Date: May 2016

OBJECT MANAGEMENT GROUP®

Application Instrumentation

V1.0

OMG Document Number: formal/2016-05-01

Standard document URL: http://www.omg.org/spec/APP-INST/1.0

Normative Machine Consumable File(s):

http://www.omg.org/spec/APP-INST/20151201/omg_appinst_pim.xmi

http://www.omg.org/spec/APP-INST/20151201/omg_appinst_c_src.zip

http://www.omg.org/spec/APP-INST/20151201/omg_appinst_java_src.zip

---

OMG's Issue Reporting

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the Issue Reporting Form listed on the main web page http://www.omg.org, under Documents, Report a Bug/Issue.

Table of Contents

Preface

OMG

Founded in 1989, the Object Management Group, Inc. (OMG) is an open membership, not-for-profit computer industry standards consortium that produces and maintains computer industry specifications for interoperable, portable, and reusable enterprise applications in distributed, heterogeneous environments. Membership includes Information Technology vendors, end users, government agencies, and academia.

OMG member companies write, adopt, and maintain its specifications following a mature, open process. OMG's specifications implement the Model Driven Architecture® (MDA®), maximizing ROI through a full-lifecycle approach to enterprise integration that covers multiple operating systems, programming languages, middleware and networking infrastructures, and software development environments. OMG's specifications include: UML® (Unified Modeling Language®); CORBA® (Common Object Request Broker Architecture); CWM™ (Common Warehouse Metamodel™); and industry-specific standards for dozens of vertical markets.

More information on the OMG is available at http://www.omg.org/.

OMG Specifications

As noted, OMG specifications address middleware, modeling and vertical domain frameworks. All OMG Specifications are available from the OMG website at:

http://www.omg.org/spec

Specifications are organized by the following categories:

Business Modeling Specifications

Middleware Specifications

- **CORBA/IIOP**
- **Data Distribution Services**
- **Specialized CORBA**

IDL/Language Mapping Specifications

Modeling and Metadata Specifications

- **UML, MOF, CWM, XMI**
- **UML Profile**

Modernization Specifications

Platform Independent Model (PIM), Platform Specific Model (PSM), Interface Specifications

- **CORBAServices**
- **CORBAFacilities**

OMG Domain Specifications

CORBA Embedded Intelligence Specifications

CORBA Security Specifications

Signal and Image Processing Specifications

All of OMG's formal specifications may be downloaded without charge from our website. (Products implementing OMG specifications are available from individual suppliers.) Copies of specifications, available in PostScript and PDF format, may be obtained from the Specifications Catalog cited above or by contacting the Object Management Group, Inc. at:

OMG Headquarters
109 Highland Avenue
Needham, MA 02494
USA
Tel: +1-781-444-0404
Fax: +1-781-444-0320
Email: *pubs@omg.org*

Certain OMG specifications are also available as ISO standards. Please consult http://www.iso.org

Typographical Conventions

The type styles shown below are used in this document to distinguish programming statements from ordinary English. However, these conventions are not used in tables or headings where no distinction is necessary.

Times/Times New Roman - 10 pt.: Standard body text, table text, bullets

Helvetica/Arial – 9 or **10 pt. Bold:** OMG Interface Definition Language (OMG IDL) and syntax elements.

`Courier new/Courier - 10 pt. Bold`: Programming Languages

Helvetica/Arial – 10 pt.: Exceptions

Issues

The reader is encouraged to report any technical or editing issues/problems with this by completing the Issue Reporting Form listed on the main web page http://www.omg.org, under documents, Report a Bug/Issue.

1 Scope

This specification defines a general API for minimally intrusive instrumentation of applications.

The API provides:

- A mechanism to define composite data types from the set of supported primitive types.
- A mechanism to create typed data sources and to collect application data through them.
- A mechanism to apply processing filters to collected data and to control its distribution to remote consumers.
- A mechanism to remotely control the instrumentation during the application's run-time.

2 Conformance

2.1 Compliance Levels

There are two conformance levels for implementations of this specification:

- **Application Instrumentation API conformance** – a complete implementation of the API described in 7.4 Application Instrumentation (API) shall be provided through either (or both) of the PSM described in 8.1 Application Instrumentation API PSMs.
- **Instrumentation Domain PSM conformance** – the implementation shall be completely compatible with the PSM described in 8.2 Instrumentation Domain PSMs.

3 Normative References

The following normative documents are referenced by this document:

- Unified Modeling Language (UML) [http://www.omg.org/spec/UML/2.4.1/]
- Data Distribution Service for Real-time Systems (DDS) [http://www.omg.org/spec/DDS/1.2/]
- C Programming Language [http://www.open-std.org/jtc1/sc22/wg14/www/standards]
- Java Programming Language [http://docs.oracle.com/javase/specs/]
- Key words for use in RFCs to Indicate Requirement Levels (RFC2119) [http://www.ietf.org/rfc/rfc2119.txt]
- IEEE Standard for Floating-Point Arithmetic (IEEE 754-2008) [http://ieeexplore.ieee.org/xpl/articleDetails.jsp?reload=true&arnumber=4610935]
- Date and Time format ISO 8601 [http://www.iso.org/iso/home/standards/iso8601.htm]

4 Terms and Definitions

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

Data Distribution Service (DDS)

An OMG distributed data communications specification that allows Quality of Service policies to be specified for data timeliness and reliability. It is independent of implementation languages.

5 Symbols

This specification does not define any symbols or abbreviations.

6 Additional Information

6.1 Overview of this Specification

This specification defines a general API for application instrumentation. The goal is to perform typed observations of internal state of application, including values of application variables, from separate monitoring applications, possibly distributed over a network environment.

This kind of instrumentation will enable accurate and flexible monitoring of custom application state information typically hidden within an application's execution environment and unavailable to external observers.

The scope of Application Instrumentation API intersects with that of other existing monitoring solutions, such as logging and system monitoring frameworks. In contrast to those solutions, this API addresses the need for precise access to custom application data and continuously changing variables as required by tasks such as application-logic debugging, testing and on-line monitoring of distributed applications. Other solutions typically fail to:

1. Support machine consumption of information by providing a sufficiently expressive type system that can accurately represent the data of interest.
2. Grant access to the specific information of interest, which may be available only within the application's execution environment.
3. Limit the degradation of performance in instrumented applications to support instrumentation of production-level system.

For example, the data-model adopted by logging tools cannot directly support extraction of numerical data and requires serialization and deserialization to and from the string format. This increases the cost of the instrumentation and it may reduce the accuracy of observed data.

Solutions for system-level monitoring typically do not have access to internal application state. They operate in separate processes and they can only observe external events of an application, often by instrumenting system calls and intercepting context-switches.

Even when access to internal state is available other limitations must be considered. For example, using a debugger to manually control the application and its memory greatly impacts an application's performance and it is usually not recommended for verifying a distributed application, since timing of the operations is often critical.

This specification defines a platform-independent instrumentation API and a distributed instrumentation infrastructure. Application developers can use the API to define the state information produced by an application, instrument the application code and then generate data from running applications. The API allows the configuration of custom data processing, performed before data is distributed outside of the application. Instrumentation may also be dynamically controlled to enable or disable specific parts of the instrumentation logic.

Data instrumented with the API will be distributed to remote consumers through the external distributed instrumentation architecture. The API's platform-independent model specifies how the processing phase may affect distribution of data, but leaves the details of how data is exchanged between distributed processes to each platform-specific models of the instrumentation architecture.

Multiple platform-specific API implementations may be created for different communication technologies. Each platform-specific API implementation shall specify how data generated using the instrumentation API is managed in its target distributed environment and how remote consumers may access it using the selected technology.

The platform-independent API describes the operations that may be invoked by client application to the instrumentation infrastructure. Client applications can use these operations to modify the configuration of available instrumentation entities dynamically during the execution of the instrumented applications.

This specification includes:

- A Platform Independent Model (PIM) for an Application Instrumentation API to instantiate and manage instrumentation entities from an application.
- A PIM for a distributed instrumentation infrastructure, which grants access to data collected by instrumented applications and remote configuration of their instrumentation.
- A C and a Java mapping for the Application Instrumentation API.
- A mapping of the distributed instrumentation infrastructure to the OMG Data Distribution Service platform.

6.2 Design Rationale

The API is designed with a particular focus on enabling efficient extraction of internal state data from running applications. The objective is to minimize the run-time overhead caused by the execution of instrumentation operations during the "steady state" of an instrumented application while providing a sufficiently rich set of instrumentation tools capable of addressing a wide variety of use cases.

A comprehensive platform-independent data-type system enables the accurate description types for the data of interest, while guaranteeing consistent access from heterogeneous execution environments. Each platform-specific model for the instrumentation API will define a mapping between the platform-independent types and data-types supported by a specific programming language and/or platform.

The model used to collect data from instrumented applications decouples data generation from data processing and the distribution to external consumers. Data is collected and processed in a separate context from the application. This context may execute asynchronously from the application code, thus reducing the impact of instrumentation on the application's critical path for example by operating in a lower priority, independent, thread.

Because of this focus on minimizing the impact on the application, the operations exposed by the interfaces of the API are not multi-thread safe unless explicitly stated in the description of a class or an operation.

Most entities created by the instrumentation API can be dynamically enabled or disabled during the execution of an instrumented application. The ability of maintaining all or part of the entities in a disabled state allows the instrumentation infrastructure to be used in an active system possibly deployed to production. The instrumentation will "lie dormant" within the execution environment until each specific entity is enabled, either by the instrumented application itself or an external application using the remote configuration interface.

6.3 Statement of Proof of Concept

The submitters have already implemented almost all elements in the specification. A prototype of the software with the specified API's and behaviors has been made available in the past months to key stakeholders as part of the US Navy SBIR N092-121 titled "Minimally Intrusive Real-time Software Instrumentation Technologies".

6.4 Acknowledgements

The following companies submitted this specification:

- Real-Time Innovations, Inc.

The following companies supported this specification:

- SimVentions

7 Platform Independent Model (PIM)

7.1 Introduction

The purpose of this clause is to provide an operational overview of the Application Instrumentation API PIM and the classifiers that it comprises.

7.2 Format and Conventions

In addition to the UML diagrams, all the classes that constitute the API are documented using tables. The format used to document these classes is shown below:

<table>
<tr><td colspan="3"><class name> [<class parameter>]
[<super classes list>]</td></tr>
<tr><td colspan="3">Attributes</td></tr>
<tr><td colspan="2"><attribute name></td><td><attribute type></td></tr>
<tr><td colspan="2">...</td><td>...</td></tr>
<tr><td colspan="3">Operations</td></tr>
<tr><td rowspan="3"><operation name></td><td></td><td><return type></td></tr>
<tr><td><parameter></td><td><parameter type></td></tr>
<tr><td>...</td><td>...</td></tr>
<tr><td>...</td><td></td><td>...</td></tr>
</table>

The operation <parameter> can contain the modifier “in,” “out,” or “inout” ahead of the parameter name. If this modifier is omitted, it is implied that the parameter is an “in” parameter.

In some cases, the operation parameters or return value(s) are a collection with elements of a given <type>. This is indicated with the notation “<type> [ ].” This notation does not imply that it will be implemented as an array. The actual implementation is defined by the PSM: it may end up being mapped to a sequence, a list, or other kind of collection.

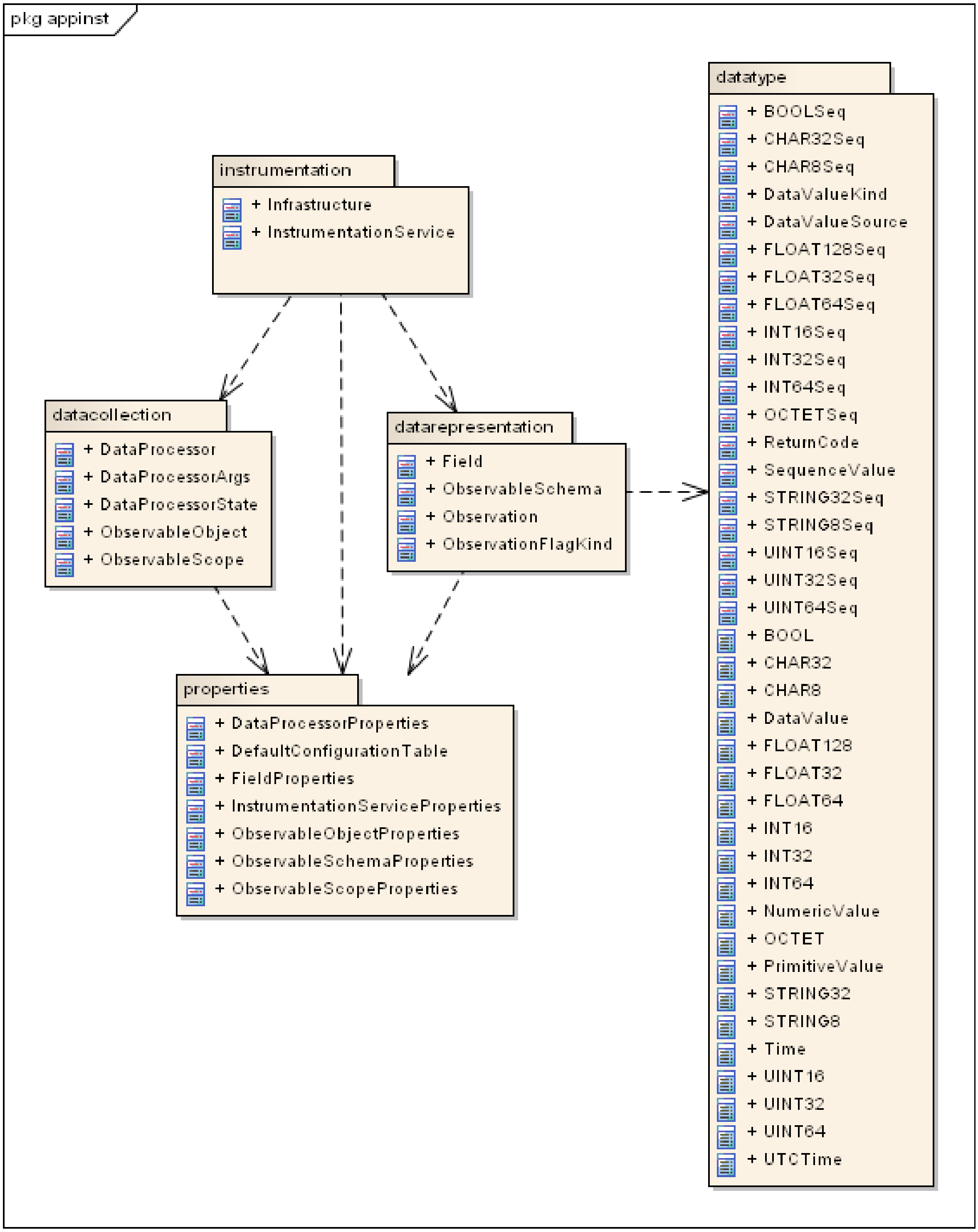

Figure 1 – Application Instrumentation PIM

7.3 PIM Overview

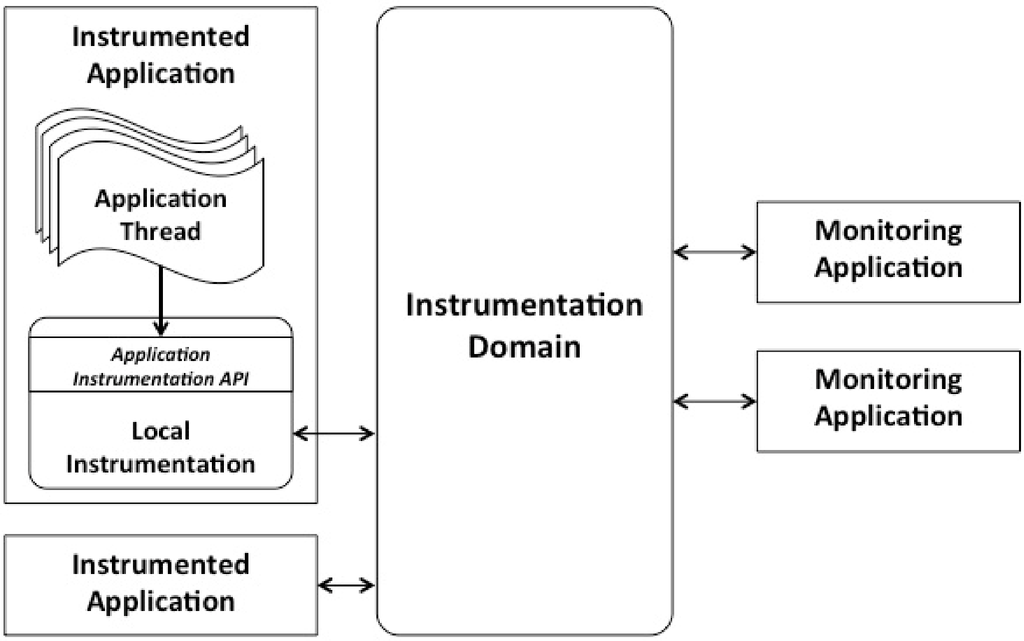

Figure 2 – Distributed instrumentation infrastructure

The platform-independent model described by this specification defines an infrastructure for the collection and consumption of internal state information from one or more application processes.

Instrumented applications use a platform-independent API to describe and generate data during their execution. Local instrumentation entities within the application's process collect data, process it, and then distribute it to an external *Instrumentation Domain*.

The *Instrumentation Domain* abstracts the communication infrastructure connecting instrumented applications and monitoring applications interested in accessing their state information. The instrumentation entities created by each application do not require prior knowledge of the observers/consumers of the data they will produce. Samples of data to be distributed outside of instrumented applications will be forwarded to the *Instrumentation Domain,* which in turn will deliver them to remote endpoints. Depending on its implementation platform, the *Instrumentation Domain* may provide decoupling in time and space between producers and consumers. Data can be stored by the *Instrumentation Domain* and made available to monitoring applications after its publication by the local instrumentation.

Local instrumentation entities may be dynamically configured during the application's execution by using the Application Instrumentation API or by issuing configuration commands from remote applications through the *Instrumentation Domain.*

7.3.1 Application Instrumentation API

The Application Instrumentation API defines an abstract data-model and a set of platform-independent entities, which can be used to instrument and collect data from a running application.

As shown in Figure 3, an *InstrumentationService* manages the entire instrumentation infrastructure local to a single application. Application state information is described by custom data-types called *ObservableSchema*. One or more *ObservableScope*s define sources of data, which will be used by the application, and control how data is processed.

Data is produced using an *ObservableObject*. An *ObservableObject* is an instantiation of an *ObservableSchema* and allows application to take snapshots of its attributes, called *Observation*s. *ObservableObject*s are logically grouped using *ObservableGroup*s.

The *ObservableScope* containing an *ObservableObject* is responsible for collecting and distributing *Observation*s to the *Instrumentation Domain*. An optional processing phase may be carried out by a customizable *DataProcessor*, which can alter the content of an *Observation* and prevent it from being published outside the instrumented application's context.

An *Observation* carries all the necessary information to identify its source and (local) time of generation. Its contents can be accessed without prior knowledge of its structure by using the information provided by it *ObservableSchema*.

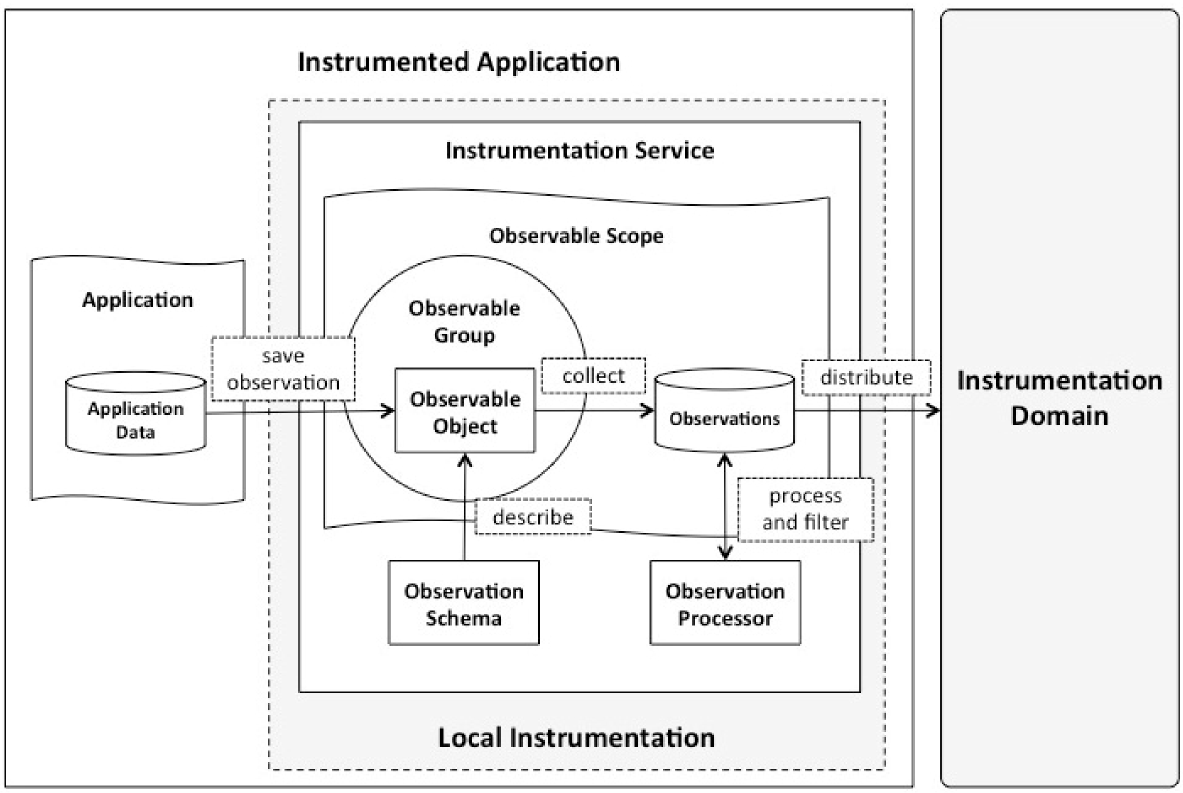

Figure 3 – Components of the local instrumentation

7.3.1.1 Data collection

Applications use the interface provided by an *ObservableObject* to pass data to the instrumentation. Figure 4 shows an example of the interactions required to extract data from an instrumented application and distribute it to the *Instrumentation Domain*.

Values from application variables can be stored inside the attributes of an *ObservableObject*. Internal memory shall store these values between multiple accesses to the *ObservableObject* so that the application may build its state incrementally from separate points in the application code if necessary.

When an *ObservableObject* contains the expected information, an application can generate a snapshot of its current state using its *save_observation* operation. This shall make a copy of the current values stored by the *ObservableObject* into a new *Observation* instance. The *ObservableObject* shall then notify its enclosing *ObservableScope* of the new *Observation*, which shall collect, process, and distribute as determined by the *ObservationProcessor.*

An *ObservableScope* operates in a separate context, independently of the application, and it periodically extracts all new *Observation*s from its *ObservableObject*s. Once collected, the new *Observation*s shall be passed (if so configured) to the *process_observations* method of a *DataProcessor.* The *Observations* shall then be handed over to the *Instrumentation Domain* to be distributed outside of the application and finally returned to the *ObservableObject*, which may reuse them to store future snapshots.

This final phase can be altered by a *DataProcessor,* which can prevent an *Observation* from both being distributed and being recycled by the original *ObservableObject*.

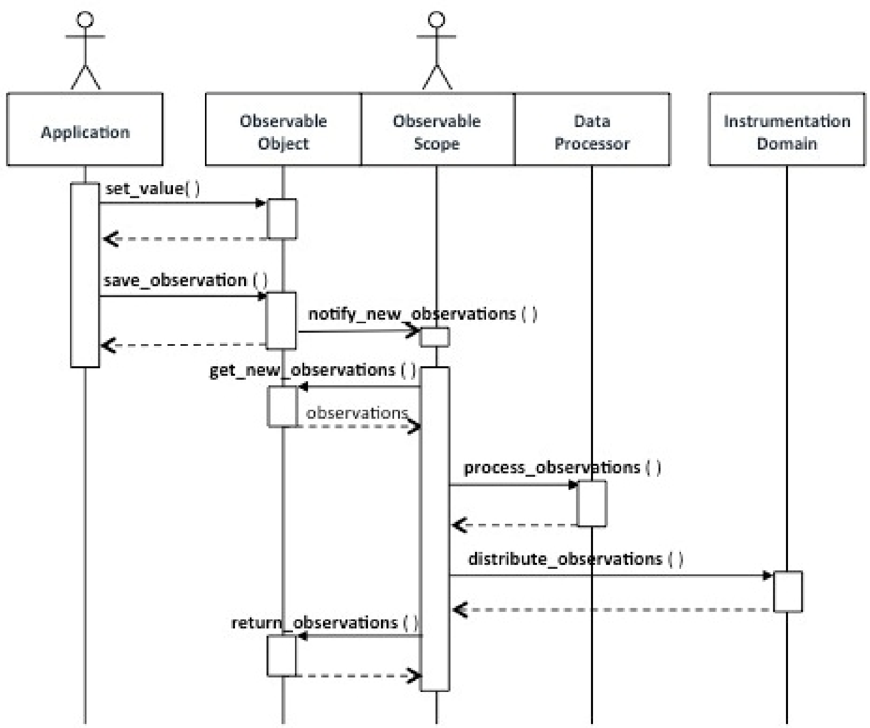

Figure 4 – Example of data collection

7.3.1.2 Data processing

The *Application Instrumentation API* processes all *Observation*s within the context of an *ObservableScope.* An *ObservableScope* implements all the logic required to receive *Observation*s from *ObservableObject*s*,* apply a customizable processing phase to each of them and then distribute those that have not been filtered out to the *Instrumentation Domain.*

The operations of an *ObservableScope* shall be executed on a separate thread. The configuration and settings of this thread are left outside this specification, however it is recommended that it be configured to run at a lower priority than the application threads. To the extent possible, application threads shall not sustain the overhead caused by processing and distribution of *Observation*s to remote applications. They should instead only be concerned with the generation of *Observation*s from application data, using the available *ObservableObject*s.

Customizable processing is provided through the *DataProcessor* interface. A *DataProcessor* is an entity created within an *InstrumentationService* that can be attached to multiple *ObservableObject* instances within the same *InstrumentationService*. Once attached to an *ObservableObject,* the *DataProcessor* can receive, through its *process_observations* method, all *Observation* samples collected by the *ObservableObject*'s *ObservableScope* for that specific *ObservableObject*.

Since the *ObservableScope* defines a single-threaded processing context, a *DataProcessor* shall process each *ObservableObject* contained in the same *ObservableScope* sequentially. Implementations of the *DataProcessor* interface can be simplified by not requiring explicit solutions for multi-thread safety provided their computations only operate within the boundaries of a single *ObservableScope,* even if they are attached to *ObservableObject* instances in a different *ObservableScope.*

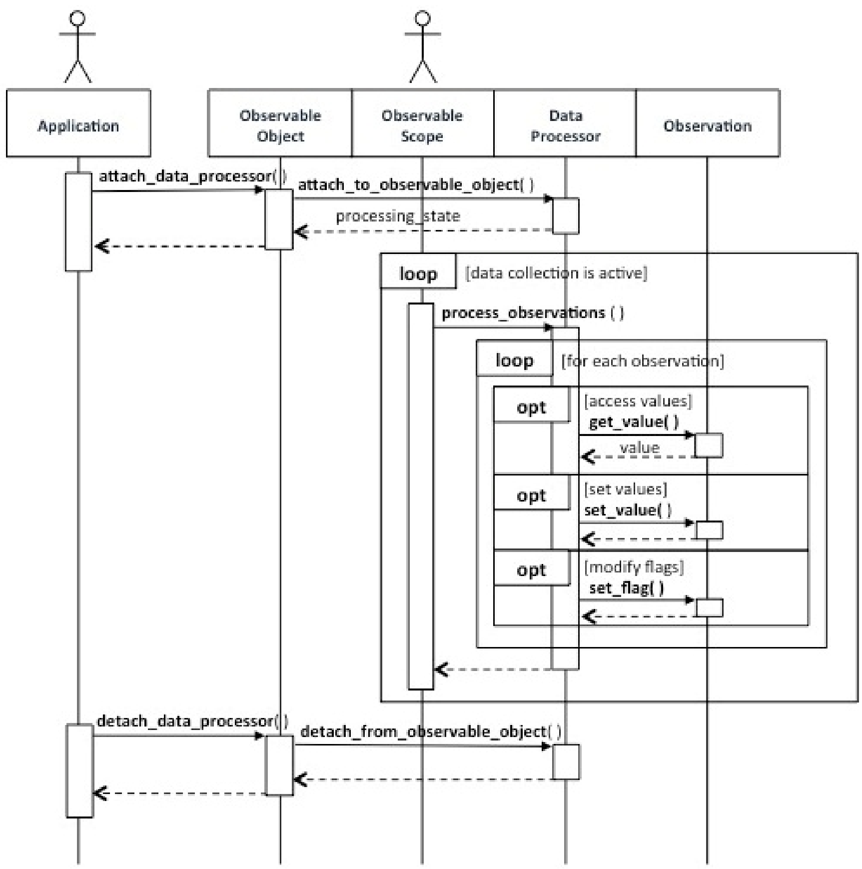

Figure 5 – Data Processing Example

A *DataProcessor* can manipulate each *Observation,* accessing its values and altering them arbitrarily. The *ObservableSchema* describing an *Observation* can also be dynamically inspected, allowing highly adaptive processing functionalities to be implemented. Additionally, a *DataProcessor* can alter the life cycle of an *Observation* by signaling the enclosing *ObservableScope* through a set of binary flags contained in the *Observation:*

- *LOCAL*: This flag allows the *DataProcessor* to prevent an *Observation* from being distributed by the *ObservableScope* to remote applications through the *Instrumentation Domain.*

- *KEEP*: This flag shall signal the *ObservableScope* that the *Observation* shall not be returned to the *ObservableObject* that generated it but instead added to the *ObservableObject's* observation history so that it is available to future invocations of *process_observations* for that *ObservableObject.*

When a *DataProcessor* instance is attached to an *ObservableObject,* it is also attached to the *ObservableGroup* and *ObservableScope* instances containing the *ObservableObject*. A *DataProcessor* may, at this time, allocate custom processing state for each of these entities, which shall be stored by the instrumentation infrastructure and passed to each invocation of *process_observations* made for *Observation*s created by that specific *ObservableObject*. This "scoped state" provides a flexible infrastructure that simplifies the implementation of complex processing logic by relieving *DataProcessor* instances from having to maintain state for each entity themselves.

Figure 5 shows an example of how a *DataProcessor* instance interacts with other instrumentation entities. In particular, it shows how a *DataProcessor* is attached to an *ObservableObject* and the multiple operations that it may perform on each *Observation* instance received from an *ObservableScope*.

7.3.2 Instrumentation Domain

The *Instrumentation Domain* is responsible for letting instrumented and monitoring applications communicate and exchange instrumentation data.

Its characteristics are intentionally left abstract by this specification and provided only as a set of high-level descriptions because many aspects of its interface and functionalities depend on the communication infrastructure chosen for its implementation.

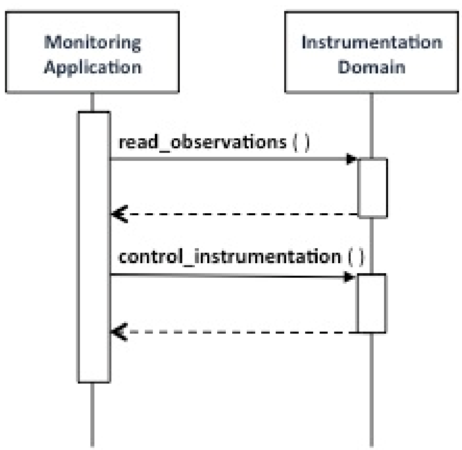

Figure 6 – Example interaction with Instrumentation Domain

Figure 6 shows an example of how remote monitoring applications can interact with the *Instrumentation Domain* in order to access the instrumentation infrastructure. While the *Instrumentation Domain*'s principal purpose is to deliver *Observation* samples received from instrumented application to the monitoring applications that requested them, an *Instrumentation Domain* can also be used to dynamically configure the instrumentation entities created by applications attached to it.

The signature of the *read_observations* and *control_instrumentation* operations are not specified in this specification. In fact, each implementation of the *Instrumentation Domain* may expose access to *Observation* instances and remote configuration through very different interfaces and communication tools, which it may not be possible to map to a programmatic interface.

For example, a very simple instrumentation domain may serialize each *Observation* to a text file, which can be then inspected by a consumer using utilities such as *tail* or *grep* or a text editor. Other solutions may leverage more complex distribution schemes such as those offered by a publish/subscribe middleware. This is the case for the platform-specific model (PSM) of the *Instrumentation Domain* using OMG Data Distribution Service and presented in 8.2.1 OMG Data Distribution Service.

7.4 Application Instrumentation (API)

The PIM of the Application Instrumentation API is organized in three functional modules, which group the API entities according to their purpose.

The responsibilities of each module are summarized in the following table. The rest of this describes each module in further detail by presenting the interface of each entity in the module.

Module	***Purpose***
Instrumentation	Define the infrastructure required to manage the local instrumentation of an application and its interaction with a distributed *Instrumentation Domain*.
Data Representation	Describe the structure of application data by defining the data schemas and provide a generic interface to manipulate samples of data.
Data Collection	Provide an interface for the generation of data from instrumented applications and support configuration of its processing and distribution to the *Instrumentation Domain*.
Data Type	Define a set of platform-independent data types to represent primitive and sequence data types supported by the instrumentation API.
Properties	Define data structures to configure instrumentation entities and provide support for defining default configuration properties for each type of entity.

7.4.1 Instrumentation Module

The Instrumentation Module is comprised of the following classifiers:

- *Infrastructure*
- *InstrumentationService*

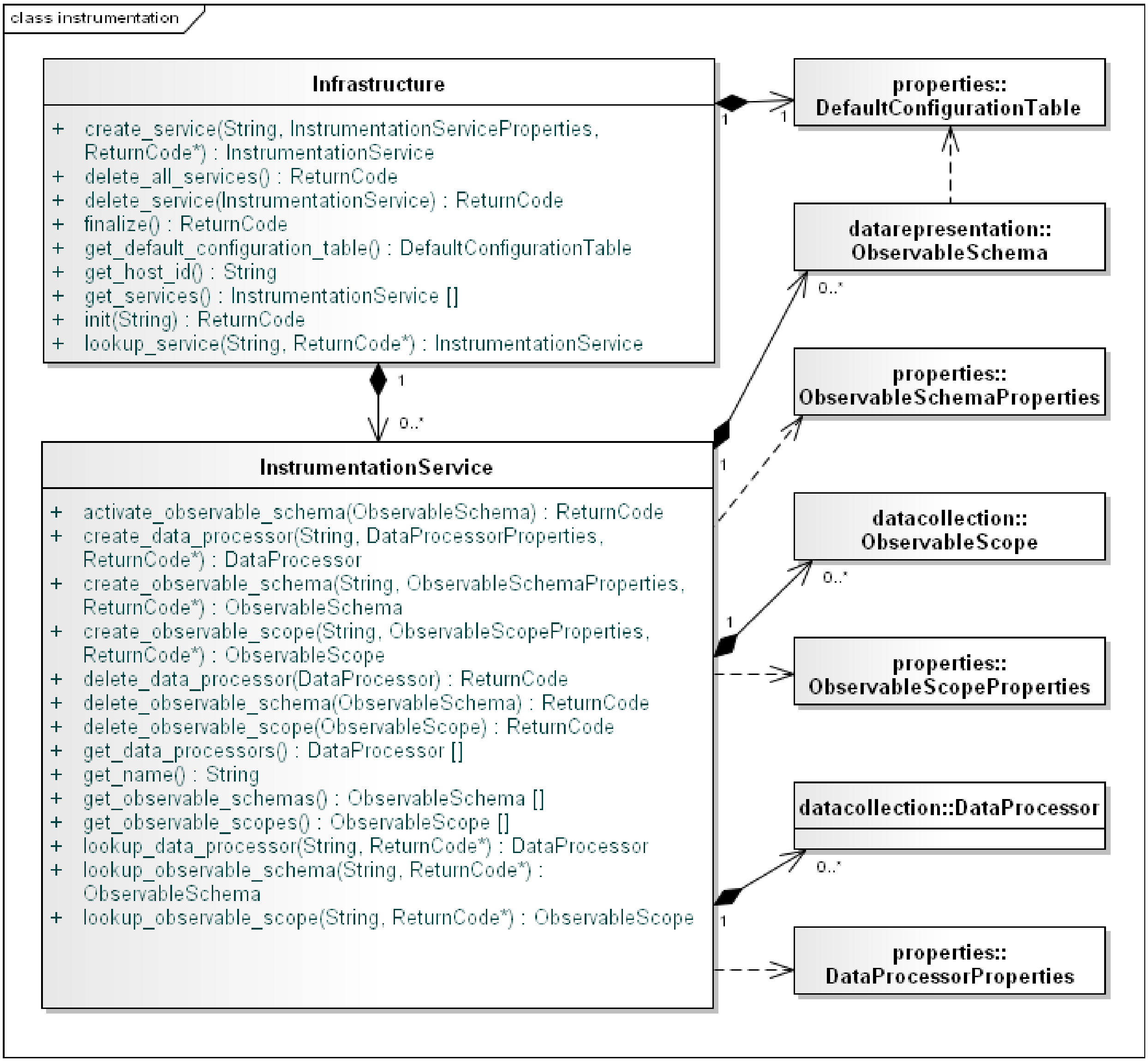

Figure 7 – Instrumentation Module

7.4.1.1 Infrastructure

The *Infrastructure* class shall be responsible for managing the life cycle of the local instrumentation used by an application. This class shall provide operations to initialize and finalize the global resources required to create and manage *InstrumentationService* instances.

Implementations of this specification shall provide access to at least one *Infrastructure* instance. It is recommended, although not mandatory, that *Infrastructure* be implemented using the singleton pattern.

Infrastructure		
No Attributes		
Operations		
create_service		InstrumentationService
	name	String
	properties	InstrumentationService Properties
	[out] retcode	ReturnCode
delete_all_services		ReturnCode
delete_service		ReturnCode
	service	InstrumentationService
finalize		ReturnCode
get_default_ configuration_table		DefaultConfigurationTable
get_host_id		String
get_services		InstrumentationService[]
init		ReturnCode
	host_id	String
lookup_service		InstrumentationService
	name	String
	[out] retcode	ReturnCode

7.4.1.1.1 create_service

This operation shall create or retrieve an instance of *InstrumentationService*.

If the creation fails, the operation shall return 'nil' (as defined by the platform [PSM]) and shall set *retcode* to RETCODE_ERROR.

If no other *InstrumentationService* instance with the specified name exists, the operation shall create a new instance, return it and shall set *retcode* to RETCODE_OK. If an instance with the specified name already exists, the operation shall return a reference to the existing *InstrumentationService* and shall set *retcode* to RETCODE_NOT_MODIFIED.

This operation shall call *init* implicitly if the application had not invoked it prior to this call, passing APPINST_HOST_ID_AUTO as *host_id*. If *init* fails, this operation shall also fail and return 'nil' (as defined by the PSM) and it shall set *retcode* to RETCODE_PRECONDITION_NOT_MET.

If the special value APPINST_INSTRUMENTATION_SERVICE_PROPERTIES_DEFAULT is passed as the *properties* argument, the operation shall substitute it with the value returned by the *get_default_service_properties* operation on the *DefaultConfigurationTable* instance.

Parameter *name*: The name to assign to the new *InstrumentationService*.

Parameter *properties*: An *InstrumentationServiceProperties* structure used to configure the new *InstrumentationService*. If an *InstrumentationService* with the specified name already exists, this parameter is ignored. APPINST_INSTRUMENTATION_SERVICE_PROPERTIES_DEFAULT may be specified in order to use the default value returned by the *DefaultConfigurationTable*'s *get_default_service_properties* operation.

Parameter *retcode*: The operation shall return RETCODE_OK if the new *InstrumentationService* was successfully created. RETCODE_NOT_MODIFIED if an existing *InstrumentationService* with the same name was found locally, RETCODE_BAD_PARAMETER if values specified for parameters *properties* and/or *name* were incorrect, and RETCODE_ERROR if there was any other type of error.

Return: The operation shall return an *InstrumentationService* in case of success (RETCODE_OK or RETCODE_NOT_MODIFIED), or 'nil' (as defined by the platform [PSM]) in case of error.

7.4.1.1.2 delete_all_services

This operation shall delete any *InstrumentationService* instance currently existing in the instrumentation infrastructure by invoking the *delete_service* operation on each of them. If the deletion of any existing *InstrumentationService* fails, this operation shall fail and return RETCODE_ERROR.

If the *init* operation has never been called successfully yet, this operation shall do nothing and return RETCODE_PRECONDITION_NOT_MET.

Return: The operation shall return RETCODE_OK upon successful deletion of all existing *InstrumentationService* instances, RETCODE_PRECONDITION_NOT_MET if the *init* operation has never been successfully called yet, RETCODE_ERROR if any other type of error occurred.

7.4.1.1.3 delete_service

This operation shall delete an instance of *InstrumentationService* and all entities it contains.

If the *InstrumentationService* contains any entity, the operation shall:

- Delete every *ObservableScope* instance (as reported by the *InstrumentationService*'s *get_observable_scopes* operation) by invoking the *InstrumentationService*'s *delete_observable_scope* operation on each of them.
- Delete every *DataProcessor* instance (as reported by the *InstrumentationService*'s *get_data_processors* operation) by invoking the *InstrumentationService*'s *delete_data_processor* operation on each of them.
- Delete every *ObservableSchema* instance (as reported by the *InstrumentationService*'s *get_observable_schemas* operation) by invoking the *InstrumentationService*'s *delete_observable_schema* operation on each of them.

If the deletion of any of the contained entities fails by returning a value other than RETCODE_OK, the operation shall fail and return RETCODE_ERROR.

If the specified *InstrumentationService* was not created by this *Infrastructure* or the value 'nil' (as specified by the platform [PSM]) is passed to the operation, the operation shall fail and return RETCODE_BAD_PARAMETER.

If the *init* operation has never been called successfully on the *Infrastructure* instance yet, the operation shall fail and return RETCODE_PRECONDITION_NOT_MET.

Parameter *service*: The *InstrumentationService* to delete.

Return: The operation shall return RETCODE_OK upon successful deletion of the *InstrumentationService* and all its contained entities, RETCODE_PRECONDITION_NOT_MET if any of the contained entities was not successfully deleted or the *init* operation has never been called successfully on the *Infrastructure*, RETCODE_BAD_PARAMETER if a bad value was specified for the *service* parameter, RETCODE_ERROR if any other error occurred.

7.4.1.1.4 finalize

This operation shall finalize the static resources required to manage the instrumentation infrastructure and all the *InstrumentationService* instances that were created locally.

If any *InstrumentationService* instance still exists in the *Infrastructure,* the operation shall fail and return RETCODE_PRECONDITION_NOT_MET.

This operation shall free any resource reserved by the invocation of operation *init.* If the freeing of any of these resources fails, the operation shall fail and return RETCODE_ERROR.

If the *init* operation has never been called successfully yet, this operation shall do nothing and return RETCODE_NOT_MODIFIED.

Return: The operation shall return RETCODE_OK upon successful finalization of all instrumentation, RETCODE_PRECONDITION_NOT_MET if any *InstrumentationService* instance still exists in the *Infrastructure*, RETCODE_ERROR if any error occurred during finalization of static resources, RETCODE_NOT_MODIFIED if the *init* operation has never been successfully called yet.

7.4.1.1.5 get_default_configuration_table

This operation shall return the singleton instance of *DefaultConfigurationTable.* If the instance does not exist, the operation shall create a new one and store it internally so that it may be returned by future invocations of this operation.

Return: a *DefaultConfigurationTable* instance or 'nil' (as defined by the platform [PSM]) if any type of error occurred.

7.4.1.1.6 get_host_id

This operation shall return a non-empty string containing the host identifier specified to the *init* operation. If *init* was called with special value APPINST_HOST_ID_AUTO, the returned value shall be the one automatically assigned by the *Infrastructure*.

If the *init* operation has never been called successfully yet, the operation shall return 'nil' (as defined by the platform [PSM]).

Return: the operation shall return a non-modifiable, non-empty string containing the identifier of the host containing the *Infrastructure* instance or 'nil' (as defined by the platform [PSM]) if the *init* operation has not been called on the *Infrastructure* instance yet or any other type of error occurred.

7.4.1.1.7 get_services

This operation returns a collection of all *InstrumentationService* instances that have been created in the local instrumentation. An empty collection will be returned if no *InstrumentationService* has yet been created locally.

If the *init* operation has never been called successfully on the *Infrastructure* instance yet, the operation shall return 'nil' (as defined by the platform [PSM]).

Return: The operation shall return a collection of *InstrumentationService* instances or 'nil' (as defined by the platform [PSM]) if *init* as not been called on the *Infrastructure* instance yet or any other type of error occurred.

7.4.1.1.8 init

This operation shall initialize the local instrumentation infrastructure and allocate the resources required to create instances of *InstrumentationService* and to manage their life cycle.

If an error prevents initialization from succeeding, this operation shall return RETCODE_ERROR.

If the initialization is completed successfully, this operation shall return RETCODE_OK.

After successful initialization, future invocations of this operation shall have no effect and return RETCODE_NOT_MODIFIED, until the operation *finalize_instrumentation* is called. The call to *finalize_instrumentation* restores the instrumentation service to its initial state as it was prior to the first call to *init_instrumentation*.

Prior to calling this operation for the first time, the instrumentation infrastructure shall be considered 'uninitialized.' After successful return from this operation, the instrumentation infrastructure shall be considered 'initialized.' The 'initialized' state shall continue until the operation *finalize_instrumentation* is called and executed successfully.

While the instrumentation infrastructure is in the 'uninitialized' state all operations, with the exception of operations *init_instrumentation, create_service* and *lookup_service*, shall fail and set *retcode* to RETCODE_PRECONDITION_NOT_MET.

The values specified by the *host_id* parameter shall be stored by the instrumentation infrastructure. It shall be used to mark all *Observation* samples generated by this instrumentation infrastructure and it will be used to identify the local instrumentation infrastructure in the distributed environment.

The value APPINST_HOST_ID_AUTO may be specified as *host_id*. In this case, the operation shall automatically generate a name for the local instrumentation infrastructure. The algorithm used to generate the string is not normative.

Parameter *host_id*: A string providing an identifier for the host on which the instrumentation infrastructure is being initialized. The special value APPINST_HOST_ID_AUTO may be specified to let the instrumentation infrastructure automatically generate a value.

Return: The operation shall return RETCODE_OK if the operation succeeds and the instrumentation infrastructure is in state 'initialize' after the return from the operation. If the value of *host_id* is an empty string, the operation shall return RETCODE_BAD_PARAMETER. If the operation has already been called successfully, the operation shall return RETCODE_NOT_MODIFIED. Otherwise it shall return RETCODE_ERROR if any other kind of error occurred.

7.4.1.1.9 lookup_service

This operation shall search among the *InstrumentationService* instances created locally and return the one identified by the specified name.

If an instance by that name is found, the operation shall return it and set *retcode* to RETCODE_OK.

If no matching instance is found and automatic creation is enabled in the *DefaultConfigurationTable*, the operation shall create a new *InstrumentationService* by invoking *create_service* with the specified name and the default value for *InstrumentationServiceProperties*, contained in the *DefaultConfigurationTable.* If the creation of the new *InstrumentationService* fails by returning a 'nil' (as defined by the platform [PSM]) value, this operation shall return 'nil' (as defined by the platform [PSM]) and set *retcode* to RETCODE_ERROR.

If no matching instance is found and automatic creation is disabled, the operation shall return 'nil' (as defined by the platform [PSM]) and set *retcode* to RETCODE_ERROR.

If the *init* operation has never been called successfully on the *Infrastructure* instance yet, the operation return 'nil' (as specified by the platform [PSM]) and set *retcode* to RETCODE_PRECONDITION_NOT_MET.

Parameter *name*: The name of the *InstrumentationService* to lookup.

Parameter *retcode*: The operation shall return RETCODE_OK if a matching *InstrumentationService* was found or successfully created. RETCODE_PRECONDITION_NOT_MET if the *InstrumentationService* could not be automatically created or *init* has not been successfully called on the *Infrastructure*, and RETCODE_ERROR if no matching instance was found or there was any other type of error.

Return: The operation shall return an *InstrumentationService* in case of success (RETCODE_OK), or 'nil' (as defined by the platform [PSM]) otherwise.

7.4.1.2 InstrumentationService

The *InstrumentationService* creates and manages all local instrumentation entities used by an application and it attaches the local instrumentation to an external *Instrumentation Domain*.

Implementations of this specification shall support creation of multiple *InstrumentationService* instances within the same *Infrastructure* instance, for example, to produce data to multiple *Instrumentation Domains.*

Each *InstrumentationService* shall be identified by a unique name, which may be used to reference it within the context of an instrumented application. This name may be specified by the application when creating an *InstrumentationService* or it may be automatically generated by the instrumentation infrastructure.

An *InstrumentationService* instance may be responsible for managing one or more threads that execute the data collection and processing operations of the *ObservableScope* instances contained in the *InstrumentationService.* It is left to implementations of this specification to define how this functionality may be configured by applications. Note that some implementations may employ separate threads to operate the *ObservableScope* instances or alternatively execute their operations on application threads.

InstrumentationService		
No Attributes		
Operations		
activate_observable_schema		ReturnCode
	schema	ObservableSchema
create_data_processor		DataProcessor
	name	String
	properties	DataProcessorProperties
	[out] retcode	ReturnCode
create_observable_schema		ObservableSchema
	name	String
	properties	ObservableSchemaProperties
	[out] retcode	ReturnCode
create_observable_scope		ObservableScope
	name	String
	properties	ObservableScopeProperties
	[out] retcode	ReturnCode
delete_data_processor		ReturnCode
	processor	DataProcessor
delete_observable_schema		ReturnCode
	schema	ObservableSchema
delete_observable_scope		ReturnCode
	scope	ObservableScope
get_data_processors		DataProcessor[]
get_name		String
get_observable_schemas		ObservableSchema[]
get_observable_scopes		ObservableScope[]
lookup_data processor		DataProcessor

	name	String
	[out] retcode	ReturnCode
lookup_observable_schema		ObservableSchema
	name	String
	[out] retcode	ReturnCode
lookup_observable_scope		ObservableScope
	name	String
	[out] retcode	ReturnCode

7.4.1.2.1 activate_observable_schema

This operation shall set the state of an *ObservableSchema* to 'active.' The 'active' state signals the *InstrumentationService* that the definition of the *ObservableSchema* is complete and *ObservableObject* instances may be created referencing the schema. When an *ObservableSchema* is set to 'active' subsequent calls to the *create_field* operation shall fail with RETCODE_PRECONDITION_NOT_MET.

Parameter *schema:* the *ObservableSchema* to be activated.

Return: The operation shall return RETCODE_OK if the *ObservableSchema* was correctly set 'active,' RETCODE_NOT_MODIFIED if the *ObservableSchema* was already in state 'active,' and RETCODE_ERROR if any type of error occurred when activating the schema.

7.4.1.2.2 create_data_processor

This operation shall create a new *DataProcessor* instance in the *InstrumentationService.*

The new *DataProcessor* shall have the specified *name* and it will be initialized using the specified *properties.* If the creation succeeds, the operation shall return the newly created *DataProcessor* and set *retcode* to RETCODE_OK. If a *DataProcessor* instance with the same *name* already exists in the *InstrumentationService*, the operation shall return the existing instance and set *retcode* to RETCODE_NOT_MODIFIED.

Each platform [PSM] must define how applications shall specify the implementation of the *DataProcessor* interface that will be used to create the new instance by extending the *DataProcessorProperties* structure.

If the special value APPINST_DATA_PROCESSOR_PROPERTIES_DEFAULT is passed as the *properties* argument, the operation shall use the value returned by the *get_default_data_processor_properties* operation on the *DefaultConfigurationTable* instance.

After successfully creating the new *DataProcessor,* the operation shall invoke its *initialize* operation passing the *DataProcessorArgs* value contained in the specified *properties* and the *InstrumentationService* instance itself as arguments. If this operation returns a value other than RETCODE_OK, the operation shall delete the newly created *DataProcessor* (as specified in the description of operation *delete_data_processor*) and set *retcode* to RETCODE_ERROR.

If the creation fails, the operation shall return 'nil' (as defined by the platform [PSM]) and set *retcode* to RETCODE_ERROR.

Parameter *name*: The name to assign to the new *DataProcessor.*

Parameter *properties*: A *DataProcessorProperties* structure used to configure the new *DataProcessor.* If a *DataProcessor* with the specified *name* already exists, this Parameter is ignored.

Parameter *retcode*: The operation shall return RETCODE_OK if the new *DataProcessor* was successfully created and initialized. RETCODE_NOT_MODIFIED if an existing *DataProcessor* with the same *name* was found locally, RETCODE_BAD_PARAMETER if values specified for parameters *properties* and/or *name* were incorrect, and RETCODE_ERROR if the newly created *DataProcessor* could not be initialized or if there was any other type of error.

Return: The operation shall return a *DataProcessor* in case of success (RETCODE_OK or RETCODE_NOT_MODIFIED), or 'nil' (as defined by the platform [PSM]) in case of error.

7.4.1.2.3 create_observable_schema

This operation shall create or retrieve an *ObservableSchema* instance in an *InstrumentationService.*

The new *ObservableSchema* shall have the specified *name* and it will be initialized using the specified *properties.* If the creation succeeds, the operation shall return the newly created *ObservableSchema* and set *retcode* to RETCODE_OK. If an *ObservableSchema* instance with the same *name* already exists in the *InstrumentationService*, the operation shall return the existing instance and set *retcode* to RETCODE_NOT_MODIFIED.

If the special value APPINST_OBSERVABLE_SCHEMA_PROPERTIES_DEFAULT is passed as the *properties* argument, the operation shall use the value returned by the *get_default_observable_schema_properties* operation on the *DefaultConfigurationTable* instance.

If the creation of the *ObservableSchema* fails, the operation shall return 'nil' and set *retcode* to RETCODE_ERROR.

Parameter *name*: The name to assign to the new *ObservableSchema*.

Parameter *properties*: An *ObservableSchemaProperties* structure used to configure the new *ObservableSchema*. If an *ObservableSchema* with the specified *name* already exists, this parameter is ignored. 'nil' (as specified by the platform [PSM]) may be specified in order to use the default value returned by the *DefaultConfigurationTable*'s *get_default_observable_schema_properties* operation.

Parameter *retcode*: The operation shall return RETCODE_OK if the new *ObservableSchema* was successfully created. RETCODE_NOT_MODIFIED if an existing *ObservableSchema* with the same *name* was found locally, RETCODE_BAD_PARAMETER if values specified for parameters *properties* and/or *name* were incorrect, and RETCODE_ERROR if there was any other type of error.

Return: The operation shall return an *ObservableSchema* in case of success (RETCODE_OK or RETCODE_NOT_MODIFIED), or 'nil' (as defined by the platform [PSM]) in case of error.

7.4.1.2.4 create_observable_scope

This operation shall create or retrieve an *ObservableScope* instance in the *InstrumentationService.*

The new *ObservableScope* shall have the specified *name* and it will be initialized using the specified *properties.* If the creation succeeds, the operation shall return the newly created *ObservableScope* and set *retcode* to RETCODE_OK. If an *ObservableScope* instance with the same *name* already exists in the *InstrumentationService*, the operation shall return the existing instance and set *retcode* to RETCODE_NOT_MODIFIED.

If the special value APPINST_OBSERVABLE_SCOPE_PROPERTIES_DEFAULT is passed as the *properties* argument, the operation shall use the value returned by the *get_default_observable_scope_properties* operation on the *DefaultConfigurationTable* instance.

If data collection is enabled in the specified *properties* (by setting attribute *enable_data_collection* to *True*), after successfully creating the new *ObservableScope* and any contained *ObservableObject* instance*,* the operation shall invoke its *enable_data_collection* operation. If this operation returns a value other than RETCODE_OK, the operation shall delete any already created *ObservableObject* (as specified by the description of operation *delete_observable_object* of *ObservableScope*), the newly created *ObservableScope* (as specified by the description of operation *delete_observable_scope*) and set *retcode* to RETCODE_ERROR.

If the creation fails, the operation shall return 'nil' (as defined by the platform [PSM]) and set *retcode* to RETCODE_ERROR.

Parameter *name*: The name to assign to the new *ObservableScope*.

Parameter *properties*: An *ObservableScopeProperties* structure used to configure the new *ObservableScope*. If an *ObservableScope* with the specified *name* already exists, this Parameter is ignored. APPINST_OBSERVABLE_SCOPE_PROPERTIES_DEFAULT may be specified in order to use the default value returned by the *DefaultConfigurationTable*'s *get_default_observable_scope_properties* operation.

Parameter *retcode*: The operation shall return RETCODE_OK if the new *ObservableScope* was successfully created. RETCODE_NOT_MODIFIED if an existing *ObservableScope* with the same *name* was found locally, RETCODE_BAD_PARAMETER if values specified for parameters *properties* and/or *name* were incorrect, and RETCODE_ERROR if data collection could not be enabled on the newly created *ObservableScope* or if there was any other type of error.

Return: The operation shall return an *ObservableScope* in case of success (RETCODE_OK or RETCODE_NOT_MODIFIED), or 'nil' (as defined by the platform [PSM]) in case of error.

7.4.1.2.5 delete_data_processor

This operation shall delete an existing *DataProcessor* instance from an *InstrumentationService*.

If the *DataProcessor* is currently attached to any *ObservableObject*, the operation shall fail and return RETCODE_PRECONDITION_NOT_MET.

The operation shall invoke the *DataProcessor*'s *finalize* operation passing the *InstrumentationService* instance itself as argument. If any value other than RETCODE_OK is returned, the operation shall return RETCODE_PRECONDITION_NOT_MET. The operation shall try to free all resources independently of the value returned by the *DataProcessor*'s *finalize* operation.

If the *DataProcessor* is successfully deleted, the operation shall return RETCODE_OK.

The operation shall fail and return RETCODE_BAD_PARAMETER if the *InstrumentationService* performing the operation did not create the *DataProcessor*.

Parameter *schema*: the *DataProcessor* to delete.

Return: The operation shall return RETCODE_OK if the *DataProcessor* was successfully deleted from the *InstrumentationService*, RETCODE_PRECONDITION_NOT_MET if the *DataProcessor* is currently attached to any *ObservableObject* or the *finalize* operation of the *DataProcessor* did not exit successfully, RETCODE_BAD_PARAMETER if the specified *DataProcessor* was not created by this *InstrumentationService*, and RETCODE_ERROR if any other type of error occurred.

7.4.1.2.6 delete_observable_schema

This operation shall delete an existing *ObservableSchema* instance from an *InstrumentationService*.

The operation shall fail and return RETCODE_PRECONDITION_NOT_MET if there are currently *ObservableObject* instances using the *ObservableSchema* in any of the *ObservableScope* instances of the *InstrumentationService*.

The operation shall fail and return RETCODE_BAD_PARAMETER if the *InstrumentationService* performing the operation did not create the *ObservableSchema*.

Parameter *schema*: the *ObservableSchema* to delete.

Return: The operation shall return RETCODE_OK if the *ObservableSchema* was successfully deleted from the *InstrumentationService*, RETCODE_PRECONDITION_NOT_MET if any *ObservableObject* referencing the *ObservableSchema* exists in the *InstrumentationService,* RETCODE_BAD_PARAMETER if the specified *ObservableSchema* was not created by this *InstrumentationService*, and RETCODE_ERROR if any other type of error occurred.

7.4.1.2.7 delete_observable_scope

This operation shall delete an existing *ObservableScope* instance from an *InstrumentationService*.

This operation shall invoke the *ObservableScope*'s *disable_data_collection* operation before deleting all *ObservableObject* instances contained in the *ObservableScope* (as specified by operation *delete_observable_object*).

If any of these operations fails with return value other than RETCODE_OK, the operation shall fail and return RETCODE_PRECONDITION_NOT_MET.

If all contained *ObservableObject* instances and the *ObservableScope* are successfully deleted, the operation shall return RETCODE_OK.

The operation shall fail and return RETCODE_BAD_PARAMETER if the *InstrumentationService* performing the operation did not create the *ObservableScope*.

Parameter *schema*: the *ObservableScope* to delete.

Return: The operation shall return RETCODE_OK if the *ObservableScope* was successfully deleted from the *InstrumentationService*, RETCODE_PRECONDITION_NOT_MET if any *ObservableObject* contained in the *ObservableScope* could not be deleted or data collection could not be disabled in the *ObservableScope,* RETCODE_BAD_PARAMETER if the specified *ObservableScope* was not created by this *InstrumentationService*, and RETCODE_ERROR if any other type of error occurred.

7.4.1.2.8 get_data_processors

This operation shall return a collection of all *DataProcessor* instances that have been created in an *InstrumentationService*. An empty collection shall be returned if the *InstrumentationService* does not contain any *DataProcessor* yet.

7.4.1.2.9 get_name

This operation shall return a string containing the name of an *InstrumentationService* instance.

Return: The operation shall return an unmodifiable, non-empty, string.

7.4.1.2.10 get_observable_schemas

This operation shall return a collection of all *ObservableSchema* instances that have been created in an *InstrumentationService*. An empty collection will be returned if the *InstrumentationService* does not contain any *ObservableSchema* yet.

Return: The operation shall return a collection of *ObservableSchema* instances.

7.4.1.2.11 get_observable_scopes

This operation shall return a collection of all *ObservableScope* instances that have been created in an *InstrumentationService*. An empty collection will be returned if the *InstrumentationService* does not contain any *ObservableScope* yet.

Return: The operation shall return a collection of *ObservableScope* instances.

7.4.1.2.12 lookup_data_processor

This operation shall search among the *DataProcessor* instances created in an *InstrumentationService* and return the one identified by the specified name.

If an instance by that name is found, the operation shall return it and set *retcode* to RETCODE_OK.

If no matching instance is found, the operation shall return 'nil' (as defined by the platform [PSM]) and set *retcode* to RETCODE_ERROR.

Parameter *name*: The name of the *DataProcessor* to lookup.

Parameter *retcode*: The operation shall return RETCODE_OK if a matching *DataProcessor* was found in the *InstrumentationService*, RETCODE_BAD_PARAMETER if a bad value was specified for the *name* parameter, and RETCODE_ERROR if no matching instance was found or there was any other type of error.

Return: The operation shall return a *DataProcessor* in case of success (RETCODE_OK), or 'nil' (as defined by the platform [PSM]) otherwise.

7.4.1.2.13 lookup_observable_schema

This operation shall search among the *ObservableSchema* instances created in an *InstrumentationService* and return the one identified by the specified name.

If an instance by that name is found, the operation shall return it and set *retcode* to RETCODE_OK.

If no matching instance is found, the operation shall return 'nil' (as defined by the platform [PSM]) and set *retcode* to RETCODE_ERROR.

Parameter *name*: The name of the *ObservableSchema* to lookup.

Parameter *retcode*: The operation shall return RETCODE_OK if a matching *ObservableSchema* was found in the *InstrumentationService*, RETCODE_BAD_PARAMETER if a bad value was specified for the *name* parameter, and RETCODE_ERROR if no matching instance was found or there was any other type of error.

Return: The operation shall return an *ObservableSchema* in case of success (RETCODE_OK), or 'nil' (as defined by the platform [PSM]) otherwise.

7.4.1.2.14 lookup_observable_scope

This operation shall search among the *ObservableScope* instances created in an *InstrumentationService* and return the one identified by the specified name.

If an instance by that name is found, the operation shall return it and set *retcode* to RETCODE_OK.

If no matching instance is found, the operation shall return 'nil' (as defined by the platform [PSM]) and set *retcode* to RETCODE_ERROR.

Parameter *name*: The name of the *ObservableScope* to lookup.

Parameter *retcode*: The operation shall return RETCODE_OK if a matching *ObservableScope* was found in the *InstrumentationService*, RETCODE_BAD_PARAMETER if a bad value was specified for the *name* parameter, and RETCODE_ERROR if no matching instance was found or there was any other type of error.

Return: The operation shall return an *ObservableScope* in case of success (RETCODE_OK), or 'nil' (as defined by the platform [PSM]) otherwise.

7.4.2 Data Representation Module

The *Data Representation Module* is comprised of the following classifiers:

- *ObservableSchema*
- *Field*
- *Observation*
- *ObservationFlagKind*

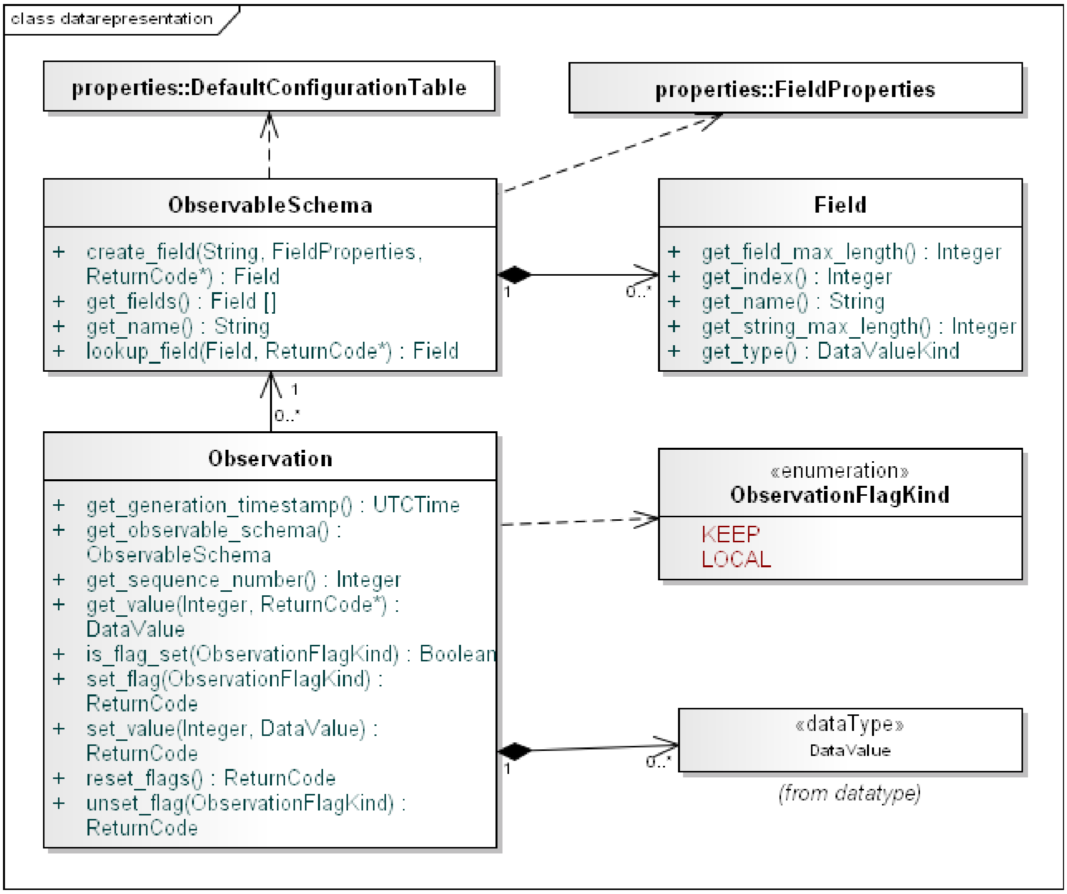

Figure 8 – Data Representation Module

7.4.2.1 ObservableSchema

An *ObservableSchema* describes the structure of application data collected at run-time.

Each *ObservableSchema* defines a complex data-type composed of a collection of named fields, each one containing a value of application data. Values stored in an *ObservableSchema*'s field may be of two types:

- A single value of a primitive type, such as numbers, characters, or strings.
- A bounded sequence of values of a primitive type.

ObservableSchema instances are created and managed by an *InstrumentationService*. They expose operations to define new *Field* entries and inspect the resulting type definition dynamically through reflection.

ObservableSchema		
No Attributes		
Operations		
`create_field`		`Field`
	`name`	`String`
	`properties`	`FieldProperties`
	`[out] retcode`	`ReturnCode`
`get_fields`		`Field[]`
`get_name`		`String`
`lookup_field`		`ReturnCode`
	`name`	`String`

7.4.2.1.1 create_field

This operation shall create a new *Field* instance and add it to the collection of fields of an *ObservableSchema.*

The *Field* will have the specified *name* and it will be initialized using the specified *properties*.

If creation succeeds, the operation shall return the new *Field* and set *retcode* to RETCODE_OK. If a *Field* with the same name already exists, the operation shall fail, returning 'nil' (as defined by the platform [PSM]) and set *retcode* to RETCODE_PRECONDITION_NOT_MET.

If the *ObservableSchema* has already been set to 'active' state in the *InstrumentationService* (by invoking the *InstrumentationService*'s *activate_observable_schema* operation), the operation shall fail, returning 'nil' and setting *retcode* to RETCODE_PRECONDITION_NOT_MET.

The operation shall assign a numerical index to the new *Field,* which uniquely identifies the new *Field* within the context of the enclosing *ObservableSchema.*

Parameter *name*: name of the new *Field* to create.

Parameter *properties*: an instance of *FieldProperties* specifying the properties for the new *Field.* If attribute *type* is one of the sequence data types, then *max_length* must be greater than 0 and specifies the maximum length of the sequence. If attribute *type* is a primitive data type, *max_length* will be ignored. If *type* is one of TYPE_STRING8, TYPE_STRING32, TYPE_STRING8_SEQ, or TYPE_STRING32_SEQ, *string_max_length* must present a value greater than 0, indicating the maximum length of a string stored in the *Field*. If *type* is any other non-string type, *string_max_length* will be ignored. If a *Field* with the same name already exists, this attribute is ignored.

Parameter *retcode*: The operation shall return RETCODE_OK if the new *Field* was successfully created and added to the *ObservableSchema*. RETCODE_PRECONDITION_NOT_MET if an existing *Field* with the same *name* was found locally or if the *ObservableSchema* is already in 'active' state and cannot be modified, RETCODE_BAD_PARAMETER if values specified for parameters *properties* and/or *name* were incorrect, and RETCODE_ERROR if there was any other type of error.

7.4.2.1.2 get_fields

This operation shall return a collection of all *Field* instances that have been created in an *ObservableSchema*. An empty collection will be returned if the *ObservableSchema* does not contain any *Field* yet.

Return: The operation shall return a collection of *DataProcessor* instances.

7.4.2.1.3 get_name

This operation shall return a string containing the name of an *ObservableSchema* instance.

Return: The operation shall return an unmodifiable, non-empty, string.

7.4.2.1.4 lookup_field

This operation shall search among the *Field* instances created in an *ObservableSchema* and return the one identified by the specified name.

If an instance by that name is found, the operation shall return it and set *retcode* to RETCODE_OK.

If no matching instance is found, the operation shall return 'nil' (as defined by the platform [PSM]) and set *retcode* to RETCODE_ERROR.

Parameter *name*: The name of the *Field* to lookup.

Parameter *retcode*: The operation shall return RETCODE_OK if a matching *Field* was found in the *ObservableSchema*, RETCODE_BAD_PARAMETER if a bad value was specified for the *name* parameter, and RETCODE_ERROR if no matching instance was found or there was any other type of error.

Return: The operation shall return a *Field* in case of success (RETCODE_OK), or 'nil' (as defined by the platform [PSM]) otherwise.

7.4.2.2 Field

A *Field* instance describes a single field in an *ObservableSchema*. It provides an interface to access its properties.

Fields shall have a unique order within an *ObservableSchema,* which is reflected by the index assigned to each of them at creation. The value of each field's index can be determined using the *get_index* operation provided by the *Field* interface.

Field		
No Attributes		
Operations		
get_field_max_length		Integer
get_index		Integer
get_name		String
get_string_max_length		Integer
get_type		DataValueKind

7.4.2.2.1 get_field_max_length

This operation shall return the maximum number of values that can be stored in a *Field*. This value is always 1 for all primitive values. *Field* instances of a sequence type shall return a positive integer corresponding to the maximum length of sequences that can be stored in the *Field*.

Return: The operation shall return an integer value greater than 0. If the *Field* is of primitive type, the operation shall return 1.

7.4.2.2.2 get_index

This operation shall return the unique, 0-based, index that identifies the *Field* within its enclosing *ObservableSchema.*

Return: The operation shall return an integer value between 0 and N-1, where N is the number of *Field* instances in the *ObservableSchema.*

7.4.2.2.3 get_name

This operation shall return a string containing the name of a *Field* instance.

Return: The operation shall return an unmodifiable, non-empty, string.

7.4.2.2.4 get_string_max_length

This operation shall return the maximum length of all strings contained in a *Field*. This value shall only be used if the *Field*'s type is one of TYPE_STRING8, TYPE_STRING32, TYPE_STRING8_SEQ, or TYPE_STRING32_SEQ.

Return: The operation shall return an integer value greater than 0 if the *Field* is of type TYPE_STRING8, TYPE_STRING32, TYPE_STRING8_SEQ, or TYPE_STRING32_SEQ. Otherwise, the return value of this operation is undefined.

7.4.2.2.5 get_type

This operation shall return the type of the value that can be stored in a *Field*. The type is represented by a value of the enumeration type *DataValueKind* defined by the Data Type Module (see 7.4.4 Data Type Module).

Return: One of the values of enumeration *DataValueKind.*

7.4.2.3 Observation

An *Observation* shall contain a sample of data collected from an instrumented application. *Observation* instances provide an interface to access instrumented data and to dynamically determine its structure by inspecting the associated *ObservableSchema*.

An *Observation* may carry a time-stamp to identify the instant when it was generated by the application's code.

A set of flags is associated with each *Observation.* Flags are used to control the life cycle of an *Observation,* for example to control its processing and distribution.

Observation		
No Attributes		
Operations		
get_generation_timestamp		UTCTime
get_observable_schema		ObservableSchema
get_sequence_number		Integer
get_value <T:DataValue>		T
	field_index	Integer
	[out]	ReturnCode

is_flag_set		Boolean
	flag	ObservationFlagKind
set_flag		ReturnCode
	flag	ObservationFlagKind
set_value <T:DataValue>		ReturnCode
	field_index	Integer
	value	T
reset_flags		ReturnCode
unset_flag		ReturnCode
	flag	ObservationFlagKind

7.4.2.3.1 get_generation_timestamp

This operation shall return a *UTCTime* value indicating when the *Observation* was generated. If a time-stamp was not collected when the *Observation* was generated, the operation shall return the special value APPINST_UTCTIME_INVALID.

Return: The operation shall return a *UTCTime* value containing the *Observation*'s generation time-stamp or special value APPINST_UTCTIME_INVALID if no generation time-stamp is present.

7.4.2.3.2 get_observable_schema

This operation shall return the *ObservableSchema* associated with the value contained in an *Observation.*

Return: The operation shall return an *ObservableSchema.*

7.4.2.3.3 get_sequence_number

This operation shall return a positive integer value representing the sequence number of the *Observation* with respect to its generating source. *Observation* instances generated from the same source can be totally ordered by increasing sequence number. The resulting order shall reflect exactly the order in which the *Observation* instances where generated by their source.

The first *Observation* generated by a source shall have sequence number 1.

Return: The operation shall return an integer value.

7.4.2.3.4 get_value<T>

This operation shall return the value stored by an *Observation* for a specific *Field* of its *ObservableSchema.* The *Observation* shall contain a value for each *Field* contained in the *ObservableSchema.* If a *Field* with the specified *field_index* exists in the *ObservableSchema,* the operation shall return the value stored for the *Field* by the *Observation* and set *retcode* to RETCODE_OK.

The operation shall expose a parameter T, which may be type *DataValue* or one of its sub-classes, specifying the type of value that must be returned to the application. If the specified type is different from the requested *Field*'s data type and the value stored in the *Observation* cannot be converted to the requested type, the operation shall return the default value of data type T and set *retcode* to RETCODE_PRECONDITION_NOT_MET.

If the specified *field_index* does not match the index of any of the *Field* instances contained in the *Observation*'s *ObservableSchema,* the operation shall return 'nil' and set *retcode* to RETCODE_BAD_PARAMETER.

Parameter *field_index*: The index of the *Field* of the *Observation*'s *ObservableSchema* whose value must be returned.

Parameter *retcode*: The operation shall return RETCODE_OK if a *Field* with the specified *index* was found in the *Observation*'s *ObservableSchema* and its value was successfully returned, RETCODE_BAD_PARAMETER if no matching *Field* was found for the specified *index*, RETCODE_PRECONDITION_NOT_MET if the data type of the selected field cannot be converted to the requested T data type, RETCODE_ERROR if any other type of error occurred.

Return: On success (RETCODE_OK), the operation shall return the value of the selected *Field*, converted to data type T, or the default value defined for T if any error occurred.

7.4.2.3.5 is_flag_set

This operation shall check if a flag (identified by a value of enumeration *ObservationFlagKind*) is currently in 'set' state in an *Observation.*

If the specified value does not identify any valid flag of this *Observation,* the operation shall do nothing and return *False.*

Parameter *flag*: The *Observation*'s flag to check.

Return: The operation shall return *True* if the specified flag is in 'set' state, *False* if it's in the 'unset' one or the specified *flag* does not exist.

7.4.2.3.6 reset_flags

This operation shall set the state of all flags in an *Observation* to 'unset.' Calling operation the *Observation*'s *is_flag_set* on any flag (identified by a value of enumeration *ObservationFlagKind*) shall always return *False* if this operation completed successfully and *set_flag* was never called yet on the specific *flag* after *reset_flags*.

If a flag is already in state 'unset,' this operation shall do nothing.

Return: The operation shall return RETCODE_OK if all flags were successfully transitioned to state 'unset' or were already in state 'unset,' RETCODE_NOT_MODIFIED if all flags were already in state 'unset,' RETCODE_ERROR if any error prevented all flags from being set to state 'unset.'

7.4.2.3.7 set_flag

This operation shall set the state of a flag (identified by a value of enumeration *ObservationFlagKind*) in an *Observation* to 'set.' Calling operation the *Observation*'s *is_flag_set* on the same flag shall always return *True* if this operation completed successfully and *unset_flag* was never called yet on the specific *flag* after *set_flag*.

If the specified value does not identify any valid flag of this *Observation,* the operation shall do nothing and return RETCODE_BAD_PARAMETER.

Parameter *flag*: The *Observation*'s flag to set.

Return: The operation shall return RETCODE_OK if the flag was successfully transitioned from state 'unset' to state 'set,' RETCODE_NOT_MODIFIED if the flag was already in 'set' state, RETCODE_BAD_PARAMETER if the specified *flag* does not exist, RETCODE_ERROR if any other error occurred while changing the flag's state.

7.4.2.3.8 set_value<T>

This operation shall store a value in an *Observation* for a specific *Field* of its *ObservableSchema.* If a *Field* with the specified *field_index* exists in the *ObservableSchema,* the operation shall store the value in the *Observation* and return RETCODE_OK. The operation shall provide all memory required to make a copy of the value and store it in the *Observation*. If an error occurs creating a copy of the value, the operation shall fail and return RETCODE_ERROR.

The operation shall expose a parameter T, which can be *DataType* or one its sub-classes of *DataValue,* specifying the type of value that is passed by the application and must be set in the *Observation*. If the specified type is incompatible with the requested *Field*'s data type and the value cannot be converted to the specified type, the operation shall fail and return RETCODE_PRECONDITION_NOT_MET.

If the specified *field_index* does not match the index of any of the *Field* instances contained in the *Observation*'s *ObservableSchema,* the operation shall fail and return RETCODE_BAD_PARAMETER.

If the operation fails, the value stored by the *Observation* for the specified *Field* shall not be modified. Calling *get_value* on the *Observation* with the same *field_index* shall return the same value before and after the operation is invoked.

Parameter *field_index*: The index of the *Field* of the *Observation*'s *ObservableSchema* whose value must be set.

Parameter *value*: The value to store in the *Observation*.

Return: The operation shall return RETCODE_OK if a *Field* with the specified *index* was found in the *Observation*'s *ObservableSchema* and the specified *value* was successfully copied into the *Observation*, RETCODE_BAD_PARAMETER if no matching *Field* was found for the specified *index*, RETCODE_PRECONDITION_NOT_MET if the specified *value* of type T cannot be converted to the type of the selected *Field*, RETCODE_ERROR if an error occurred while creating a copy of the *value* or any other type of error occurred.

7.4.2.3.9 unset_flag

This operation shall set the state of a flag (identified by a value of enumeration *ObservationFlagKind*) in an *Observation* to 'unset.' Calling operation the *Observation*'s *is_flag_set* on the same flag shall always return *False* if this operation completed successfully and *set_flag* was never called yet on the specific *flag* after *unset_flag*.

If the specified value does not identify any valid flag of this *Observation,* the operation shall do nothing and return RETCODE_BAD_PARAMETER.

Parameter *flag*: The *Observation*'s flag to unset.

Return: The operation shall return RETCODE_OK if the flag was successfully transitioned from state 'set' to state 'unset,' RETCODE_NOT_MODIFIED if the flag was already in 'unset' state, RETCODE_BAD_PARAMETER if the specified *flag* does not exist, RETCODE_ERROR if any other error occurred while changing the flag's state.

7.4.2.4 ObservationFlagKind

This enumeration shall define all the valid flags that may be manipulated in an *Observation* instance using its *is_flag_set, set_flag, unset_flag,* and *reset_flags* operations.

7.4.2.4.1 KEEP

This flag shall have the following meaning:

- 'unset' state: The *Observation* may be reused to store new values as soon as the instrumentation infrastructure has finished processing it and, if requested, distributing it to the *Instrumentation Domain.*
- 'set' state: The *Observation* shall not be reused to store new values until it is explicitly disposed.

7.4.2.4.2 LOCAL

This flag shall have the following meaning:

- 'unset' state: The *Observation* may be made visible outside of the *InstrumentationService* where it was generated and distributed to the *Instrumentation Domain.*
- 'set' state: The *Observation* shall not be distributed to the *Instrumentation Domain* and it should not be exposed outside the boundaries of the *InstrumentationService* where it was generated.

7.4.3 Data Collection Module

The Data Collection Module is comprised of the following classifiers:

- *ObservableScope*
- *ObservableObject*
- *DataProcessor*
- *DataProcessorArgs*
- *DataProcessorState*

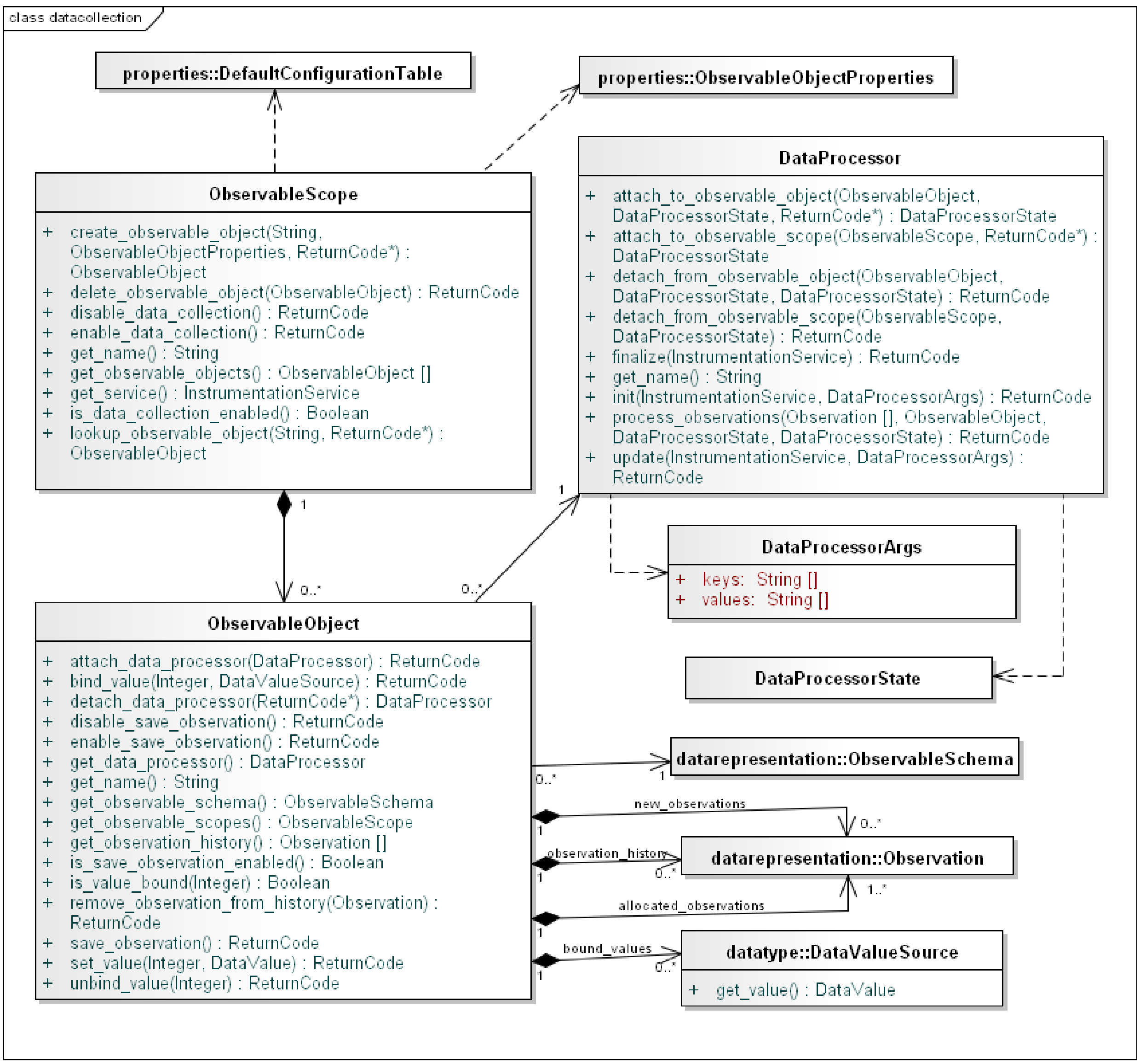

Figure 9 – Data Collection Module

7.4.3.1 ObservableScope

An *ObservableScope* shall define a single-threaded execution context where *Observation* instances generated by multiple *ObservableObject* instances are collected, processed, and distributed to the *Instrumentation Domain*.

An *ObservableScope* manages a set of *ObservableObject* instances. It collects new *Observation* objects created by each *ObservableObject* and it may invoke the services of *DataProcessor* instances attached to the *ObservableObject* instances.

An *ObservableScope* processes data per-*ObservableObject,* extracting new *Observation* values from each of its managed *ObservableObject* instances. Implementations of this specification may provide configurable policies to control how the *ObservableObject* are polled for new data values. These policies may control when an *ObservableObject* is polled for new *Observation* objects (e.g., periodically, upon notification, etc.) and may also enable configuration of advanced aspects, such as, collection ordering. Extensions to the *ObservableScopeProperties* structure and/or the *ObservableObjectProperties* structure may be defined by implementations to configure these policies. These extended policies are not specified in this specification. This specification only provides means to control how many *Observation* instances an *ObservableScope* may collect at most each time it processes an *ObservableObject.*

An *ObservableScope* guarantees that *Observation* objects obtained from two *ObservableObject* instances belonging to the same scope will never be processed concurrently. Collection, processing, and distribution of newly generated *Observation* objects shall occur within the single-threaded context associated with the *ObservableScope*. Independently of the policies controlling the frequency of these operations and the order in which *ObservableObject* instances are processed, the *ObservableScope* shall always process *Observation* objects from a single *ObservableObject* instance at a time.

An *ObservableScope* shall periodically perform the following operations for each *ObservableObject* instance:

- Collect new *Observation* objects, generated by that *ObservableObject,* that have not been processed yet in a collection ordered by increasing sequence number (as reported by each *Observation*'s *get_sequence_number* operation). If the size of the collection is limited[1], it shall include the oldest unprocessed *Observation* (the one with the lowest sequence number) and any following *Observation* fitting within the collection's boundaries.
- Invoke the *process_observations* operation of the *DataProcessor* attached to the *ObservableObject*, if a *DataProcessor* has been attached to the *ObservableObject.* The operation shall be invoked passing the following values to its parameters:
  - *observations*: the collection of newly extracted unprocessed *Observation* objects.
  - *object*: the *ObservableObject* currently being processed by the *ObservableScope*.
  - *scope_state*: the *DataProcessorState* returned when the *ObservableScope* invoked the *DataProcessor*'s *attach_to_observable_scope* operation.
  - *object_state*: the *DataProcessorState* returned when the *ObservableObject* invoked the *DataProcessor*'s *attach_to_observable_object* operation.
- Distribute any *Observation* object contained in the collection that does not have flag LOCAL in state 'set' to the *Instrumentation Domain*.
- Store any *Observation* object contained in the collection that has flag KEEP in state 'set' into the *ObservableObject*'s observation history, so that it shall become part of the collection returned by the *ObservableObject*'s *get_observation_history* operation.
- Return any *Observation* object contained in the collection that has flag KEEP in state 'unset' to the *ObservableObject* that generated it so that it may be reused to store new observations.

An *ObservableScope* shall guarantee that all *Observation* objects created by a successful invocation of an

[1] Recall that a limit on the maximum number of *Observation* values to collect per-*ObservableObject* is specified in the *ObservableScope*'s initialization properties.

ObservableObject's *save_observation* operation will be eventually collected and processed in the same order as they were generated by the *ObservableObject*.

The execution context of an *ObservableScope* and the collection of data from *ObservableObject* objects may be dynamically enabled or disabled. If data-collection is disabled, the execution context of an *ObservableScope* shall be stopped, interrupting the extraction of new *Observation* objects from any of its contained *ObservableObject* instances.

An *ObservableScope* shall not be required to provide a multi-thread safe interface. Only operations that explicitly state so may be safely invoked when the execution context of an *ObservableScope* is enabled and performing data-collection.

Implementations of this specification may decide to execute the data-collection operations performed by an *ObservableScope* on any thread, as long as the requirement for single-threaded execution context is satisfied (i.e., all the operations are performed by the same thread). This will allow an *ObservableScope*'s operation to be naturally supported on different threads than the application's ones, possibly limiting the overhead caused by instrumentation code added to the application. Mapping of *ObservableScope* instances and threads is not specified by this specification. Implementations may provide additional parameters in *InstrumentationServiceProperties* and/or *ObservableScopeProperties* to configure how many threads should be used by all the *ObservableScope* instances of an *InstrumentationService* and on which thread(s) each *ObservableScope* should be executed.

ObservableScope		
No Attributes		
Operations		
create_observable_object		ObservableGroup
	name	String
	properties	ObservableObjectProperties
	[out] retcode	ReturnCode
delete_observable_object		ReturnCode
	object	ObservableObject
disable_data_collection		ReturnCode
enable_data_collection		ReturnCode
get_name		String
get_observable_objects		ObservableObject[]
get_service		InstrumentationService
is_data_collection_enabled		ObservableScope
lookup_observable_object		ObservableObject
	name	String
	[out] retcode	ReturnCode

7.4.3.1.1 create_observable_object

This operation shall create or retrieve an *ObservableObject* instance.

The new *ObservableObject* shall have the specified *name* and it shall be initialized using the specified *properties.* If the creation succeeds, the operation shall return the newly created *ObservableObject* and set *retcode* to RETCODE_OK. If an *ObservableObject* instance with the same *name* already exists in the *ObservableScope*, the operation shall return the existing instance and set *retcode* to RETCODE_NOT_MODIFIED.

If the special value APPINST_OBSERVABLE_OBJECT_PROPERTIES_DEFAULT is passed as the *properties* argument, the operation shall use the value returned by the *get_default_observable_object_properties* operation on the *DefaultConfigurationTable* instance.

If the *observable_schema_name* attribute of the specified *properties* is 'nil' (as specified by the platform [PSM]) or no *ObservableSchema* instance is returned by invoking the *lookup_observable_schema* operation on the enclosing *InstrumentationService* with the specified name*,* the operation shall fail, returning 'nil' (as specified by the platform [PSM]) and set *retcode* to RETCODE_BAD_PARAMETER.

If the *data_processor_name* attribute of the specified *properties* contains a non-empty string, the operation shall retrieve the specified *DataProcessor,* using the enclosing *InstrumentationService*'s *lookup_data_processor* operation, and pass it to the newly created *ObservableObject*'s *attach_data_processor* operation.

If the *DataProcessor* cannot be found in the *InstrumentationService*, the operation shall not create an *ObservableObject*, return 'nil' (as specified by the platform [PSM]) and set *retcode* to RETCODE_BAD_PARAMETER.

If the *ObservableObject*'s *attach_data_processor* fails, the *create_observable_object* operation shall undo any side effects performed by operation, including deleting the newly created *ObservableObject* if one had been created. In this situation the operation shall return 'nil' (as specified by the platform [PSM]) and set *retcode* to RETCODE_ERROR.

If generation of *Observation* instances is enabled in the specified *properties* (by setting attribute *enable_save_observation* to *True*), after successfully creating the new *ObservableObject* and possibly attaching a *DataProcessor* to it*,* the operation shall invoke the *ObservableObject*'s *enable_save_observation* operation. If this operation returns a value other than RETCODE_OK, the operation shall undo any side effects performed by operation, including deleting the newly created *ObservableObject* if one had been created. In this situation the operation shall return 'nil' (as specified by the platform [PSM]) and set *retcode* to RETCODE_ERROR.

If the creation fails, the operation shall return 'nil' (as defined by the platform [PSM]) and set *retcode* to RETCODE_ERROR.

Parameter *name*: The name to assign to the new *ObservableObject*.

Parameter *properties*: An *ObservableObjectProperties* structure used to configure the new *ObservableObject*. If an *ObservableObject* with the specified *name* already exists, this Parameter is ignored. The value APPINST_OBSERVABLE_OBJECT_PROPERTIES_DEFAULT may be specified in order to use the default value returned by the *DefaultConfigurationTable*'s *get_default_observable_object_properties* operation.

Parameter *retcode*: The operation shall return RETCODE_OK if the new *ObservableObject* was successfully created. RETCODE_NOT_MODIFIED if an existing *ObservableObject* with the same *name* was found locally, RETCODE_PRECONDITION_NOT_MET if data-collection is enabled in the *ObservableScope*, RETCODE_BAD_PARAMETER if values specified for parameters *properties* and/or *name* were incorrect, and RETCODE_ERROR if a *DataProcessor* could not be attached to the *ObservableObject,* if the generation of *Observation* instances could not be enabled on the newly created *ObservableObject* of if there was any other type of error.

Return: The operation shall return an *ObservableObject* in case of success (RETCODE_OK or RETCODE_NOT_MODIFIED), or 'nil' (as defined by the platform [PSM]) in case of error.

7.4.3.1.2 delete_observable_object

This operation shall delete an existing *ObservableObject* instance from an *ObservableScope*.

If data-collection is enabled in the *ObsersableScope,* the operation shall do nothing and return RETCODE_PRECONDITION_NOT_MET.

If the *ObservableObject* has a DataProcessor attached, the operation shall do nothing and return RETCODE_PRECONDITION_NOT_MET.

If the *ObservableObject* instance is successfully deleted, the operation shall return RETCODE_OK.

The operation shall fail and return RETCODE_BAD_PARAMETER if the *ObservableScope* performing the operation did not create the *ObservableObject* to be deleted.

Parameter *schema*: the *ObservableObject* to delete.

Return: The operation shall return RETCODE_OK if the *ObservableObject* was successfully deleted from the *ObservableScope*, RETCODE_PRECONDITION_NOT_MET if data collection is currently enabled in the *ObservableScope* or a *DataProcessor* could not be detached from the *ObservableObject,* RETCODE_BAD_PARAMETER if the specified *ObservableObject* was not created by this *ObservableScope*, and RETCODE_ERROR if any other type of error occurred.

7.4.3.1.3 disable_data_collection

This operation shall disable data-collection in an *ObservableScope,* deactivating its single-threaded execution context. If data-collection is successfully disabled, the operation shall return RETCODE_OK.

If data-collection and the *ObservableScope*'s execution context are already disabled, the operation shall do nothing and return RETCODE_NOT_MODIFIED.

If any error prevents data-collection from being disabled, the operation shall return RETCODE_ERROR.

Return: The operation shall return RETCODE_OK if data-collection was successfully disabled in the *ObservableScope,* RETCODE_NOT_MODIFIED if data-collection was already disabled, RETCODE_ERROR if any error prevented data-collection from being disabled.

7.4.3.1.4 enable_data_collection

This operation shall enable data-collection in an *ObservableScope,* activating its single-threaded execution context. If data-collection is successfully enabled, the operation shall return RETCODE_OK.

If data-collection and the *ObservableScope*'s execution context are already enabled, the operation shall do nothing and return RETCODE_NOT_MODIFIED.

If any error prevents data-collection from being enabled, the operation shall return RETCODE_ERROR.

Return: The operation shall return RETCODE_OK if data-collection was successfully enabled in the *ObservableScope,* RETCODE_NOT_MODIFIED if data-collection was already enabled, RETCODE_ERROR if any error prevented data-collection from being enabled.

7.4.3.1.5 get_name

This operation shall return a string containing the name of an *ObservableScope* instance.

This operation may be safely invoked while data-collection is enabled in the *ObservableScope.*

Return: The operation shall return an unmodifiable, non-empty, string.

7.4.3.1.6 get_observable_objects

This operation shall return a collection of all *ObservableObject* instances that have been created in an *ObservableScope*. An empty collection shall be returned if the *ObservableScope* does not contain any *ObservableObject* yet.

This operation may be safely invoked while data-collection is enabled in the *ObservableScope.*

Return: The operation shall return a collection of *ObservableObject* instances.

7.4.3.1.7 get_service

This operation shall return the *InstrumentationService* that created an *ObservableScope* instance.

This operation may be safely invoked while data-collection is enabled in the *ObservableScope.*

Return: The operation shall return an *InstrumentationService* instance.

7.4.3.1.8 is_data_collection_enabled

This operation shall check the current state of data-collection in an *ObservableScope.*

Return: The operation shall return *True* if the *ObservableScope*'s execution context is currently active and data-collection is enabled*, False* otherwise.

7.4.3.1.9 lookup_observable_object

This operation shall search among the *ObservableObject* instances created in an *ObservableScope* and return the one identified by the specified name.

If an instance by that name is found, the operation shall return it and set *retcode* to RETCODE_OK.

If no matching instance is found, the operation shall return 'nil' (as defined by the platform [PSM]) and set *retcode* to RETCODE_ERROR.

This operation may be safely invoked while data-collection is enabled in the *ObservableScope.*

Parameter *name*: The name of the *ObservableObject* to lookup.

Parameter *retcode*: The operation shall return RETCODE_OK if a matching *ObservableObject* was found in the *ObservableScope*, and RETCODE_ERROR if no matching instance was found or there was any other type of error.

Return: The operation shall return an *ObservableObject* in case of success (RETCODE_OK), or 'nil' (as defined by the platform [PSM]) otherwise.

7.4.3.2 ObservableObject

An *ObservableObject* represents a source of instrumented application data. Applications may generate samples of instrumented data during their execution by using the interface of *ObservableObject* to associate application data-objects with any field of an *ObservableSchema* and then capture a snapshot of these values in an *Observation*.

An *ObservableObject* shall be associated with a single *ObservableSchema,* which describes the structure and nature of data that can be provided by applications through that *ObservableObject*.

Applications shall use the *ObservableObject* to specify values contained in each field of an *Observation* and then invoke the *ObservableObject*'s *save_observation* operation to store these values in an *Observation*, which may be collected by the *ObservableScope* managing the *ObservableObject.*

Applications may provide data for a specific field using two types of operations:

- A setter operation, called *set_value,* which shall store the specified value (of one of the supported sub-types of *DataValue*) in internal buffers of the *ObservableObject.*
- A binding operation, called *bind_value,* which shall associate the values of a field of the *ObservableObject* with an external data source (represented by a *DataValueSource* instance) contained in the application's data space.

An *ObservableObject* shall maintain in its internal state any value successfully stored using the *set_value* operation and any reference to external date sources provided using the *bind_value* operation. This allows the values of fields in an *ObservableObject* to be specified incrementally by the application, before a snapshot is captured in an *Observation* using the *save_observation* operation.

When *save_observation* is invoked, the values associated with each field by the *ObservableObject* shall be copied to the corresponding fields of the *Observation object* being generated. The value of fields that were bound by *bind_value* shall be determined by sampling the associated external data source at the time the operation *save_observation* is executed.

An *ObservableObject* shall provide configuration parameters to control the number of *Observation* objects it may allocate during the application's execution. This limit controls the amount of resources used to store instrumentation data and offers a trade-off between performance and accuracy of the observed behavior. An *ObservableObject* shall reuse any *Observation* objects that its *ObservableScope* returns to it, after processing and distribution, once the maximum number of *Observation* objects have been allocated. The *save_observation* operation may fail if all available *Observation* objects have been already allocated and the associated *ObservableScope* is still currently processing all of them. Application may set a maximum amount of time (ranging from 0 to infinity) that the *save_observation* operation may block execution of an application waiting for an *Observation* object to be made available, before failing.

Generation of *Observation* objects may also be dynamically enabled and disabled by an application. When generation of *Observation* objects is disabled, the *save_observation* operation shall have no effect.

The operations of an *ObservableObject* do not offer safety with respect to invocation from multiple threads of execution for the generation of *Observation* objects. An *ObservableObject*'s *save_observation* shall be used to generate data only from a single thread at a time.

An *ObservableObject* shall allow invocation of its *enable_observation* and *disable_observation* operations from threads concurrent to any other thread using the *ObservableObject* to generate data. The *ObservableObject* shall guarantee that the change of status will be eventually propagated to the thread generating data.

ObservableObject		
No Attributes		
Operations		
attach_data_processor		ReturnCode
	processor	DataProcessor
bind_value <T:DataValue>		ReturnCode
	field_index	Integer
	value_source	DataValueSource<T>
detach_data_processor		DataProcessor
	[out] retcode	ReturnCode
disable_save_observation		ReturnCode
enable_save_observation		ReturnCode
get_data_processor		DataProcessor
get_name		String
get_observable_schema		ObservableSchema
get_observable_scope		ObservableScope
get_observation_history		Observation[]
is_save_observation_ enabled		Boolean
is_value_bound		Boolean

	field_index	Integer
remove_observation_ from_history		ReturnCode
	observation	Observation
save_observation		ReturnCode
set_value <T:DataValue>		ReturnCode
	field_index	Integer
	value	T
unbind_value		ReturnCode
	field_index	Integer

7.4.3.2.1 attach_data_processor

This operation shall attach a *DataProcessor* to an *ObservableObject.* The *DataProcessor* shall be used by the *ObservableScope* that manages the *ObservableObject* to process all *Observation* objects generated by the *ObservableObject*.

If the same *DataProcessor* is already attached to the *ObservableObject*, the operation shall return RETCODE_NOT_MODIFIED.

If another *DataProcessor* is already attached to the *ObservableObject,* the operation shall fail and return RETCODE_PRECONDITION_NOT_MET.

If this is the first *ObservableObject* instance attached to this *DataProcessor* within the enclosing *ObservableScope*, the operation shall invoke *attach_to_observable_scope* operation of the *DataProcessor,* with the following parameters:

- *scope*: The *ObservableScope* that created the *ObservableObject.*

The *DataProcessorState* value returned by the *DataProcessor* shall be stored by the *ObservableScope* if it is different than 'nil' (as specified by the platform [PSM]).

If the *attach_to_observable_scope* operation returns a return code different from RETCODE_OK, the *attach_data_processor* operation shall fail and return RETCODE_ERROR.

After having successfully attached the *DataProcessor* to the *ObservableObject's ObservableScope,* the operation shall invoke the *DataProcessor*'s *attach_to_observable_object* operation with the following parameters:

- *object*: the *ObservableObject* performing the *attach_data_processor* operation.
- *scope_state*: the *DataProcessorState* value returned when the *ObservableScope* invoked the *DataProcessor*'s *attach_to_observable_scope* operation.

The *DataProcessorState* value returned by the *DataProcessor* shall be stored by the *ObservableObject* if it is different than 'nil' (as specified by the platform [PSM]).

If the *attach_to_observable_object* operation returns a return code different from RETCODE_OK, the *attach_data_processor* operation shall fail and return RETCODE_ERROR.

If all operations succeed with return code RETCODE_OK, the *attach_data_processor* operation shall return RETCODE_OK.

This operation may be safely invoked while data-collection is enabled in the enclosing *ObservableScope*. Implementations shall guarantee that the newly attached *DataProcessor* will be eventually used to process the *Observation* instances generated by the *ObservableObject* and collected by the *ObservableScope*'s execution context.

Parameter *processor*: The *DataProcessor* to be attached to the *ObservableObject.*

Return: The operation shall return RETCODE_OK if the *DataProcessor* was successfully attached to the *ObservableObject* (and possibly the enclosing *ObservableScope*), RETCODE_ERROR if any of the *DataProcessor*'s *attach_to_observable_scope* and *attach_to_observable_object* failed, RETCODE_ERROR if any other error prevented the *DataProcessor* from being attached to the *ObservableObject.*

7.4.3.2.2 bind_value<T>

This operation shall create a binding inside an *ObservableObject* instance*,* between an external data source of type T (represented by a *DataValueSource* instance of type T) and one of the fields of an *ObservableObject*'s *ObservableSchema*. The value reported by the *FieldValueSource* shall be sampled by the *ObservableObject*'s *save_observation* operation and copied to any new *Observation*.

If the specified *index* does not match the index of any of the *Field* instances contained in the *ObservableObject*'s *ObservableSchema,* the operation shall fail and return RETCODE_BAD_PARAMETER.

If the specified T data type is not compatible with the value of the selected *Field*, the operation shall fail and return RETCODE_ERROR.

The operation shall release any memory that might have been previously required to store a value for the *Field* specified using the *set_value* operation.

Parameter *field_index*: An integer identifying the *index* of the *ObservableSchema*'s *Field* to which the *DataValueSource* must be bound.

Parameter *source*: a *FieldValueSource* instance that will be bound to the selected *Field.*

Return: The operation shall return RETCODE_OK if the specified *source* was successfully bound in the *ObservableObject* and is now associated with the specified *Field* of its *ObservableSchema,* RETCODE_BAD_PARAMETER if the specified *index* does not identify any *Field* of the *ObservableSchema*, RETCODE_ERROR if the specified *source* is of an incompatible data type or if any other error prevented the *ObservableObject* from binding the specified *source*.

7.4.3.2.3 detach_data_processor

This operation shall detach a *DataProcessor* from an *ObservableObject,* causing the enclosing *ObservableScope* to stop using the *DataProcessor* to process any *Observation* instance generated by the *ObservableObject*.

If the *ObservableObject* has no *DataProcessor* instance currently attached to it, the operation shall do nothing and return RETCODE_NOT_MODIFIED.

The operation shall invoke the *DataProcessor's detach_from_observable_object* operation, passing the following parameters:

- *object*: the *ObservableObject* performing the *detach_data_processor* operation.
- *scope_state*: the *DataProcessorState* value returned when the enclosing *ObservableScope* invoked the *DataProcessor*'s *attach_to_observable_scope* operation.
- *object_state:* the *DataProcessorState* value returned when the *ObservableObject invoked the DataProcessor*'s *attach_to_observable_object* operation.

If this operation returns a value other than RETCODE_OK, the *detach_data_processor* operation shall fail and return *RETCODE_ERROR*.

If this is the last *ObservableObject* instance the *DataProcessor* is attached to within the enclosing *ObservableScope,* the operation shall invoke the *DataProcessor*'s *detach_from_observable_scope* operation with the following parameters:

- *scope*: the *ObservableScope* containing the *ObservableObject* performing the *detach_data_processor* operation.
- *scope_state*: the *DataProcessorState* value returned when the *ObservableScope* invoked the *DataProcessor's attach_to_observable_scope* operation.

If this operation returns a value other than RETCODE_OK, the *detach_data_processor* operation shall fail and return *RETCODE_ERROR*.

Implementations are not required to support invocation of this operation while data-collection is enabled in the enclosing *ObservableScope.* Undetermined behavior may arise if a *DataProcessor* instance is detached from an *ObservableObject* while the *ObservableScope* is processing it (and possibly using the *DataProcessor*'s services).

Parameter *retcode*: The operation shall return RETCODE_OK if a *DataProcessor* instance was detached from the *ObservableObject,* RETCODE_NOT_MODIFIED if no *DataProcessor* instance was previously attached to the *ObservableObject*, RETCODE_ERROR if either of the *DataProcessor*'s *detach_from_observable_object* and *detach_from_observable_scope* operations failed, or if any other type of error occurred.

Return: upon success (RETCODE_OK), the operation shall return a *DataProcessor* instance. In any other case, 'nil' (as specified by the platform [PSM]) shall be returned.

7.4.3.2.4 disable_save_observation

This operation shall disable generation of *Observation* objects from an *ObservableObject.* If generation is successfully disabled, the operation shall return RETCODE_OK.

If generation of *Observation* objects is already disabled, the operation shall do nothing and return RETCODE_NOT_MODIFIED.

If any error prevents the generation of *Observation* instances from being disabled, the operation shall return RETCODE_ERROR

Return: The operation shall return RETCODE_OK if generation of *Observation* objects was successfully disabled in the *ObservableObject,* RETCODE_NOT_MODIFIED if generation was already disabled, RETCODE_ERROR if any error prevented generation from being disabled.

7.4.3.2.5 enable_save_observation

This operation shall enable the generation of *Observation* objects in an *ObservableObject.* If generation is successfully enabled, the operation shall return RETCODE_OK.

If generation of *Observation* objects is already enabled, the operation shall do nothing and return RETCODE_NOT_MODIFIED.

If any error prevents generation from being enabled, the operation shall return RETCODE_ERROR.

Return: The operation shall return RETCODE_OK if generation of *Observation* objects was successfully enabled in the *ObservableObject,* RETCODE_NOT_MODIFIED if generation was already enabled, and RETCODE_ERROR if any error prevented generation from being enabled.

7.4.3.2.6 get_name

This operation shall return a string containing the name of an *ObservableObject* instance.

Return: The operation shall return an unmodifiable, non-empty, string.

7.4.3.2.7 get_observable_schema

This operation shall return the *ObservableSchema* associated with the *ObservableObject*.

Return: The operation shall return the *ObservableSchema* associated with the *ObservableObject.*

7.4.3.2.8 get_observable_scope

This operation shall return the *ObservableScope* that created an *ObservableObject* instance.

Return: The operation shall return the *ObservableScope* that created the *ObservableObject*.

7.4.3.2.9 get_observation_history

This operation shall provide access to an ordered collection of *Observation* objects that have been stored in an *ObservableObject*'s observation history. These *Observation* objects were generated by the *ObservableObject,* collected and processed by its *ObservableScope,* and then stored by a *DataProcessor* by using the KEEP flag.

Observation objects contained in the returned collection shall be ordered by increasing value of sequence number.

Only a *DataProcessor* accessing the *ObservableObject* within the execution context of an *ObservableScope* (i.e., in its *attach_to_observable_object*, *detach_from_observable_object* and *process_observations* operations) may safely invoke this operation. Instrumented applications may only safely invoke this operation directly if data-collection is disabled in the *ObservableScope* that contains the *ObservableObject.*

The returned collection shall not be modified. *Observation* objects contained in the collection may be individually removed (and returned to the *ObservableObject* for storing new values) by using the *ObservableObject*'s *remove_observation_from_history* operation. In order to retrieve an updated version of the collection (without the removed element), the *get_observation_history* operation needs to be invoked again. It shall be possible to safely iterate over the value returned by *get_observation_history* and remove *Observation* instances using *remove_observation_from_history*.

Return: An unmodifiable collection of *Observation* instances. The collection may be empty if no *Observation* has been saved in the *ObservableObject*'s observation history.

7.4.3.2.10 is_save_observation_enabled

This operation shall check the current state of the generation of *Observation* instances in an *ObservableObject.*

Return: The operation shall return *True* if the generation is enabled in the *ObservableObject* and the *save_observation* operation may be used to generate new *Observation* instances*, False* otherwise.

7.4.3.2.11 is_value_bound

This operation shall check whether a *Field* of an *ObservableObject*'s *ObservableSchema* is currently bound to a *DataValueSource.*

If the specified *index* does not match the index of any of the *Field* instances contained in the *ObservableObject*'s *ObservableSchema,* the operation shall fail and return *False*.

Parameter *field_index*: An integer identifying the *index* of the *ObservableSchema*'s *Field* to which the value must be associated.

Return: The operation shall return *True* if the specified *Field* is currently bound to a *DataValueSource, False* if the *Field* is not bound or an error occurred while determining the *Field*'s status.

7.4.3.2.12 remove_observation_from_history

This operation shall remove an *Observation* from an *ObservableObject*'s observation history. The *Observation* shall be disposed and the values stored for each *Field* reverted to the defaults specified by each data type, so that it may be used by future invocations of the *save_observation* operation.

If the *Observation* is not currently stored in the *ObservableObject*'s observation history or if the *ObservableObject* did not generate it, the operation shall fail and return RETCODE_BAD_PARAMETER.

If an error occurs while removing the *Observation* from the observation history or while disposing it, the operation shall fail and return RETCODE_ERROR.

It shall be possible to safely invoke this operation while iterating over the elements of the collection returned by operation *get_observation_history*.

Return: The operation shall return RETCODE_OK if the specified *Observation* was successfully removed and returned to the *ObservableObject* for reuse, RETCODE_BAD_PARAMETER if the *Observation* was not in the observation history or the *ObservableObject* did not generate it, RETCODE_ERROR if any error occurred while removing or disposing the *Observation* or any other part of the operation's implementation.

7.4.3.2.13 save_observation

This operation shall generate an *Observation* object containing the values that an *ObservableObject* currently associates with the fields of its *ObservableSchema.*

If generation of *Observation* objects is disabled in the *ObservableObject*, the operation shall do nothing and return RETCODE_NOT_MODIFIED.

If enabled in the *ObservableObject*'s initialization properties, the operation shall collect a time-stamp of the time at which the operation started and store it in as a *UTCTime* value, which will be copied to the new *Observation*.

The operation shall retrieve an *Observation* object to store the values by either:

- Reusing an *Observation* object previously allocated by the *ObservableObject* that has not been used to store values yet or that has already been processed by the *ObservableObject*'s *ObservableScope* and returned to the *ObservableObject.*
- Dynamically allocating a new *Observation* object, if the maximum number of *Observation* objects that can be allocated by the *ObservableObject* has not been reached yet.

The operation may block the execution for a configurable amount of time (specified in the *ObservableObject*'s initialization properties) in order to allow the enclosing *ObservableScope* to complete processing of existing *Observation* objects. If no *Observation* is made available during this time (or no wait time is allowed in case of no *Observation* instance available), *save_observation* will fail and return RETCODE_PRECONDITION_NOT_MET.

For each *Field* instance contained in the *ObservableObject*'s *ObservableSchema*, the *Observation* instance shall contain:

- The value associated to the *Field* by the latest successful invocations of the *ObservableObject*'s *set_value* or *bind_value* operations.
- The default value specified by the *Field*'s data type.

For bound fields, the operation shall sample the value of each field by invoking its bound *DataValueSource*'s *get_value* operation. If an error occurs during the sampling of any bound *DataValueSource,* the operation shall dispose the *Observation* instance, restoring default values for each *Field,* and return RETCODE_ERROR.

The operation shall retrieve any memory required to copy the values into the *Observation* object. If an error occurs during the copy of the value of any *Field*, the operation shall dispose the *Observation* object, restoring default values for each *Field* and releasing any memory previously acquired, and return RETCODE_ERROR.

The operation shall assign a sequence number to the new *Observation*, starting from 1 for each *ObservableObject*.

After successfully generating the new *Observation*, the operation shall add it to a queue associated with the *ObservableObject,* where the enclosing *ObservableScope* might later extract it for processing, and notify the *ObservableScope* of the new *Observation.* If any error occurs in adding the *Observation* to the queue or notifying the *ObservableScope* of its generation, the operation shall dispose the *Observation* object, restoring default values for each *Field* and releasing any memory previously acquired, and return RETCODE_ERROR.

After successfully notifying the new *Observation* to the *ObservableScope*, the operation shall increment the sequence number assigned to the *Observation* and store the value internally to the *ObservableObject* for the next invocation of *save_observation*.

Return: The operation shall return RETCODE_OK if a new *Observation* containing the values of the *ObservableObject* was successfully generated and notified to the enclosing *ObservableScope*, RETCODE_NOT_MODIFIED if the generation of *Observation* objects is disabled in the *ObservableObject,* RETCODE_PRECONDITION_NOT_MET if no *Observation* object could be retrieved or be allocated to store the values, RETCODE_ERROR if any error occurred during the generation of the *Observation,* its notification to the *ObservableScope* or any other part of the operation's implementation.

7.4.3.2.14 set_value<T>

This operation shall store a value of type T inside the memory of an *ObservableObject,* associating it with one of the fields of the *ObservableObject*'s *ObservableSchema*. The value shall be copied to any new *Observation* created by the *ObservableObject*'s *save_observation* operation, until a new one is supplied through a new invocation of this operation or the field is bound to an external source using the *bind_value* operation.

If the specified *index* does not match the index of any of the *Field* instances contained in the *ObservableObject*'s *ObservableSchema,* the operation shall fail and return RETCODE_BAD_PARAMETER.

If the specified T data type is not compatible with the value of the selected *Field* and an error occurred while converting it, the operation shall fail and return RETCODE_ERROR.

If the specified *Field* has been currently bound to an external *DataValueSource* by using the *bind_value* operation, the operation shall fail and return RETCODE_PRECONDITION_NOT_MET.

The operation shall retrieve any memory required to store a copy of the specified value independently from the application and to make it available to following invocations of the *ObservableObject*'s *save_observation* operation. The memory may be dynamically allocated or accessed from pre-allocated buffers.

Parameter *field_index*: An integer identifying the *index* of the *ObservableSchema*'s *Field* to which the value must be associated.

Parameter *value*: the value to set in the *ObservableObject.*

Return: The operation shall return RETCODE_OK if the specified value was successfully copied into the *ObservableObject* and is now associated with the specified *Field* of its *ObservableSchema,* RETCODE_BAD_PARAMETER if the specified *index* does not identify any *Field* of the *ObservableSchema*, RETCODE_ERROR if the specified *value* is of an incompatible data type or the *Field* is currently bound to a *DataValueSource* or if any other error prevented the *ObservableObject* from storing the specified *value*.

7.4.3.2.15 unbind_value

This operation shall delete an existing binding between a *DataValueSource* and one of the *Fields* of an *ObservableObject*'s *ObservableSchema.* The default value of the *Field*'s data type shall be copied into any new *Observation* created by the *ObservableObject*'s *save_observation* operation, until a new value is supplied using the *set_field* operation or the field is bound to an external source using the *bind_value* operation.

If the specified *index* does not match the index of any of the *Field* instances contained in the *ObservableObject*'s *ObservableSchema,* the operation shall fail and return RETCODE_BAD_PARAMETER.

If the selected *Field* is not currently bound to any *FieldValueSource* in the *ObservableObject,* the operation shall do nothing and return RETCODE_NOT_MODIFIED.

If the *Field* is bound to a *DataValueSource*, the operation shall discard the *DataValueSource* and revert the value of the *Field* in the *ObservableObject* to its data type's default value. The operation shall retrieve any memory necessary to store this value.

Parameter *field_index*: An integer identifying the *index* of the *ObservableSchema*'s *Field* to which the value must be associated.

Return: The operation shall return RETCODE_OK if the specified *Field* was successfully unbound from its *DataValueSource*, RETCODE_BAD_PARAMETER if the specified *index* does not identify any *Field* of the *ObservableSchema*, RETCODE_NOT_MODIFIED if the *Field* was not bound to any *DataValueSource*, RETCODE_ERROR if any other error prevented the *Field* from being unbound.

7.4.3.3 DataProcessor

The *DataProcessor* interface shall provide support for the implementation of custom processing of *Observation* objects collected by an *ObservableScope.* Applications shall be able to create instances of *DataProcessor* within an *InstrumentationService,* using its *create_data_processor* operation, and then attach these instances to any

ObservableObject contained in the *InstrumentationService,* by means of the *ObservableObject*'s *attach_data_processor* operation.

An *ObservableScope* shall use the *DataProcessor* that has been attached to an *ObservableObject* to process all *Observation* objects collected from that *ObservableObject.* As detailed in 7.4.3.1 ObservableScope, the *ObservableScope* shall invoke the *DataProcessor*'s *process_observations* every time *Observation* objects are collected from an *ObservableObject* which that *DataProcessor* is currently attached to.

In order to facilitate the implementation of custom processing functionalities, the instrumentation infrastructure shall provide a *DataProcessor* with convenient *attach* and *detach* callback notifications that allow the *DataProcessor* to associate (and conversely delete) custom processing state associated to any *ObservableScope* and/or *ObservableObject.*

DataProcessor		
No Attributes		
Operations		
attach_to_ observable_object		DataProcessorState
	obj	ObservableObject
	scope_state	DataProcessorState
	[out] retcode	ReturnCode
attach_to_ observable_scope		DataProcessorState
	scope	ObservableScope
	[out] retcode	ReturnCode
detach_from_ observable_object		ReturnCode
	obj	ObservableObject
	scope_state	DataProcessorState
	obj_state	DataProcessorState
detach_from_ observable_scope		ReturnCode
	scope	ObservableScope
	scope_state	DataProcessorState
finalize		ReturnCode
	service	InstrumentationService
get_name		String
initialize		ReturnCode
	service	InstrumentationService
	init_args	DataProcessorArgs
process_observations		ReturnCode
	observations	Observation[]

	obj	ObservableObject
	scope_state	DataProcessorState
	obj_state	DataProcessorState
update		ReturnCode
	service	InstrumentationService
	args	DataProcessorArgs

7.4.3.3.1 attach_to_observable_object

This operation shall be invoked only when a *DataProcessor* is attached to an *ObservableObject* (see 7.4.3.2.1 attach_data_processor). The invocation of this operation shall notify the *DataProcessor* that it is going to be attached to an *ObservableObject* such that it has the opportunity to reserve any resource it may require.

The *DataProcessor* shall return 'nil' (as defined by the platform [PSM]) and set *retcode* to RETCODE_OK, if it does not need to associate any specific state with the *ObservableObject*. Alternatively, the *DataProcessor* shall return a value of type *DataProcessorState* instead, if explicit state for this *ObservableObject* needs to be maintained across invocations of its *process_observations* operation.

The value returned by this operation shall be stored by the *ObservableObject* and passed to later invocations of the *DataProcessor*'s operations.

Parameter *obj*: The *ObservableObject* that the *DataProcessor* is being attached to.

Parameter *scope_state*: The custom processing state associated with the *ObservableScope* that was returned by the *DataProcessor*'s interface *attach_to_observable_scope*.

Parameter *retcode*: The operation shall return RETCODE_OK if the *DataProcessor* was successfully attached to the *ObservableObject*, any other return value shall be interpreted as an error and it is up to implementations to define the specific semantics of each one.

Return: the operation shall return 'nil' (as defined by the platform [PSM]) if no state should be associated with the *ObservableObject* or a value of type *DataProcessorState* representing processing state that shall be stored by the instrumentation infrastructure.

7.4.3.3.2 attach_to_observable_scope

This operation shall be invoked when a *DataProcessor* is attached to an *ObservableObject* and it is not yet attached to any other *ObservableObject* instance contained in the same *ObservableScope* (see 7.4.3.2.1 attach_data_processor). The invocation of this operation shall notify the *DataProcessor* that it is going to be attached to an *ObservableObject* within the specified *ObservableScope* such that it has the opportunity to reserve any resource it may require.

The *DataProcessor* shall return 'nil' (as defined by the platform [PSM]) and set *retcode* to RETCODE_OK, if it does not need to associate any specific state with the *ObservableScope*. The *DataProcessor* shall return a value of type *DataProcessorState* instead, if explicit state for this *ObservableScope* needs to be maintained across invocations of its *process_observations* operation.

The value returned shall be stored by the *ObservableScope* and passed to later invocations of the *DataProcessor*'s operations.

Parameter *scope*: The *ObservableScope* that the *DataProcessor* is being attached to.

Parameter *retcode*: The operation shall return RETCODE_OK if the *DataProcessor* was successfully attached to the *ObservableScope*, any other return value shall be interpreted as an error and it is up to implementations to define the specific semantics of each one.

Return: the operation shall return 'nil' (as defined by the platform [PSM]) if no state should be associated with the

ObservableScope or a value of type *DataProcessorState* representing processing state that shall be stored by the instrumentation infrastructure.

7.4.3.3.3 detach_from_observable_object

This operation shall be invoked when a *DataProcessor* is detached from an *ObservableObject* instance by using the *ObservableObject*'s *detach_data_processor* operation. The implementation of this operation by the *DataProcessor* shall release any resource that had been previously reserved by the *DataProcessor* to create processing state associated with the *ObservableObject,* contained in the *object_state* parameter.

The operation shall return RETCODE_OK if freeing of resources was completed successfully.

Parameter *obj*: The *ObservableObject* that the *DataProcessor* is being detached from.

Parameter *scope_state*: The custom processing state associated with the *ObservableScope* that was returned by the *DataProcessor'*s interface *attach_to_observable_scope*.

Parameter *obj_state*: The custom processing state associated with the *ObservableObject* that was returned by the *DataProcessor*'s operation *attach_to_observable_object*.

Return: The operation shall return RETCODE_OK if the *DataProcessor* was successfully attached to the *ObservableObject*, any other return value shall be interpreted as an error and it is up to implementations to define the specific semantics of each one.

7.4.3.3.4 detach_from_observable_scope

This operation shall be invoked when a *DataProcessor* is detached from the last *ObservableObject* instance it was attached to within a certain *ObservableScope.* The implementation of this operation by the *DataProcessor* shall release any resource that had been previously reserved by the *DataProcessor* to create processing state associated with the *ObservableScope,* contained in the *scope_state* parameter.

The operation shall return RETCODE_OK if freeing of resources was completed successfully.

Parameter *scope*: The *ObservableScope* that the *DataProcessor* is being detached from.

Parameter *scope_state*: The custom processing state associated with the *ObservableScope* that was returned by the *DataProcessor*'s operation *attach_to_observable_scope*.

Return: The operation shall return RETCODE_OK if the *DataProcessor* was successfully attached to the *ObservableScope*, any other return value shall be interpreted as an error and it is up to implementations to define the specific semantics of each one.

7.4.3.3.5 finalize

This operation shall be invoked by an *InstrumentationService*'s *delete_data_processor* operation before the *DataProcessor* is de-allocated (see 7.4.1.2.5 delete_data_processor). The invocation of this operation shall notify a *DataProcessor* instance that it is going to be deleted by the *InstrumentationService* that created it.

The operation shall finalize and release any resource previously required by the *DataProcessor* to carry out its operations.

This operation shall return RETCODE_OK if all resources used by the *DataProcessor* were successfully finalized. The *InstrumentationService* shall interpret any other return code value as an error, but implementation may return any valid value defined by type *ReturnCode.*

Parameter *service*: The *InstrumentationService* instance deleting the *DataProcessor.*

Return: The operation shall return RETCODE_OK if the finalization of the *DataProcessor*'s resources was successful, any other return value shall be interpreted as an error and it is up to implementations to define the specific semantics of each one.

7.4.3.3.6 get_name

This operation shall return a string containing the name of a *DataProcessor*.

Return: The operation shall return an unmodifiable non-empty string.

7.4.3.3.7 initialize

This operation shall be invoked by an *InstrumentationService*'s *create_data_processor* operation after the *DataProcessor* has been allocated (see 7.4.1.2.2 create_data_processor). The invocation of this operation shall notify a *DataProcessor* instance that it is being created within a certain *InstrumentationService* and provide custom configuration parameters, in the form of a *DataProcessorArgs* value specified in the *DataProcessor*'s initialization properties passed to the *InstrumentationService*'s *create_data_processor* operation.

The operation shall initialize any resource required by the *DataProcessor* to carry out its operations within the context of the enclosing *InstrumentationService*.

This operation shall return RETCODE_OK if the initialization was completed successfully. The *InstrumentationService* shall interpret any other return code value as an error, but implementation may return any valid value defined by type *ReturnCode.*

Parameter *service*: The *InstrumentationService* instance creating the *DataProcessor*.

Return: The operation shall return RETCODE_OK if the initialization of the *DataProcessor* was successful, any other return value shall be interpreted as an error and it is up to implementations to define the specific semantics of each one.

7.4.3.3.8 process_observations

This operation shall process a collection of *Observation* objects generated by an *ObservableObject* to which the *DataProcessor* is attached. The operation may manipulate the values contained in an *Observation,* using the *Observation*'s *get_value* and *set_value* operations, and control its distribution and life cycle, by setting and unsetting the flags contained in the *Observation*.

The operation shall set flag LOCAL to 'set' state for each *Observation* instance that must not be distributed outside of the *InstrumentationService*.

The operation shall set flag KEEP to 'set' state for each *Observation* instance that must not be returned to the original *ObservableObject* that created it, so that it may be used to store new values, but instead added to the *ObservableObject*'s observation history.

This operation may safely iterate over the observation history of each *ObservableObject* it processes by using the *ObservableObject*'s *get_observation_history*. As explained in 7.4.3.2.12 remove_observation_from_history, elements may be removed from the observation history if passed to the *remove_from_observation_history* operation.

Parameter *observations*: a collection of *Observation* instances that have not been processed yet. It may be empty.

Parameter *obj*: The *ObservableObject* that generated the *Observation* instances to be processed.

Parameter *scope_state*: The custom processing state associated with the *ObservableScope* that was returned by the *DataProcessor*'s interface *attach_to_observable_scope*.

Parameter *obj_state*: The custom processing state associated with the *ObservableObject* that was returned by the *DataProcessor*'s interface *attach_to_observable_object*.

Return: The operation shall return RETCODE_OK if the *DataProcessor* successfully processed the *ObservableObject*'s *Observation* instances, any other return value shall be interpreted as an error and it is up to implementations to define the specific semantics of each one.

7.4.3.3.9 update

This operation shall be invoked by the instrumentation infrastructure to request the reconfiguration of an existing *DataProcessor* instance. The *DataProcessor* shall adapt its configuration and use the specified *DataProcessorArgs* value for its parameters.

This operation shall return RETCODE_OK if the *DataProcessor*'s configuration was successfully updated. The *InstrumentationService* shall interpret any other return code value as an error, but implementation may return any valid value defined by type *ReturnCode.*

Parameter *service*: The *InstrumentationService* instance deleting the *DataProcessor.*

Return: The operation shall return RETCODE_OK if the update of the *DataProcessor* was successful, any other return value shall be interpreted as an error and it is up to implementations to define the specific semantics of each one.

7.4.3.4 DataProcessorArgs

This class is used to provide custom configuration arguments to a *DataProcessor.*

DataProcessorArgs	
Attributes	
keys	String[]
values	String[]

7.4.3.4.1 keys

This attribute contains a collection of strings representing keys identifying the nature of the values stored at corresponding indices in the collection contained by the *values* attribute.

7.4.3.4.2 values

This attribute contains a collection of strings representing arguments for the configuration of the *DataProcessor* instance.

7.4.3.5 DataProcessorState

DataProcessorState shall represent any generic state objects created by a *DataProcessor* instance in its *attach_to_observable_scope* or *attach_to_observable_object* operations. No attribute or operation shall be specified for this class, implementations must map it to a construct available in the target implementation language that will support any custom data returned by a *DataProcessorState* (e.g., a pointer to type void in C or a value of type *Object* in Java).

7.4.4 Data Type Module

The Data Type Module is comprised of the following classifiers:

- *DataValueKind*
- *DataValue*
- *PrimitiveValue*
- *NumericValue*
- *SequenceValue*
- *BOOL*
- *OCTET*
- *INT16*
- *INT32*
- *INT64*

- *UINT16*
- *UINT32*
- *UINT64*
- *FLOAT32*
- *FLOAT64*
- *FLOAT128*
- *CHAR8*
- *CHAR32*
- *STRING8*
- *STRING32*
- *BOOLSeq*
- *OCTETSeq*
- *INT16Seq*
- *INT32Seq*
- *INT64Seq*
- *UINT16Seq*
- *UINT32Seq*
- *UINT64Seq*
- *FLOAT32Seq*
- *FLOAT64Seq*
- *FLOAT128Seq*
- *CHAR8Seq*
- *CHAR32Seq*
- *STRING8Seq*
- *STRING32Seq*
- *DataValueSource*
- *ReturnCode*
- *Time*
- *UTCTime*

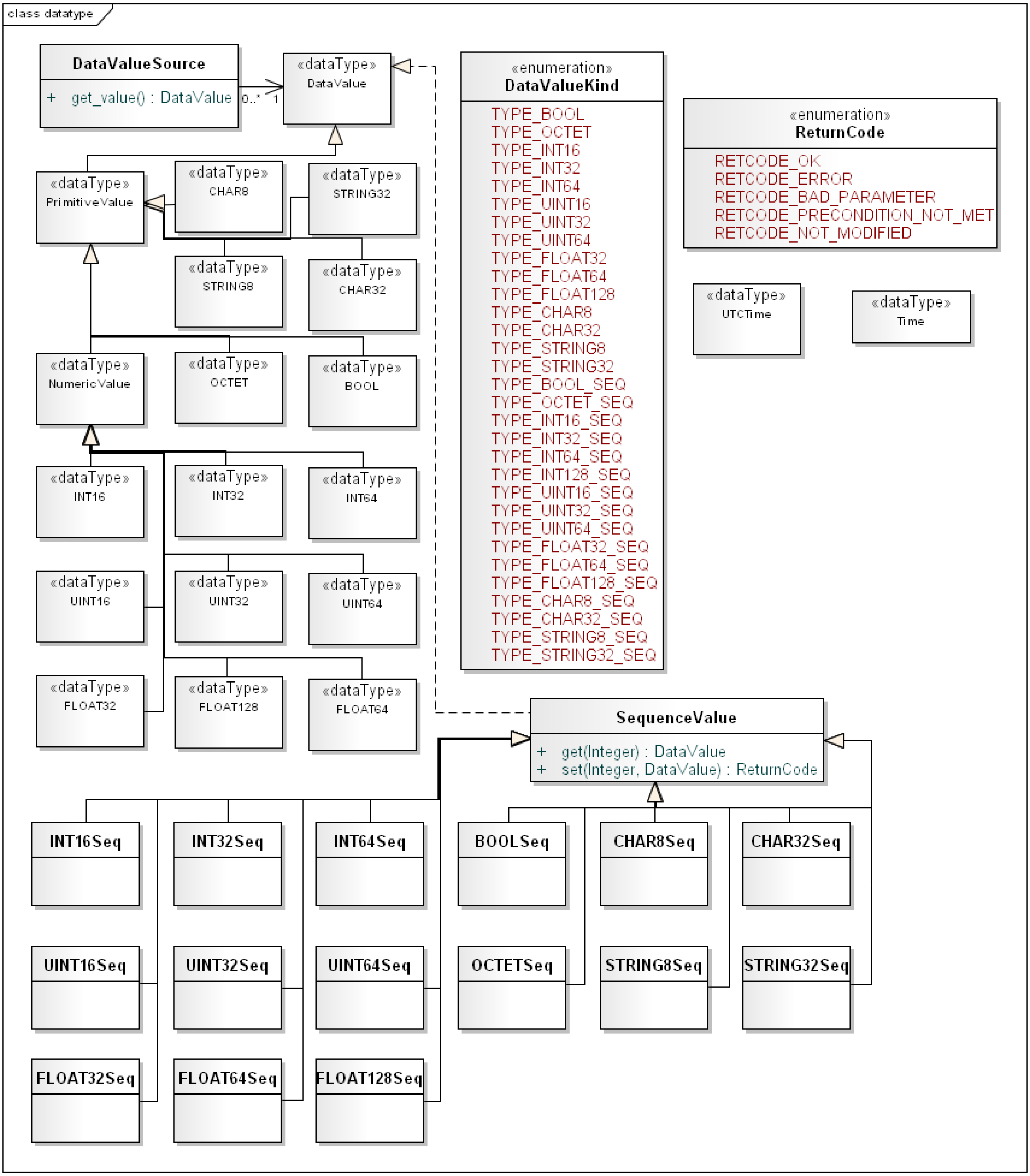

Figure 10 – Data Type Module

7.4.4.1 DataValueKind

The enumeration *DataValueKind* shall provide a list of all basic data-types that Application Instrumentation API supports to represent application data and configuration properties. Each value of defined by the enumeration shall correspond to a concrete data type available in the API.

DataValueKind	DataValue type	Description
TYPE_BOOL	BOOL	Boolean value
TYPE_OCTET	OCTET	8-bit quantity
TYPE_INT16	INT16	16-bit integer value
TYPE_INT32	INT32	32-bit integer value
TYPE_INT64	INT64	64-bit integer value
TYPE_UINT16	UINT16	16-bit unsigned integer value
TYPE_UINT32	UINT32	32-bit unsigned integer value
TYPE_UINT64	UINT64	64-bit unsigned integer value
TYPE_FLOAT32	FLOAT32	IEEE single-precision floating point number
TYPE_FLOAT64	FLOAT64	IEEE double-precision floating point number
TYPE_FLOAT128	FLOAT128	IEEE double-extended floating point number
TYPE_CHAR8	CHAR8	8-bit character
TYPE_CHAR32	CHAR32	32-bit character
TYPE_STRING8	STRING8	8-bit character string
TYPE_STRING32	STRING32	32-bit character string
TYPE_BOOL_SEQ	BOOLSeq	Array of Boolean values
TYPE_OCTET_SEQ	OCTETSeq	Array of 8-bit quantities
TYPE_INT16_SEQ	INT16Seq	Array of 16-bit integer values
TYPE_INT32_SEQ	INT32Seq	Array of 32-bit integer values
TYPE_INT64_SEQ	INT64Seq	Array of 64-bit integer values
TYPE_UINT16_SEQ	UINT16Seq	Array of 16-bit unsigned integer values
TYPE_UINT32_SEQ	UINT32Seq	Array of 32-bit unsigned integer values
TYPE_UINT64_SEQ	UINT64Seq	Array of 64-bit unsigned integer values
TYPE_FLOAT32_SEQ	FLOAT32Seq	Array of IEEE single-precision floating point numbers

TYPE_FLOAT64_SEQ	FLOAT64Seq	Array of IEEE double-precision floating point numbers
TYPE_FLOAT128_SEQ	FLOAT128Seq	Array of IEEE double-extended floating point numbers
TYPE_CHAR8_SEQ	CHAR8Seq	Array of 8-bit characters
TYPE_CHAR32_SEQ	CHAR32Seq	Array of 32-bit characters
TYPE_STRING8_SEQ	STRING8Seq	Array of 8-bit character strings
TYPE_STRING32_SEQ	STRING32Seq	Array of 32-bit character strings

7.4.4.2 DataValue

The *DataValue* data type is an abstract type that subsumes all other data types defined by the PIM to represent application data.

DataValue is the base class of a platform-independent hierarchy of data-types, which includes both primitive values, such as numbers, characters and strings, and limited sequences of primitive values.

PSMs are only required to provide representations for concrete sub-types of the type hierarchy. These types will typically be mapped to data types natively supported by the target platform to ease integration of the instrumentation in existing applications. The set of supported data types may also be restricted to guarantee easier integration. Custom complex data types will have to be defined if data types not available natively are to be supported on certain target platforms.

7.4.4.3 PrimitiveValue

PrimitiveValue is the base abstract type for primitive values.

7.4.4.4 NumericValue

NumericValue is the base abstract type for all numeric primitive values, which include integers, unsigned integers, and floating-point numbers of different size. The default value for all sub-types of *NumericValue* is *0*.

7.4.4.5 SequenceValue<T>

SequenceValue is the base abstract type for all complex values containing a finite collection of primitive values. It defines the minimal interface to access the contained values and determine the length of the sequence.

This class is parameterized on the primitive type of its element, specified as one of the sub-types of *PrimitiveValue*. The operations specified by this class shall be implemented as native operators on target platforms whenever possible. Values may, for example, be represented as native arrays and random access provided using the array operator.

The default value for all sub-types of *SequenceValue* is 'nil' (as specified by the platform [PSM]).

SequenceValue <T:PrimitiveValue>		
No Attributes		
Operations		
get		T
	index	Integer
set		ReturnCode

	index	Integer
	value	T
length		Integer

7.4.4.5.1 get<T>

This operation shall perform random access on the sequence's elements. If the specified *index* presents a value between 0 and the sequence's *length* – 1, the operation shall return the value stored at that position by the collection.

Parameter *index*: the *index* of the element to access in the sequence.

Return: the operation shall return the value of type T contained by the sequence at the specified index or the default value specified for type T if the index is not within the boundaries of sequence.

7.4.4.5.2 set<T>

This operation shall store an element at a particular index in the sequence. If the specified *index* presents a value between 0 and the sequence's *length* – 1, the operation shall store the specified value at that position in the collection.

Parameter *index*: the *index* of the element of the collection to set.

Parameter *value*: the value of type T to be set at the specified position in the sequence.

Return: The operation shall return RETCODE_OK if the value was successfully set at the requested position of the sequence, RETCODE_BAD_PARAMETER if the specified *index* was not within the boundaries of the sequence, RETCODE_ERROR if any other type of error occurred.

7.4.4.5.3 length

This operation shall return the current length of the sequence.

Return: The operation shall return an integer representing the total number of elements currently stored in the sequence. It may be 0 if no element as been added to the sequence yet.

7.4.4.6 BOOL

The BOOL data type represents a Boolean value. The default value for this type is *False*.

7.4.4.7 OCTET

The OCTET data type represents an 8-bit value. The default value for this type is *0x00*.

7.4.4.8 INT16

The INT16 data type represents a 16-bit integer value.

7.4.4.9 INT32

The INT32 data type represents a 32-bit integer value.

7.4.4.10 INT64

The INT64 data type represents a 64-bit integer value.

7.4.4.11 UINT16

The UINT16 data type represents a 16-bit unsigned integer value.

7.4.4.12 UINT32

The UINT32 data type represents a 32-bit unsigned integer value.

7.4.4.13 UINT64

The UINT64 data type represents a 64-bit unsigned integer value.

7.4.4.14 FLOAT32

The FLOAT32 data type represents a 32-bit IEEE floating-point value (as defined by the IEEE 754-2008 specification).

7.4.4.15 FLOAT64

The FLOAT64 data type represents a 64-bit IEEE floating-point value (as defined by the IEEE 754-2008 specification).

7.4.4.16 FLOAT128

The FLOAT128 data type represents a 128-bit IEEE floating-point value (as defined by the IEEE 754-2008 specification).

7.4.4.17 CHAR8

The CHAR8 data type represents an 8-bit character value. The default value for this type is 0x00.

7.4.4.18 CHAR32

The CHAR32 data type represents a wide character value, typically Unicode. The default value for this type is 0x00000000.

7.4.4.19 STRING8

The STRING8 data type represents strings of 8-bit characters. The default value for this type is 'nil' (as specified by the platform [PSM]).

7.4.4.20 STRING32

The STRING32 data type represents strings of wide characters. The default value for this type is 'nil' (as specified by the platform [PSM]).

7.4.4.21 BOOLSeq

The BOOLSeq data type represents SequenceValues containing BOOL elements.

7.4.4.22 OCTETSeq

The OCTETSeq data type represents SequenceValues containing OCTET elements.

7.4.4.2 INT16Seq

The INT16Seq data type represents SequenceValues containing INT16 elements.

7.4.4.24 INT32Seq

The INT32Seq data type represents SequenceValues containing INT32 elements.

7.4.4.25 INT64Seq

The INT64Seq data type represents SequenceValues containing INT64 elements.

7.4.4.26 UINT16Seq

The UINT16Seq data type represents SequenceValues containing UINT16 elements.

7.4.4.27 UINT32Seq

The UINT32Seq data type represents SequenceValues containing UINT32 elements.

7.4.4.28 UINT64Seq

The UINT64Seq data type represents SequenceValues containing UINT64 elements.

7.4.4.29 FLOAT32Seq

The FLOAT32Seq data type represents SequenceValues containing FLOAT32 elements.

7.4.4.30 FLOAT64Seq

The FLOAT64Seq data type represents SequenceValues containing FLOAT64 elements.

7.4.4.3 FLOAT128Seq

The FLOAT128Seq data type represents SequenceValues containing FLOAT128 elements.

7.4.4.32 CHAR8Seq

The CHAR8Seq data type represents SequenceValues containing CHAR8 elements.

7.4.4.33 CHAR32Seq

The CHAR32Seq data type represents SequenceValues containing CHAR32 elements.

7.4.4.34 STRING8Seq

The STRING8Seq data type represents SequenceValues containing STRING8 elements.

7.4.4.35 STRING32Seq

The STRING32Seq data type represents SequenceValues containing STRING32 elements.

7.4.4.36 DataValueSource<T>

A *DataValueSource* shall encapsulate a single value of application data and provide an interface for the instrumentation infrastructure to dynamically sample it, for example when generating an *Observation* from an *ObservableObject* whose fields have been bound using the *bind_value* operation.

***DataValueSource* <T:DataValue>**		
No Attributes		
Operations		
get_value		T

7.4.4.36.1 get_value<T>

This operation shall return the current value of the data source wrapped by the *DataValueSource* instance. The returned value shall be the value contained by the wrapped source as sampled at the time this operation was invoked.

Return: a value of type T representing the current value of the data source wrapped by the *DataValueSource* instance or the default value for type T, if an error occurred while sampling the wrapped data source.

7.4.4.37 ReturnCode

ReturnCode is an enumerated type that is used by methods of the Application Instrumentation API to indicate the outcome of an operation. The following table describes its possible values and provides a description of their meaning with respect to the operation's execution.

VALUE	DESCRIPTION
`RETCODE_OK`	The requested operation was executed correctly.
`RETCODE_ERROR`	The operation could not be carried out because an error occurred.
`RETCODE_BAD_PARAMETER`	The operation could not be carried out because one or more of the supplied arguments did not present acceptable values for the operation.
`RETCODE_PRECONDITION_NOT_MET`	The operation could not be carried out because one or more of the operation's preconditions were not met at the time the operation was executed.
`RETCODE_NOT_MODIFIED`	The operation was not performed and target resources were not modified.

All operations of the API that perform non-trivial behavior include an output parameter of type *ReturnCode* (or they return a value of type *ReturnCode* when no other return value is required) that applications can use to control the outcome of the operation.

Any operation may return RETCODE_OK or RETCODE_ERROR. Any operation that takes an input parameter may additionally return RETCODE_BAD_PARAMETER.

Operations that may return any other error code shall state so explicitly in their description.

7.4.4.38 Time

Time is a generic data type that shall represent an interval of time. It is left up to implementation of this specification to define its attributes and operations. The implementing data type shall support the representation of finite and infinite durations of time.

7.4.4.39 UTCTime

UTCTime is a generic data type that shall represent an instant of time using the UTC standard. It is left up to implementation of this specification to define its attributes and operations. The implementing data type shall support the representation of any date that can be expressed using the UTC standard.

7.4.5 Properties Module

The Properties Module is comprised of the following classifiers:

- *DefaultConfigurationTable*
- *InstrumentationServiceProperties*
- *ObservableSchemaProperties*
- *FieldProperties*
- *ObservableScopeProperties*
- *ObservableObjectProperties*
- *DataProcessorProperties*

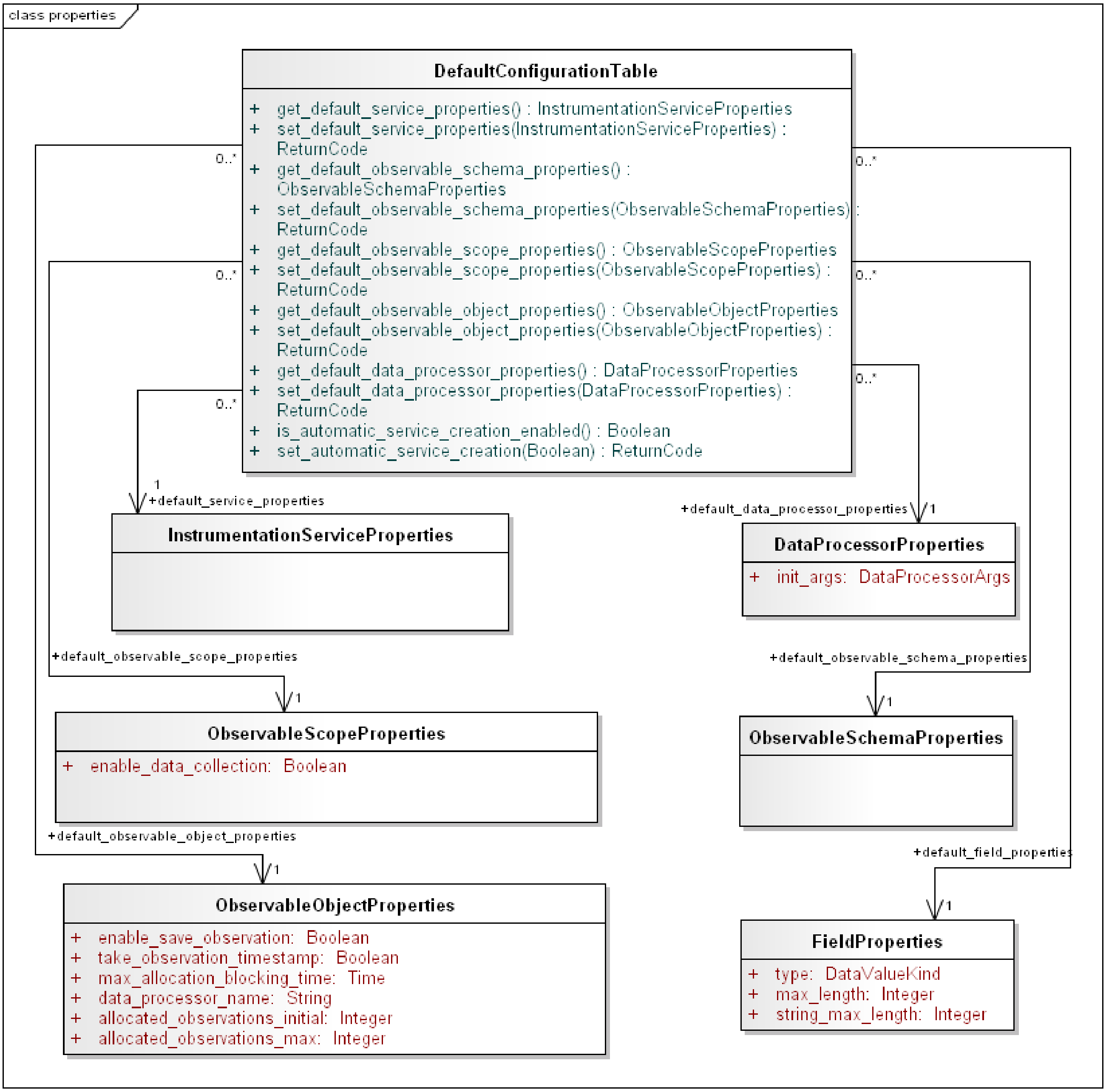

Figure 11 – Properties Module

7.4.5.1 DefaultConfigurationTable

The *DefaultConfigurationTable* class shall provide a single location where default configuration properties for each instrumentation entities may be stored and accessed by applications.

The *DefaultConfigurationTable* shall be implemented as a singleton, accessible using the provided *get_instance* static operation.

Any operation that creates an instrumentation entity, and accepts one of the properties structures defined by this module, shall also accept 'nil' (as defined by the platform [PSM]) as the value for the entities initialization properties. In this case, the operation shall use the default value reported for the specific type of properties by one of the methods of the *DefaultConfigurationTable* singleton instance.

DefaultConfigurationTable		
No Attributes		
Operations		
get_default_data_processor_properties		DataProcessorProperties
get_default_observable_object_properties		ObservableObjectProperties
get_default_observable_schema_properties		ObservableSchemaProperties
get_default_observable_scope_properties		ObservableScopeProperties
get_default_service_properties		InstrumentationServiceProperties
is_automatic_service_creation_enabled		Boolean
set_automatic_service_creation		ReturnCode
	enabled	Boolean
set_default_data processor_properties		ReturnCode
	properties	DataProcessorProperties
set_default_observable_object_properties		ReturnCode
	properties	ObservableObjectProperties
set_default_observable_schema_properties		ReturnCode
	properties	ObservableSchemaProperties
set_default_observable_scope_properties		ReturnCode
	properties	ObservableScopeProperties
set_default_service_properties		ReturnCode
	properties	InstrumentationServiceProperties

7.4.5.1.1 get_default_data_processor_properties

This operation shall return the default *DataProcessorProperties* instance stored by the *DefaultConfigurationTable.*

If the operation *set_default_service_properties* was never called by the application, this operation shall return an instance containing the default values for *DataProcessorProperties,* as defined by this specification.

Return: an instance of *DataProcessorProperties*.

7.4.5.1.2 get_default_observable_object_properties

This operation shall return the default *ObservableObjectProperties* instance stored by the *DefaultConfigurationTable.*

If the operation *set_default_observable_object_properties* was never called by the application, this operation shall return an instance containing the default values for *ObservableObjectProperties,* as defined by this specification.

Return: an instance of *ObservableObjectProperties*.

7.4.5.1.3 get_default_observable_schema_properties

This operation shall return the default *ObservableSchemaProperties* instance stored by the *DefaultConfigurationTable.*

If the operation *set_default_observable_schema_properties* was never called by the application, this operation shall return an instance containing the default values for *ObservableSchemaProperties,* as defined by this specification.

Return: an instance of *ObservableSchemaProperties*.

7.4.5.1.4 get_default_observable_scope_properties

This operation shall return the default *ObservableScopeProperties* instance stored by the *DefaultConfigurationTable.*

If the operation *set_default_observable_scope_properties* was never called by the application, this operation shall return an instance containing the default values for *ObservableScopeProperties,* as defined by this specification.

Return: an instance of *ObservableScopeProperties*.

7.4.5.1.5 get_default_service_properties

This operation shall return the default *InstrumentationServiceProperties* instance stored by the *DefaultConfigurationTable.*

If the operation *set_default_service_properties* was never called by the application, this operation shall return an instance containing the default values for *InstrumentationServiceProperties,* as defined by this specification.

Return: an instance of *InstrumentationServiceProperties*.

Parameter *properties*: the value of *InstrumentationServiceProperties* to store in the *DefaultConfigurationTable.*

Return: the operation shall return RETCODE_OK if the specified values were successfully stored inside the *DefaultConfigurationTable,* RETCODE_ERROR if any error occurred.

7.4.5.1.6 is_automatic_service_creation_enabled

This operation shall check whether *InstrumentationService* instances should be automatically created with default *InstrumentationServiceProperties* (as returned by the *get_default_service_properties*) when looked up by name using the *lookup_service* operation of the *Infrastructure* class.

If *set_automatic_service_creation* has never been called yet, this operation shall return *False*. Otherwise, the operation shall return the last value specified using *set_automatic_service_creation*.

Return: The operation shall return *True* if *InstrumentationService* instances should be automatically created.

7.4.5.1.7 set_automatic_service_creation

This operation shall set whether *InstrumentationService* instances should be automatically created with default *InstrumentationServiceProperties* (as returned by the *get_default_service_properties*) when looked up by name using the *lookup_service* operation of the *Infrastructure* class.

Parameter *enabled*: a Boolean value specifying whether the property is enabled or not.

Return: The operation shall return RETCODE_OK if the specified value was correctly stored, RETCODE_ERROR if any type of error occurred.

7.4.5.1.8 set_default_data_processor_properties

This operation shall set the default *DataProcessorProperties* instance stored by the *DefaultConfigurationTable.*

If the value was successfully saved, this operation shall return RETCODE_OK. If an error prevented the value from being stored in the *DefaultConfigurationTable,* the operation shall fail and return RETCODE_ERROR.

Every successive invocation of *get_default_data_processor_properties* shall return the values saved by this operation, until it is successfully invoked again with different ones.

Parameter *properties*: the value of *DataProcessorProperties* to store in the *DefaultConfigurationTable.*

Return: the operation shall return RETCODE_OK if the specified values were successfully stored inside the *DefaultConfigurationTable,* RETCODE_ERROR if any error occurred.

7.4.5.1.9 set_default_observable_object_properties

This operation shall set the default *ObservableObjectProperties* instance stored by the *DefaultConfigurationTable.*

If the value was successfully saved, this operation shall return RETCODE_OK. If an error prevented the value from being stored in the *DefaultConfigurationTable,* the operation shall fail and return RETCODE_ERROR.

Every successive invocation of *get_default_observable_object_properties* shall return the values saved by this operation, until it is successfully invoked again with different ones.

Parameter *properties*: the value of *ObservableObjectProperties* to store in the *DefaultConfigurationTable.*

Return: the operation shall return RETCODE_OK if the specified values were successfully stored inside the *DefaultConfigurationTable,* RETCODE_ERROR if any error occurred.

7.4.5.1.10 set_default_observable_schema_properties

This operation shall set the default *ObservableSchemaProperties* instance stored by the *DefaultConfigurationTable.*

If the value was successfully saved, this operation shall return RETCODE_OK. If an error prevented the value from being stored in the *DefaultConfigurationTable,* the operation shall fail and return RETCODE_ERROR.

Every successive invocation of *get_default_observable_schema_properties* shall return the values saved by this operation, until it is successfully invoked again with different ones.

Parameter *properties*: the value of *ObservableSchemaProperties* to store in the *DefaultConfigurationTable.*

Return: the operation shall return RETCODE_OK if the specified values were successfully stored inside the *DefaultConfigurationTable,* RETCODE_ERROR if any error occurred.

7.4.5.1.11 set_default_observable_scope_properties

This operation shall set the default *ObservableScopeProperties* instance stored by the *DefaultConfigurationTable.*

If the value was successfully saved, this operation shall return RETCODE_OK. If an error prevented the value from being stored in the *DefaultConfigurationTable,* the operation shall fail and return RETCODE_ERROR.

Every successive invocation of *get_default_observable_scope_properties* shall return the values saved by this operation, until it is successfully invoked again with different ones.

Parameter *properties*: the value of *ObservableScopeProperties* to store in the *DefaultConfigurationTable.*

Return: the operation shall return RETCODE_OK if the specified values were successfully stored inside the *DefaultConfigurationTable,* RETCODE_ERROR if any error occurred.

7.4.5.1.12 set_default_observable_schema_properties

This operation shall set the default *FieldProperties* instance stored by the *DefaultConfigurationTable.*

If the value was successfully saved, this operation shall return RETCODE_OK. If an error prevented the value from being stored in the *DefaultConfigurationTable,* the operation shall fail and return RETCODE_ERROR.

Every successive invocation of *get_default_field_properties* shall return the values saved by this operation, until it is successfully invoked again with different ones.

Parameter *properties*: the value of *FieldProperties* to store in the *DefaultConfigurationTable.*

Return: the operation shall return RETCODE_OK if the specified values were successfully stored inside the *DefaultConfigurationTable,* RETCODE_ERROR if any error occurred.

7.4.5.1.13 set_default_service_properties

This operation shall set the default *InstrumentationServiceProperties* instance stored by the *DefaultConfigurationTable.*

If the value was successfully saved, this operation shall return RETCODE_OK. If an error prevented the value from being stored in the *DefaultConfigurationTable,* the operation shall fail and return RETCODE_ERROR.

Every successive invocation of *get_default_service_properties* shall return the values saved by this operation, until it is successfully invoked again with different ones.

7.4.5.2 InstrumentationServiceProperties

This class shall define all configuration properties that may be used to control the initialization of an *InstrumentationService* and its functionalities.

InstrumentationServiceProperties
Attributes
No attributes

7.4.5.3 ObservableSchemaProperties

This class shall define all configuration properties that may be used to control the initialization of an *ObservableSchema* and its functionalities.

ObservableSchemaProperties
Attributes
No attributes

7.4.5.4 FieldProperties

This class defines the properties of a *Field* instance and allows their specification when creating the *Field* in an *ObservableSchema.*

FieldProperties	
Attributes	
type	DataValueKind
max_length	Integer
string_max_length	Integer

7.4.5.4.1 type

This attribute specifies the type of *DataValue* that can be stored in a *Field*. The default value of this attribute is TYPE_INT32.

7.4.5.4.2 max_length

This attribute specifies the maximum length of a single value contained in a *Field*. The default value of this attribute is 1.

7.4.5.4.3 string_max_length

This attribute specifies the maximum number of single values that can be contained in a *Field*. The default value of this attribute is 0.

7.4.5.5 ObservableScopeProperties

This class shall define all configuration properties that may be used to control the initialization of an *ObservableScope* and its functionalities.

ObservableScopeProperties	
Attributes	
enable_data_collection	Boolean

7.4.5.5.1 enable_data_collection

This attribute controls whether data-collection in the *ObservableScope* will be enabled automatically upon initialization or if applications must enable it manually. If *True* is specified, data-collection will be enabled by invoking operation *ObservableScope::enable_data_collection*. The default value of this attribute is *True*.

7.4.5.6 ObservableObjectProperties

This class shall define all configuration properties that may be used to control the initialization of an *ObservableObject* and its functionalities.

ObservableObjectProperties	
Attributes	
enable_save_observation	Boolean
take_observation_timestamp	Boolean
max_allocation_blocking_time	Time
data_processor_name	String
allocated_observations_initial	Integer
allocated_observations_max	Integer

7.4.5.6.1 enable_save_observation

If this attribute is set to *True*, an *ObservableObject* will enable the generation of *Observation* instances as soon as it is initialized using operation *ObservableObject::enable_save_observation*. If this attribute is *False,* applications must enable generation of *Observation* instances explicitly.

The default value for this attribute is *True.*

7.4.5.6.2 take_observation_timestamp

If this attribute is set to True, an *ObservableObject* will take a time-stamp of the current time when generating a new *Observation* and store in the new *Observation* instance as an *UTCTime* value.

The default value for this attribute is *True.*

7.4.5.6.3 allocated_observations_initial

This attribute controls how many *Observation* instances an *ObservableObject* should initially allocate. The default value for this attribute is 1.

7.4.5.6.4 allocated_observations_max

This attribute controls how many *Observation* instances an *ObservableObject* can allocate at the most. This value must be greater or equal than *allocated_observations_initial*. This attribute can be set to -1 to indicate that an unlimited number of *Observation* instances can be allocated if required to store new values from the *ObservableObject*.

The default value for this attribute is -1.

7.4.5.6.5 max_allocation_blocking_time

This attribute defines the maximum period of time that an *ObservableObject* can block the execution of an application thread during the execution of its *save_observation* operation, after the maximum number of *Observation* instances that can be allocated as been reached and no *Observation* instance is available to store the new values. The value shall be represented by type *Time*.

The default value for this attribute is a period of length 0.

7.4.5.6.6 data_processor_name

This attribute specifies the name of an optional *DataProcessor* that will be used by the *ObservableScope* managing the *ObservableObject* to process all *Observation* instances it creates.

The default value for this attribute is 'nil'.

7.4.5.7 DataProcessorProperties

This class is a placeholder for properties that each PSM may need to specify for the proper initialization of a *DataProcessor* instance.

DataProcessorProperties	
Attributes	
init_args	DataProcessorArgs

7.4.5.7.1 init_args

This attribute contains an optional instance of *DataProcessorArgs* that will be passed to the *DataProcessor*'s *initialize* method when the new *DataProcessor* instance is initialized.

7.5 Instrumentation Domain

7.5.1 Distributed Architecture

An *Instrumentation Domain* is an abstract service layer that interconnects instrumented applications with remote monitoring applications. It models a distributed instrumentation infrastructure and encapsulates its underlying communication infrastructure, providing all functionalities required by remote applications to interact with the local instrumentation of each monitored application.

An *Instrumentation Domain* allows applications to access *Observations* generated using *ObservableObjects* and published by an *ObservableScope.* It also exposes a *Remote Service Interface* that can be used to perform dynamic reconfiguration of the remote instrumentation entities.

An *Instrumentation Domain* must define a naming scheme to allow remote applications to identify specific instrumentation entities in the distributed infrastructure. This scheme may leverage the names assigned by the API to each entity and used within the local instrumentation to uniquely reference them. If this were the case, some additional strategy, such as also considering host-specific information, should be considered to avoid potential name clashes between multiple instrumented applications in the same *Instrumentation Domain*.

7.5.2 Data Distribution Model

Applications interact with the *Instrumentation Domain* to access observations of data generated by instrumented applications.

An *Instrumentation Domain* receives processed *Observation* objects from *ObservableScope* instances and it implements the communication logic required for their distribution. Once an *Observation* has been successfully handed to the *Instrumentation Domain*, an *ObservableScope* can safely recycle it for new contents. If future external consumers must access the *Observation*, it is the *Instrumentation Domain*'s responsibility to store its values and meta-data information.

Transformations may be applied to the data in order to map it to the technology selected for the implementation of the distribution to remote consumers.

The interface offered to consumers for accessing the data depends on the selected communication technology.

7.5.3 Addressing of Instrumentation Entities

Since the *Instrumentation Domain* enables applications to re-configure remote instrumentation entities and access the observations they generate, each implementation shall define how instrumentation entities residing on distributed instrumented application may be uniquely identified by a remote application.

The *Application Instrumentation API* assigns each entity a name, which is constrained to be unique within the context of the entity that created it. This naming support must be leveraged to correctly address entities in an *Instrumentation Domain*.

The following table summarizes how each type of entity is uniquely identified in the *Instrumentation Domain.* Implementations shall allow applications to identify entity using this information. How the information contained in each identifying tuple is concretely expressed by the application is left unspecified. As an example, an ad-hoc URI scheme may be defined to properly address entities in a concise manner.

The identifiers are presented in the form of tuples of strings.

Type of Entity	Identifier
InstrumentationService	(ID of host; name)
ObservableSchema	(ID of *InstrumentationService*; name)
DataProcessor	(ID of *InstrumentationService*; name)
ObservableScope	(ID of *InstrumentationService*; name)
ObservableObject	(ID of *ObservableScope*; name)

7.5.4 Remote Service Interface

An *Instrumentation Domain* lets applications interact with the instrumentation entities of instrumented applications through a remote service interface.

The following sub clauses provide a high-level description of the operations that shall be supported through this interface by implementations of an *Instrumentation Domain.*

Each Platform Specific Model (PSM) must specify how this interface is exposed to applications and how the remote interaction is implemented.

In particular, different platforms may support different invocation semantics for the operations. Depending on the naming scheme adopted to reference remote entities and the support of the underlying communication architecture, a platform [PSM] may support invocation of these operations on multiple instrumentation entities, from multiple *InstrumentationService* instances at a time. Other implementations may adopt a 1 to 1 invocation semantic, more similar to classic Remote Procedure Call or Remote Method Invocation architectures.

7.5.4.1 Description of operations

The description of each operation summarizes the actions performed by the local instrumentation of an instrumented application when an operation is requested to the *Instrumentation Domain*.

The input and output parameters of each operation are described in a qualitative way, leaving their formal definition to implementations. Similarly to the Application Instrumentation API, non-trivial operations are assumed to have a *ReturnCode* output parameter that can be used to convey the outcome of the operation.

Most operations take an input parameter specifying a reference to an instrumentation entity. This instrumentation entity will be the target of the operation and control the actual operations performed by the local instrumentation.

As mentioned in 7.5.3 Addressing of Instrumentation Entities, a way to uniquely identify instrumentation entities created by each instrumented application is assumed to be available. All operations shall fail and return RETCODE_BAD_PARAMETER if the target entity is not found in the local instrumentation of an application.

Each implementation may map input and output data required by each operation in different ways and adopt different strategies to perform the operations.

7.5.4.1.1 Check Save Observation Status

This operation shall allow a remote application to verify whether the generation of new *Observation* objects is enabled in one or more *ObservableObject* instances. The remote application must specify the target *ObservableObject* instances on which to perform the verification.

For each referenced *ObservableObject* instance found within the local instrumentation that is performing the operation, the result shall include a tuple of the form:

(ID of *ObservableObject*, RETCODE_OK, *status*)

Status is the result of invoking operation *is_save_observation_enabled* on the *ObservableObject*.

If the remote application specifies the identifier of an *InstrumentationService* and/or an *ObservableScope* as the target for this operation, the result shall include information as if the operation had been invoked on all *ObservableObject* instances contained by the referenced entity (i.e. all *ObservableObject* instances created by an *ObservableScope* or all *ObservableObject* instances created by every *ObservableScope* instance created by the *InstrumentationService*).

For each entity included in the operation's *target* that is not found within the local instrumentation, the result shall include a tuple of the form:

(ID of entity, RETCODE_BAD_PARAMETER)

If an error occurred while analyzing an entity (e.g., determining the *ObservableObject* instances it contains or verifying the status of an *ObservableObject*), the result shall include a tuple of the form:

(ID of entity, RETCODE_ERROR)

Parameter *target*: A collection of identifiers of entities in the local instrumentation; the referenced entities may be of type *InstrumentationService*, *ObservableScope*, *ObservableObject.*

Return: In case of success, this operation shall return a tuple of the form (ID of *ObservableObject*, RETCODE_OK, *status*) for each *ObservableObject* instance referenced by *target* whose status was successfully verified in the local instrumentation, where *status* is the Boolean value returned by invoking *is_save_observation_enabled* on the *ObservableObject*. In case of failure, the operation shall return a tuple of the form (ID of entity, RETCODE_BAD_PARAMETER) for each entity referenced by *target* that was not found in the local instrumentation, and a tuple of the form (ID of entity, RETCODE_ERROR) for each entity for which error occurred while performing the operation. In case of failure, implementations may also return a *status* value, which shall be ignored by applications receiving the result.

7.5.4.1.2 Enable Save Observation

This operation shall allow a remote application to enable the generation of *Observation* samples on one or more *ObservableObject* instances. The remote application must specify the target *ObservableObject* instances on which to perform the operation.

For each referenced *ObservableObject* instance found within the local instrumentation that is performing the operation, the operation shall invoke the *ObservableObject*'s *enable_save_observation* operation and include a tuple of the following form in the result:

(ID of *ObservableObject*, *retcode*)

Retcode is the result value returned by the invocation of *enable_save_observation* on the *ObservableObject*.

If the remote application specifies the identifier of an *InstrumentationService* and/or an *ObservableScope* as the target for this operation, the operation shall target all *ObservableObject* instances contained by the referenced entity (i.e., all *ObservableObject* instances created by an *ObservableScope* or all *ObservableObject* instances created by every *ObservableScope* instance created by the *InstrumentationService*) and include a tuple for each *ObservableObject* in its result.

For each entity included in the operation's *target* that is not found within the local instrumentation, the result shall include a tuple of the form:

(ID of entity, RETCODE_BAD_PARAMETER)

If an error occurred while analyzing an entity (e.g., determining the *ObservableObject* instances it contains or enabling generation of *Observation* samples on an *ObservableObject*), the result shall include a tuple of the form:

(ID of entity, RETCODE_ERROR)

Parameter *target*: A collection of identifiers of entities in the local instrumentation; the referenced entities may be of type *InstrumentationService*, *ObservableScope*, *ObservableObject.*

Return: In case of success, this operation shall return a tuple of the form (ID of *ObservableObject*, *retcode*) for each *ObservableObject* instance referenced by *target*, where *retcode* is the value returned by invoking *enable_save_observation* on the *ObservableObject*. In case of failure, the operation shall return a tuple of the form (ID of entity, RETCODE_BAD_PARAMETER) for each entity referenced by *target* that was not found in the local instrumentation, and a tuple of the form (ID of entity, RETCODE_ERROR) for each entity for which an error occurred while performing the operation.

7.5.4.1.3 Disable Save Observation

This operation shall allow a remote application to disable the generation of *Observation* samples on one or more *ObservableObject* instances. The remote application must specify the target *ObservableObject* instances on which to perform the operation.

For each referenced *ObservableObject* instance found within the local instrumentation that is performing the operation, the operation shall invoke the *ObservableObject*'s *disable_save_observation* operation and include a tuple of the following form in the result:

(ID of *ObservableObject*, *retcode*)

Retcode is the result value returned by the invocation of *enable_save_observation* on the *ObservableObject*.

If the remote application specifies the identifier of an *InstrumentationService* and/or an *ObservableScope* as the target for this operation, the operation shall target all *ObservableObject* instances contained by the referenced entity (i.e., all *ObservableObject* instances created by an *ObservableScope* or all *ObservableObject* instances created by every *ObservableScope* instance created by the *InstrumentationService*) and include a tuple for each *ObservableObject* in its result.

For each entity included in the operation's *target* that is not found within the local instrumentation, the result shall include a tuple of the form:

(ID of entity, RETCODE_BAD_PARAMETER)

If an error occurred while analyzing an entity (e.g., determining the *ObservableObject* instances it contains or disabling generation of *Observation* samples on an *ObservableObject*), the result shall include a tuple of the form:

(ID of entity, RETCODE_ERROR)

Parameter *target*: A collection of identifiers of entities in the local instrumentation; the referenced entities may be of type *InstrumentationService*, *ObservableScope*, *ObservableObject.*

Return: In case of success, this operation shall return a tuple of the form (ID of *ObservableObject*, *retcode*) for each *ObservableObject* instance referenced by *target*, where *retcode* is the value returned by invoking *disable_save_observation* on the *ObservableObject*. In case of failure, the operation shall return a tuple of the form (ID of entity, RETCODE_BAD_PARAMETER) for each entity referenced by *target* that was not found in the local instrumentation, and a tuple of the form (ID of entity, RETCODE_ERROR) for each entity for which an error occurred while performing the operation.

7.5.4.1.4 Check Data Collection Status

This operation shall allow a remote application to verify if data collection is currently enabled in one or more *ObservableScope* instances. The remote application must specify the target *ObservableScope* instances on which to perform the verification.

For each referenced *ObservableScope* instance found within the local instrumentation that is performing the operation, the operation shall invoke the *ObservableScope*'s *is_data_collection_enabled* operation and include a tuple of the following form in the result:

(ID of *ObservableScope*, RETCODE_OK*, status*)

Status is the result value returned by the invocation of *is_data_collection_enabled* on the *ObservableScope*.

If the remote application includes the identifier of an *InstrumentationService* in the target of this operation, the operation shall target all *ObservableScope* instances contained by the referenced entity (i.e., all *ObservableScope* instances created by the *InstrumentationService*) and include a tuple for each *ObservableScope* in its result.

For each entity included in the operation's *target* that is not found within the local instrumentation, the result shall include a tuple of the form:

(ID of entity*,* RETCODE_BAD_PARAMETER)

If an error occurred while analyzing an entity (e.g., determining the *ObservableScope* instances it contains or verifying the status of data collection on an *ObservableScope*)*,* the result shall include a tuple of the form:

(ID of entity*,* RETCODE_ERROR)

Parameter *target*: A collection of identifiers of entities in the local instrumentation; the referenced entities may be of type *InstrumentationService*, *ObservableScope.*

Return: In case of success, this operation shall return a tuple of the form (ID of *ObservableScope*, RETCODE_OK, *status*) for each *ObservableScope* instance referenced by *target*, where *status* is the value returned by invoking *is_data_collection_enabled* on the *ObservableScope*. In case of failure, the operation shall return a tuple of the form (ID of entity, RETCODE_BAD_PARAMETER) for each entity referenced by *target* that was not found in the local instrumentation, and a tuple of the form (ID of entity, RETCODE_ERROR) for each entity for which an error occurred while performing the operation.

7.5.4.1.5 Enable Data Collection

This operation shall allow a remote application to enable data collection on one or more *ObservableScope* instances. The remote application must specify the target *ObservableScope* instances on which to perform the operation.

For each referenced *ObservableScope* instance found within the local instrumentation that is performing the operation, the operation shall invoke the *ObservableScope*'s *enable_data_collection* operation and include a tuple of the following form in the result:

(ID of *ObservableScope*, *retcode*)

Retcode is the result value returned by the invocation of *enable_data_collection* on the *ObservableScope*.

If the remote application includes the identifier of an *InstrumentationService* in the target of this operation, the operation shall target all *ObservableScope* instances contained by the referenced entity (i.e., all *ObservableScope* instances created by the *InstrumentationService*) and include a tuple for each *ObservableScope* in its result.

For each entity included in the operation's *target* that is not found within the local instrumentation, the result shall include a tuple of the form:

(ID of entity*,* RETCODE_BAD_PARAMETER)

If an error occurred while analyzing an entity (e.g., determining the *ObservableScope* instances it contains or enabling data collection on an *ObservableScope*)*,* the result shall include a tuple of the form:

(ID of entity*,* RETCODE_ERROR)

Parameter *target*: A collection of identifiers of entities in the local instrumentation; the referenced entities may be of type *InstrumentationService*, *ObservableScope.*

Return: In case of success, this operation shall return a tuple of the form (ID of *ObservableScope*, *retcode*) for each *ObservableScope* instance referenced by *target*, where *retcode* is the value returned by invoking *enable_data_collection* on the *ObservableScope*. In case of failure, the operation shall return a tuple of the form (ID of entity, RETCODE_BAD_PARAMETER) for each entity referenced by *target* that was not found in the local instrumentation, and a tuple of the form (ID of entity, RETCODE_ERROR) for each entity for which an error occurred while performing the operation.

7.5.4.1.6 Disable Data Collection

This operation shall allow a remote application to disable data collection on one or more *ObservableScope* instances. The remote application must specify the target *ObservableScope* instances on which to perform the operation.

For each referenced *ObservableScope* instance found within the local instrumentation that is performing the operation, the operation shall invoke the *ObservableScope*'s *disable_data_collection* operation and include a tuple of the following form in the result:

(ID of *ObservableScope*, *retcode*)

Retcode is the result value returned by the invocation of *disable_data_collection* on the *ObservableScope*.

If the remote application includes the identifier of an *InstrumentationService* in the target of this operation, the operation shall target all *ObservableScope* instances contained by the referenced entity (i.e., all *ObservableScope* instances created by the *InstrumentationService*) and include a tuple for each *ObservableScope* in its result.

For each entity included in the operation's *target* that is not found within the local instrumentation, the result shall include a tuple of the form:

(ID of entity*,* RETCODE_BAD_PARAMETER)

If an error occurred while analyzing an entity (e.g., determining the *ObservableScope* instances it contains or disabling data collection on an *ObservableScope*)*,* the result shall include a tuple of the form:

(ID of entity*,* RETCODE_ERROR)

Parameter *target*: A collection of identifiers of entities in the local instrumentation; the referenced entities may be of type *InstrumentationService*, *ObservableScope.*

Return: In case of success, this operation shall return a tuple of the form (ID of *ObservableScope*, *retcode*) for each *ObservableScope* instance referenced by *target*, where *retcode* is the value returned by invoking *disable_data_collection* on the *ObservableScope*. In case of failure, the operation shall return a tuple of the form (ID of entity, RETCODE_BAD_PARAMETER) for each entity referenced by *target* that was not found in the local instrumentation, and a tuple of the form (ID of entity, RETCODE_ERROR) for each entity for which an error occurred while performing the operation.

7.5.4.1.7 Update DataProcessor

This operation shall allow a remote application to update the configuration of one or more *DataProcessor* instances. The remote application must specify the identifiers of the *DataProcessor* instances on which to perform the operation.

For each referenced *DataProcessor* instance found within the local instrumentation that is performing the operation, the operation shall invoke the *DataProcessor*'s *update* operation passing the *DataProcessorArgs* value specified by the remote application. The result shall include a tuple of the following form:

(ID of *DataProcessor*, *retcode*)

Retcode is the result value returned by the invocation of *update* on the *DataProcessor* with the specified arguments.

If the remote application includes the identifier of an *InstrumentationService* in the target of this operation, the operation shall target all *DataProcessor* instances contained by the referenced entity (i.e., all *DataProcessor* instances created by the *InstrumentationService*) and include a tuple for each *DataProcessor* in its result.

For each entity included in the operation's *target* that is not found within the local instrumentation, the result shall include a tuple of the form:

(ID of entity, RETCODE_BAD_PARAMETER)

If an error occurred while analyzing an entity (e.g., determining the *ObservableScope* instances it contains or disabling data collection on an *ObservableScope*), the result shall include a tuple of the form:

(ID of entity, RETCODE_ERROR)

Parameter *target*: A collection of identifiers of entities in the local instrumentation; the referenced entities may be of type *InstrumentationService*, *DataProcessor*.

Parameter *args*: an instance of *DataProcessorArgs* that will be passed to each referenced *DataProcessor*'s *update* operation.

Return: In case of success, this operation shall return a tuple of the form (ID of *DataProcessor*, *retcode*) for each *DataProcessor* instance referenced by *target*, where *retcode* is the value returned by invoking *update* on the *DataProcessor* with the specified argument. In case of failure, the operation shall return a tuple of the form (ID of entity, RETCODE_BAD_PARAMETER) for each entity referenced by *target* that was not found in the local instrumentation, and a tuple of the form (ID of entity, RETCODE_ERROR) for each entity for which an error occurred while performing the operation.

This page intentionally left blank.

8 Platform Specific Model (PSM)

8.1 Application Instrumentation API PSMs

8.1.1 C PSM

8.1.1.1 PIM to PSM Mapping Rules

8.1.1.1.1 Naming conventions

The name of all functions and data-types defined by the API shall start with the prefix "AppInst_".

The name of classifier that only present data attributes (i.e., Value Types) shall be terminated with the suffix "_t".

8.1.1.1.2 Interfaces Types

All classifiers of the PIM that only define methods in their description shall be mapped to opaque data-types.

The operations defined by each type shall be mapped to C functions that prepend the name of the entity to the operation's name. All non-static functions shall take as the first parameter a "self" parameter of the type of the entity, representing the instance on which the operation is being invoked. Static operations shall not take a "self" parameter.

8.1.1.1.3 Value Types

Classifiers of the PIM that define only public data attributes in their description and no operation shall be mapped to C *struct* data-types.

A macro shall be defined for each one of these data-types, which shall be used by applications to properly initialize the attributes to their default values.

When required, implementation may include support functions to properly handle initialization and finalization of any type that requires dynamic allocation of memory to store its values (e.g., *DataProcessorArgs).*

8.1.1.1.4 Enumeration Types

All enumeration types shall be mapped to *enum* types. Values of the enumeration shall be prepended with the "*APPINST_"* prefix.

8.1.1.1.5 Supported DataValue Types

Each concrete sub-class of *DataValue* defined by the *Application Instrumentation API*'s PIM shall be mapped to a native C types. Abstract types of the *DataValue* hierarchy shall not be mapped to specific C types.

The following table illustrates how each supported type is mapped.

***DataValue* type**	**C Type**
`BOOL`	`unsigned char`
`OCTET`	`unsigned char`
`INT16`	`short`
`INT32`	`long`
`INT64`	`long long`

UINT6	unsigned short
UINT32	unsigned long
UINT64	unsigned long long
FLOAT32	float
FLOAT64	double
FLOAT128	long double
CHAR8	char
CHAR32	wchar_t
STRING8	char*
STRING32	wchar_t*
BOOLSeq	unsigned char*
OCTETSeq	unsigned char*
INT16Seq	short*
INT32Seq	long*
INT64Seq	long long*
UINT6Seq	unsigned short*
UINT32Seq	unsigned long*
UINT64Seq	unsigned long long*
FLOAT32Seq	float*
FLOAT64Seq	double*
FLOAT128Seq	long double*
CHAR8Seq	char*
CHAR32Seq	wchar_t*
STRING8Seq	char**
STRING32Seq	wchar_t**

In order to correctly represent the BOOL type, two macros, APPINST_BOOL_TRUE and APPINST_BOOL_FALSE shall be defined to represent its two possible values.

8.1.1.1.6 Function Signatures

All output parameters will be mapped to inout parameters in C passed by reference to the function. The caller must supply a pointer to an object of the appropriate type where the output value will be stored. The value NULL may be passed in place of a pointer to indicate that the output parameter is not of interest to the caller. The operation shall thus ignore the parameter.

For each collection returned by a function, the function's signature shall include an inout parameter of type UINT32 that will store the length of the retuned collection. For each collection accepted as input parameter, the function's signature shall include an in parameter of type UINT32 specifying the length of the supplied collection.

Parameters of type *String* shall be mapped to STRING8 values. Parameters of type *Boolean* shall be mapped to BOOL values.

8.1.1.1.7 DataValueSource

The type *DataValueSource* that defines the source of a bound field in an *ObservableObject* shall be mapped to a C pointer of type *void*. The *get_value* operation shall be implemented by casting the pointer to the expected type (the C data-type that maps the *DataValueSource*'s type T) and the use of the indirection operator (*).

8.1.1.1.8 DataProcessor

Each operation defined by the *DataProcessor* interface shall be mapped to a function pointer type. When implementing the interface, an application shall define a function with the appropriate signature for each of the operations.

The implementation of each operation that will be used by a new *DataProcessor* instance shall be specified in the *DataProcessorProperties* value specified at the creation of the *DataProcessor*.

The *DataProcessorProperties* class shall be extended to include attributes of the appropriate function pointer type and string attributes for each of the operations of the *DataProcessor* interface. The following attributes will be added to *DataProcessorProperties*:

- *initialize_fn:* function pointer to the implementation of the *initialize* operation.
- *finalize_fn:* function pointer to the implementation of the *finalize* operation.
- *update_fn:* function pointer to the implementation of the *update* operation.
- *attach_to_observable_scope_fn*: function pointer to the implementation of the *attach_to_observable_scope* operation.
- *detach_from_observable_scope_fn*: function pointer to the implementation of the *detach_from_observable_scope* operation.
- *attach_to_observable_object_fn*: function pointer to the implementation of the *attach_to_observable_object* operation.
- *detach_from_observable_object_fn*: function pointer to the implementation of the *detach_from_observable_object* operation.
- *process_observations_fn:* function pointer to the implementation of the *process_observations* operation.

The *DataProcessorState* class is mapped to a C pointer of type *void*. The instrumentation infrastructure shall treat values of this type as opaque and delegate any management of memory associated with them to the application.

8.1.2 Java PSM

8.1.2.1 PIM to PSM Mapping Rules

8.1.2.1.1 Packages Organization

All modules in the PIM are mapped to package `org.omg.appinst`.

8.1.2.1.2 Naming Conventions

All names of operations replace the "underscore-based" naming convention used by the PIM in favor of the "Camel-case" convention, which is more familiar to Java developers. For example, operation "my_class_operation" in the PIM will be mapped to operation "myClassOperation" in the Java PSM.

8.1.2.1.3 Entity representation

All entities in the PIM are mapped to Java interfaces.

8.1.2.1.4 Data type support

The set of supported sub-types of *PrimitiveValue* is limited to those that can be mapped to data types natively supported by the Java platform. This limitation also affects the supported sub-types of *SequenceValue,* which are similarly limited to collections of native primitive types.

Each supported type is mapped to both a native primitive type and its corresponding primitive wrapper class.

Supported sub-types of SequenceValue are instead mapped to both an array of a primitive type and an array of its primitive wrapper class. The API operations support the specification of value in any of these two forms and as a `java.util.list.Collection` of the correct primitive wrapper class. Implementation are allowed to operate internal conversions between these formats to a selected internal representation, although they should state when conversions may take place and provide at least one optimized representation.

The following tables lists concrete sub-types of *DataValue* and their mapping to Java types:

***DataValue* type**	**Java Type**
BOOL	boolean
OCTET	byte
INT16	short
INT32	int
INT64	long
UINT6	NOT SUPPORTED
UINT32	NOT SUPPORTED
UINT64	NOT SUPPORTED
FLOAT32	float
FLOAT64	double
FLOAT128	NOT SUPPORTED
CHAR8	char
CHAR32	char
STRING8	String
STRING32	String
BOOLSeq	boolean[], Boolean[], Collection<Boolean>
OCTETSeq	byte[], Byte[], Collection<Byte>
INT16Seq	Short[], Short[], Collection<Short>
INT32Seq	Integer[], Integer[], Collection<Integer>
INT64Seq	Long[], Long[], Collection<Long>
UINT6Seq	NOT SUPPORTED

UINT32Seq	NOT SUPPORTED
UINT64Seq	NOT SUPPORTED
FLOAT32Seq	float[], Float[], Collection<Float>
FLOAT64Seq	double[], Double[], Collection<Double>
FLOAT128Seq	NOT SUPPORTED
CHAR8Seq	char[], Character[], Collection<Character>
CHAR32Seq	char[], Character[], Collection<Character>
STRING8Seq	String[], Collection<String>
STRING32Seq	String[], Collection<String>

8.1.2.1.5 Return Codes

Type *ReturnCode* is mapped to an enum and an exception class.

Operations of the PIM declaring a *ReturnCode* output parameter throw a *ReturnCodeException* in case the *ReturnCode* is not RETCODE_OK.

Operations that declare a *ReturnCode* return value present the same return value in the PSM too.

8.1.2.1.6 Collection return values

An array of the mapped type is returned by operations of the PIM that return a collection of elements. Implementations may operate a transformation between their internal representations of the collections to the array form, for example by using the *toArray* operation of one of the classes of the *java.util.list* package.

For performance reasons, an overloaded version of the operation is also provided, which accepts an input array where the result returned by the operation will be stored. Only the elements that can fit in the length of the array will be returned.

8.1.2.1.7 Factory methods

Factory methods that create instrumentation entities and accept properties objects to configure them also present an overloaded version of the method, which does not declare the properties parameter. These methods will use the default properties and avoid the caller to have to specify a *null* argument in order to request the default values.

8.2 Instrumentation Domain PSMs

8.2.1 OMG Data Distribution Service

8.2.1.1 PIM to PSM Mapping Rules

8.2.1.1.1 Data type representation

The following table illustrates how sub-classes of *DataValue* are mapped to IDL data types:

Instrumentation Data Type	IDL Data Type
BOOL	boolean
OCTET	octet
INT16	short
INT32	long
INT64	long long
UINT6	unsigned short
UINT32	unsigned long
UINT64	unsigned long long
FLOAT32	float
FLOAT64	double
FLOAT128	long double
CHAR8	char
CHAR32	wchar
STRING8	string
STRING32	wstring
BOOLSeq	sequence<boolean>
OCTETSeq	sequence<octet>
INT16Seq	sequence<short>
INT32Seq	sequence<long>
INT64Seq	sequence<long long>
UINT6Seq	sequence<unsigned short>
UINT32Seq	sequence<unsigned long>
UINT64Seq	sequence<unsigned long long>
FLOAT32Seq	sequence<float>
FLOAT64Seq	sequence<double>
FLOAT128Seq	sequence<long double>
CHAR8Seq	sequence<char>
CHAR32Seq	sequence<wchar>
STRING8Seq	sequence<string>
STRING32Seq	sequence<wstring>

8.2.1.1.2 Instrumentation entities

Instrumentation entities defined by the API are uniquely associated with entities in the DDS middleware.

API entity	Middleware entity
InstrumentationService	*DDS_DomainParticipant*
ObservableScope	*DDS_Publisher and one DDS_DataWriter for each ObservableSchema of contained ObservableObject instances*
ObservableSchema	*DDS data-type and a DDS_Topic*
ObservableObject	Instance on *DDS_Topic*

The instance associated with an *ObservableObject* is identified by the following attributes:

- Host Identifier
- Name of the *InstrumentationService*
- Name of the *ObservableScope*
- Name of the *ObservableObject*

It is up to the user to guarantee that *ObservableObject* from multiple application map to separate instances in the *DDS_Topic* associated with their *ObservableSchema.*

8.2.1.1.3 Local instrumentation operations

This sub clause defines the operations performed by the data-distribution middleware in response to operations requested by an application using the Application Instrumentation API.

The mappings are presented in terms of "events" in the local instrumentation and corresponding operations carried out by the middleware.

API action	Middleware action
A new *InstrumentationService* is created	A *DDS_DomainParticipant* is created and uniquely associated with the *InstrumentationService/.*
An *InstrumentationService* is deleted	The *DDS_DomainParticipant* associated with the *InstrumentationService* and all its contained entities are deleted.
An *ObservableSchema* is activated in an *InstrumentationService*	1. A DDS data-type is created by mapping the *ObservableSchema* with the rules described in 8.2.1.1.4 ObservableSchema
	2. The DDS data-type is registered on the *DDS_DomainParticipant* associated with the *InstrumentationService,* using the *ObservableSchema*'s name.

	3. A new *DDS_Topic* is created using the registered data-type and it is uniquely associated with the *ObservableSchema;* the name of the topic is derived by prefixing the *ObservableSchema*'s name with the string "*AppInst::*"
An *ObservableScope* is created	A *DDS_Publisher* is created in the *DDS_DomainParticipant* associated with the *InstrumentationService;* the publisher is uniquely associated with the *ObservableScope.*
An *ObservableScope* is deleted	The *DDS_Publisher* associated with the *ObservableScope* is deleted, along with all its contained *DDS_DataWriter.*
An *ObservableObject* is created	A *DDS_DataWriter* to the *DDS_Topic* associated with the *ObservableObject*'s *ObservableSchema* is created in the *DDS_Publisher* associated with the *ObservableScope* containing the *ObservableObject,* if it does not exist already*;* A new instance, uniquely associated with the *ObservableObject*, is registered in the *ObservableSchema*'s *DDS_DataWriter.*
An *ObservableObject* is deleted	The instance associated with the *ObservableObject* is disposed. If no other *ObservableObject* of the same *ObservableSchema* exists within the enclosing *ObservableScope*, *DDS_DataWriter* for the *ObservableSchema* is deleted.
An *ObservableScope* distributes an *Observation* to the Instrumentation Domain	1. The Observation is converted to a sample of the *ObservableSchema's* DDS data-type.
	2. The sample is written to the DDS Global Data Space using the *DDS_DataWriter* associated with the *ObservableGroup* containing the *ObservableObject* that generated the *Observation.*

8.2.1.1.4 ObservableSchema

Each *ObservableSchema* is mapped to a complex DDS data-type. The *Field* entries contained in the *ObservableSchema* are converted to single attributes. A key attribute identifying the *ObservableObject* that generated the *Observation* is automatically added to the data-type.

The key attribute is added first to the generated type and it can be described in the following way, using IDL:

```
typedef string<MAX_STRING_LEN> SupportedString;

struct ObservationSource {
   SupportedString host_id;  //@key
   SupportedString service;  //@key
   SupportedString scope; //@key
```

```
   SupportedString object; //@key
}

struct ObservationHeader {
   ObservationSource source; //@key
   long sequence_number;
};
```

MAX_STRING_LEN is left unspecified.

The data-type resulting from the mapping of an example *ObservationSchema* named “Foo” can be generically represented in IDL as:

```
struct Foo {
   ObservationHeader header; //@key
   //<field mappings >
};
```

Fields are added to the DDS data-type according to the creation order in their *ObservableSchema*.
Each field is mapped to a corresponding entry with the same name and type determined using the table in 8.2.1.1.1 Data type representation.

8.2.1.1.4.1 Quality of Service of associated DDS Topic

The DDS Topic created for an *ObservableSchema* shall be configured with the following default Quality of Service (QoS) configuration.

QoS Property	Default Value
Durability	TRANSIENT_LOCAL
Deadline	Infinite
Latency Budget	0
Ownership	EXCLUSIVE
Ownership Strength	0
Liveliness	AUTOMATIC
Time-based Filter	0
Reliability	RELIABLE
Transport Priority	0
Lifespan	Infinite
Destination Order	BY_SOURCE_TIMESTAMP
History	KEEP_ALL
Resource Limits (max_samples, max_instances, max_samples_per_instance)	(LENTGH_UNLIMITED, LENTGH_UNLIMITED, LENTGH_UNLIMITED)

Writer Data Life-cycle (autodispose_unregistered_instances)	(True)
Reader Data Life-cycle (autopurge_nowriter_samples_delay, autopurge_disposed_samples_delay)	(Infinite, Infinite)
User Data	None

8.2.1.1.5 Extensions to *ObservableSchemaProperties*

The following attributes shall be added to *ObservableSchemaProperties* to allow configuration of the Quality of Service properties of every *DataWriter* publishing *Observation* samples from *ObservableObject* instances of a certain *ObservableSchema*.

ObservableSchemaProperties	
Attributes	
qos_reliability	String
qos_history	String
qos_history_depth	Integer
qos_resource_limits_max_samples	Integer
qos_resource_limits_max_instances	Integer
qos_resource_limits_max_samples_per_instance	Integer

8.2.1.1.5.1 qos_reliability

This attribute shall set the *Reliability* QoS policy of any *DataWriter* associated with the *ObservableSchema.* Values of this attribute shall be interpreted according to the following table. The default value for this attribute shall be "RELIABLE."

Value	**QoS Policy Value**
"RELIABLE"	RELIABLE
"BEST_EFFORT"	BEST_EFFORT
any other value	error

8.2.1.1.5.2 qos_history

This attribute shall set the *History* QoS policy of any *DataWriter* associated with the *ObservableSchema.* Values of this attribute shall be interpreted according to the following table. The default value for this attribute shall be "KEEP_ALL."

Value	QoS Policy Value
“KEEP_ALL”	KEEP_ALL
“KEEP_LAST”	KEEP_LAST
any other value	error

8.2.1.1.5.3 qos_history_depth

This attribute shall specify the *depth* property of the *History* QoS policy for any *DataWriter* associated with an *ObservableSchema*. The default value for this attribute is 0.

8.2.1.1.5.4 qos_resource_limits_max_samples

This attribute shall set the *max_samples* property of the *ResourceLimits* QoS policy of any *DataWriter* associated with the *ObservableSchema.* The default value for this attribute shall be -1, to indicate “LENTGH_UNLIMITED.”

8.2.1.1.5.5 qos_resource_limits_max_instances

This attribute shall set the *max_instances* property of the *ResourceLimits* QoS policy of any *DataWriter* associated with the *ObservableSchema.* The default value for this attribute shall be -1, to indicate “LENTGH_UNLIMITED.”

8.2.1.1.5.6 qos_resource_limits_max_samples_per_instance

This attribute shall set the *max_samples_per_instance* property of the *ResourceLimits* QoS policy of any *DataWriter* associated with the *ObservableSchema.* The default value for this attribute shall be -1, to indicate “LENTGH_UNLIMITED.”

8.2.1.2 Remote Service Interface

The *Remote Service Interface* provided by the *Instrumentation Domain* to remote application can be implemented using two DDS topics to receive operation requests and output their outcome from each local instrumentation in the distributed system.

These topics have a fixed name and corresponding data-types, one for operation requests and one for responses.

The data-type for requests and responses may be represented in IDL as:

```
const long MAX_STRING_LEN = 255;
const long MAX_SEQ_SIZE = 255;

typedef string<MAX_STRING_LEN> SupportedString;
typedef sequence<SupportedString, MAX_SEQ_SIZE> SupportedStringSeq;
typedef SupportedString InstrumentationId;
typedef SupportedString RequesterId;
typedef long RequestId;
```

```
enum RequestType {
   APPINST_OPERATION_CHECK_SAVE_OBSERVATION,
   APPINST_OPERATION_ENABLE_SAVE_OBSERVATION,
   APPINST_OPERATION_DISABLE_SAVE_OBSERVATION,
   APPINST_OPERATION_CHECK_DATA_COLLECTION,
   APPINST_OPERATION_ENABLE_DATA_COLLECTION,
   APPINST_OPERATION_DISABLE_DATA_COLLECTION,
   APPINST_OPERATION_UPDATE_DATA_PROCESSOR
};

typedef SupportedStringSeq RequestArgsKeys;
typedef SupportedStringSeq RequestArgsValues;
typedef SupportedStringSeq RequestOutcomeKeys;
typedef SupportedStringSeq RequestOutcomeValues;

struct RemoteServiceRequest {
   InstrumentationId instrumentation_id; //uniquely identifies the target instrumentation
   RequesterId requester_id; //uniquely identifies the application that request the operation
   RequestId request_id; //uniquely identifies the request with respect to its requester
   RequestType request_type; //indicates which operation should be performed by the service
   RequestArgsKeys request_args_keys; //names of the argument to be passed to the operation
   RequestArgsValues request_args_values; //arguments to be passed to the operation
};

struct RemoteServiceResponse {
   InstrumentationId instrumentation_id; //instrumentation where the request was handled
   RequesterId requester_id; //application that generated the request
   RequestId request_id; //id assigned to the request by the requester
   RequestOutcomeKeys request_outcome_keys; //name of output values of the operation
   RequestOutcomeValues request_outcome_values; //output values of the operation
};
```

The two Topics used to implement the Remote Service Interface shall have the following names and data types:

Type of topic	Name	Data Type
Requests	AppInst_RemoteServiceRequests	RemoteServiceRequest
Responses	AppInst_RemoteServiceResponses	RemoteServiceResponse

8.2.1.2.1 Quality of Service of associated DDS Topics

The DDS Topics created to implement the Remote Service Interface shall be configured with the following default Quality of Service (QoS) configuration.

QoS Property	Default Value
Durability	TRANSIENT_LOCAL
Deadline	Infinite
Latency Budget	0
Ownership	EXCLUSIVE
Ownership Strength	0
Liveliness	AUTOMATIC
Time-based Filter	0
Reliability	RELIABLE
Transport Priority	0
Lifespan	Infinite
Destination Order	BY_SOURCE_TIMESTAMP
History	KEEP_ALL
Resource Limits (max_samples, max_instances, max_samples_per_instance)	(LENTGH_UNLIMITED, LENTGH_UNLIMITED, LENTGH_UNLIMITED)
Writer Data Life-cycle (autodispose_unregistered_instances)	(True)
Reader Data Life-cycle (autopurge_nowriter_samples_delay, autopurge_disposed_samples_delay)	(Infinite, Infinite)
User Data	None

This page intentionally left blank.

9 Instrumentation Example (Non-normative)

9.1 General

This clause presents a complete example of a software system instrumented using the Application Instrumentation API. The example is included specification to provide better understanding of the scope of this specification and how it may be effectively used to instrument applications. It is not to be considered part of the normative specification.

The following sub clauses are laid out as follows:

- *Example Overview*: presents the example distributed system and its instrumentation requirements.
- *Instrumentation Configuration*: describes how the instrumentation requirements of the system may be modeled using the Application Instrumentation API.

9.2 Example Overview

The software system chosen for the example instrumentation is a radar track-management system. This kind of application usually has near real-time or real-time requirements, with critical need for minimizing the instrumentation's intrusiveness into the system's resources.

The instrumentation presented in the example includes a variety of information extracted from the application. These different types of information will provide the opportunity to show how the API may be leveraged to easily expose application data to remote consumers.

9.2.1 Instrumented System

The software system used in this example is a generic radar track-management system, described only in a qualitative way that includes a general description of each software component.

No assumption is made on the implementation platform of the system. Behavior of each component is presented in a programming language independent way, providing an abstract description of the module's purpose.

Figure 12 shows a possible architecture for the example system.

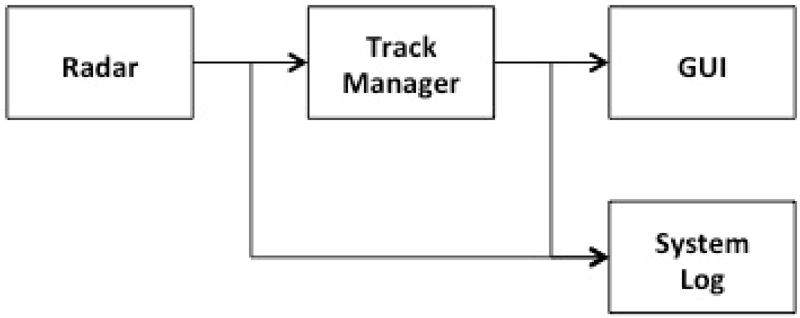

Figure 12 – Example Track-Management System

The system comprises four types of components:

- *Radar*: hardware radar equipment and attached control software, which periodically produces data samples containing the latest measurements.
- *Track Manager*: a software component responsible for monitoring data samples produced by the *Radar;* the component detects new objects in the measurements and creates new tracks in its state; it correlates new data samples with existing objects and updates the associated track's status; finally it marks tracks as disposed once their objects are not detected by the *Radar* anymore; track data is produced in output so that it may consumed by other components in the system.
- *GUI*: a display application used by end users of the system to monitor the state of the *Track Manager* and observe track data in real-time.
- *System Log*: a logging component, which records radar and track data for off-line analysis and general auditing of the system.

Each component is a software module that runs as an independent application on a physical system. An application communicates with other applications through a network infrastructure.

Each application accepts a set of configuration parameters. Parameters are specified at start-up and possibly updated during the execution of the system through some kind of remote configuration interface. The available configuration parameters depend on the software component. They include resources allocated to each application, such as memory storage, threads, and network bandwidth, and other parameters related to the specific role of each component.

Each application can consume and produce data, exchanging information through the network with other applications in the system. Components that produce data include the *Radar* and the *Track Manager*, which will operate on a periodic basis, with more or less stringent requirements on the maximum latency allowed between successive updates produced in output. Components that consume information (*Track Manager, GUI, System Log*) can all be abstracted as some processing logic that must be executed whenever the remote producers make new samples available.

9.2.2 Instrumentation Requirements

The components of the track-management system in the example are assumed to be operating under near real-time/real-time requirements. The system is expected to be able to consume a certain amount of radar measurements and produce track information in output with a guaranteed update rate.

These requirements make the monitoring of the performance of each component of critical importance. On-line monitoring of the system's performance is leveraged during the development and testing stages of the system. There it can help with exposing bottlenecks in the processing pipeline, optimizing configuration parameters and debugging the system.

The monitoring infrastructure is also useful once the system is deployed into production. Monitoring information may be accessed when suspicious behavior is observed in the system. Data extracted by the instrumentation may provide helpful hindsight on the operations of each application in the system and help identify unexpected problems.

The following table describes the information that will be extracted via instrumentation from the system.

Type of Information	**Description**
Module Performance	The timing and details of the operations performed by each component of the system must be monitored; the information should include initial and final time when an operation was carried out, the number of data samples processed by the operation and their size.
Track Processing Throughput	The *Track Manager* represents the core component of the system and its performance in processing track information must be monitored by computing the throughput of the updates to each track and the total aggregated throughput.
Application Configuration	Since each component's behavior depends on its configuration parameters, it is useful to be able to inspect the actual parameters used by an application and to monitor any changes that may occur to them over the application's execution.

All types of information shall be updated whenever they change in the instrumented applications. Aggregated values of relevant statistics shall also be provided, offering monitoring applications with averages over different time spans and tracking of min/max values for each stream.

9.3 Instrumentation Configuration

This sub clause presents a possible configuration of the instrumentation used by the example track-management system, described using the constructs and concepts of the Application Instrumentation API.

First, the *ObservableSchemas* that describe the information that will be produced by the instrumentation are presented. Then *ObservableScopes* and *ObservableObjects* used to produce information from the instrumented applications are described and finally the *DataProcessors* that will process collected *Observations* and control their distribution to remote monitoring applications.

9.3.1 Instrumentation Service

A single *InstrumentationService* will be created for each instrumented application. The *InstrumentationService* will be named after a unique, alphanumeric identifier assigned to the application.

9.3.2 Data Types

Each data-type used to describe monitored application data must be described by an *ObservableSchema*. This sub-clause describes all *ObservableSchemas* required to model the instrumented information of the system.

The *ObservableSchemas* include both *Fields* meant to be filled by instrumented applications and *Fields* that will be used by *DataProcessors* to store additional information computed from the one provided by the applications. Computed *Fields* are marked with an italic font in the tables describing each *ObservableSchema*'s *Fields*.

9.3.2.1 Module Performance

Performances of the instrumented applications are abstractly modeled through the number and size of items processed by each “operation” performed by the software modules contained in the applications. As introduced in 9.2.1 Instrumented System, applications operate in periodic loops, where they typically receive data from their inputs and process it, possibly generating output in response. Nevertheless, data may be internally passed through several processing modules, each one performing some operation whose performance should be tracked by the instrumentation.

The *OperationLog ObservableSchema* models the performance of each operation, providing fields to store the name of the operation, its timing information, and data about the items it processed.

Observations of this *ObservableSchema* will be processed by the instrumentation to produce an aggregated performance analysis of each software module. The *ModulePerformance ObservableSchema* contains only computed *Fields,* which will provide the average, minimum and maximum values of the performance of single operations.

Observations of *OperationLog* can be kept local to instrumented application to avoid burdening monitoring applications, and the interconnecting network infrastructure, with unnecessary data. Only aggregated performance will be distributed outside of the application’s instrumentation.

ObservableSchema **OperationLog**		
Field Name	**Field Type**	**Description**
module_name	STRING8	Name of the software module that performed the operation.
operation_name	STRING8	Name of the operation performed by the software module.
time_in	UINT64	Time when the operation started.
time_out	UINT64	Time when the operation was completed.
processed_items	UINT32	Number of items processed by the operation.
processed_items_size	UINT64	Total size of the items processed by the operation.

ObservableSchema **ModulePerformance**		
Field Name	**Field Type**	**Description**
module_name	STRING8	Name of the software module.
operations_total	UINT64	Total number of operations performed by the module.
time_total	UINT64	Total time taken by operations performed by the module.
operation_duration_avg	FLOAT64	Average time required to perform a single operation.
processed_items_total	UINT64	Total number of items processed by the module.
processed_items_avg	FLOAT64	Running average of the number of items processed

		by each operation performed by the module.
processed_items_min	UINT32	Minimum number of items processed by the module in a single operation.
processed_items_max	UINT32	Maximum number of items processed by the module in a single operation.
processed_items_size_total	UINT64	Total size of the items processed by the module.
processed_items_size_avg	FLOAT64	Running average of the size of the items processed by the module in a single operation.
processed_items_size_min	UINT64	Minimum size of items processed by the module in a single operation.
processed_items_size_max	UINT64	Maximum size of items processed by the module in a single operation.

9.3.2.2 Track Update Throughput

This *ObservableSchema* is used to describe the current state maintained by the *Track Manager* of each single track discovered from the radar measurements. *Observations* of this *ObservableSchema* will be generated whenever a track's status on the *Track Manager* is updated and they will be used to compute throughput of the track managing logic.

The information modeled by this *ObservableSchema* is only used by the local instrumentation installed in the *Track Manager* to compute the application's throughput. *Observations* will not be distributed to remote monitoring applications, which will only consume the generated throughput information.

The track update throughput will be automatically computed from *Observations* of the *TrackState ObservableSchema* produced by the instrumented applications.

Throughput will be calculated for each single track managed by the *Track Manager* as well as an aggregated statistic that includes all tracks. The *TrackThroughput ObservableSchema* models the throughput information for a single track*,* while *ObservableSchema AggregatedTrackThroughput* models the aggregated throughput of the processing of updates to all tracks in the system. Both *ObservableSchema* contain computed *Fields* to store running average, minimum, and maximum values of each type of throughput.

ObservableSchema **TrackState**		
Field Name	**Field Type**	**Description**
track_number	INT32	A unique identifier assigned by the system to each track.
update_time	UINT64	Time when the track's state was last updated.
track_discovered	BOOL	A boolean value signaling that a track has just been discovered and updated for the first time.
track_deleted	BOOL	A boolean value signaling that the track has been deleted and updated for the last time.

ObservableSchema **TrackThroughput**		
Field Name	**Field Type**	**Description**
tracks_count	UINT64	Total number of tracks currently managed by the system.
track_number	INT32	The unique identifier of the last track to be updated.
update_time	UINT64	Time of the last update made to the state of one of the tracks.
throughput	FLOAT64	The total update throughput currently measured for all tracks managed by the system.
throughput_avg	FLOAT64	Running average of the total update throughput.
throughput_min	FLOAT64	Minimum total update throughput.
throughput_max	FLOAT64	Maximum total update throughput.

9.3.2.3 Application Configuration

The configuration of applications in the track-management system is contained in a text file, which is loaded by the application at start-up. No assumption is made on the format of the configuration file's contents.

ObservableSchema **ApplicationConfiguration**		
Field Name	**Field Type**	**Description**
config_file_src	STRING8	A string, possibly following an URI scheme, identifying the source where the application read its configuration parameters (file, database, network location, etc.).
config_file_contents	STRING8	Contents of the configuration file read by the application.
system_name	STRING8	Name of the host where the application is deployed.
cpu_count	UINT16	Total number of CPU allocated to the application.
run_mode	INT16	How the application should operate (test, training, live, etc.).
thread_pool_mode	INT16	How the thread pool allocated to the application should be managed (dynamic, fixed-size, etc.).
thread_pool_initial	UINT16	The initial size of the application's thread pool.
thread_pool_max	UINT16	The maximum number of threads that can be allocated to the application's thread pool.
memory_alloc_mode	INT16	How memory allocation should be managed by the application (pre-allocate, dynamic, etc.).

`memory_alloc_initial`	`UINT64`	The initial memory allocated to the application.
`memory_alloc_max`	`UINT64`	The maximum amount of memory that can be allocated to the application.

9.3.3 Data Collection

In order to generate data that may be accessed by remote consumers, instrumented applications must create *ObservableScope* and *ObservableObject* instances. *ObservableObjects* will be used to store data and generate *Observations,* while their enclosing *ObservableScopes* will collect these *Observations,* pass them through a *DataProcessor,* and possibly distribute them to the remote monitoring applications.

Since an *ObservableScope* defines an independent, single-threaded, data processing context for the *Observations* of all *ObserableObjects* it contains, it is a good usage pattern to map only instrumented information that should be correlated together to the same *ObservableScope*. This will avoid tying uncorrelated *Observations*, which may be processed independently, to the same execution context, thus serializing their processing unnecessarily. *ObservableObjects* that represent unrelated information should be placed into separate *ObservableScope*, so that their *Observations* may be processed in separate threads if the implementation supports it. Even if processing is implemented using a single thread, the resulting serialization will be equal to the one that would have resulted from placing all *ObservableObjects* into the same *ObservableScope*.

For this reason, this example instrumentation adopts three separate *ObservableScope*, one for each category of instrumented information that will be collected from the system:

- *Module Performance*: this *ObservableScope* will contain *ObservableObjects* that generate *Observations* related to the application's operative performance.
- *Track Update Throughput*: this *ObservableScope* will contain *ObservableObjects* that generate *Observations* on the state of track processing and its throughput.
- *Application Configuration*: this *ObservableScope* will contain *ObservableObjects* that generate *Observations* of the configuration parameters of the application.

The following sub clauses will describe the purpose of each *ObservableScope,* the *ObservableObjects* that will be created, and how they will be used by the application and the instrumentation.

9.3.3.1 Module Performance

A single *ObservableScope* is dedicated to the monitoring of the performance of the instrumented applications modules.

Within this *ObservableScope*, two *ObservableObjects,* of *ObservableSchemas OperationLog* and *ModulePerformance* respectively, are created for each module registered with the instrumented application. This choice is made to allow separate modules to access their dedicated *ObservableObjects* independently of other modules, possibly from within a concurrent execution context. Sharing *ObservableObject* instances between concurrent application threads could lead to unexpected behavior, since the *ObservableObject* interface offers only safe multi-thread access from a single application thread and the execution context of its *ObservableScope*.

If the number, and names, of modules used by the application are known at start-up, *ObservableObject* instances may be created immediately during the initialization of the instrumentation. If the application is allowed to load new modules dynamically, new *ObservableObjects* can also be created dynamically, provided data collection is first disabled on the *ObservableScope*. While the *ObservableScope*'s processing context is disabled, *Observations* will be accumulated in the internal queues of *ObservableObjects* associated with other modules. If the *Observation* allocation is properly configured, the loss of data should me minimal once the *ObservableObject* has been created and data collection is enabled again in the *ObservableScope.*

Note that the code of the software modules will only access the *ObservableObjects* to generate *OperationLog Observations*. *ObservableObjects* of the *ModulePerformance ObservableSchema* will be used by a *DataProcessor* to automatically compute the aggregated statistics from the single operation ones.

ObservableScope **ModulePerformanceMonitor**		
ObservableObject Name	**ObservableSchema**	**Description**
<module_name>_ operations_log	OperationLog	Produces logs of every (high level) operation performed by the software module with name *<module_name>*.
<module_name>_ module_performance	ModulePerformance	Produces aggregated statistics for the software module with name *<module_name>*, computed by a *DataProcessor* from the *Observations* of *ObservableObject* *<module_name>_operations_log*

9.3.3.2 Track Update Throughput

An *ObservableScope* is dedicated to the collection of *Observations* of track updates and the computation of the resulting update throughput.

Two *ObservableObjects* are created within this *ObservableScope*, each one producing data about multiple tracks.

An *ObservableObject* of type *TrackState* will be used to produce *Observations* whenever a track is updated. A *DataProcessor* will pick up these *Observations* to compute the total resulting throughput. The results will be produced via the other *ObservableObject,* of types *TrackThroughput.*

The choice of not creating an *ObservableObject* per track to generate the track update logs and the single track throughput statistics is motivated by the typically great number of tracks managed by a system of this type during its life-cycle. Dedicating an *ObservableObject* to a single track would cause the number of *ObservableObjects* to explode. The associated overhead caused by the creations, and possibly deletion, of the *ObservableObject* would require an unacceptable cost from the application's performance.

Similarly to 9.3.3.1 Module Performance, it is best to consider the creation of multiple *ObservableObjects* only if they must be accessed from different application contexts. It is assumed in this case that the track management will occur in a single-threaded software module. If that were not to be the case, than multiple *TrackState ObservableObjects* should be created, to safely collect information from multiple sources.

ObservableScope **TrackThrougputMonitor**		
ObservableObject Name	**ObservableSchema**	**Description**
track_state_log	TrackState	Produces logs of every updated made to a track.
track_throughput	TrackThroughput	Produces throughput information, computed by a *DataProcessor* from the *Observations* of *ObservableObject track_state_log.*

9.3.3.3 Application Configuration

An *ObservableScope* is dedicated to the collection of *Observations* of the configuration parameters of each instrumented application.

The *ObservableScope* contains a single *ObservableObject* of type *ApplicationConfiguration,* which will be used by the application to produce snapshots of its configuration parameters, when it is first loaded and whenever they are modified during its execution.

ObservableScope **AppConfigMonitor**		
ObservableObject Name	**ObservableSchema**	**Description**
app_config	Application Configuration	Produces logs of the configuration parameters loaded by the instrumented applications, at start-up and whenever they are modified.

9.3.4 Data Processing

One of the advantages offered by the Application Instrumentation API over manual instrumentation of an application is the support offered for easy manipulation and control of collected data by means of the *DataProcessor* interface.

This example makes use of *DataProcessors* to automatically compute interesting information from the *Observations* generated by the code of instrumented applications. While the implementation of the custom processing function may be considered an additional cost of the instrumentation, it is a one-time cost that may greatly reduce the intrusiveness of the instrumentation into the application code (by limiting the additional logic that must be added to the application to generated the required information, which is instead encapsulated by the *DataProcessor*), and it may also be typically reused in instrumenting multiple applications (as long as the same *ObservableSchema* are used and *ObservableObjects* are accessed with the same semantics by the applications).

Moreover, several processing functions can be sufficiently generalized so that they might be implemented independently of the *ObservableSchema* of the *Observation*. These operations are presented in 9.3.4.1 General Processing.

The following sub clauses present processing operation specific to each type of instrumented information.

9.3.4.1 General Processing

Some processing operations are sufficiently general that they can be made independent of the *ObservableSchema* of an *Observation* and only depend on specific *Fields* to be available to provide input to the operation and store its output.

In this example, these operation include:

- Computation of the cumulative running average of a *Field*.
- Tracking of the minimum and maximum value of a *Field.*
- Computation of the throughput of track update events.

The description of each operation presents a general algorithm for its computation, characterizes its inputs and outputs in terms of *Fields,* and defines that necessary state that must be stored to correctly carry out the processing.

9.3.4.1.1 Running Average

Input Fields

- **latest_val**: the latest value that must be used to update the average.

Output Fields

- **current_avg**: the cumulative running average of all values.

Processing State:

- **last_avg**: a floating point value, initialized to 0, containing the last average value computed.
- **total_items**: an integer value, initialized to 0, counting the number of values averaged so far.

Algorithm:

- Compute the current average using the formula:
    - **current_avg = (latest_value + total_items * latest_avg) / (total_items + 1)**

- Increment **total_items** by one unit.
    - **total_items += 1**

- Store the latest average value.
    - **last_avg** = **current_avg**

9.3.4.1.2 Min/Max

Input Fields

- **latest_val**: the latest value that must be used to update the average.

Output Fields

- **min**: the minimum value observed so far.
- **max**: the maximum value observed so far.

Processing State:

- **current_min**: a floating point or integer value (depending on the type of **latest_val**), initialized to its type's maximum value, containing the minimum value observed so far.
- **current_max**: a floating point or integer value (depending on the type of **latest_val**), initialized to its type's minimum value, containing the maximum value observed so far.

Algorithm:

- Check and update minimum:
    - **if (latest_val < current_min) min = latest_val**

- Check and update minimum:
    - **if (latest_val > current_max) max = latest_val**

- Store the latest for the next value:
    - **current_min = min**
    - **current_max = max**

9.3.4.1.3 Throughput

Input Fields

- **update_time**: time-stamp of the latest event

Output Fields

- **current_throughput**: the current throughput of the monitored events

Processing State:

- **last_update_time**: the time-stamp of the last event processed

Algorithm:

- Calculate the period of time elapsed between this event and the previous:
    - **elapsed_time = update_time - last_update_time**

- Compute the instant throughput in number of events per unit of time:
    - **current_throughput = 1 / elapsed_time**

- Store the event's time for following processing:
    - **last_update_time = update_time**

9.3.4.2 Module Performance

An implementation of the *DataProcessor* interface will be created to properly handle the computation of performance information about a module.

An instance of this type of *DataProcessor* will be attached to all *ObservableObjects* of type *OperationLog* contained in *ObservableScope ModulePerformanceMonitor.*

When processing an *Observation* produced by one of these *ObservableObjects,* the *DataProcessor* will be responsible for computing the following values:

- Duration of the operation logged by the *Observation,* computed as the difference between fields *time_out* and *time_in*.
- The average duration of all operations logged so far.
- The total sum of items processed by all operations logged so far.
- The average number of items processed by a single logged operation.
- The minimum and maximum number of items processed by a single logged operation so far.
- The total sum of the size of the items porcessed by all operation logged so far.
- The average size of items processed by a single logged operation.
- The minimum and maximum number of items processed by a single logged operation so far.

All values can be computed by using the generic processing functions presented in 9.3.4.1 General Processing on *Fields* contained in the *Observations.*

The *DataProcessor* will attach a data structure of type *OperationLogProcessingState* (described in the table at the end of the sub clause) to every *ObservableObject* of type *OperationLog* to maintain the necessary processing state between successive invocations of its *process_observations* operation on the same *ObservableObject.*

The results computed at each iteration will be stored in the *ObservableObject* of type *ModulePerformance* associated with the same module. After storing the results in the *ObservableObject,* the *DataProcessor* will generate an *Observation* of it that will be distributed to remote applications without further processing. No *DataProcessor* is attached to *ObservableObjects* of type *ModulePerformance*.

All *Observations* of type *OperationLog* are marked with flag LOCAL and they are not distributed to remote applications.

OperationLogProcessingState		
Field Name	**Field Type**	**Description**
`total_operations`	`UINT64`	A counter of the total number of operations logged, used when computing all running averages.
`last_duration_avg`	`FLOAT64`	The last value computed for the average duration of a single operation.
`processed_items_total`	`UINT64`	The total sum of items processed by logged operations.
`processed_items_avg`	`FLOAT64`	The last value computed for the average number of items processed by a single operation.
`processed_items_min`	`UINT64`	The current minimum number of processed items observed so far in a single operation.
`processed_items_max`	`UINT64`	The current maximum number of processed items observed so far in a single operation.
`processed_items_size_total`	`UINT64`	The total sum of the size of items processed by logged operations.
`processed_items_size_avg`	`FLOAT64`	The last value computed for the average size of items processed by a single operation.
`processed_items_size_min`	`UINT64`	The current minimum size of items processed observed so far in a single operation.
`processed_items_size_max`	`UINT64`	The current maximum size of items processed observed so far in a single operation.

9.3.4.3 Track Update Throughput

An implementation of the *DataProcessor* interface will be created to compute the throughput of track updates carried out by the system.

An instance of this type of *DataProcessor* will be attached to all *ObservableObjects* of type *TrackState* contained in *ObservableScope TrackThroughputMonitor.*

When processing an *Observation* produced by one of these *ObservableObjects,* the *DataProcessor* will be responsible for computing the following values:

- Total number of tracks currently managed by the system; this number is increased whenever an *Observation* with *track_discovered* set to *True* is processed, and decreased whenever an *Observation* with *track_deleted* set to *True* is received instead.
- The current throughput of track state updates carried out by the systems.
- The average value for the computed throughput.
- The minimum/maximum throughput values observed so far.

Similarly to module performance, all values can be computed by using the generic processing functions presented in 9.3.4.1 General Processing on *Fields* contained in the *Observations.*

The *DataProcessor* will attach a data structure of type *TrackThroughputProcessingState* (described in the table at the end of the sub clause) to the *ObservableObject* of type *TrackState* to maintain the necessary processing state between successive invocations of its *process_observations* operation on the *ObservableObject.*

The results computed at each iteration will be stored in the *ObservableObject* of type *TrackThroughput*. After storing the results in the *ObservableObject,* the *DataProcessor* will generate an *Observation* of it that will be distributed to remote applications without further processing. No *DataProcessor* is attached to *ObservableObjects* of type *TrackThroughput*.

All *Observations* of type *TrackState* are marked with flag LOCAL and they are not distributed to remote applications.

TrackThroughputProcessingState		
Field Name	**Field Type**	**Description**
tracks_count	UINT64	Total number of tracks currently managed by the system.
total_updates	UINT64	Total number of track state updates received so far.
last_update_time	UINT64	Time of the last update.
throughput_avg	FLOAT64	The last value computed for the average throughput of update events.
throughput_min	FLOAT64	The current minimum throughput value observed so far.
throughput_max	FLOAT64	The current maximum throughput value observed so far.

This page intentionally left blank.