defines.h</code>
Kernel, Plugin and base class identifiers	Used mainly for introspection.	<code>./openvibe/include/openvibe/ov_defines.h</code>

Ex: OV_ClassId_Matrix OV_ClassId_Plugins_Algorithm		
Attribute Identifiers	Used to add or retrieve attributes from attributable classes at runtime (see §0 for information on attributability).	./openvibe/include/openvibe/ov_defines.h
Stream Node Identifiers	Used by codec algorithms to identify nodes in the EBML structure.	./toolkit/include/toolkit/ovtk_defines.h
Measurement Units Identifiers	Used by channel measurement units codec algorithms to identify measurement units.	./toolkit/include/toolkit/ovtk_defines.h

 `ovp_global_defines.h` is auto-generated by the plugin inspector that inspects all plugins to generate global defines usable by other modules.

 **Type Manager**

The table above mentions the fact that some types are registered in the type manager. As explained in SDD, the kernel provides its services through a set of managers accessed via `IKernelContext` interface. The type manager (interface `ITypeManager`) is mainly used to:

- Handle string-based box settings simple types:
  - Types are registered in the manager;
  - The manager can be queried for type existence or to convert type identifier into string.
- Handle string-based box settings enum types:
  - Types are registered in the manager;
  - Enum values are registered in the manager;
  - The manager can be queried to retrieve enum entries.
- Handle stream types:
  - Types are registered in the manager with the parent type;
  - The manager can be queried for stream type existence or stream castability.



### 3. System Logical Units

The previous chapter focuses on structures that contain data. Once data are available, the main purpose of the system is to process them. This chapter deals with the entities responsible for performing processing on data: the logical units.

There are three types of logical units within the system:

- **Algorithms** are generic low-level components that perform operations on data (*ex: algorithm to read file, algorithm to encode/decode stream*). It can be used by box algorithms, other algorithms, the Kernel module or applications based on OpenViBE;
- **Box algorithms** are the processing engines each box relies on;
- **Box listeners** are logical units responsible for performing specific actions when boxes state changes.

Implementation of algorithms, box algorithms and box listeners are part of the plugin mechanism that is presented in the next section.

#### 3.1 Plugin Mechanism

In SDD, it is explained that a plugin can contain multiple components definitions.

A plugin component definition consists of:

- A descriptive part: description of the component metadata (author, name etc.) and prototype (input, output, parameters);
- An operational part: definition of the logical unit actually responsible for performing operations on data.

A plugin gathers together a set of component definitions that aim at providing a specific service or a coherent set of services (*ex: classification plugin with classification related algorithms and box algorithms*).

The list of plugins components is available in appendix (§10.3 Standard Configuration Tokens).

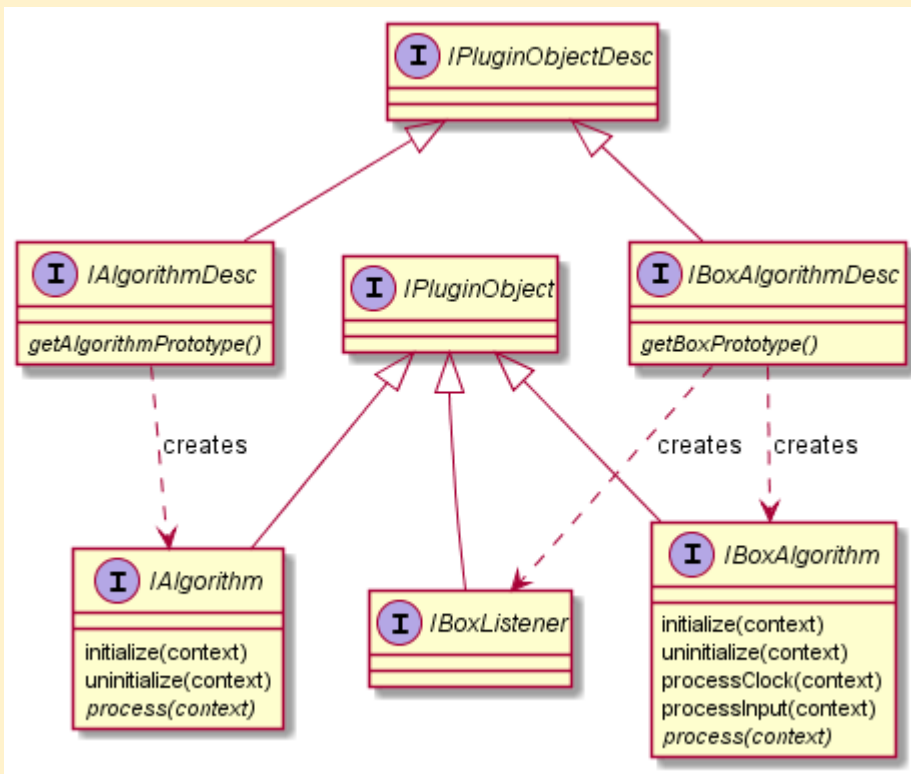
The base requirement for a plugin mechanism is to make the application responsible for loading the plugins able to communicate with them. It is achieved through published interfaces that must be implemented by plugin components and a set callbacks provided by each plugin and used as an entry point by plugin loaders to explore the plugin content.

##### 3.1.1 Plugin API

The following view presents the set of interfaces that must be implemented by plugins components.

CertiViBE - v1.0 Modules Detailed Design Specifications	CERT-01 MSD-01	Page 23 / 86
--	----------------	--------------

## Primary Presentation



## Element Catalog / Description

The plugin object descriptor interfaces are implemented to fulfill the descriptive part of component definition while the plugin object interfaces are implemented to fulfill the operational part of component definition.

**IPluginObject:** Base interface for algorithms or box algorithms operational definition.

**IAlgorithm:** Base class for algorithms operational definition. Implementations of this interface aim at containing the algorithm logic (basically reading from input, computing data to produce and writing to output) divided into three steps: initialization, processing, and uninitialization (see §3.2 Algorithm).

**IBoxAlgorithm:** Base class for box algorithms operational definition. Implementations of this interface aim at containing the box algorithm logic (basically reading from input, computing data to produce and writing to output) divided into three steps: initialization, processing, and uninitialization (see §3.3 Box Algorithm).

**IBoxListener:** Base class for box listeners. Implementations of this interface aim at performing specific actions related to boxes on notification (see §3.4 Box Listener).

**IPluginObjectDesc:** Base interface for algorithms or box algorithms descriptive definition. Implementations of this interface aims at providing metadata information about the logical unit. They must also implement the method responsible for creating instances of plugin object classes.

**IAlgorithmDesc:** Base interface for algorithms descriptive definition. Implementations of this class must implement the method responsible for providing the algorithm prototype to external modules (e.g. the Kernel module).

**IBoxAlgorithmDesc:** Base interface for box algorithms descriptive definition. Implementations of this class must implement the method responsible for providing the box algorithm prototype (input, output and settings) to external modules (e.g. the Kernel module) and, optionally, implement the method responsible for box listener class instances creation.



The plugin mechanism is designed so that the plugin loader loads and manipulates descriptors. These descriptors provide enough information to let the system know how the logical unit is structured. Moreover, these descriptors are also responsible for providing instances of the actual processing class to the system. It has some consequences especially for box algorithm:

**Box algorithm descriptors provide enough structural information to create a scenario (see §0**

- Scenario Management);
- Creation of a plugin object instance is just needed when a scenario is played (see §7 Scenario Playback) and boxes have to process data.



Algorithm, box algorithm and box listener implementations should not inherit directly from `IAlgorithm` and `IBoxAlgorithm`. The `openvibe-toolkit` component provides wrappers classes (`TAlgorithm`, `TBoxAlgorithm` and `TBoxListener`) that implement `IAlgorithm`, `IBoxAlgorithm` and `IBoxListener` context related calls and provides to subclasses a controlled access to context features. They act as guards to prevent misuse of the context at the algorithm implementation level.

### 3.1.2 Plugin Callbacks

Providing a plugin API allows the loader to manipulate and communicate with plugin components. But before any communication is setup, the loader must be aware of what components are available in a plugin.

As explained in SDD, plugins are dynamically loaded libraries. The plugin content discovery relies on a set of callbacks that each plugin must provide in order to be explorable. The dynamic loader looks for the following callbacks at runtime:

- `onInitialize` is responsible for registering all the components provided by the plugin;
- `onUninitialize` is responsible for releasing the component list;
- `onGetPluginObjectDescription` is responsible for retrieving plugin descriptors.

These callbacks are defined in the `ovp_main.cpp` file of each plugin through a macro mechanism. The use of these callbacks by the dynamic loader is described in the next section.

### 3.1.3 Plugin Management

Once a list of plugins is available, the system needs a way to load them at runtime. Plugin management is part of the Kernel module and dedicated to the plugin manager (`CPluginManager`). This class is responsible for:

- Loading plugins;
- Managing plugin object lifetime (plugin object creation and destruction);

Internally, it relies on a `CPluginModule` objects and their platform-specific implementations to load plugins at runtime.

`CPluginModule` instances first look for the callback symbols (described in previous section) in a dynamically loaded library. Then, it calls the `onInitialize` callback so that the list of plugin components is created. Eventually, it loops plugin's `onGetPluginObjectDescription` callback to get each registered component description.



#### FS Module

The plugin manager relies on `openvibe-fs-module` component to retrieve the list of plugins from a wildcard path (e.g. `path/to/plugin/*.so`). The fs module uses a visitor-like pattern to enumerate paths (called entries) that match a given wildcard pattern and perform specific processing on each match. *Example of use with by plugin manager:*

CertiViBE - v1.0 Modules Detailed Design Specifications	CERT-01 MSD-01	Page 26 / 86
--	----------------	--------------

- The plugin manager implements a specific callback class to process new entry (path to plugin) in a `callback` method;
- An instance of the specific callback class is given to the fs module enumerator;
- The enumerator enumerates paths that match the plugin path pattern and calls the specific callback class `callback` method that tries to load the plugin and add it to the plugin manager module list.

## 3.2 Algorithm

An algorithm can be viewed from different perspectives. The [Black Box](#) view, which describes the algorithm interfaces, is presented in the first section Algorithm Prototype. The second section, Algorithm Core, discusses the internal logic of algorithms.

### 3.2.1 Algorithm Prototype

Algorithm interfaces description is called **algorithm prototype** in the system.

Prototypes are characterized by the definition of the following properties:

- Input/Output parameters: each algorithm defines a number of I/O parameters characterized by a unique identifier (see §2.3 Structures Identification), a name and a type;
- Input/Output triggers: each algorithm defines a number of I/O triggers characterized by a unique identifier and a name. Input triggers are used to control the algorithm (ex: "Process") while output triggers represent events risen by the algorithm (ex: "Processing Done"). Triggers definition consists of the specification of messages that can be exchange between the algorithm and, typically, the Kernel module. However, the real exchange of messages at runtime is perform by activating a given trigger.

Algorithm prototypes are defined in algorithm descriptors (see §3.1.1 Plugin API) through the `IAlgorithmProto` interface.

### 3.2.2 Algorithm Core

Algorithm logic is implemented in plugin object classes which must implement `TAlgorithm` interface as stated in §3.1.1 Plugin API.

Algorithm logic is implemented through three methods:

- `initialize`: mainly used to initialize algorithm input and output parameters as well as for resources allocations;

CertiViBE - v1.0 Modules Detailed Design Specifications	CERT-01 MSD-01	Page 27 / 86
--	----------------	--------------

All rights reserved. Passing on or copying of this document, use and communication of its contents not permitted without written authorization

- `uninitialize`: used for cleaning up input and output parameters as well as for releasing resources allocations;
- `process`: used to perform operations on input data according to input triggers, produce output data and rise output triggers.

Algorithms make use of the execution context (`IAgorithmContext`) provided by `TAlgorithm` interface. This context gives access to kernel managers, algorithm I/O parameters and active input triggers, and, provides features to control output triggers activation.



## Algorithm Parameters

Algorithm parameter types are defined in `ovkCParameter.h`. Each parameter implements a set of interfaces linearized using mixin inheritance (see SDD for details on mixin inheritance).

*Example for matrix-type parameter:*

```
CMatrixParameter -> TBaseParameter -> TKernelObject -> IParameter ->
IKernelObject -> IObject
```

`TBaseParameter` is an implementation of `IParameter`. The main purpose of these classes is to provide some sort of type erasure. This convenience enables a uniform management of input and output parameters through the `IParameter` interface without concern for the real type they contain.

Algorithms access their own I/O parameters via the algorithm context provided by `TAlgorithm` interface. Other algorithms I/O parameters are accessed via `AlgorithmProxy` interface (see §5 Algorithm Management). Parameters are returned as `IParameter` pointers. As manipulation of `IParameter` is error-prone, algorithm parameters should be manipulated through `TParameterHandler` objects that are typically initialized in the `initialize` method.

*Example:*

```
class MyAlgorithm : public
OpenViBEToolkit::TAlgorithm<OpenViBE::Plugins::IAgorithm>
{
    ...
    OpenViBE::Kernel::TParameterHandler<OpenViBE::CString*> m_Input;
    ...
    void initialize()
    {
        m_Input.initialize(this->getInputParameter(ParameterId));
    }
}
```

CertiViBE - v1.0 Modules Detailed Design Specifications	CERT-01 MSD-01	Page 28 / 86
--	----------------	--------------

### 3.3 Box Algorithm

Box algorithm concept can be analyzed from the same perspectives as algorithm concept. The [Black Box](#) view, which describes box algorithm interfaces, is presented in the first section Box Algorithm Prototype. The second section, Box Algorithm Core, discusses box algorithms internal logic.

#### 3.3.1 Box Algorithm Prototype

Box algorithm interfaces description is called **box algorithm prototype** in the system.

Prototypes are characterized by the definition of the following properties:

- **Input/Outputs:** box algorithms define a number of inputs and outputs characterized by a name and a type. I/O types are meant to be stream type (see §2.2 Stream ) represented by a unique identifier (see §2.3 Structures Identification for stream type identifier);



Box input and output can be referred as connector or port in the context of box connection.

- **Settings:** box algorithms define a number of settings characterized by a name, a type represented by a unique identifier (see §2.3 Structures Identification for box settings identifiers) and a default string-based value. Settings are used to customize box algorithm behavior.
- **Flags:** box algorithms set a number of flags that represent true/false conditions. These flags represent box options that can be set or cleared. Flags can deal with prototype modification permissions given to the Kernel (see §3.4 Box Listener ) or simple implementation state (e.g. informing box algorithm users that a box is deprecated or unstable).

Box algorithm prototypes are defined in descriptors (see §3.1.1 Plugin API) through the `IBoxProto` interface.



#### String-based settings value

Settings values are set as string (although settings are given an expected type) so that they can be straightforwardly serialized. The main goal is to be able to set settings values as configuration tokens in a configuration file. So, it is possible to customize a runtime session without any scenario modification.



Adding inputs, outputs and settings is done in sequence. As a result, indexes used to access them afterwards match the declaration order in the descriptor.

### 3.3.2 Box Algorithm Core

Box algorithms logic is implemented in plugin object classes that should implement the `TBoxAlgorithm` interface as stated in §3.1.1 Plugin API.

Box algorithm logic is implemented through a set of fundamental methods:

- `initialize`: mainly used to initialize and retrieve box algorithm settings, initialize internal algorithms and connect algorithms inputs and outputs.



During initialization, internal algorithms can be chained simply by binding input to output. Here

*Typical example where an encoder input is connected to decoder output:*

```
void initialize()
{
    ...
    m_SignalEncoder.getInputMatrix().setReferenceTarget(
        m_SignalDecoder.getOutputMatrix()
    );
    ...
}
```

Note that a given algorithm output can feed multiple algorithms input.

- `uninitialize`: used for cleaning up resources;
- `processClock`: callback called periodically by Kernel module. It is used for box algorithms whose processing is not triggered by data available on one of to their inputs (time-driven box algorithm);
- `processInput`: callback called by Kernel module each time data is available on one of the box algorithm input (data-driven box algorithm). Box algorithms decide if the processing should be triggered or not (e.g. a box algorithm with two inputs can wait to have data available on the first AND the second before it triggers the processing);
- `process`: perform the actual processing job. This callback is triggered only if the box algorithm informs Kernel module it is ready to process.

CertiViBE - v1.0 Modules Detailed Design Specifications	CERT-01 MSD-01	Page 30 / 86
--	----------------	--------------



Box algorithms expect streams as inputs and outputs but needs to manipulate regular objects within the processing step. As explained in §2 System Data, they rely on codec algorithms for that. This leads to significant consequences on box algorithm core implementation:

- `initialize` callback almost always contains code to initialize codec algorithms;
- `process` callback workflow always include decoding and encoding steps.

Note that each box algorithms input/output must have its own decoder/encoder instance.

A typical processing workflow follows these steps:

- Kernel module calls box algorithms `initialize` callbacks:
  - Internal algorithms are initialized;
  - Algorithms inputs and outputs are connected;
  - Box algorithm settings are retrieved.
- Kernel module calls `processClock/processInput` callbacks periodically:
  - Once ready, box algorithms inform the Kernel module that they are ready to process.
- Kernel module calls the `process` callbacks on box algorithms ready to process:
  - Box algorithms retrieve the dynamic context that gives access to data waiting on their inputs and allows to write data on outputs;
  - For each chunk of data available on their inputs:
    - The chunk is decoded;

The nature of the chunk is analyzed (head, buffer, tail chunk as described in §0

- Stream Structure Specification) and processing is performed accordingly:
  - Head and tail chunks are re-encoded directly and sent to outputs;
  - Payload chunks are processed, output data produced, encoded and sent to outputs.



Box algorithms make use of the box algorithm execution context (`IBoxAlgorithmContext`) provided by `TBoxAlgorithm` interface to access kernel services and communicate with the Kernel module.

Note that box algorithms have restricted access to Kernel features and cannot directly share information with other box algorithms.

### 3.4 Box Listener

As stated in §3.1.1 Plugin API, boxes can rely on box listeners to perform specific actions on event notification. Box listener mechanism is specially implemented to react to box modifications by the Kernel module.

The set of events a listener can react to is defined by `EBoxModification` enum. It is closely related to flags that are set in box algorithms prototype. In a nutshell:

- ➔ Flags define the modifiability of a box by the Kernel module:
  - Can the Kernel modify inputs/outputs/settings?
  - Can the Kernel add inputs/outputs/settings?
- ➔ Listeners define the reaction to boxes modification by the Kernel module;
- ➔ The kernel module performs authorized modifications and notify the listener (if available).

Listeners implements `TBoxListener` interface and override only “reaction” they want to customize.

Note that defining a box listener is optional.

CertiViBE - v1.0 Modules Detailed Design Specifications	CERT-01 MSD-01	Page 32 / 86
--	----------------	--------------

## 4. Kernel Management

The kernel module is the central module of the system. As it is explained in SDD, it provides the base services to build and play scenarios. Previous and next chapters describe some of these services (§3.1.3 Plugin Management, §5 Algorithm Management, §6 Scenario Management and §7 Scenario Management Scenario Playback). This chapter details how the kernel services are made available to Kernel consumers (i.e. client code).

The Kernel component is loaded at runtime by a specific class (`CKernelLoader`, part of OpenViBE Base framework as described in SDD).



Why is the kernel a dynamic loaded (DL) library?

The idea is to be able to use different Kernel implementations without the need to recompile the system. At compile-time, only the set of Kernel interfaces (abstract classes) is used by Kernel consumers. At runtime, the DL library containing an implementation of these interfaces is loaded and provide a definition for all required symbols.

Currently, only one default Kernel implementation is available in the framework.

Kernel loading follows the same principles as plugins loading. The dynamic loader is looking for a set of callbacks:

- `onInitialize` is responsible for initializing resources;
- `onUninitialize` is responsible for releasing resources;
- `onGetKernelDesc` is responsible for retrieving the kernel descriptor. The descriptor is used to generate the kernel context (`IKernelContext` implementation). Once the kernel context is created and initialized, it is the access point to all managers (algorithm, configuration, player, plugin, scenario, type, and log manager) and, thus, to all services.

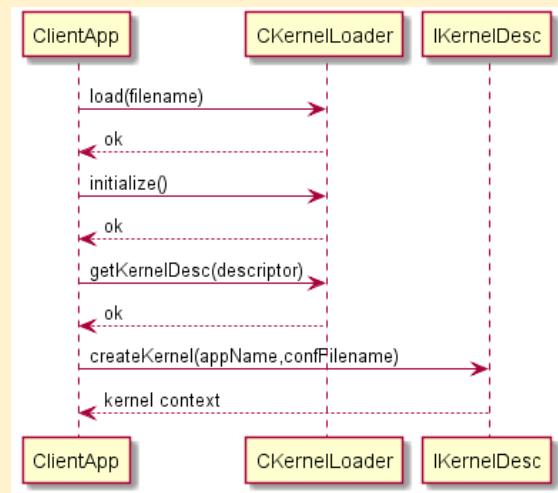
Definition of these callbacks as well as Kernel descriptor implementation are available in `ovk_main.cpp`.

The following views describes the sequence of calls that leads to kernel context creation and retrieval.

CertiViBE - v1.0 Modules Detailed Design Specifications	CERT-01 MSD-01	Page 33 / 86
--	----------------	--------------

## BV-SEQUENCEDIAGRAM-KERNELLOADING

### Primary Presentation



### Element Catalog / Description

Call to `load` with the path to the Kernel DL library triggers the search for callbacks by the kernel loader. Then, the two next calls are just forwarded to corresponding callbacks (`initialize` -> `onInitialize` and `getKernelDesc` -> `onGetKernelDesc`).

#### 4.1 Once the Kernel descriptor is retrieved, it is mainly used to generate an instance of the Kernel context. The `createKernel` call is crucial because it feeds the context with the path to the configuration file (see §7.4 Scenario Player application)

Scenario player is the application responsible for loading and playing a scenario using a command line without launching any graphical user interface.

Here are available options of the application:

Option	Description	Mandatory
<code>--command-file</code>	Path to command file (command mode only)	Yes
<code>--config-file</code>	Path to configuration file (express mode only)	No
<code>--dg</code>	Global user-defined token: -dg="(token:value)" (express mode only)	No

<code>--ds</code>	Scenario user-defined token: -ds="(token:value)" (express mode only)	No
<code>--max-time</code>	Scenarios playing execution time limit (express mode only)	No
<code>--mode</code>	Execution mode: 'x' for express, 'c' for command	Yes
<code>--play-mode</code>	Play mode: std for standard and ff for fast-foward (express mode only) [default=std]	No
<code>--scenario-file</code>	Path to scenario file (express mode only)	Yes

#### 4.1.1 Scenario player execution workflow

A straightfoward commands workflow is built according to command-line (or a command file).

Configuration Management).

Initializing the kernel context instance can be delayed. It will automatically be performed on the first call to any context method. However, it is important in most case to initialize the toolkit module from the created context (`OpenViBEToolkit::initialize`).

## 5. Algorithm Management

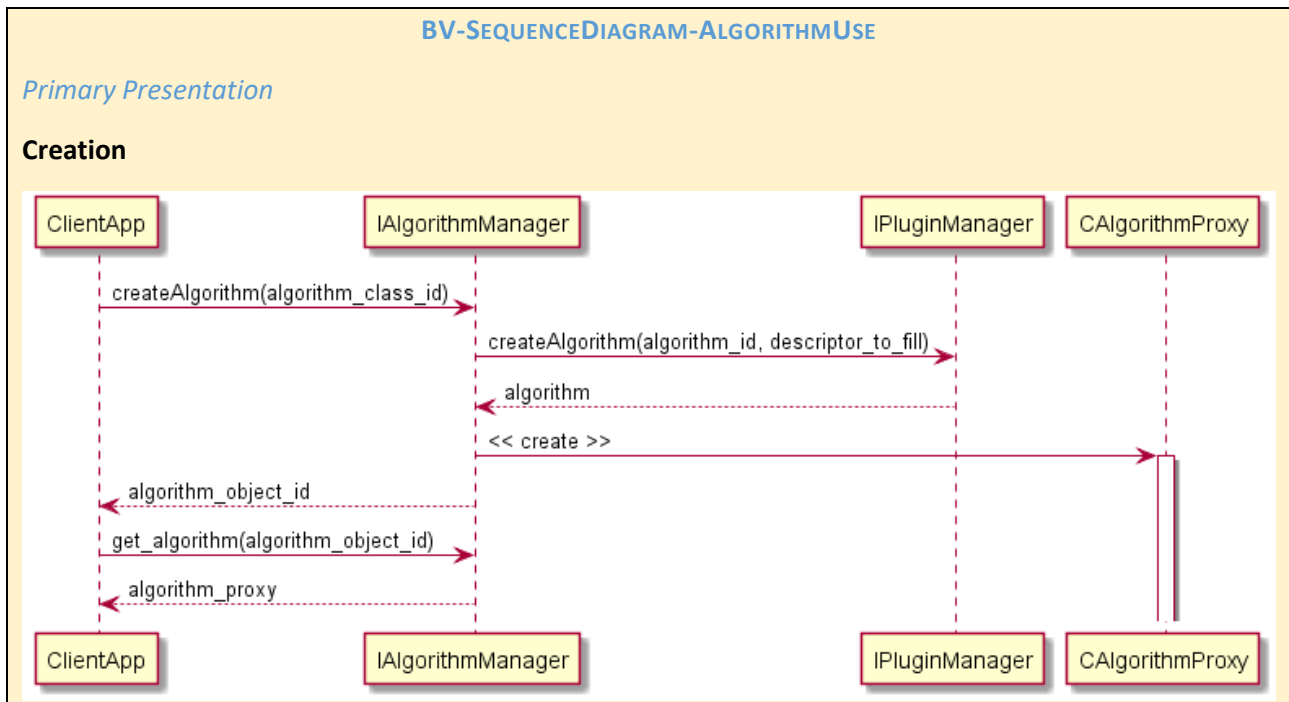
One of the services provided by the Kernel module is the management of algorithms.

As stated in section §3.2 Algorithm, algorithms are low-level logical units that can be used within boxes, by the Kernel or by an application to perform specific tasks. Once algorithms are loaded (see §3.1 Plugin Mechanism) and usable, an interface is needed to manipulate these specific objects. The Kernel module provides a set of classes to manipulate algorithms centered on the algorithm manager (CAlgorithmManager) which is responsible for creating and destroying algorithm instances.

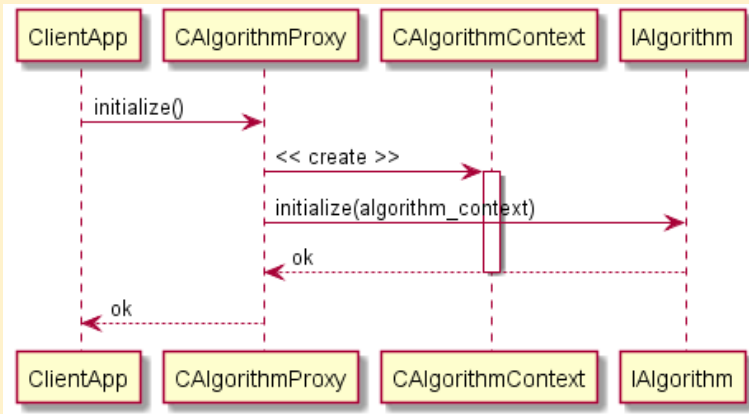
Use of an algorithm follows a rather typical sequence:

- Algorithm creation;
- Algorithm initialization;
- Processing;
- Algorithm uninitialization;
- Algorithm release.

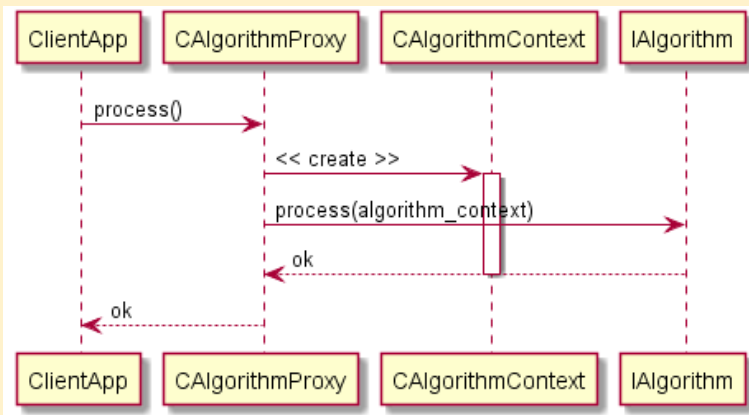
The following view describes some steps of the sequence above in details..



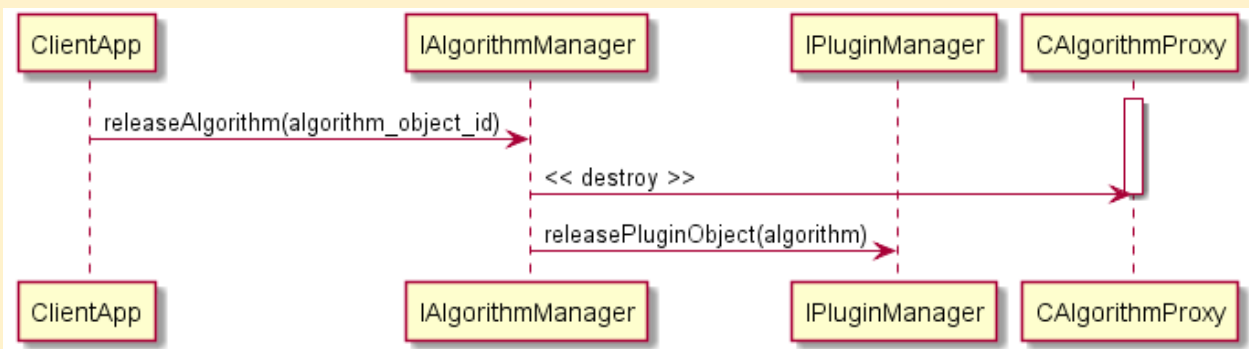
## Initialization



## Processing



## Release



## Element Catalog / Description

### Creation

Algorithms are plugin components. As described in §3.1.3 Plugin Management, the plugin manager is in charge of managing plugin object instances lifetime. Thus, the scenario manager forwards the creation of algorithm objects (i.e. `IAlgorithm` instances) to the plugin manager. Internally, the algorithm manager wraps created algorithm object instances in `CAlgorithmProxy` objects.

`CAlgorithmProxy` objects are responsible for exposing an algorithm prototype (inputs/outputs/triggers) to Kernel consumers and supervising calls to algorithm processing-related methods (i.e. creating the algorithm context needed by algorithms to perform their tasks (see §3.2.2 Algorithm Core), calling `initialize/uninitialize/process` methods of the algorithm and handling potential errors).

An algorithm object identifier (see §2.3 Structures Identification) is returned back to Kernel consumers so that he can use it to retrieve a handle on a proxy object afterwards.

#### Initialization, Process:

Calls are forwarded from proxy objects to algorithm objects. `CAlgorithmProxy` objects create a new algorithm context for each new call to the underlying algorithm.

❗ As the context is only valid during a single call to the underlying algorithm, implementation of algorithm logical units should not store the algorithm context for a later use.

#### Release

This is the creation step inverse operation. Algorithm managers are, as other managers, resources managers. The `createAlgorithm` method is paired with a `releaseAlgorithm` method. Note that it is essential to release an algorithm once it has been used.

#### Notes

Uninitialization steps is not shown because it mirrors the initialization step.



#### Algorithm Parameters Handling

As it is presented in the previous view, `CAlgorithmProxy` instances are responsible for exposing algorithm prototypes to Kernel consumers. To achieve that, `CAlgorithmProxy` objects have to retrieve prototypes (see §3.2.1 Algorithm Prototype) from algorithm descriptors to expose it to Kernel consumers.

This requirement is fulfilled through a temporary object (`CAlgorithmProto`). At construction time, `CAlgorithmProxy` objects call `getAlgorithmPrototype` on descriptors with this temporary object as parameter.

CAlgorithmProto is a simple implementation of IAlgorithmProto that forwards each call to the corresponding CAlgorithmProxy method. Therefore, each time an input, output or trigger is added to a CAlgorithmProto, it is automatically added to the underlying CAlgorithmProxy object.

Internally, CAlgorithmProxy handles the list of algorithm input and output parameters with CConfigurable objects that are used to handle list of parameters.

## 6. Scenario Management

As stated in SDD, one of the main aim of the system is the creation of personalized chains of processing. It is achieved through one of the Kernel service dedicated to scenario management.

This service is centered on the scenario manager (`CScenarioManager`) which is responsible for creating and destroying instances of scenarios. Scenario management involves:

- Scenario lifetime management;
- Scenario populating (adding boxes and connecting boxes);
- Scenario configuration;
- Scenario loading/saving;

This first section focuses on the creation and configuration of new scenarios while the second section details scenarios storage capabilities.

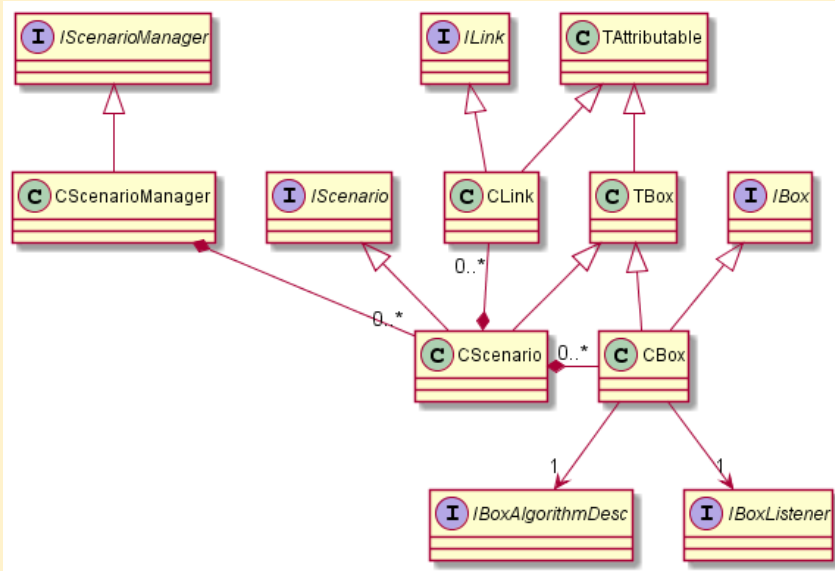
CertiViBE - v1.0 Modules Detailed Design Specifications	CERT-01 MSD-01	Page 40 / 86
--	----------------	--------------

## 6.1 Scenario Creation

Scenario creation involves the manipulation of multiple objects. The following view presents the main classes involved in scenario creation.

LV-CLASSDIAGRAM-SCENARIOMANAGEMENT

### Primary Presentation



### Element Catalog / Description

**CScenarioManager:** Kernel manager dedicated to scenario management. This class manages scenario lifetime (creation, destruction) and handles scenario import/export. Internally, it maintains a list of scenarios (`CScenario`).

**CScenario:** This class represents a scenario. At it is presented in SDD, a pipeline is a chain of processing elements. This class implements this concept through a list of boxes (`CBox`, the “processing elements”) and a list of links (`CLink`, the “chains”). This class provides an interface to build the pipeline by adding new elements and connecting them.

**CBox:** This class is responsible for exposing boxes prototype (inputs/outputs/settings) to Kernel consumers. Actually, it is similar to the part of `CAlgorithm` (see §5 Algorithm Management) that exposes algorithm prototype. Internally, it wraps a box algorithm descriptor and a box listener.

**i** Unlike `CAlgorithmProxy` class that wraps into one interface the exposure of the algorithm prototype and the supervision of calls to the underlying algorithm processing methods, `CBox` just implements the exposure part. The processing part is dedicated to another class that is presented in §7 Scenario Playback).

**i** Retrieval of the algorithm prototype by `CAlgorithmProxy` instances is explained in the focus “**Algorithm Parameters Handling** - §5 Algorithm Management”. The behavior is similar for `CBox` instances expect that the class involved in forwarding the prototype is `CBoxProto`. In addition to filling `CBox` instances from box algorithm prototypes defined in box descriptors, `CBoxProto` creates some new attributes (e.g. input count, output count). For instance, `CBoxProto` is responsible for converting box algorithm flags to attributes (see `TAttributable` just below).

**CLink**: This class represents a connection between a box input and another box output.

**TBox**: This is the base class for scenarios and boxes. Both inherit `TBox` because both represent the same [Black Box](#) concept.

**TAttributable**: This is the base class for “attributable” objects in the system. This very simple implementation of the properties pattern allows (key, value) properties to be added to these objects at runtime. Attributes are identified with 64-bits identifier (see §2.3 Structures Identification). `CBoxProto` makes use of the “attributability” of `CBox` to handle flags (flags are transformed from an enum type to an attribute identifier `OV_AttributeId_Box_XXX`) and add additional properties when it explores the box algorithm prototype. `CScenario` “attributability” is also used in the system.

### Notes

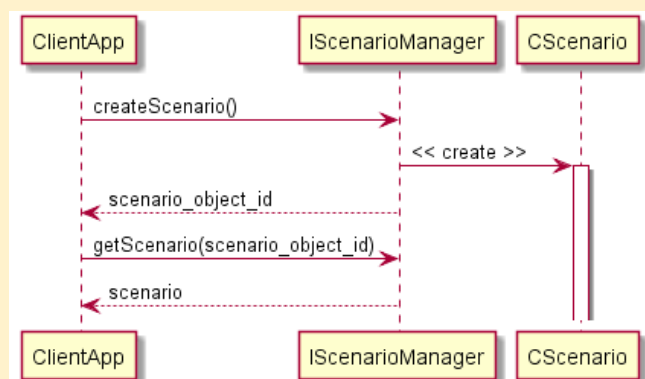
The primary presentation shows multiple inheritance artifacts for `CLink`, `CBox` and `CScenario`. At implementation level, multiple inheritance is not implemented with the default C++ support but with mixin inheritance as described in SDD.

The previous view shows the organization of Kernel classes regarding scenario management. The following one presents the typical sequence of calls involved in scenario creation.

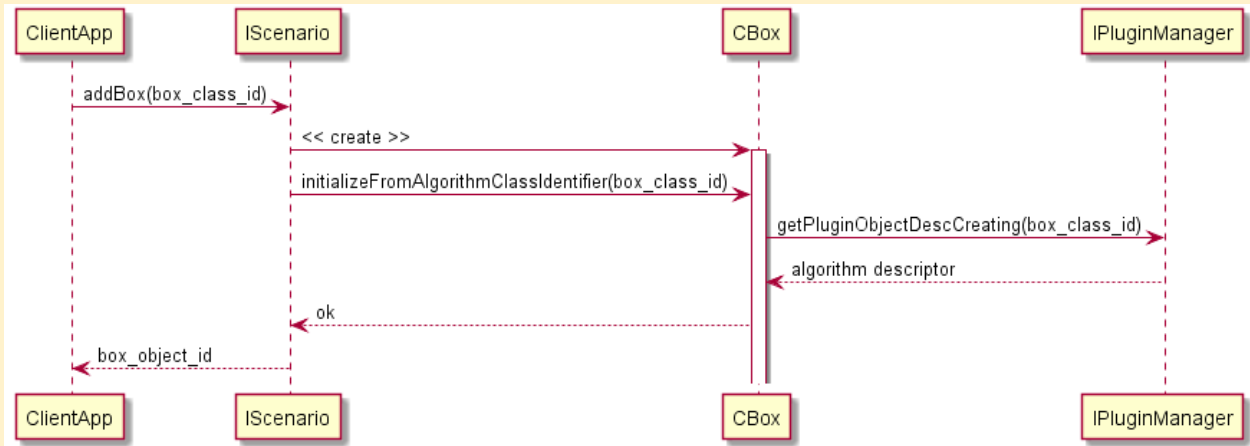
## BV-SEQUENCEDIAGRAM-SCENARIOMANAGEMENT

### Primary Presentation

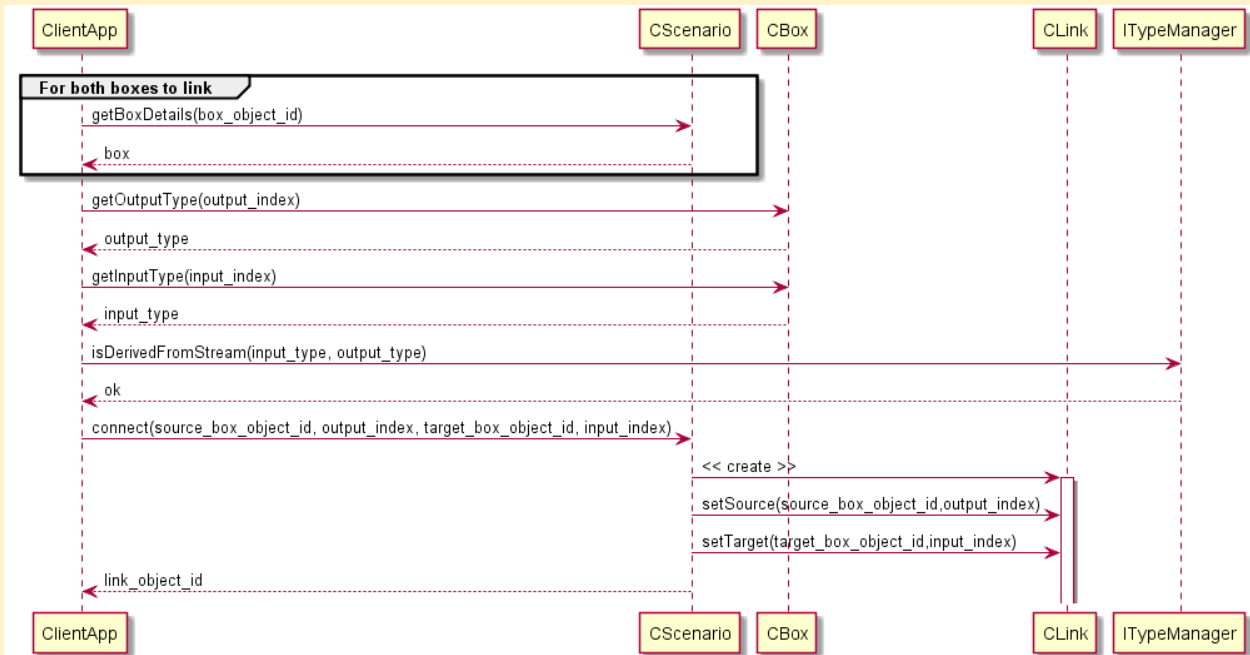
#### Creation



## Adding Boxes



## Connecting Boxes



## Element Catalog / Description

### Creation

In the creation phase, the scenario manager is in charge of scenario creation. A scenario object identifier (see §2.3 Structures Identification) is returned back to the Kernel consumer so that he can use it to retrieve a handle on a scenario object (CScenario) afterwards.

### Adding Boxes

CertiViBE - v1.0	CERT-01 MSD-01	Page 43 / 86
Modules Detailed Design Specifications		

All rights reserved. Passing on or copying of this document, use and communication of its contents not permitted without written authorization

Once scenario handle is retrieved, populating the scenario is performed through the `IScenario` interface. Adding a box involves the creation of a box object (`CBox`) given a box type (see §2.3 Structures Identification for box class identifier). As it was described, a `CBox` object is a wrapper around a box algorithm descriptor. As box algorithm descriptors are plugins components, the plugin manager is responsible for managing their lifetime (see §3.1.3 Plugin Management). Therefore, the box forwards the creation of the descriptor (i.e. `IBoxAlgorithmDesc` instances) to the plugin manager.

**i** To create a scenario, only box algorithm prototypes are needed as no processing is performed. That is the reason why `CBox` objects only need a reference to their corresponding box algorithm descriptor.

**i** The box class identifier is needed to create a box. To retrieve the list of available box algorithm identifiers, Kernel consumers can use `getNextPluginObjectDescIdentifier` method in a loop. Feeding this method with `OV_UndefinedIdentifier` as first parameter and `OV_ClassId_Plugins_BoxAlgorithmDesc` as second parameter returns the first box class identifier and, from that, all identifiers can be retrieved. With an additional call to `getPluginObjectDesc` in the loop, it is easy to retrieve the entire list of box algorithm descriptors. The list of default box algorithms provided by the system to build a scenario is available in appendix (§10.3 Standard Configuration Tokens) and can be grouped in the following categories: Data Generation, Data I/O, Classification, and Signal Processing.

**Q** The system is distributed with a single LDA classifier. Although it is widely used for BCI, system end-users might be interested in adding new classifiers to the framework. This is made very easy thanks to base extension interfaces provide by the `openvibe-toolkit` component:

- `CAlgorithmClassifier`: should be inherited for binary classification;
- `CAlgorithmPairingStrategy`: should be inherited for multiclass (> 2) classification strategies.

### Connecting Boxes

Connecting boxes is straightforward. The goal is to connect a box output (source box) to another box input (target box). Kernel consumers must retrieve `CBox` object handles to get details on the output and input they want to connect. From this information, the type manager can be queried to check output and input are compatible. Finally, the connection operation is performed and the created connection is stored in a `CLink` object within the scenario.

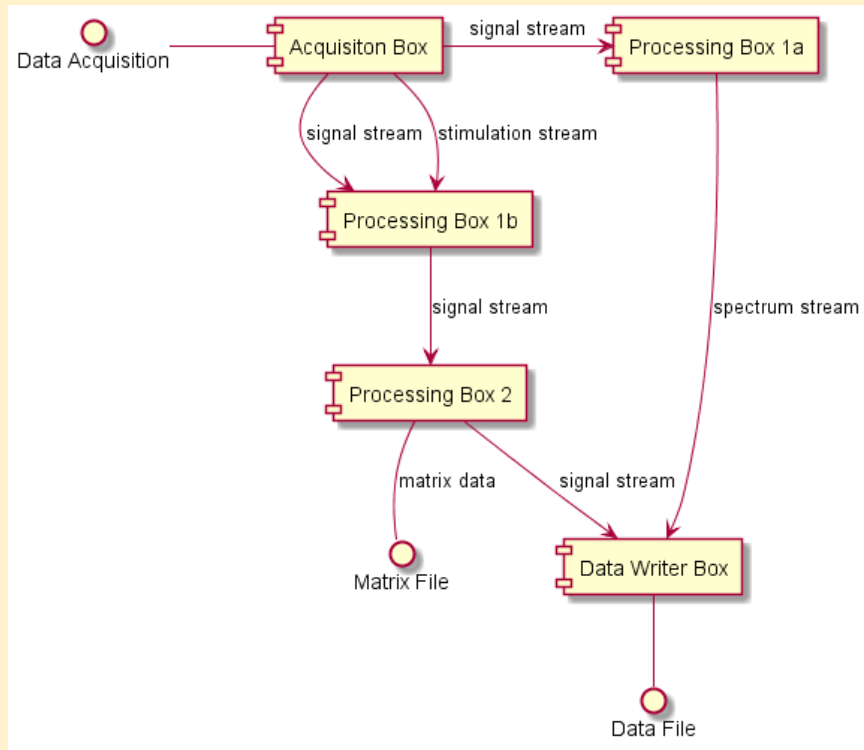
### Notes

Sequence diagrams are high-level representation of the exchange of calls between entities. Therefore, when the diagram shows a return call with an object, it does not imply the called entity returns the object by value in C++. It means that this object is returned in some manner to the caller (e.g. return value, reference parameter etc.).

Scenarios creation within the system is flexible. However, there are some rules scenarios have to comply with to ensure playing a scenario will be stable and reliable at runtime. The following view describes some of this connection rules.

#### LV-ACTIVITYDIAGRAM-CONNECTIONRULES

##### Primary Presentation



##### Element Catalog / Description

The primary presentation shows an example of scenario with boxes and connections. It gives an illustration of what is allowed or not when building scenarios. Here are the basic connection rules that scenarios have to comply with:

- A box output cannot be connected to another box output;
- A box input cannot be connected to another box input;
- A box output can be connected to multiple boxes inputs;
- A box input cannot receive more than one connection;
- A pipeline must be acyclic meaning that boxes behavior cannot impose further dependencies on their antecedents. The acyclic nature of the pipeline removes the possibility of deadlocks between tasks, provided the tasks are truly independent.

CertiViBE - v1.0 Modules Detailed Design Specifications	CERT-01 MSD-01	Page 45 / 86
--	----------------	--------------

All rights reserved. Passing on or copying of this document, use and communication of its contents not permitted without written authorization



## Scenario Configuration

As it was presented in §3.3.1 Box Algorithm Prototype, a number of flags can be set in box prototypes. Some of these flags deal with the modifiability of boxes (possibility to add/remove/modify input/output/setting).

Kernel consumers can retrieve handles to boxes via scenarios (`getBoxDetails`) and perform some modifications to box prototypes. When such a modification is performed, boxes (`TBox`) notify their box listener (see §3.4 Box Listener).

## 6.2 Scenario Loading/Saving

Scenario importing/exporting is handled by the scenario manager. Internally the scenario manager uses a specific importer/exporter algorithm to load/save the scenario. In that way, any scenario format can be imported/exported if the corresponding importer/exporter algorithms are implemented (note that all importers/exporters must inherit toolkit `CAlgorithmScenarioImporter/Exporter`).



## XML Scenario

The importer/exporter algorithms for XML scenario are defined in plugin FileIO. Note that the xml scenario importer performs xsd schema validation on input scenario to validate it conforms to a given format. The validation is implemented through a fallback mechanism:

- Scenario is validated against the newer schema version
- ...
- Scenario is validated against the older (legacy) schema version

The idea behind this mechanism is to be able to discard older versions smoothly in time (first issue a warning for deprecation, then an error).

XSD schemas can be found in the source tree `{Path_Root}/share/opencvibe/kernel.`

## 6.3 Metaboxes definition

A metabox is simply a scenario that exposes the same prototype as a `CBox` class (input/output and settings). Therefore `CScenario` class inherits from `TBox`. `CScenario` Inputs/Outputs are connected to some `boxes` Inputs/Outputs in the scenario. Settings are defined by the user and are handled in the scenario boxes using a `$var{setting_name}` macro.

CertiViBE - v1.0 Modules Detailed Design Specifications	CERT-01 MSD-01	Page 46 / 86
--	----------------	--------------

When a scenario is started, it is cloned and the metaboxes are expanded recursively along with their settings. It is thus possible to have a metabox inside a metabox. Each metabox is identified by a unique identifier assigned to it by its author.



### Metaboxes

When output and/or outputs are defined in a scenario, it can be saved under the `.mxb` extension in order to be considered as a metabox. Settings of the scenario will be considered as settings of the metabox.

During initialization, the kernel checks for metaboxes available in the source tree `{Path_Root}/share/opencvibe/kernel/metaboxes`.

A metabox can be manipulated in a scenario like other boxes.

## 7. Scenario Playback

As stated in SDD, the next step after building a processing pipeline is the ability to execute it. It is achieved through one of the Kernel service dedicated to scenario playback.

This service is centered on the player manager (`CPlayerManager`) which is responsible for creating and destroying instances of players, each payer being responsible for the execution of a single scenario.

The chapter focuses on three critical concepts regarding scenario execution:

- Input data production ( §7.1Data Acquisition describes the module responsible for feeding processing pipeline with input data from EEG devices);
- Execution workflow (§7.2 explains how the processing pipeline is executed by the player);
- Synchronization (§7.3 provides insights on the way time is handled within the system).

### 7.1 Data Acquisition

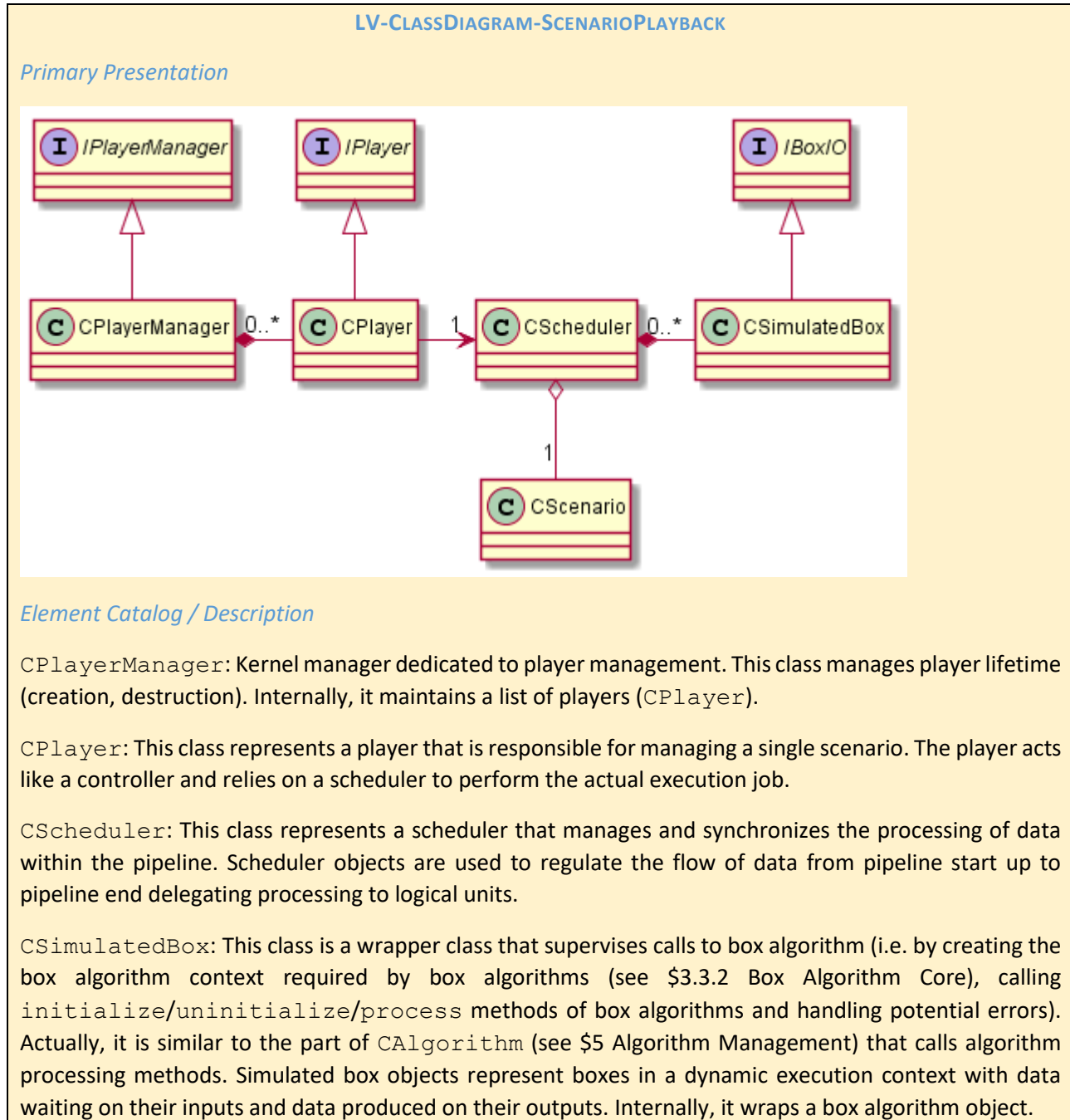
Processing pipeline aims at performing operations on input data. Although the system is able to process data recorded into a file (see §2.2 Stream Structures for stream recording file format), this section focuses on data acquired from EEG devices.

CertiViBE - v1.0 Modules Detailed Design Specifications	CERT-01 MSD-01	Page 47 / 86
--	----------------	--------------

## 7.2 Execution Workflow

This section assumes a scenario is created or loaded (see §6 Scenario Management for scenario creation or loading).

The execution workflow relies on a set of fundamental classes that are described in the following view.



**i** Unlike `CAlgorithm` class that wraps into one interface the exposure of the algorithm prototype and the call to algorithm processing methods, `CSimulatedBox` just implements the call-forwarding part. The prototype exposure part is dedicated to another class (`CBox`) that is presented in §6 Scenario Management .

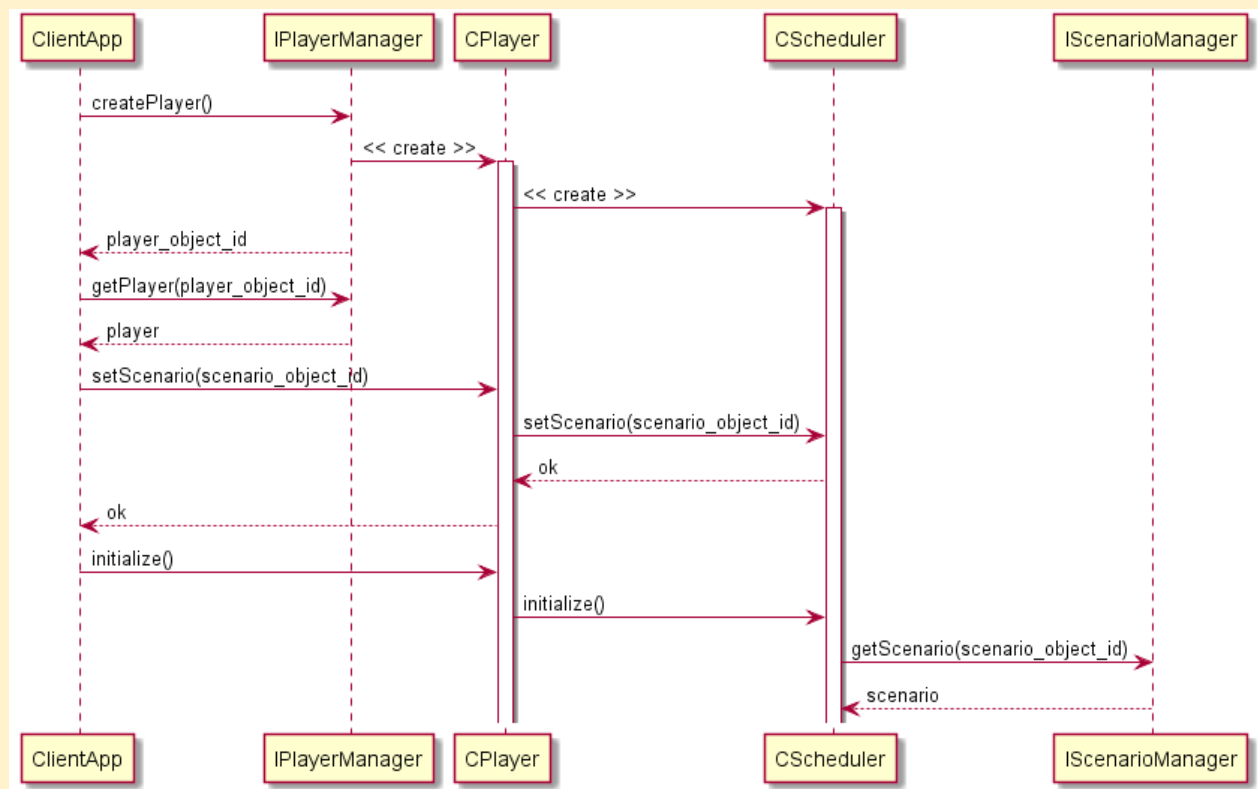
The previous view presents the main actors involved in scenario playback. The next one describes how these actors are involved in the initialization of the pipeline. The initialization of the pipeline is a critical step that must be performed before its execution to initialize resources and setup the execution environment configuration.

**i** Although only the initialization step is presented in this section, an uninitialization step must be performed after a scenario execution to release resources.

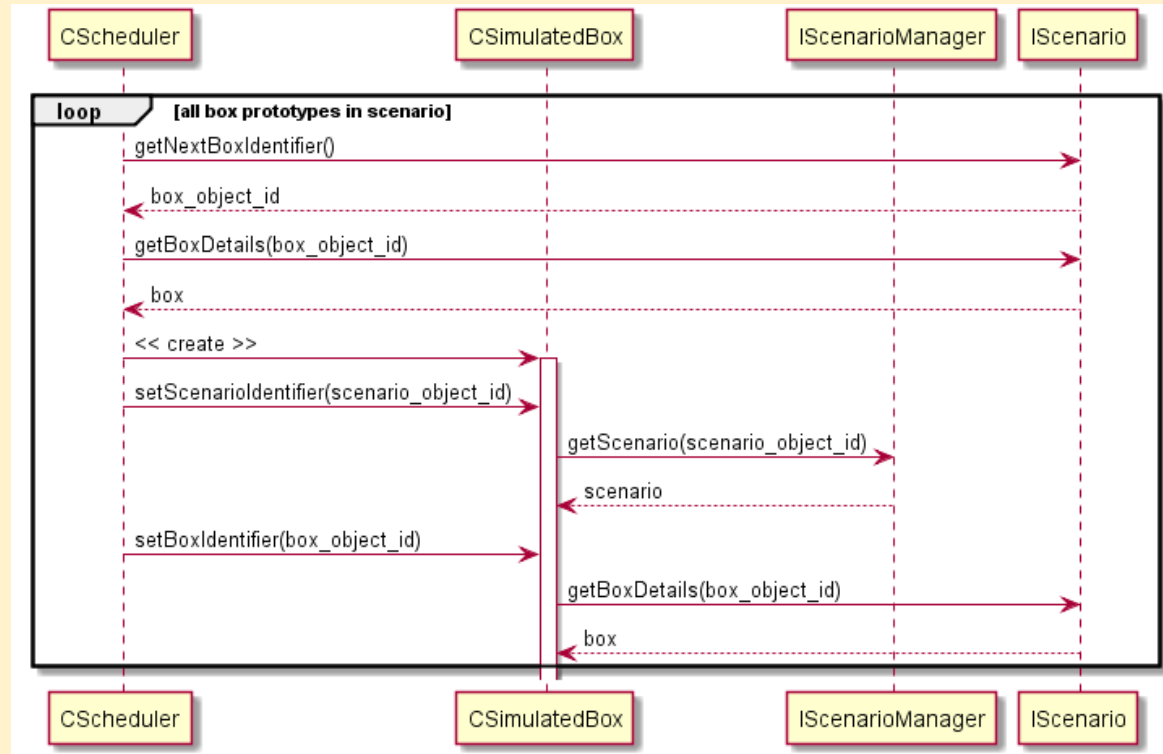
## BV-SEQUENCEDIAGRAM-PLAYBACKINITIALIZATION

### Primary Presentation

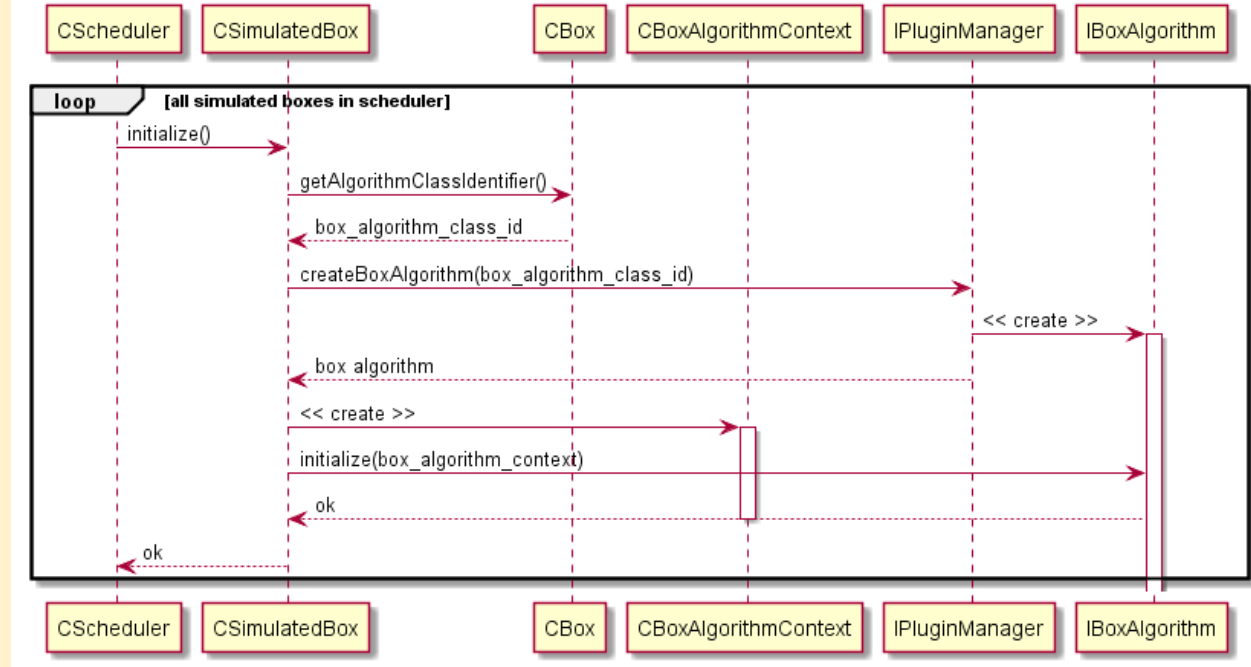
#### Player Creation



## Simulated Boxes Creation



## Simulated Boxes Initialization



### Player Creation


The player manager is responsible for creating players. Each player relies on a scheduler instantiated at creation time. Both scheduler and player own a reference on the scenario to be executed.


### Simulated Boxes Creation

Simulated boxes creation occurs within `initialize` call. The scheduler retrieves all box prototypes from the scenario. For each box prototype (`CBox`), a simulated box (`CSimulatedBox`) is created.

### Simulated Boxes Initialization

During their initialization step, simulated boxes use their `CBox` reference to retrieve the box algorithm class identifier. They use this identifier to request the plugin manager for box algorithm plugin object creation.

 Again, this step illustrates the difference between `CBox` objects and `CSimulatedBox` objects. In §6 Scenario Management, it is explained that `CBox` objects only need a box descriptor instance as no processing is involved at scenario creation time. Unlike `CBox` objects, `CSimulatedBox` objects are execution time representation of boxes. At runtime, box logic is needed to actually process data. Therefore, `CSimulatedBox` objects need a reference to the box algorithm plugin object.

 As described in §5 Algorithm Management, `CAlgorithm` objects are responsible for creating the context needed by algorithms to perform their tasks. In the same manner, `CSimulatedBox` objects create a box algorithm context (see §3.3.2 Box Algorithm Core for description of box algorithm context) for each new call to box algorithm methods. As the context is only valid during a single call to a box algorithm object method, implementation of box algorithm logical units should not store the box algorithm context for a later use.

Once the execution environment is setup, a player is ready to execute a scenario. For that, it provides a `loop` method that must be invoked periodically by Kernel consumers.



### Player `loop` method

The `loop` method is a short-time function that must be called by client application repeatedly. The `loop` method is the heartbeat of scenario execution. Each `loop` execution “*can lead*” to the update of the processing pipeline state (i.e. input data update, calling processing methods of every boxes in the pipeline).

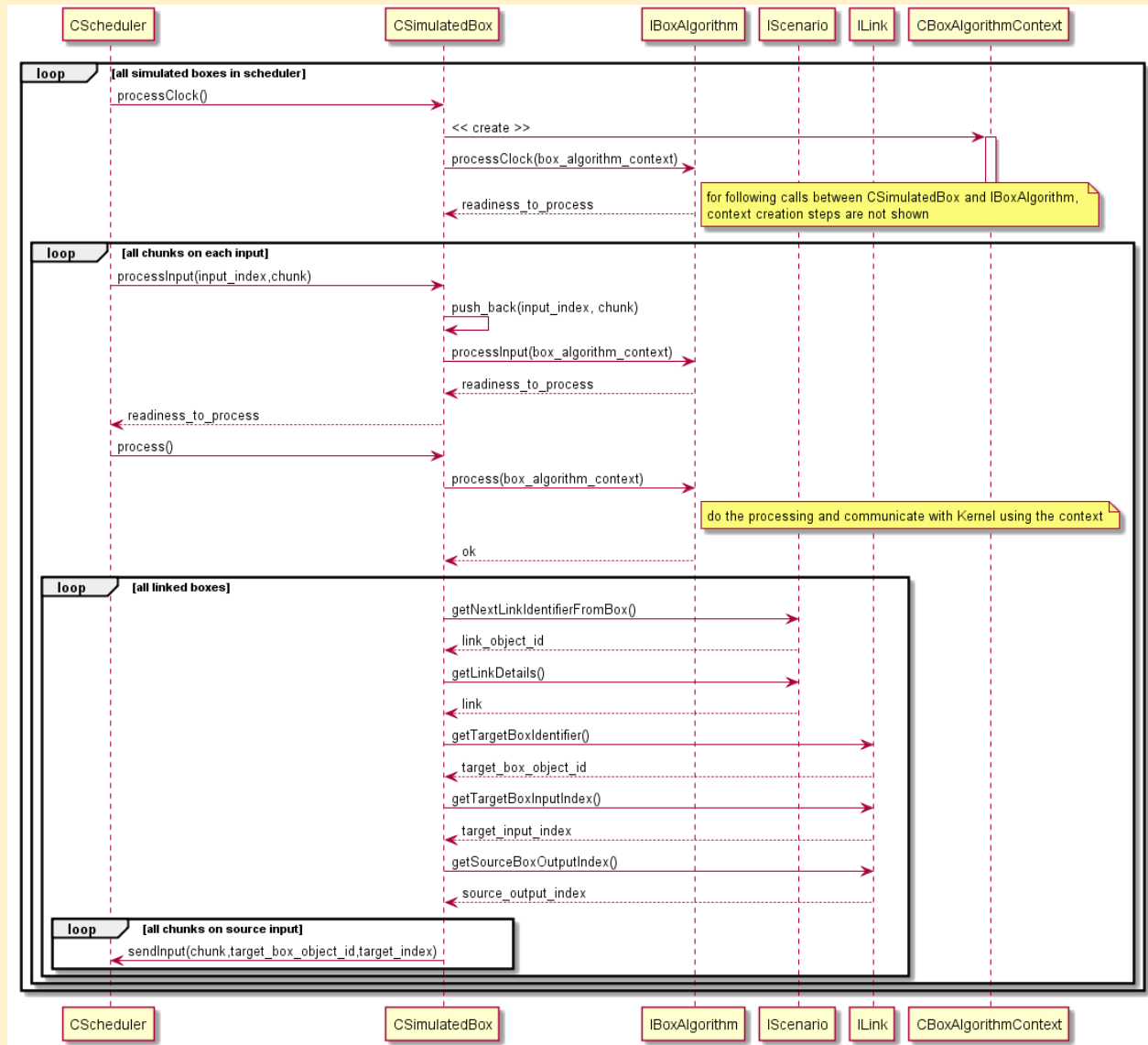
“*Can lead*” means an update of the pipeline can be triggered or not (see Timing section for details on synchronization). If an update is needed, it is done through a call to the scheduler `loop` method. The following view describes in detail what occurs within a scheduler loop.

CertiViBE - v1.0 Modules Detailed Design Specifications	CERT-01 MSD-01	Page 51 / 86
--	----------------	--------------

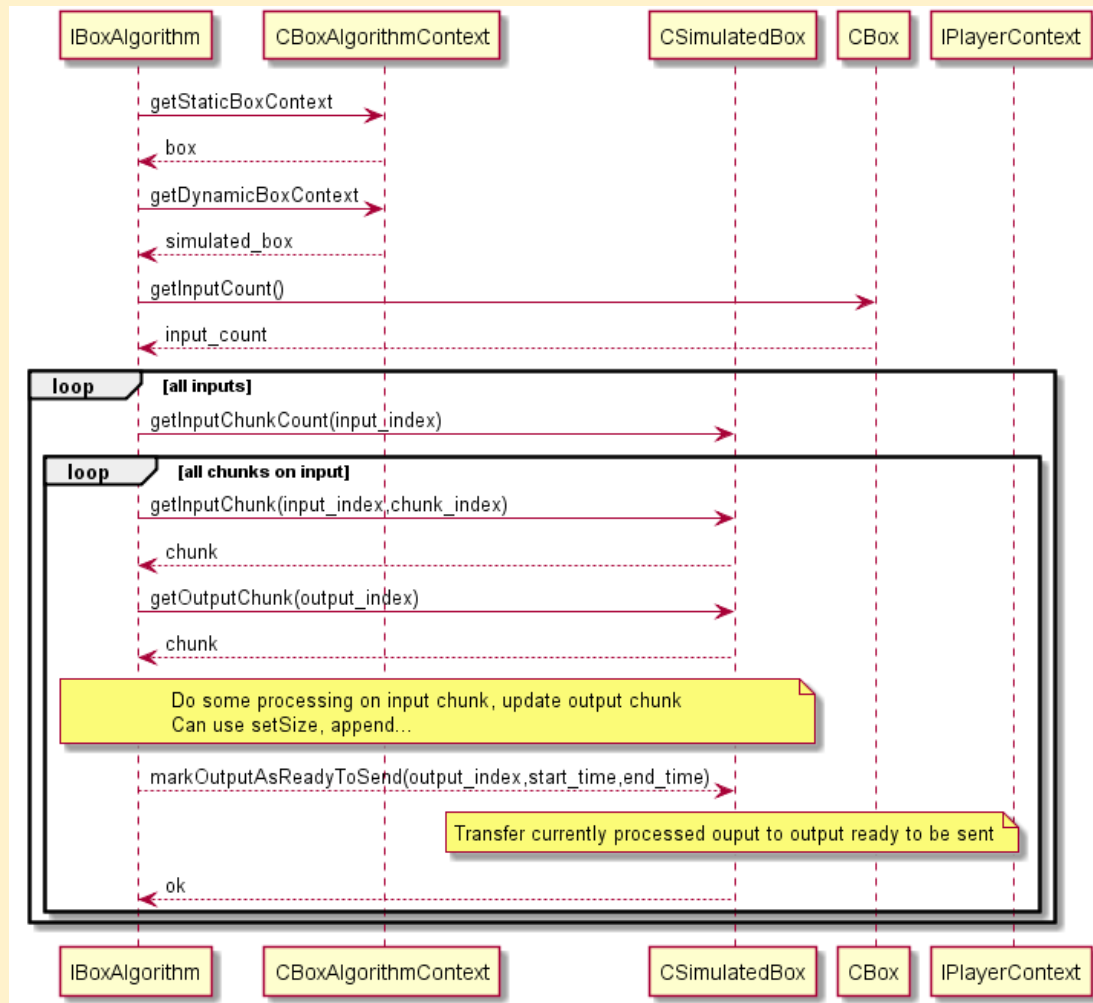
## BV-SEQUENCEDIAGRAM-PLAYBACKLOOP

### Primary Presentation

#### Scheduler Loop



## Box Algorithm Process



## Element Catalog / Description


### Scheduler Loop

The scheduler loops through all simulated boxes in the scenario. Calling the `process` method of a box algorithm to perform the real processing on data is a 3-steps sequence:

- 1: The scheduler asks the box algorithm (manipulated through the simulated box) if it is ready to process data (`processClock` for time-driven boxes and `processInput` for data-driven boxes).
- 2: The box algorithm checks its internal state and inform the simulated box if it is ready or not to process. For instance, a data-driven box aiming at processing signal data when a stimulation occurs could only be

ready to process once the stimulation chunk is received. At the meantime, signal data would be buffered within the simulated box (`push_back`).


-3: The scheduler decides to trigger the processing only if the box algorithm is ready. Actual processing is forwarded by simulated boxes to box algorithms. Once processing is performed, simulated boxes retrieve all the boxes their outputs are connected to and use the scheduler to send produced output data to the right input of next boxes in the pipeline (`sendInput`).

 For data-driven boxes, step 2 and 3 are performed for all data chunks waiting on each input.

### Box Algorithm Process

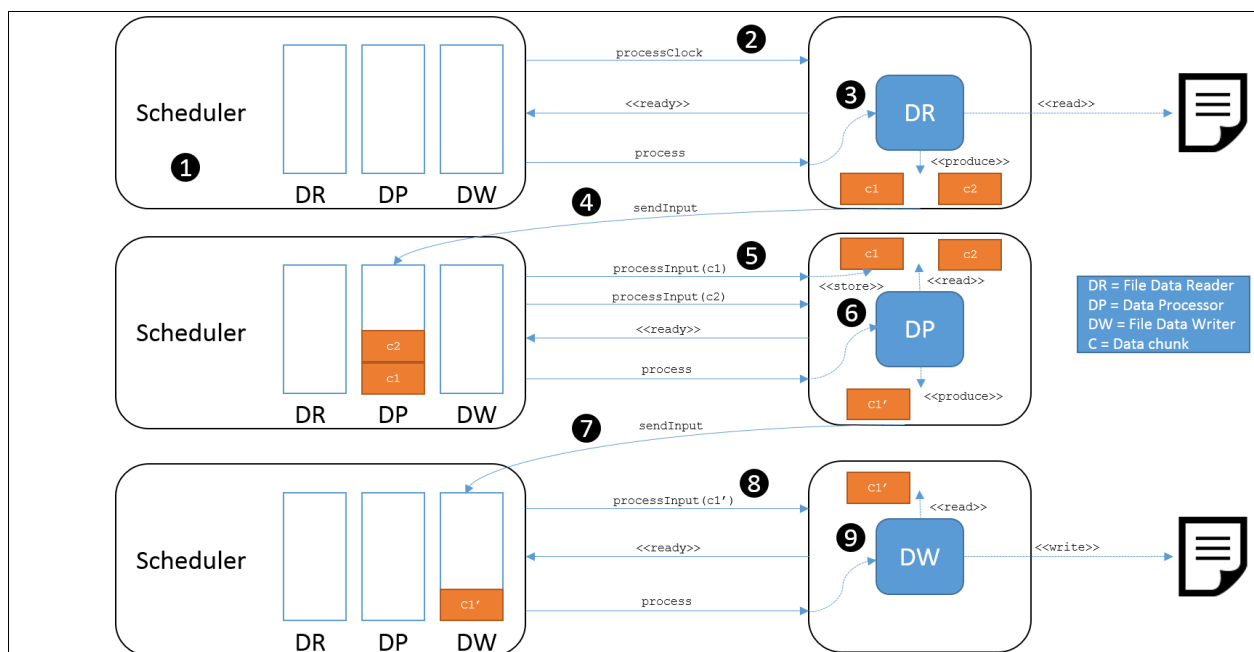
This sequence focuses on box algorithms `process` method. It illustrated the use of the box algorithm context to communicate with the Kernel module. Box algorithms can retrieve the static box context (`CBox` reference), the dynamic box context (`CSimulatedBox` reference) or the player context from the box algorithm context.

Typical uses of static context is to retrieve some prototype information (e.g. number of inputs). Box algorithms mostly make use of the dynamic context to manipulate input and output data as well as for communication with the Kernel.

 Some communication steps between box algorithms and the Kernel module are automatically performed by codec algorithms (see §3.3.2 Box Algorithm Core): decoders retrieve input chunks to decode them and mark them as deprecated while encoders retrieve output chunk references to fill their buffer.



### Data Flow



This example shows the data flow for a single scheduler loop. The basic scenario consists of reading data from a file, processing data and writing data into a file. The emphasis is put on the the data flow between the scheduler, its associated simulated boxes and their associated box algorithm (blue chip).

1. The scheduler keeps input data for each simulated box in separate containers. At the initial stage, there is no data available in the pipeline.
2. The scheduler calls `processClock` on the data reader box (time-driven box). When the box is ready to process, the scheduler triggers the processing.
3. The data reader box reads data from a file and produces two output chunks `c1` and `c2`.
4. Once processing is finished, the produced output chunks are transmitted to the scheduler that stores them in next box's input data container.
5. The scheduler then deals with the next box in the pipeline. For each input chunk in the input data container, it calls `processInput`. The first chunk `c1` is stored in the simulated box. The data processor box informs the scheduler it is ready to process after receiving the second chunk `c2`.
6. The data processor box processes `c1` and `c2` and produces an output chunk `c1'`.
7. Once processing is finished, the produced output chunk is transmitted to the scheduler that stores it in next box's input data container. The scheduler cleans DP container.

8. The scheduler then deals with the last box in the pipeline. The data writer box informs the scheduler it is ready to process after receiving the chunk  $c1'$ .
9. The data writer box processes  $c1'$  and writes data into a file.

## 7.3 Timing

The previous section explained that a player loop can or cannot lead to an update of the pipeline. The decision to update is tightly linked to the time model used within the system and described in the next section.

### 7.3.1 Time Model

Internally, the system uses simulated real-time as time model in order to handle homogeneously input data coming from an acquisition device (real-time data) or from a data file (simulation data). It is a well-known pattern used in the game industry (fixed time-step) and in any field where simulation is involved. The basic idea is to update the logic of the system at fixed intervals.

Here is some pseudo-code to illustrate the simulated-time concept:

```
while(app_running) 1
{
    int realTime = GetTime();
    while (simulatedTime < realTime) 2
    {
        simulatedTime += time_step; 3
        update(); 4
    }
    // do some work (rendering, etc...)
}
```

The main loop (**1**) represents a basic application loop (API client code).

In the system, the second loop (**2**) is implemented in the `player loop` method. Update of the logic (**4**) consists of a scheduler loop (a scheduler loop is a period during which each box in a scenario is executed exactly once as explained in §7.2 Execution Workflow). During a scheduler loop, each box in the scenario work as if the time was the simulated real time. At the end of the loop, the scheduler increases the simulated time by a time step value (**3**).



Every time value handled at box algorithm level is simulated real-time, be it acquired from a device or replayed from a file. Only the application layer has real real-time awareness.

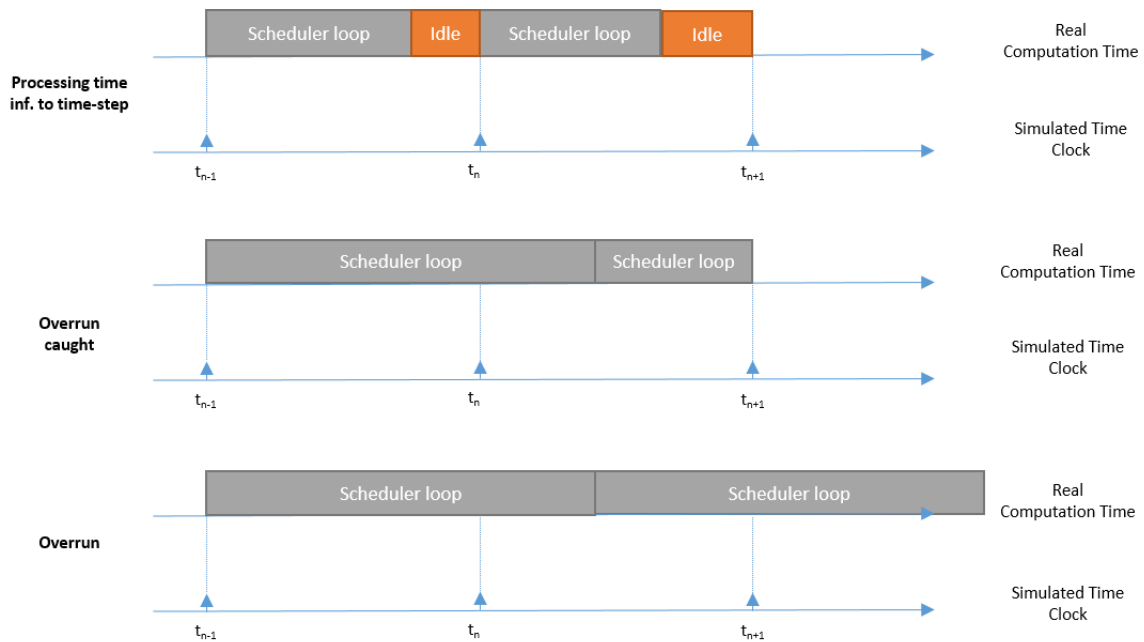
Amount of real time needed for a scheduler loop can be:

- longer than the time step (overruns);
- shorter than the time step.

The following picture illustrates both cases.

CertiViBE - v1.0 Modules Detailed Design Specifications	CERT-01 MSD-01	Page 57 / 86
--	----------------	--------------

All rights reserved. Passing on or copying of this document, use and communication of its contents not permitted without written authorization



In the first case, real processing time is inferior to the fixed time step. During each time step, the system performs a scheduler loop. As the loop takes less time than the time step, the processor remains idle during the remainder of the time.

In the last case, real processing time is superior to the fixed time step what is called overrun. The system cannot ensure outputs produced by the pipeline (e.g. visual feedbacks) are real-time.

In the middle case, the overrun is corrected in the next iteration by a lower processing time. The simulated real-time manages to get back to supposed real real-time after a temporary long processing.

The following table presents theoretical examples that illustrate the first and last cases. In these examples, it is assumed that real processing time is constant (**4** is constant-time), elapsed time only depends on processing time (i.e. all operation durations are negligible in loop **1** except **4**) and synchronization was perfect up to  $t_{n-1}$  (real processing time = time step).

	Real Processing Time = 0.1s (time step = 1s)	Real Processing Time = 10s (time step = 1s)
Loop ( <b>1</b> ) Iteration n-1	Real time = $t_{n-1}$ 1 scheduler loop Simulated time += 1s ( $=t_{n-1}$ )	Real time = $t_{n-1}$ 1 scheduler loop Simulated time += 1s ( $=t_{n-1}$ )
Loop ( <b>1</b> ) Iteration n	Real time += 0.1s ( $=t_{n-1} + 0.1s$ ) 1 scheduler loop Simulated time += 1s ( $=t_{n-1} + 1s$ )	Real time += 10s 10 scheduler loops Simulated time += 10s ( $=\text{Real time}$ )
Loop ( <b>1</b> ) Iteration n+1	Idle time	Real time += 100s ( $10 \times 10s$ ) 100 scheduler loops Simulated time += 100s ( $=\text{Real time}$ )



Setting of the time step is crucial. Reducing the step time reduces idle time raising the risk of overrun.

Based on the time model, the player offers multiple execution modes:

- Normal speed mode that is used to illustrate the time model above (the player requests the right number of scheduler loops according to real elapsed time; see pseudo-code above);
- Accelerated mode (the player requests as many scheduler loops as possible);
- Step-by-step mode (the player requests a single scheduler loop).

### 7.3.2 Time Representation

Time within the system is represented on 64 bits integer in 32:32 [fixed point arithmetic](#). The choice of fixed-point arithmetic over floating-point arithmetic is driven by the need to know exactly the error that is included in computations in order to be able to correct the error or chose the computations in a way that has no derivation (see [What Every Computer Scientist Should Know About Floating-Point Arithmetic](#) for details about floating-point rounding errors). As fixed-point time value are represented as integer, the gap between two adjacent values is always equals to one (what leads to maximum rounding error of  $1/2^{32}$  which is a sufficient precision in the system) while this distance varies a lot with floating-point representation.

As a consequence, arithmetic operations on fixed-point numbers (e.g. addition, subtraction etc.) introduce no error beyond that in their arguments.



One issue that must be taken into account with operations on fixed-point numbers is overflow. Care has to be taken on the operation order (especially when multiplications and divisions are involved). For conversion to/from fixed-point representation, the system provides a special class (`ITimeArithmetics`).

### 7.3.3 System Clock

In section §7.3.1 Time Model, the use of simulated time within the system is illustrated. The scheduler and all box algorithms manipulate simulated time. However, the notion of “real” time is essential to some subsystems:

- The acquisition module;
- The `CPlayer` class;
- Benchmarking features (`CChrono` in `openvibe-module-system`);

CertiViBE - v1.0 Modules Detailed Design Specifications	CERT-01 MSD-01	Page 59 / 86
--	----------------	--------------

- Client code to know the elapsed time between two calls to the player loop.

In the system retrieving the “real” time relies on the OS time measured by the OS clock. This service is provided by the `zgetTime` method in the `openvibe-module-system` component.

#### System time on Linux

On Linux, time retrieval is based on `gettimeofday` which has a granularity of one microsecond.

#### System time on Windows

On Windows, time retrieval is based on a dual strategy based on `timeGetTime` (5/6 milliseconds precision) and high precision counters (`QueryPerformanceCounter` and `QueryPerformanceFrequency`).



The system relies on the OS clock while a potentially connected device has its own internal clock. At the present time, there is no way to synchronize both clocks in the system.

## 7.4 Scenario Player application

Scenario player is the application responsible for loading and playing a scenario using a command line without launching any graphical user interface.

Here are available options of the application:

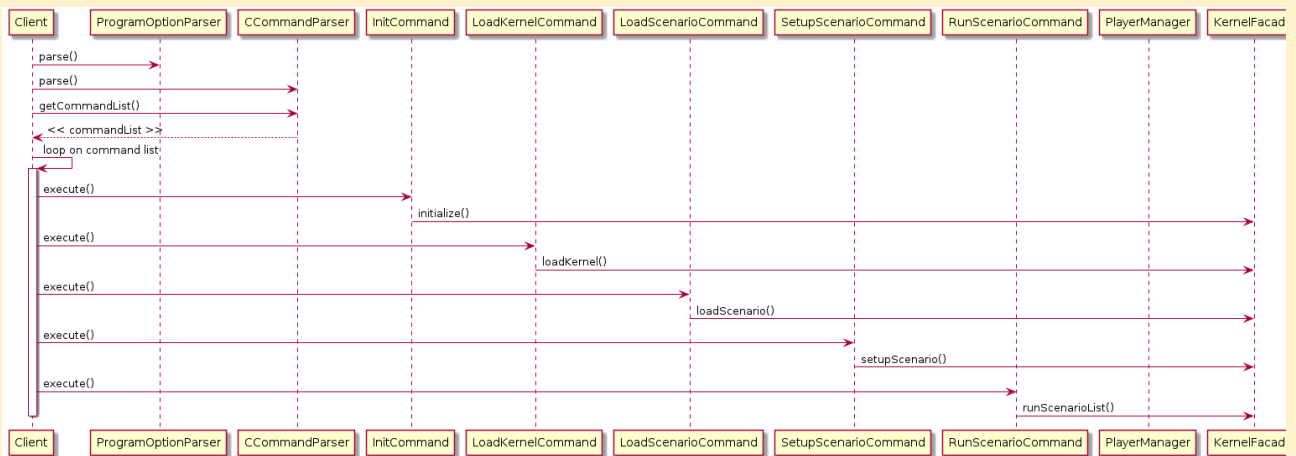
Option	Description	Mandatory
<code>--command-file</code>	Path to command file (command mode only)	Yes
<code>--config-file</code>	Path to configuration file (express mode only)	No
<code>--dg</code>	Global user-defined token: -dg="(token:value)" (express mode only)	No
<code>--ds</code>	Scenario user-defined token: -ds="(token:value)" (express mode only)	No
<code>--max-time</code>	Scenarios playing execution time limit (express mode only)	No
<code>--mode</code>	Execution mode: 'x' for express, 'c' for command	Yes
<code>--play-mode</code>	Play mode: std for standard and ff for fast-foward (express mode only) [default=std]	No
<code>--scenario-file</code>	Path to scenario file (express mode only)	Yes

### 7.4.1 Scenario player execution workflow

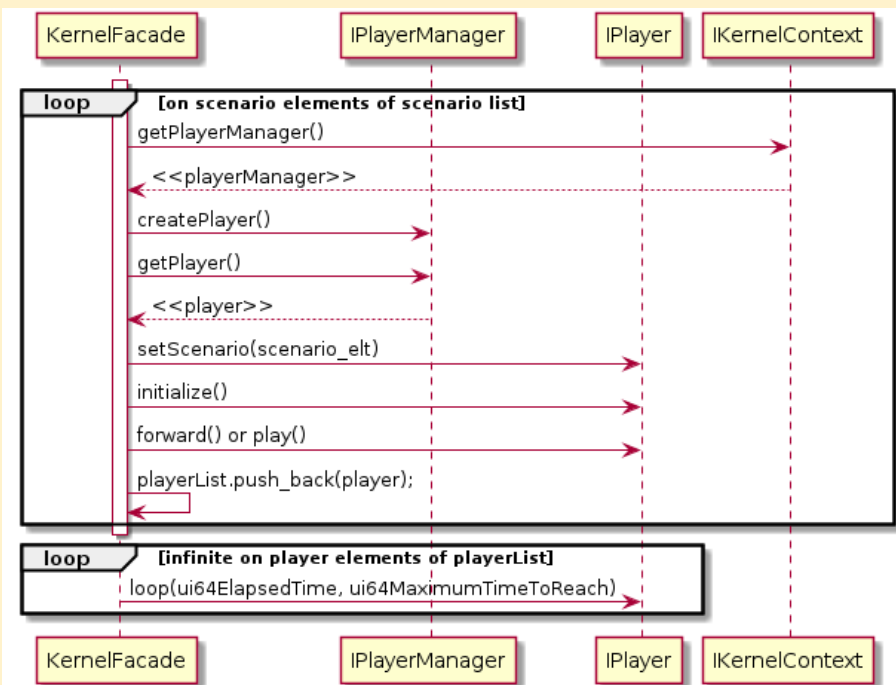
A straightforward commands workflow is built according to command-line (or a command file).

#### Scenario Player execution workflow

##### Commands workflow



##### Scenario(s) running



## 8. Configuration Management

Configuration management is about setting up an environment to execute a scenario. It is achieved through one of the Kernel class dedicated to configuration management: the configuration manager (`CConfigurationManager`).

### 8.1 Configuration Token

Configuration is handled with tokens that consist of name/value pairs. To access token values in other variables, the following syntax is used: `${Token_Name}`.

The configuration manager provides an expansion mechanism that analyzes a string and substitutes any token reference (`${Token_Name}`) with the token value if the token name matches an internally registered one. There are many ways to register tokens within the configuration manager.

The manager can be manipulated directly to configure the overall runtime session:

- Tokens can be added individually to the manager (`createConfigurationToken`);
- A set of tokens can be loaded by the manager from a configuration file (`addConfigurationFromFile`).

Scenario-specific configuration tokens can be set indirectly in two ways:

- Tokens can be added individually through the player object in charge of executing the scenario (`CPlayer setScenario` method takes a list of name/value pairs as parameters);
- A set of tokens can be defined in custom configuration files located in the same directory as the loaded scenario file (this feature is only available for scenarios loaded from a file). To activate this feature, client code must ensure `OV_AttributeId_ScenarioFilename` attribute is added to the scenario before the corresponding player is initialized (`TAttributable, addAttribute`).

The following view describes the expected file format for configuration files.

CertiViBE - v1.0 Modules Detailed Design Specifications	CERT-01 MSD-01	Page 62 / 86
--	----------------	--------------

### Primary Presentation

```
# Comments
# Root path expressed relative to execution directory dir
Path_Root = MyPath
# Declared token reuse in value
Path_Bin = ${Path_Root}/bin
# Token value inside a token name
ExperimentName = P300
ExperimentP300Path = /openvibe/data/p300
ExperimentPath = ${Experiment${ExperimentName}Path}
# Environment variable used in value
Path_Env = $Environment{ENV}
# Inclusion of a sub configuration file
Include = c:/Demo/${ExperimentName}.cfg
```

### Element Catalog / Description

Token pairs are declared with `name = value` statements. The manager allows token names to be expanded as token value by using `${name}`.

In the snapshot, `${Path_Root}` is expanded to `MyPath` when the manager interprets the `Path_Bin` token.

Token value can also be used within token name. In the snapshot, `ExperimentPath` will be expanded to `${ExperimentP300Path}` that is equal to `/openvibe/data/p300`.

Environment variables can also be used as value by using the `$Environment{ENV_NAME}` syntax.

Specific tokens with `core` prefix have their value updated each time it is retrieved from the manager.

Note that a configuration file can be included from another configuration file to override some standard settings or add specific settings by using the `Include` token name.

### Notes

The list of standard OpenViBE tokens is available in §10.3 Standard Configuration Tokens.

CertiViBE - v1.0 Modules Detailed Design Specifications	CERT-01 MSD-01	Page 63 / 86
--	----------------	--------------



A standard session configuration file is available in `/share/opencvibe/kernel/*.conf`. This configuration file uses the `Include` token to load other configuration files. Among these included configuration file, there is a custom file that can be edited depending on the user needs and preferences.

Custom file location:

- Linux: `~/.Config/opencvibe/opencvibe.rc`.
- Windows : `%APPDATA%/opencvibe/opencvibe.conf`

For scenario-specific configuration, files must be located in the same directory as the scenario file and follow this naming convention:

- `*.conf` and preferentially `scenario.conf`

**Note that custom files are not created by the system and must be created manually.**

## 8.2 Box Settings Customization

As explained in §3.3.1 Box Algorithm Prototype, box settings are handled as string so they can be expanded by the configuration manager. In this way, the system can achieve late-binding configuration of boxes without any scenario modification. Here are some details on how this is achieved:

1. A box algorithm setting is set with a token-dependent value as default value in the prototype declaration or as a new value (for modifiable settings) via the `TBox` interface (`setSettingValue`). Note that the token can be a standard token (§10.3 Standard Configuration Tokens) or a custom token.

*Settings value example:* `${Custom_Token}/experiment/`

2. The token is either defined in a configuration file loaded by the configuration manager (`addFromConfigurationFile`) or added individually to the manager (`createConfigurationToken`).

*Configuration token example:* `Custom_Token = ~/opencvibe/p300`

3. At runtime, during the initialization or processing phase, box algorithms use the box algorithm context to request settings values (`CBoxAlgorithmContext`, `getSettingValue`). Internally, the context uses the configuration manager to expand each value (i.e. replace every token name reference with the corresponding token value).

*Retrieved setting example:* `~/opencvibe/p300/experiment`

CertiViBE - v1.0 Modules Detailed Design Specifications	CERT-01 MSD-01	Page 64 / 86
--	----------------	--------------



## Box Settings File

Box settings can also be overridden from settings specified in a file. For that, application code (i.e. client code) must retrieve the box (CBox object) and add the attribute `OV_AttributeId_Box_SettingOverrideFilename` with the settings filename to the box.

The scheduler checks for the presence of this attribute to load the settings from the file. For that, it makes use of the visitor pattern implemented in the base class of all OpenViBE objects (IObject). During the initialization step, the current scenario is visited by a specific visitor (CBoxSettingsModifierVisitor) in charge of checking for each box the existence of the attribute mentioned above.

If the attribute is found, the settings file is loaded and parsed. As settings files follow XML format, the visitor makes use of `openvibe-module-xml` to parse the file. Settings file must comply with the following structure with settings being declared in the same order as in the prototype:

```
<OpenViBE-SettingsOverride>
    <SettingValue>Algo2</SettingValue>
    <SettingValue>18.4</SettingValue>
    <SettingValue>>false</SettingValue>
</OpenViBE-SettingsOverride>
```

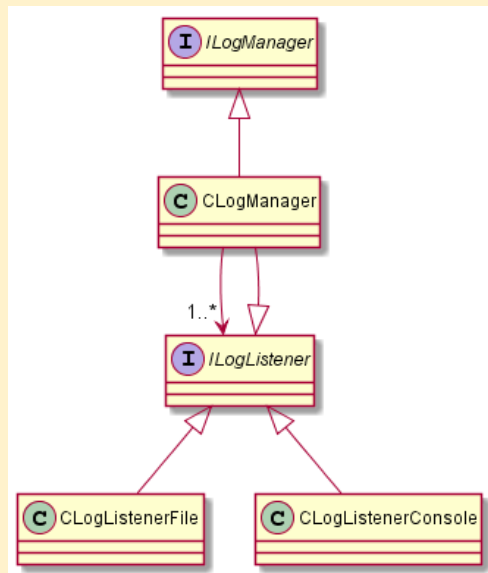
## 9. Log and Error Management

### 9.1 System Logging

System logging is handled by the Kernel log manager (`CLogManager`). The following view describes this class in details.

LV-CLASSDIAGRAM-LOGMANAGEMENT

#### Primary Presentation



#### Element Catalog / Description

**CLogManager:** Central class used to trace messages. It implements the `ILogListener` interface. The manager forwards each log request to attached listeners that are registered at runtime. The log manager provides different level of activation:

- Debug: Use in debug mode. Add more information than Info mode.
- Benchmark: Use for benchmark testing.
- Info: The software behavior is as expected but the information is valuable.
- Warning: Either to alert a fault is about to happen or to report the current behavior might be different from the expected one.
- Error: A fault appeared somewhere in the system. The fault is detected and reported. Usually, the component cannot proceed and is subsequently cancelled/disabled.

CertiViBE - v1.0 Modules Detailed Design Specifications	CERT-01 MSD-01	Page 66 / 86
--	----------------	--------------

All rights reserved. Passing on or copying of this document, use and communication of its contents not permitted without written authorization

- Fatal: Something that should never happen actually happened. Crash could not be avoided. It might be used by client application to handle exceptions/crashes not caught by the Kernel module.

Log levels can be activated/inactivated either at manager level or at listener level. Inactivation at manager level leads to inactivation of this log level for all listeners. Log levels can be setup from a configuration file (see §10.3 Standard Configuration Tokens).

`CLogListenerFile`: Listener class used to log messages in a file.

`CLogListenerConsole`: Listener class used to log messages on console.

The system is easily extensible as new listeners can be implemented and attach to the logger at runtime.

#### Notes

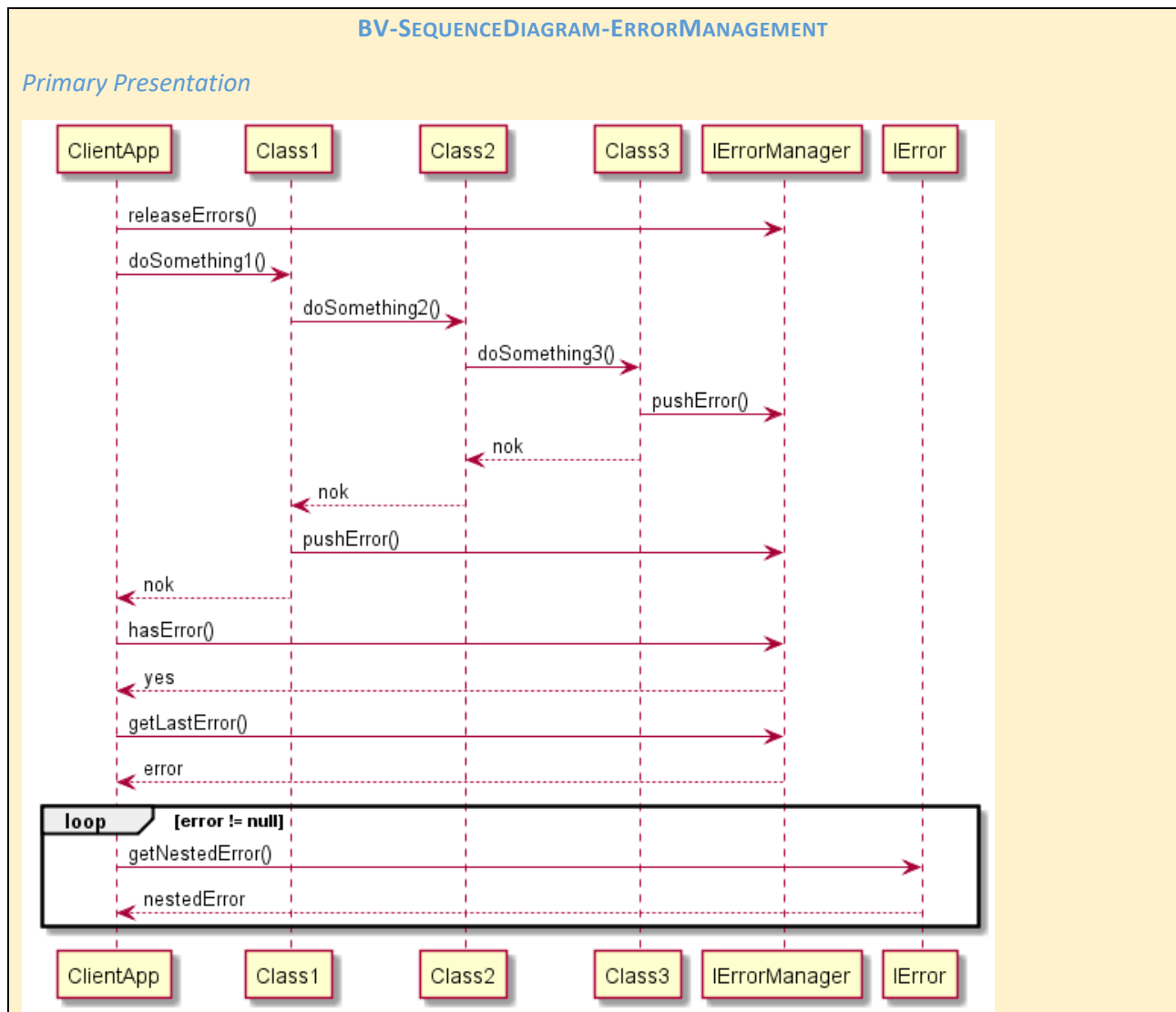
Some layers in the inheritance hierarchy tree are discarded for the sake of clarity.

## 9.2 Error Management

Error management is handled with an enhanced errno-like mechanism. Rationales behind this design were:

- Interface must respect the abi compatibility design implemented in the whole framework
  - ➔ Implementation with pure abstract interfaces using standard types (pointers, integer etc.)
- Adding error management must not break source compatibility with older versions
  - ➔ Method return type (usually Boolean) and parameters left unchanged

As a consequence, the current implementation is built around a Kernel manager (`IErrManager`) that stores errors (`IErr`) and can be queried to retrieve the error stack.



### Element Catalog / Description

- 1: The error manager is released
- 2: The public API is queried
- 3: Failure in `doSomething3()`. A new error with low-level details is pushed to the manager and `nok` is returned.
- 4: `doSomething2()` checks the result of call to `doSomething3()` and just returns accordingly.
- 5: `doSomething()` checks the result of call to `doSomething3()`. As adding information is valuable here, a new error with high-level information is pushed to the manager.
- 6: `ClientApp` checks the result of `doSomething()`. As it is `nok`, it queries the top level error from the manager.
- 7: `ClientApp` retrieve low-level errors thanks to the nested error mechanism.

The error manager holds an error stack. Each time a new error is pushed to the manager, it is pushed on the top of the stack. In this way, an error can be enhanced or not at each level and the calling code can retrieve the whole error backtrace.



A set of utility macros was implemented to make handling errors easier in the framework.

10.Appendix

10.1 Stream Structure Specifications

LV-EBMLSTREAM

Primary Presentation

I

EBMLStream

HEADER

stream\_type

stream\_version

BUFFER

END

OVTK\_NodeId\_Header

OVTK\_NodeId\_Header\_StreamType (integer:)

OVTK\_NodeId\_Header\_StreamVersion (integer:)

OVTK\_NodeId\_Buffer

OVTK\_NodeId\_Buffer

...

OVTK\_NodeId\_End

Element Catalog / Description

Base stream not intended to be used directly.

**Stream description (left view)**

Version: 1

stream\_type: Type identifier.

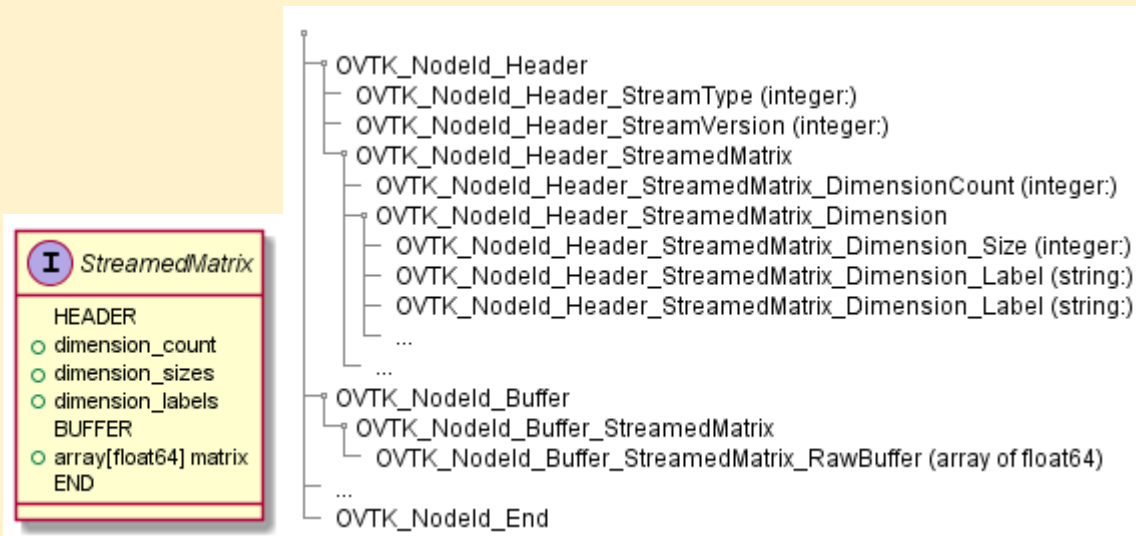
stream\_version: Version number.

**Stream structure (right view)**

EBML nodes tree (see §2.3 Structures Identification for references about node tree identifiers).

CertiViBE - v1.0 Modules Detailed Design Specifications	CERT-01 MSD-01	Page 70 / 86
--	----------------	--------------

All rights reserved. Passing on or copying of this document, use and communication of its contents not permitted without written authorization

*Primary Presentation**Element Catalog / Description*

Base stream for stream conveying data matrices.

**Stream description (left view)**

Version: 1

dimension\_count: Numbers of dimensions for the matrix.

dimension\_sizes: Size of each dimension.

dimension\_labels: List of labels for each dimension.

matrix: Matrix values.

**Stream structure (right view)**

EBML nodes tree (see §2.3 Structures Identification for references about node tree identifiers).

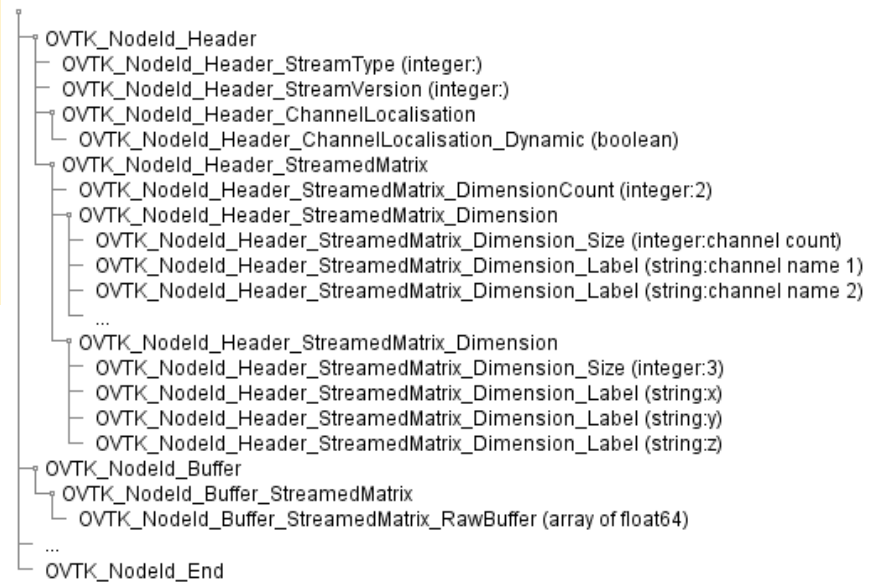
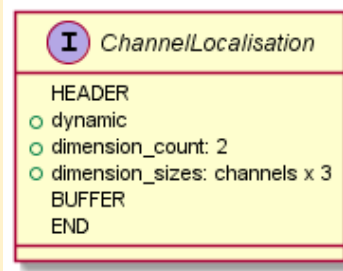
Dimension field and its subfields should recur as many times as dimension count.

*Notes*

Inherited properties are left out in the stream description view.

## LV-CHANNELLOCALISATIONSTREAM

### Primary Presentation



### Element Catalog / Description

#### Stream description (left view)

Version: 1

dynamic: false if sensor coordinates are static, true if it can changes over time (more than one buffer received).

dimension\_count: Channels and positions.

dimension\_sizes: dim1 = channel count, dim2 = 3 (normalized Cartesian coordinates in reference frame Xright, Yfront, Zup).

#### Stream structure (right view)

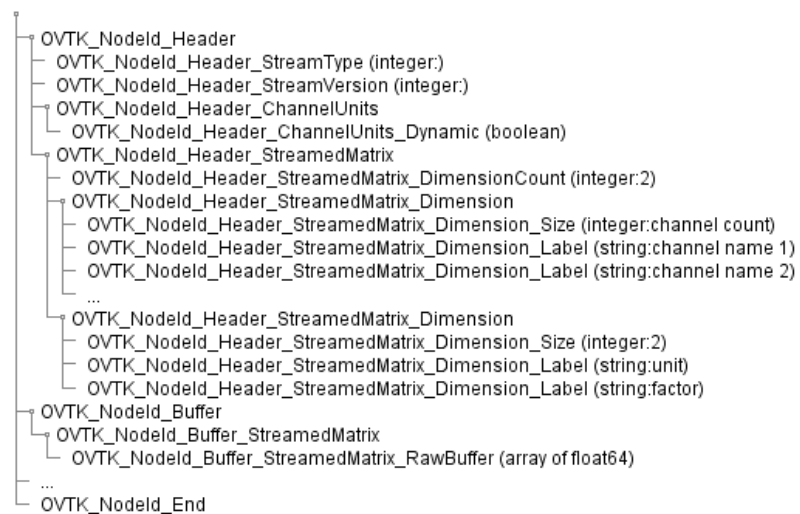
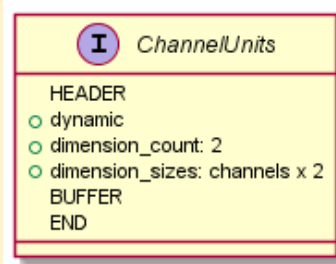
EBML nodes tree (see §2.3 Structures Identification for references about node tree identifiers).

#### Notes

Inherited properties are left out in the stream description view.

CertiViBE - v1.0	CERT-01 MSD-01	Page 72 / 86
Modules Detailed Design Specifications		

All rights reserved. Passing on or copying of this document, use and communication of its contents not permitted without written authorization



Stream intended to carry channel measurement unit information. The matrix contains on each row unit and scaling factor for a given channel.

## Version: 1

dynamic: false if channel units are static, true if it can changes over time (more than one buffer received).

dimension\_count: Channels and their unit properties.

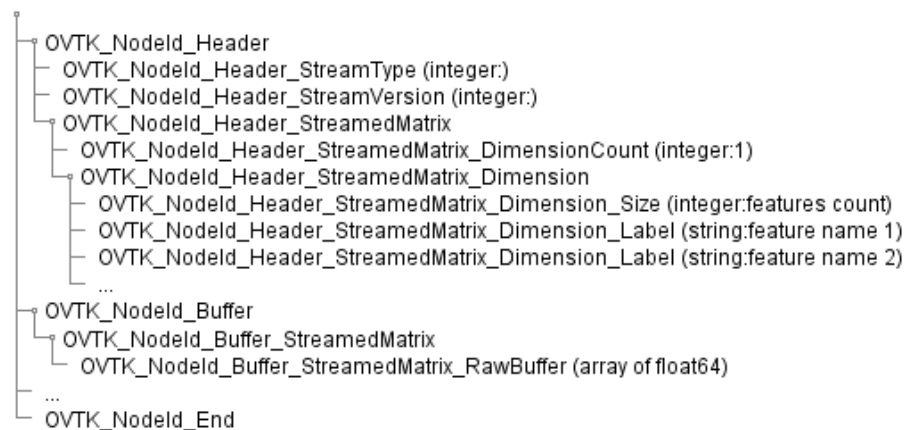
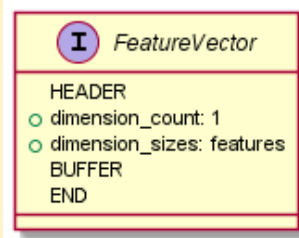
dimension\_sizes: dim1 = channel count / dim2 = 2 (unit and scaling factor).

EBML nodes tree (see §2.3 Structures Identification for references about node tree identifiers).

## Notes

Inherited properties are left out in the stream description view.

See §2.3 Structures Identification for references about measurement units identifiers.

*Primary Presentation**Element Catalog / Description*

Stream intended to carry a feature vector for classification purpose. The matrix contains one row containing the list of numerical features describing an object.

**Stream description (left view)**

Version: 1

dimension\_count: 1.

dimension\_sizes: dim1 = the features.

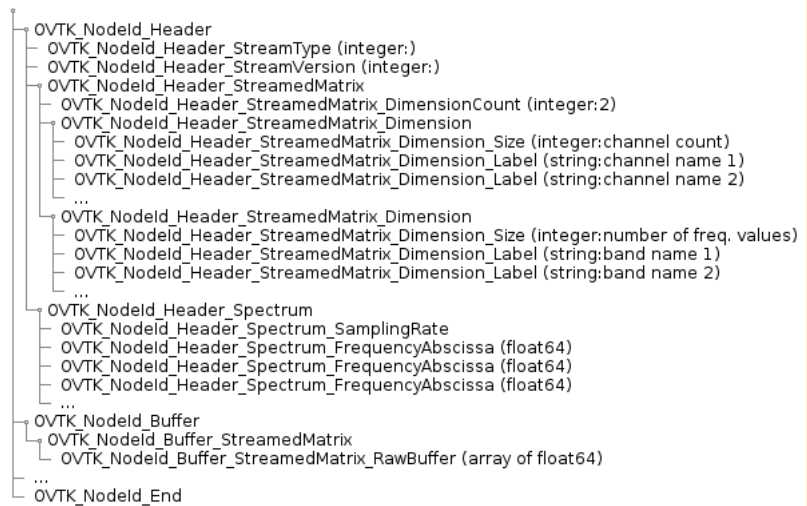
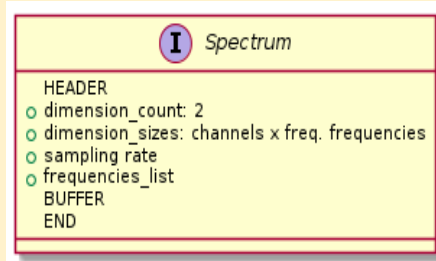
**Stream structure (right view)**

EBML nodes tree (see §2.3 Structures Identification for references about node tree identifiers).

*Notes*

Inherited properties are left out in the stream description view.

## Primary Presentation



## Element Catalog / Description

Stream intended to carry EEG signal spectral analysis results. The matrix contains on each row the list of spectral powers of each frequency band for a given channel.

### Stream description (left view)

Version: 2

dimension\_count: Channels and frequency bands.

dimension\_sizes: dim1 = channel count, dim2 = number of frequency bands.

sampling rate: the signal sampling rate

frequencies\_list: List of frequencies. Each value represents the center of a frequency band.

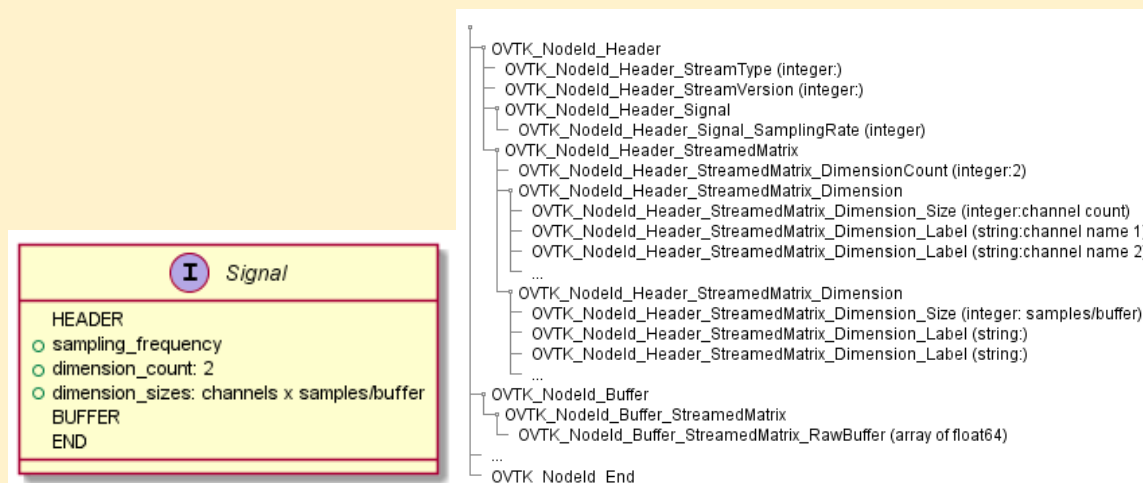
### Stream structure (right view)

EBML nodes tree (see §2.3 Structures Identification for references about node tree identifiers).

## Notes

Inherited properties are left out in the stream description view.

## Primary Presentation



## Element Catalog / Description

Stream intended to carry the EEG signals on one or multiple channels. The matrix contains on each row the sample data of a given channel. Rows grows top down with index notation from 1 to n while columns, representing time or sample count, grows from left to right. One matrix buffer is one signal chunk.

### Stream description (left view)

Version: 2

sampling\_frequency: sampling frequency of the signal.

dimension\_count: Channels and samples.

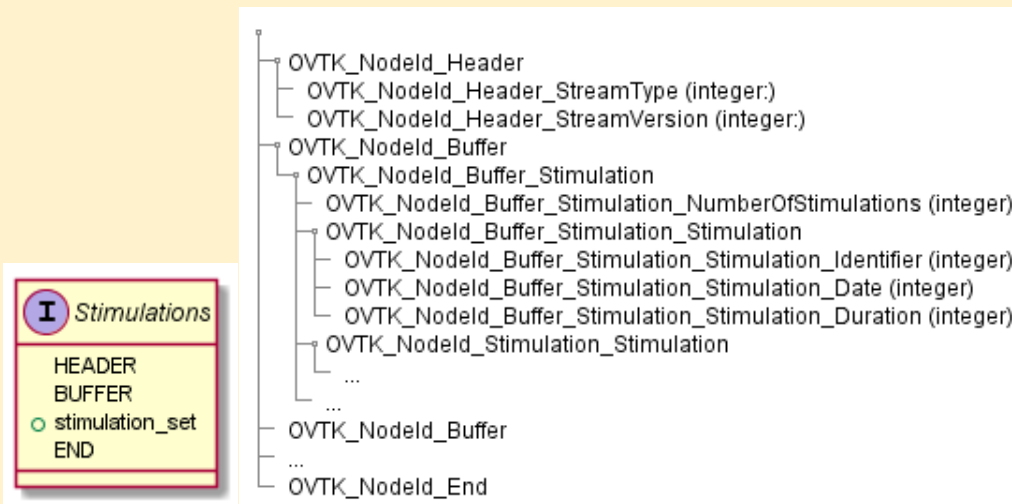
dimension\_sizes: dim1 = channel count, dim2 = sample count per buffer.

### Stream structure (right view)

EBML nodes tree (see §2.3 Structures Identification for references about node tree identifiers).

## Notes

Inherited properties are left out in the stream description view.

*Primary Presentation**Element Catalog / Description*

Stream intended to carry stimulations information. A typical used is to select part of a signal based on the stimulation type and occurrence time.

**Stream description (left view)**

Version: 3

stimulation\_set: A stimulation set contains:

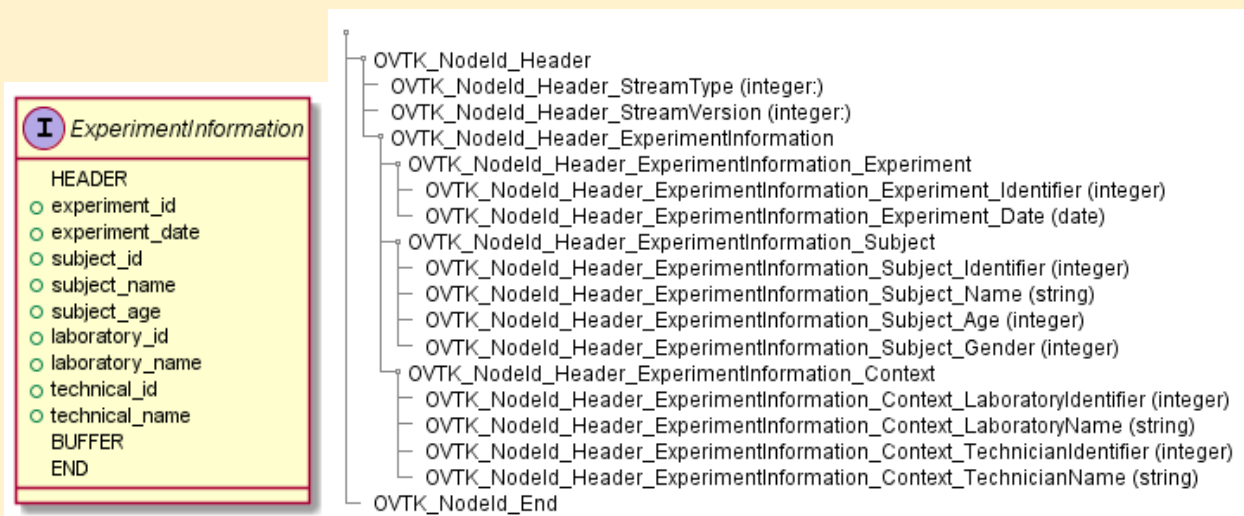
- The stimulation date as 64 bits unsigned integer with 32:32 fixed point precision.
- The stimulation identifier (see §2.3 Structures Identification for references about stimulations identifiers).
- The stimulation duration as 64 bits unsigned integer with 32:32 fixed point precision.

**Stream structure (right view)**

EBML nodes tree (see §2.3 Structures Identification for references about node tree identifiers).

*Notes*

Inherited properties are left out in the stream description view.

*Primary Presentation**Element Catalog / Description*

Stream intended to carry information on the experiment being conducted.

**Stream description (left view)**

Version: 1

Experiment description: identifier and date.

Subject description: identifier, name and age.

Context: laboratory identifier and name, technician identifier and name.

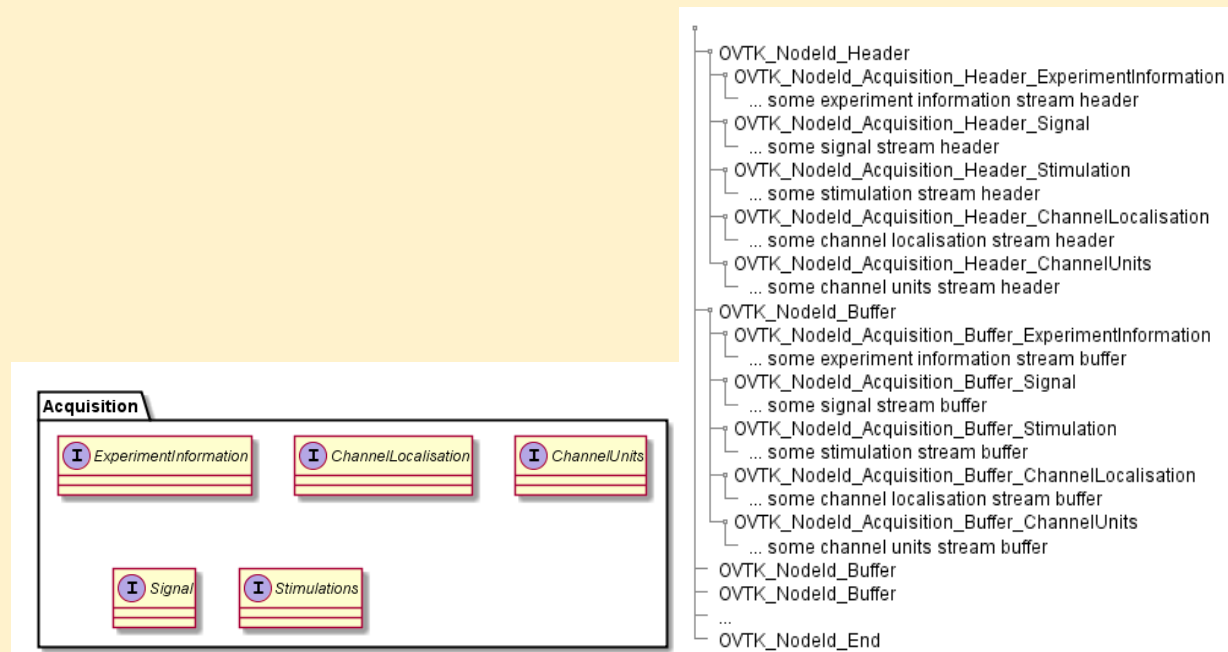
**Stream structure (right view)**

EBML nodes tree (see §2.3 Structures Identification for references about node tree identifiers).

*Notes*

Inherited properties are left out in the stream description view.

## Primary Presentation



## Element Catalog / Description

Multiplexed stream encapsulating five types of streams. It is intended to be used by an acquisition module to convert raw data to data usable in the processing pipeline.

### Stream description (left view)

Version: 3

This is a multiplexed stream encapsulating five streams.

### Stream structure (right view)

EBML nodes tree (see §2.3 Structures Identification for references about node tree identifiers).

## 10.2 Plugins Components List

The following table presents the list of plugins with their main components.

Plugin name	Type	Plugin Object Class Name / Identifier	Reference Doc
openvibe-plugins-classification	Algorithm	CAlgorithmClassifierShrinkageLDA OVP_ClassId_Algorithm_ClassifierShrinkageLDA	<a href="#">link</a>
openvibe-plugins-classification	Algorithm	CAlgorithmClassifierConditionedCovariance OVP_ClassId_Algorithm_ConditionedCovariance	<a href="#">link</a>
openvibe-plugins-classification	Box Algorithm	CBoxAlgorithmClassifierProcessor OVP_ClassId_BoxAlgorithm_ClassifierProcessor	<a href="#">link</a>
openvibe-plugins-classification	Box Algorithm	CBoxAlgorithmClassifierTrainer OVP_ClassId_BoxAlgorithm_ClassifierTrainer	<a href="#">link</a>
openvibe-plugins-classification	Box Algorithm	CBoxAlgorithmVotingClassifier OVP_ClassId_BoxAlgorithm_VotingClassifier	<a href="#">link</a>
openvibe-plugins-file-io	Algorithm	CAlgorithmOVMatrixFileReader OVP_ClassId_Algorithm_OVMatrixFileReader	
openvibe-plugins-file-io	Algorithm	CAlgorithmOVMatrixFileWriter OVP_ClassId_Algorithm_OVMatrixFileWriter	
openvibe-plugins-file-io	Algorithm	CAlgorithmXMLScenarioExporter OVP_ClassId_Algorithm_XMLScenarioExporter	
openvibe-plugins-file-io	Algorithm	CAlgorithmXMLScenarioImporter OVP_ClassId_Algorithm_XMLScenarioImporter	<a href="#">link</a>
openvibe-plugins-file-io	Box Algorithm	CBoxAlgorithmGenericStreamReader OVP_ClassId_BoxAlgorithm_GenericStreamReader	<a href="#">link</a>
openvibe-plugins-file-io	Box Algorithm	CBoxAlgorithmGenericStreamWriter OVP_ClassId_BoxAlgorithm_GenericStreamWriter	<a href="#">link</a>
openvibe-plugins-file-io	Box Algorithm	CBoxAlgorithmOVCSVFileReader OVP_ClassId_BoxAlgorithm_OVCSVFileReader	<a href="#">link</a>
openvibe-plugins-file-io	Box Algorithm	CBoxAlgorithmOVCSVFileWriter OVP_ClassId_BoxAlgorithm_OVCSVFileWriter	<a href="#">link</a>
openvibe-plugins-file-io	Box Algorithm	CBoxAlgorithmElectrodeLocalisationFileReader OVP_ClassId_BoxAlgorithm_ElectrodeLocalisationFileReader	<a href="#">link</a>

openvibe-plugins-file-samples	Box Algorith m	CBoxAlgorithmClockStimulator OVP_ClassId_BoxAlgorithm_ClockStimulator	<a href="#">link</a>
openvibe-plugins-file-samples	Box Algorith m	CIdentity OVP_ClassId_Identity	<a href="#">link</a>
openvibe-plugins-file-samples	Box Algorith m	CTimeSignalGenerator OVP_ClassId_TimeSignalGenerator	<a href="#">link</a>
openvibe-plugins-signal-processing	Algorith m	CAlgorithmStimulationBasedEpoching OVP_ClassId_Algorithm_StimulationBasedEpoching	<a href="#">link</a>
openvibe-plugins-signal-processing	Box Algorith m	CEpochAverage OVP_ClassId_BoxAlgorithm_EpochAverage	<a href="#">link</a>
openvibe-plugins-signal-processing	Box Algorith m	CBoxAlgorithmSpectrumAverage OVP_ClassId_BoxAlgorithm_SpectrumAverage	<a href="#">link</a>
openvibe-plugins-signal-processing	Box Algorith m	CSignalAverage OVP_ClassId_SignalAverage	<a href="#">link</a>
openvibe-plugins-signal-processing	Box Algorith m	CBoxAlgorithmCommonAverageReference OVP_ClassId_BoxAlgorithm_CommonAverageReference	<a href="#">link</a>
openvibe-plugins-signal-processing	Box Algorith m	CBoxAlgorithmReferenceChannel OVP_ClassId_BoxAlgorithm_ReferenceChannel	<a href="#">link</a>
openvibe-plugins-signal-processing	Box Algorith m	CBoxAlgorithmChannelRename OVP_ClassId_BoxAlgorithm_ChannelRename	<a href="#">link</a>
openvibe-plugins-signal-processing	Box Algorith m	CBoxAlgorithmChannelSelector OVP_ClassId_BoxAlgorithm_ChannelSelector	<a href="#">link</a>
openvibe-plugins-signal-processing	Box Algorith m	CBoxAlgorithmCrop OVP_ClassId_BoxAlgorithm_Crop	<a href="#">link</a>
openvibe-plugins-signal-processing	Box Algorith m	CSimpleDSP OVP_ClassId_SimpleDSP	<a href="#">link</a>
openvibe-plugins-signal-processing	Box Algorith m	CBoxAlgorithmSignalDecimation OVP_ClassId_BoxAlgorithm_SignalDecimation	<a href="#">link</a>
openvibe-plugins-signal-processing	Box Algorith m	CBoxAlgorithmSpatialFilter OVP_ClassId_BoxAlgorithm_SpatialFilter	<a href="#">link</a>
openvibe-plugins-signal-processing	Box Algorith m	CBoxAlgorithmXDAWNSpatialFilterTrainer OVP_ClassId_BoxAlgorithm_XDAWNSpatialFilterTrainer	<a href="#">link</a>

openvibe-plugins-signal-processing	Box Algorithm	CBoxAlgorithmStimulationBasedEpoching OVP_ClassId_BoxAlgorithm_StimulationBasedEpoching	<a href="#">link</a>
openvibe-plugins-signal-processing	Box Algorithm	CTimeBasedEpoching OVP_ClassId_TimeBasedEpoching	<a href="#">link</a>
openvibe-plugins-signal-processing	Box Algorithm	CBoxAlgorithmSpectrumAverage OVP_ClassId_BoxAlgorithm_SpectrumAverage	<a href="#">link</a>
openvibe-plugins-signal-processing	Box Algorithm	CBoxAlgorithmFrequencyBandSelector OVP_ClassId_BoxAlgorithm_FrequencyBandSelector	<a href="#">link</a>
openvibe-plugins-signal-processing	Box Algorithm	CBoxAlgorithmRegularizedCSPTrainer OVP_ClassId_BoxAlgorithm_RegularizedCSPTrainer	<a href="#">link</a>
openvibe-plugins-signal-processing	Box Algorithm	CBoxAlgorithmSignalResampling OVP_ClassId_BoxAlgorithm_SignalResampling	<a href="#">link</a>
openvibe-plugins-signal-processing	Box Algorithm	CBoxAlgorithmSpectralAnalysis OVP_ClassId_SpectralAnalysis	<a href="#">link</a>
openvibe-plugins-signal-processing	Box Algorithm	CBoxAlgorithm_TemporalFilter OVP_ClassId_BoxAlgorithm_TemporalFilter	<a href="#">link</a>
openvibe-plugins-signal-processing	Box Algorithm	CBoxAlgorithmWindowing OVP_ClassId_Windowing	<a href="#">link</a>
openvibe-plugins-signal-processing	Box Algorithm	CBoxAlgorithm_ZeroCrossingDetector OVP_ClassId_BoxAlgorithm_ZeroCrossingDetector	<a href="#">link</a>
openvibe-plugins-stimulation	Box Algorithm	CBoxAlgorithmStimulationMultiplexer OVP_ClassId_BoxAlgorithm_StimulationMultiplexer	<a href="#">link</a>
openvibe-plugins-stimulation	Box Algorithm	CBoxAlgorithmPlayerController OVP_ClassId_BoxAlgorithm_PlayerController	<a href="#">link</a>
openvibe-plugins-stimulation	Box Algorithm	CBoxAlgorithmTimeout OVP_ClassId_BoxAlgorithm_Timeout	<a href="#">link</a>
openvibe-plugins-stimulation	Box Algorithm	CBoxAlgorithm_StimulationVoter OVP_ClassId_BoxAlgorithm_StimulationVoter	<a href="#">link</a>
openvibe-plugins-stream-codecs	Algorithm	All codec algorithms classes.	\$2.2.4 Stream Encoding /Decodin g

openvibe-plugins-streaming	Box Algorithm	CBoxAlgorithmSignalMerger OVP_ClassId_BoxAlgorithm_SignalMerger	<a href="#">link</a>
openvibe-plugins-streaming	Box Algorithm	CBoxAlgorithmStreamedMatrixMultiplexer OVP_ClassId_BoxAlgorithm_StreamedMatrixMultiplexer	<a href="#">link</a>
openvibe-plugins-tools	Box Algorithm	CBoxAlgorithmEBMLStreamSpy OVP_ClassId_BoxAlgorithm_EBMLStreamSpy	<a href="#">link</a>
openvibe-plugins-tools	Box Algorithm	CBoxAlgorithmMatrixValidityChecker OVP_ClassId_BoxAlgorithm_MatrixValidityChecker	<a href="#">link</a>
openvibe-plugins-tools	Box Algorithm	CBoxAlgorithmStimulationListener OVP_ClassId_BoxAlgorithm_StimulationListener	<a href="#">link</a>
openvibe-plugins-feature-extraction	Box Algorithm	CBoxAlgorithmFeatureAggregator OVP_ClassId_BoxAlgorithm_FeatureAggregator	<a href="#">link</a>

### 10.3 Standard Configuration Tokens

Token Name	Token Description
<code>\${Path_Root}</code>	Root installation directory of OpenViBE
<code>\${Path_Bin}</code>	Binaries directory
<code>\${Path_Lib}</code>	Libraries directory
<code>\${Path_Data}</code>	Data directory. (default is <code>\${Path_Root}/share/openvibe</code> )
<code>\${Path_Samples}</code>	Scenarios directory. (default is <code>\${Path_Data}/scenarios</code> )
<code>\${Path_UserData}</code>	Writable location for data (default is <code>%APPDATA%/openvibe</code> on Windows and <code>\$HOME/.config/openvibe</code> on Linux)
<code>\${Path_Log}</code>	Log directory (default is <code>\${Path_UserData}/log</code> )
<code>\${Path_Tmp}</code>	Temporary directory (default is <code>\${Path_UserData}/tmp</code> )
<code>\${Kernel_PluginsPatternLinux}</code>	Linux openvibe plugin regex pattern ( <code>libopenvibe-plugins-*.so</code> )
<code>\${Kernel_PluginsPatternWindows}</code>	Windows openvibe plugin regex pattern ( <code>openvibe-plugins-*.dll</code> )
<code>\${Kernel_PluginsPatternMacOS}</code>	Mac OS X openvibe plugin regex pattern ( <code>libopenvibe-plugins-*.dylib</code> )
<code>\${Kernel_Plugins}</code>	Full plugins path regex pattern ( <code>\${Path_Lib}/\${Kernel_PluginsPattern\${OperatingSystem}}</code> )
<code>\${Kernel_MainLogLevel}</code>	Log level threshold below which all messages are ignored.
<code>\${Kernel_ConsoleLogLevel}</code>	Specific log level threshold for console output (must be greater than main one)
<code>\${Kernel_ConsoleLogWithHexa}</code>	Add hexadecimal (0x...) value after time log for console output when set to <code>True</code> ( <code>False</code> by default)
<code>\${Kernel_ConsoleLogTimeInSeconds}</code>	Time log in <u>seconds</u> for console output when set to <code>True</code> ( <code>True</code> by default)

CertiViBE - v1.0 Modules Detailed Design Specifications	CERT-01 MSD-01	Page 84 / 86
--	----------------	--------------

<code>\${Kernel_ConsoleLogTimePrecision}</code>	Time log precision for console output (3 by default)
<code>\${Kernel_FileLogLevel}</code>	Specific log level threshold for file output (must be greater than main one).
<code>\${Kernel_FileLogWithHexa}</code>	Add hexadecimal (0x...) value after time log for file output when set to <code>True</code> ( <code>False</code> by default).
<code>\${Kernel_FileLogTimeInSecond}</code>	Time log in <u>seconds</u> for file output when set to <code>True</code> ( <code>True</code> by default).
<code>\${Kernel_FileLogTimePrecision}</code>	Time log precision for file output (3 by default).
<code>\${Kernel_PlayerFrequency}</code>	Player frequency in Hz.
<code>\${Kernel_DelayedConfiguration}</code>	File loaded later when a scenario execution starts ( <code>\${Path_Data}/kernel/openvibe-delayed.conf</code> )
<code>\${Kernel_AbortPlayerWhenBoxNeedsUpdate}</code>	When <code>True</code> , do not start the processing if at least one box has to be updated ( <code>False</code> by default).
<code>\${Kernel_AbortScenarioImportWhenBoxNeedsUpdate}</code>	When <code>True</code> , do not import the scenario if at least one box has to be updated ( <code>False</code> by default).
<code>\${Kernel_AbortScenarioImportOnUnknownSetting}</code>	When <code>True</code> , do not import the scenario if at least one setting is unknown ( <code>False</code> by default).
<code>\${Kernel_AbortPlayerWhenBoxIsDisabled}</code>	When <code>True</code> , abort player if at least one box is disabled ( <code>False</code> by default).
<code>\${Kernel_Metabox}</code>	Directory(ies) from which the kernel can load metaboxes files ( <code>\${Path_Data}/metaboxes/</code> ; <code>\${Path_UserData}/metaboxes/</code> by default)
<code>\$score{random}</code>	Random number
<code>\$score{index}</code>	Incremental index
<code>\$score{time}</code>	Current time
<code>\$score{date}</code>	Current data

\$core{real-time}	Time since configuration manager creation
\$core{process-id}	Current process id