Date: December 2005

Life Sciences Analysis Engine

OMG Available Specification
Version 1.0

formal/05-12-02

OBJECT MANAGEMENT GROUP

USE OF SPECIFICATION - TERMS, CONDITIONS & NOTICES

The material in this document details an Object Management Group specification in accordance with the terms, conditions and notices set forth below. This document does not represent a commitment to implement any portion of this specification in any company's products. The information contained in this document is subject to change without notice.

LICENSES

The companies listed above have granted to the Object Management Group, Inc. (OMG) a nonexclusive, royalty-free, paid up, worldwide license to copy and distribute this document and to modify this document and distribute copies of the modified version. Each of the copyright holders listed above has agreed that no person shall be deemed to have infringed the copyright in the included material of any such copyright holder by reason of having used the specification set forth herein or having conformed any computer software to the specification.

Subject to all of the terms and conditions below, the owners of the copyright in this specification hereby grant you a fully-paid up, non-exclusive, nontransferable, perpetual, worldwide license (without the right to sublicense), to use this specification to create and distribute software and special purpose specifications that are based upon this specification, and to use, copy, and distribute this specification as provided under the Copyright Act; provided that: (1) both the copyright notice identified above and this permission notice appear on any copies of this specification; (2) the use of the specifications is for informational purposes and will not be copied or posted on any network computer or broadcast in any media and will not be otherwise resold or transferred for commercial purposes; and (3) no modifications are made to this specification. This limited permission automatically terminates without notice if you breach any of these terms or conditions. Upon termination, you will destroy immediately any copies of the specifications in your possession or control.

PATENTS

The attention of adopters is directed to the possibility that compliance with or adoption of OMG specifications may require use of an invention covered by patent rights. OMG shall not be responsible for identifying patents for which a license may be required by any OMG specification, or for conducting legal inquiries into the legal validity or scope of those patents that are brought to its attention. OMG specifications are prospective and advisory only. Prospective users are responsible for protecting themselves against liability for infringement of patents.

GENERAL USE RESTRICTIONS

Any unauthorized use of this specification may violate copyright laws, trademark laws, and communications regulations and statutes. This document contains information which is protected by copyright. All Rights Reserved. No part of this work covered by copyright herein may be reproduced or used in any form or by any means--graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems--without permission of the copyright owner.

DISCLAIMER OF WARRANTY

WHILE THIS PUBLICATION IS BELIEVED TO BE ACCURATE, IT IS PROVIDED "AS IS" AND MAY CONTAIN ERRORS OR MISPRINTS. THE OBJECT MANAGEMENT GROUP AND THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS PUBLICATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTY OF TITLE OR OWNERSHIP, IMPLIED WARRANTY OF MERCHANTABILITY OR WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE OR USE. IN NO EVENT SHALL THE OBJECT MANAGEMENT GROUP OR ANY OF THE COMPANIES LISTED ABOVE BE LIABLE FOR ERRORS CONTAINED

HEREIN OR FOR DIRECT, INDIRECT, INCIDENTAL, SPECIAL, CONSEQUENTIAL, RELIANCE OR COVER DAMAGES, INCLUDING LOSS OF PROFITS, REVENUE, DATA OR USE, INCURRED BY ANY USER OR ANY THIRD PARTY IN CONNECTION WITH THE FURNISHING, PERFORMANCE, OR USE OF THIS MATERIAL, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

The entire risk as to the quality and performance of software developed using this specification is borne by you. This disclaimer of warranty constitutes an essential part of the license granted to you to use this specification.

RESTRICTED RIGHTS LEGEND

Use, duplication or disclosure by the U.S. Government is subject to the restrictions set forth in subparagraph (c) (1) (ii) of The Rights in Technical Data and Computer Software Clause at DFARS 252.227-7013 or in subparagraph (c)(1) and (2) of the Commercial Computer Software - Restricted Rights clauses at 48 C.F.R. 52.227-19 or as specified in 48 C.F.R. 227-7202-2 of the DoD F.A.R. Supplement and its successors, or as specified in 48 C.F.R. 12.212 of the Federal Acquisition Regulations and its successors, as applicable. The specification copyright owners are as indicated above and may be contacted through the Object Management Group, 250 First Avenue, Needham, MA 02494, U.S.A.

TRADEMARKS

The OMG Object Management Group Logo®, CORBA®, CORBA Academy®, The Information Brokerage®, XMI® and IIOP® are registered trademarks of the Object Management Group. OMG™, Object Management Group™, CORBA logos™, OMG Interface Definition Language (IDL)™, The Architecture of Choice for a Changing World™, CORBAservices™, CORBAfacilities™, CORBAmed™, CORBAnet™, Integrate 2002™, Middleware That's Everywhere™, UML™, Unified Modeling Language™, The UML Cube logo™, MOF™, CWM™, The CWM Logo™, Model Driven Architecture™, Model Driven Architecture Logos™, MDA™, OMG Model Driven Architecture™, OMG MDA™ and the XMI Logo™ are trademarks of the Object Management Group. All other products or company names mentioned are used for identification purposes only, and may be trademarks of their respective owners.

COMPLIANCE

The copyright holders listed above acknowledge that the Object Management Group (acting itself or through its designees) is and shall at all times be the sole entity that may authorize developers, suppliers and sellers of computer software to use certification marks, trademarks or other special designations to indicate compliance with these materials.

Software developed under the terms of this license may claim compliance or conformance with this specification if and only if the software compliance is of a nature fully matching the applicable compliance points as stated in the specification. Software developed only partially matching the applicable compliance points may claim only that the software was based on this specification, but may not claim compliance or conformance with this specification. In the event that testing suites are implemented or approved by Object Management Group, Inc., software developed using this specification may claim compliance or conformance with the specification only if the software satisfactorily completes the testing suites.

OMG's Issue Reporting Procedure

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 (http://www.omg.org/technology/agreement.htm).

Table of Contents

Preface

About the Object Management Group

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. A catalog of all OMG Specifications Catalog is available from the OMG website at:

http://www.omg.org/technology/documents/spec_catalog.htm

Specifications within the Catalog are organized by the following categories:

OMG Modeling Specifications

- UML
- MOF
- XMI
- CWM
- Profile specifications.

OMG Middleware Specifications

- CORBA/IIOP
- IDL/Language Mappings
- Specialized CORBA specifications
- CORBA Component Model (CCM).

Platform Specific Model and Interface Specifications

- CORBAservices

- CORBAfacilities
- OMG Domain specifications
- OMG Embedded Intelligence specifications
- OMG Security 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
250 First Avenue
Suite 100
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 section headings where no distinction is necessary.

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

Helvetica/Arial - 10 pt. Bold: OMG Interface Definition Language (OMG IDL) and syntax elements.

`Courier - 10 pt. Bold:` Programming language elements.

Helvetica/Arial - 10 pt: Exceptions

Note – Terms that appear in *italics* are defined in the glossary. Italic text also represents the name of a document, specification, or other publication.

Issues

The reader is encouraged to report any technical or editing issues/problems with this specification to *http://www.omg.org/technology/agreement.htm*.

1 Scope

In January 2000, the Biomolecular Sequence Analysis specification (BSA) was made available. It has two, almost independent parts. This specification modernizes and extends what was previously defined in the BSA part defining the components for supporting sequence analysis through a generic analysis design.

2 Conformance

The normative parts of this specification are:

- Platform independent model expressed in the attached XML file created according to XMI format rules, v1.1, using Unisys Rose XMI Exporter JCR.2, version 1.3.2.
- Platform specific model for Web Services, for those who are implementing Web Services model.
- Platform specific model for Java, for those who are implementing Java model.
- At least one platform specific model, from those listed above, must be implemented.

If there is any inconsistency, or discrepancy between generality and specificity, between the platform independent (PIM) and platform specific (PSM) models, the platform specific model has precedence.

The normative specifications of the platform specific models are expressed in the accompanied files in a document whose number is given in the History section (or elsewhere in this document). Parts of these files may also appear in the explanatory text of this document. If they do and if there are some differences or discrepancies, the contents of the accompanied files has precedence.

The specification uses XML structures to define notification events, notification descriptor, and minimal analysis metadata. These XML structures are the same in any platform-specific model. Any compliant implementation must use these structures. Additionally, this specification also suggests optional metadata (extended analysis metadata) that may be used.

There are also several optional methods (see their details elsewhere in this document):

- Method run_and_wait_for, depending on the underlying platform-specific protocol and the time-outs of other components involved (e.g., proxy servers), may be less reliable for longer times. Therefore, an implementation may refuse to implement this method without losing compliancy with this specification, assuming it raises proper exception as defined in platform-specific models.
- The server-push notification is optional because for some platform-specific implementations it may be difficult to achieve it.

3 Normative References

1. Web Services Notification, http://www-106.ibm.com/developerworks/library/specification/ws-notification/
2. CORBA Event Service, http://www.omg.org/technology/documents/corbaservices_spec_catalog.htm#Events

3. CORBA Notification Service, http://www.omg.org/technology/documents/corbaservices_spec_catalog.htm#Notification
4. Web Services Glossary (W3C), W3C Working Draft, August 2003: http://www.w3.org/TR/ws-gloss/
5. Basic Profile Version 1.0a (W3C): http://www.ws-i.org/Profiles/Basic/2003-08/BasicProfile-1.0a.htm
6. Web Services Basic Profile: http://www.ws-i.org/docs/charters/WSBasic_Profile_Charter1-1.pdf
7. Apache Axis implementation of the SOAP submission to W3C, http://ws.apache.org/axis/

4 Terms and Definitions

There are no specific terms or definitions used in this specification.

5 Symbols

There are no specific symbols included in this specification.

6 Additional Information

6.1 Relationship to Existing OMG Specifications

- Biomolecular Sequence Analysis (formal/2001-11-01)

  The Biomolecular Sequence Analysis specification (BSA) defines the components for supporting sequence analysis through a generic analysis design as CORBA interfaces and structure, and XML structures. The Life Sciences Analysis Engine (this specification) adapts the same XML structures almost without changes, and re-uses and extends concepts from CORBA interfaces, making them technology-neutral. It does not, however, addresses the issues related to the biological objects.

- OMG: Model Driven Architecture

  MDA is used as a fundamental concept. Both platform-independent (PIM) and platform-specific models (PSM) are defined here.

- OMG: XML Interdata Interchange

  The platform-independent model for this specification for designed as a UML model that was converted into XMI model exchange format as defined in the XML Interdata Interchange specification.

- OMG: CORBA Event Service

  CORBA Event Service is defined here as one possible protocol for the client-server negotiation - but far from being the only one. This specification does not dictate any CORBA-only technologies.

- OMG: CORBA Notification Service

  The same applies here as for CORBA Event Service above.

- OMG: Life Sciences Identifiers

  Even though it is not normative, the implementations are encouraged to adopt LSID specification for naming entities, especially those that represent more permanent objects (such as names of analysis services).

6.2 Acknowledgements

The authors of this document wish to express their sincere appreciation to those listed below (in alphabetic order) for their invaluable contributions of ideas and experience. Ultimately, the conclusions expressed in this document are those of the authors and do not necessarily reflect the views or ideas of these individuals, nor does the inclusion of their names imply an endorsement of the final product.

Neil Blue	Neil.Blue@biowizdom.com
Mark Greenwood	markg@cs.man.ac.uk
Tom Oinn	tmo@ebi.ac.uk
Peter Rice	pmr@ebi.ac.uk
Ugis Sarkans	ugis@ebi.ac.uk

There were also people involved in the evaluation of our submissions. They did not only evaluate the proposed specification but also contributed with suggestions and ideas.

7 Introduction

In the Life Sciences domains there are many computer methods for analyzing and deriving data. The goal of this specification is to describe how to access them in an inter-operable way in a distributed environment. It includes the execution and control of computer methods (represented as applications, programs, processes, threads, or otherwise) and dealing with their inputs, output, and functionality. Particularly, it addresses:

- both synchronous and asynchronous invocation of the remote tools.
- the ability to transport data to and from the remote tools both in its entirety and in smaller chunks.
- the ability to interact with the tools that already have been started.

This specification addresses the issues described above.

7.1 UML Representation

A UML diagram can summarize the whole response. The platform independent model is expressed in two UML packages:

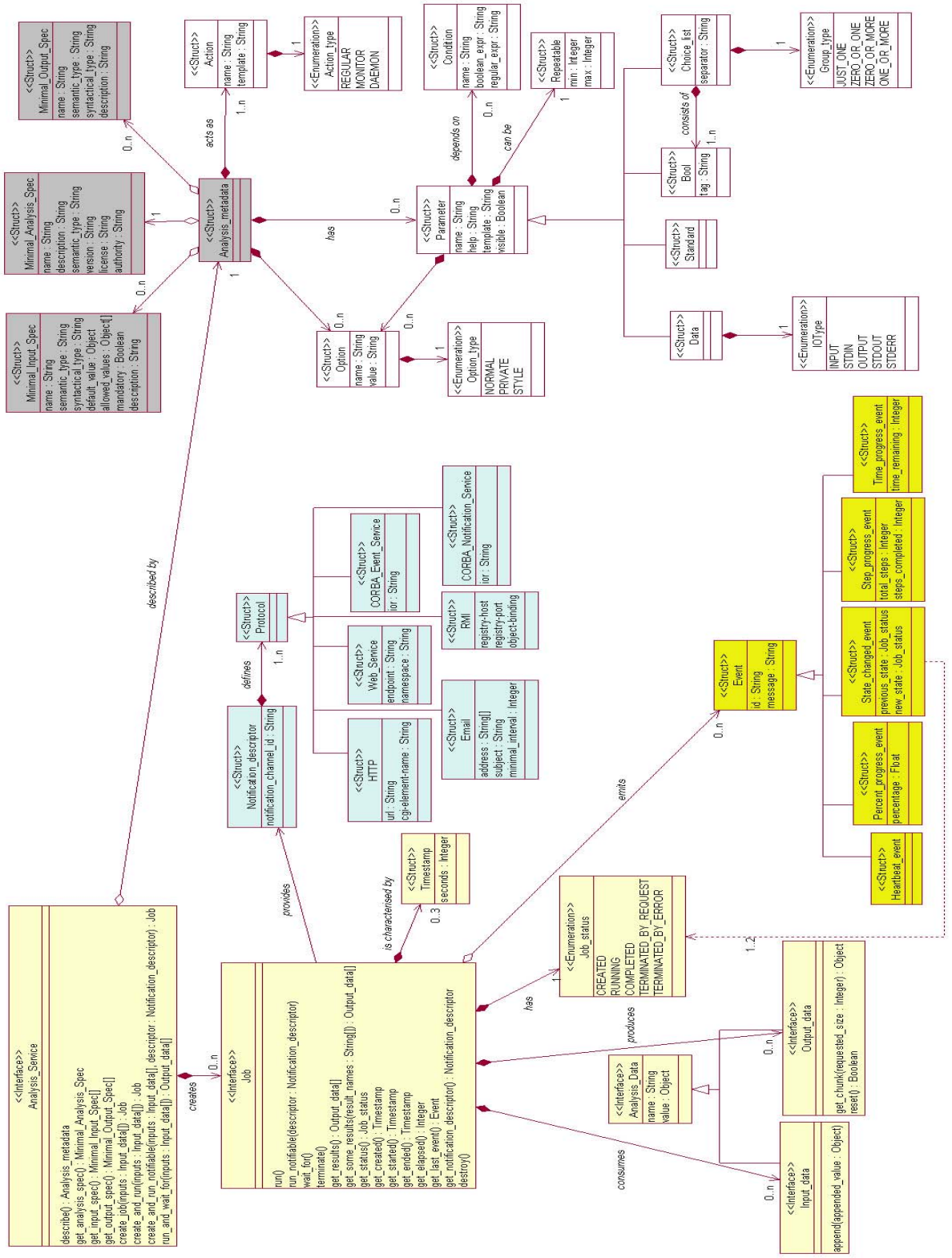
- AnalysisService that has all interfaces and structure, except analysis metadata. The picture is available in the accompanied files as **`AnalysisServicePackage.jpg`**.
- AnalysisMetadata that contains minimal metadata model. The picture is available as **`AnalysisMetadataPackage.jpg`**.

Both packages are available in a Rational Rose compatible file LSAE.mdl, and as a picture in **`LSAE_all.jpg`**. The normative, however, is the file **`LSAE-XMI-1.1.xml`**.

In order to describe individual interfaces and data structures of the model this document uses both UML diagrams and pseudo-code whose resemblance to Java language is only accidental.

Some methods and structures use a data type Object. This data type represents any type, and its usage indicates that the real type used in the run-time is not defined by this specification (the implementation is allowed to use here any data type). The concrete run-time data type is defined in minimal analysis metadata.

The UML Model of LSAE is shown in the figure on the next page.


<<Interface>>
Analysis_Service
describe() : Analysis_metadata
get_analysis_spec() : Minimal_Analysis_Spec
get_input_spec() : Minimal_Input_Spec[]
get_output_spec() : Minimal_Output_Spec[]
create_job(inputs : Input_data[]) : Job
create_and_run(inputs : Input_data[]) : Job
create_and_run_notifiable(inputs : Input_data[], descriptor : Notification_descriptor) : Job
run_and_wait_for(inputs : Input_data[]) : Output_data[]
<<Interface>>
Job
run()
run_notifiable(descriptor : Notification_descriptor)
wait_for()
terminate()
get_results() : Output_data[]
get_some_results(result_names : String[]) : Output_data[]
get_status() : Job_status
get_created() : Timestamp
get_started() : Timestamp
get_ended() : Timestamp
get_elapsed() : Integer
get_last_event() : Event
get_notification_descriptor() : Notification_descriptor
destroy()
<<Interface>>
Input_data
append(appended_value : Object)
<<Interface>>
Output_data
get_chunk(requested_size : Integer) : Object
reset() : Boolean
<<Interface>>
Analysis_Data
name : String
value : Object
<<Struct>>
Minimal_Output_Spec
name : String
semantic_type : String
syntactical_type : String
description : String
<<Struct>>
Minimal_Analysis_Spec
name : String
description : String
semantic_type : String
version : String
license : String
authority : String
<<Struct>>
Minimal_Input_Spec
name : String
semantic_type : String
syntactical_type : String
default_value : Object
allowed_values : Object[]
mandatory : Boolean
description : String
<<Struct>>
Analysis_metadata
<<Struct>>
Action
name: String
template : String
<<Enumeration>>
Action_type
REGULAR
MONITOR
DAEMON
<<Struct>>
Condition
name : String
boolean_expr : String
regular_expr : String
<<Struct>>
Repeatable
min : Integer
max : Integer
<<Struct>>
Parameter
name : String
help : String
template : String
visible : Boolean
<<Struct>>
Option
name : String
value : String
<<Enumeration>>
Option_type
NORMAL
PRIVATE
STYLE
<<Struct>>
Choice_list
separator : String
<<Struct>>
Bool
tag : String
<<Struct>>
Standard
<<Struct>>
Data
<<Enumeration>>
Group_type
JUST_ONE
ZERO_OR_ONE
ZERO_OR_MORE
ONE_OR_MORE
<<Enumeration>>
IOType
INPUT
STDIN
OUTPUT
STDOUT
STDERR
<<Struct>>
Notification_descriptor
notification_channel_id : String
<<Struct>>
Protocol
<<Struct>>
HTTP
url : String
cgi-element-name : String
<<Struct>>
Web_Service
endpoint : String
namespace : String
<<Struct>>
CORBA_Event_Service
ior : String
<<Struct>>
CORBA_Notification_Service
ior : String
<<Struct>>
RMI
registry-host
registry-port
object-binding
<<Struct>>
Email
address : String[]
subject : String
minimal_interval : Integer
<<Struct>>
Timestamp
seconds : Integer
<<Enumeration>>
Job_status
CREATED
RUNNING
COMPLETED
TERMINATED_BY_REQUEST
TERMINATED_BY_ERROR
<<Struct>>
Event
id : String
message : String
<<Struct>>
Time_progress_event
time_remaining : Integer
<<Struct>>
Step_progress_event
total_steps : Integer
steps_completed : Integer
<<Struct>>
State_changed_event
previous_state : Job_status
new_state : Job_status
<<Struct>>
Percent_progress_event
percentage : Float
<<Struct>>
Heartbeat_event
creates
provides
defines
described by
acts as
has
depends on
can be
consists of
is characterised by
emits
produces
consumes

8 LSAE Interfaces

8.1 Introduction

The interfaces and data structures described in this platform-independent model consist of several parts:

- Metadata describing analysis tools and interfaces to access them.
- Interfaces and structures to control analysis tools.
- Interfaces and structures for notification events.

8.2 Metadata Describing Analysis Tools

8.2.1 Rationale

The eternal problem with metadata is that they are expected to be both flexible and extendible and at the same time to be highly interoperable. Any solution for this dilemma also indicates the place where to define metadata. They can be defined as a part of the API of the analysis services. Such solutions make metadata more explicit, more interoperable, but also much less flexible. Or, they can be defined independently on the analysis interfaces, for example as an XML DTD or XML Schema - supposedly shared by all platform-specific models. Such a solution will make them very flexible and extendible because they are not part of the MDA architecture at all. That also makes them less interoperable.

Being aware of the metadata dilemma outlined above, this specification treats metadata in the following way:

- It defines a platform-independent, minimal metadata model that includes only the most crucial, and therefore, mandatory metadata. It contains metadata that are necessary to understand how an analysis tool can be used and what it does.
- However, it does not express this minimal metadata model in the platform-specific data types. The minimal metadata model is always, disregarding the technology used by a particular PSM, expressed as an XML string, whose DTD is normative and is defined by this specification.
- Additionally, this specification includes an optional (non-normative) part describing additional metadata, focusing primarily on metadata that are useful (but not crucial) not only for the service providers, but also for the clients (for example, metadata describing available parameters of an analysis service can be used to build a better GUI for such service).

8.2.2 Minimal Metadata Model

The mandatory part of metadata (called a minimal metadata model or a minimal analysis metadata) is defined in XML DTD **`MinimalAnalysisMetadata.dtd`** file. The document with accompanied files includes also **`MinimalAnalysisMetadata.xsd`** file that was generated from the previous one and is there for convenience.

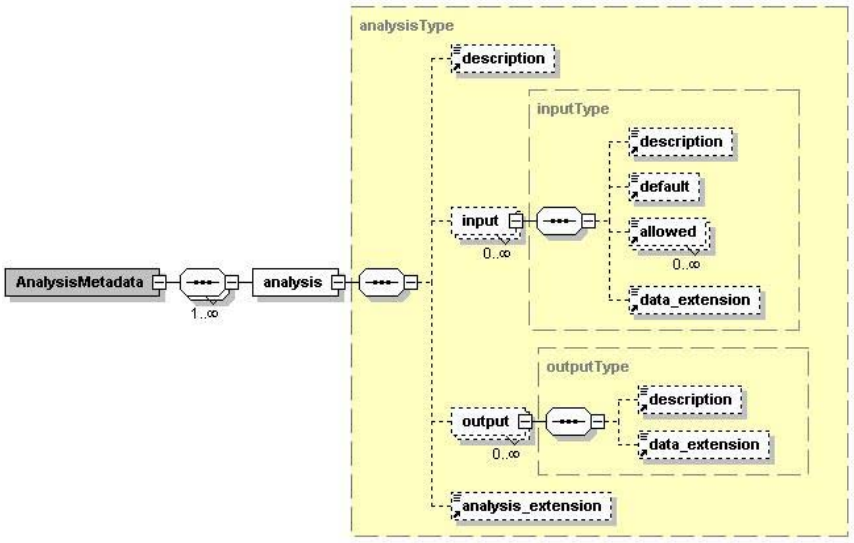


Figure 8.1 - Minimal Metadata Model

Note that minimal metadata model is represented as an XML document independently on the platform-specific model. The description below uses more readable names as they appear in the UML diagram - but the normative names are those in the XML DTD.

8.2.2.1 Analysis_metadata

This is a starting structure defining the whole analysis service.

Minimal_Analysis_Spec

The `name` should be reasonably unique, the `semantic_type` qualifies the type of the analysis service, the authority points to an author or to a broader group where the service belongs to, the `description` is a human-readable text, the version further distinguishes the analysis and the `license` defines copyright and other rules dealing with intellectual property.

Only the `name` is mandatory. For `semantic_type`, it is recommended to re-use established ontologies.

Minimal_Input_Spec

Each input describes a piece of data that is entering given analysis tool. It must be uniquely named within this analysis, and its `syntactical_type` must be specified. This specification does not define how the type should be expressed. It is recommended to re-use established data type systems, and the implementation should document what types it uses.

The `semantic_type` expresses the meaning of this input. The ultimate goal is to use these semantic types to classify analysis tools and to be able to establish how individual analysis services can be chained together. Again, this specification does not define its own semantic types and recommends re-using existing ontologies.

Any input can be `mandatory`. It can have a `default_value` and a list of `allowed_values`. A human-readable `description` is recommended.

Minimal_Output_Spec

Each output describes a result produced by given analysis tool. It must be uniquely named within this analysis tool, and its `syntactical_type` must be specified. There are two pre-defined names indicating results with specific contents - see details in Section 8.3.1.1, “Special output names.” Similarly with the inputs, it has a `semantic_type` and a `description`.

8.2.3 Extended Metadata Model (optional)

This part is optional. All methods for controlling analysis tools can be correctly used without these metadata. All crucial information (such as how to name inputs, what types of results to expect and what kind of analysis tool is invoked) is in the minimal metadata model described above. Therefore, the basic interoperability between clients and servers is guaranteed by mandatory metadata.

By recommending additional metadata, this specification aims to provide metadata that can be exchanged between individual implementations and that can be re-used. The extended metadata also may be useful for clients to build richer GUIs and to be able to validate (some) input data independently on server, and in advance.

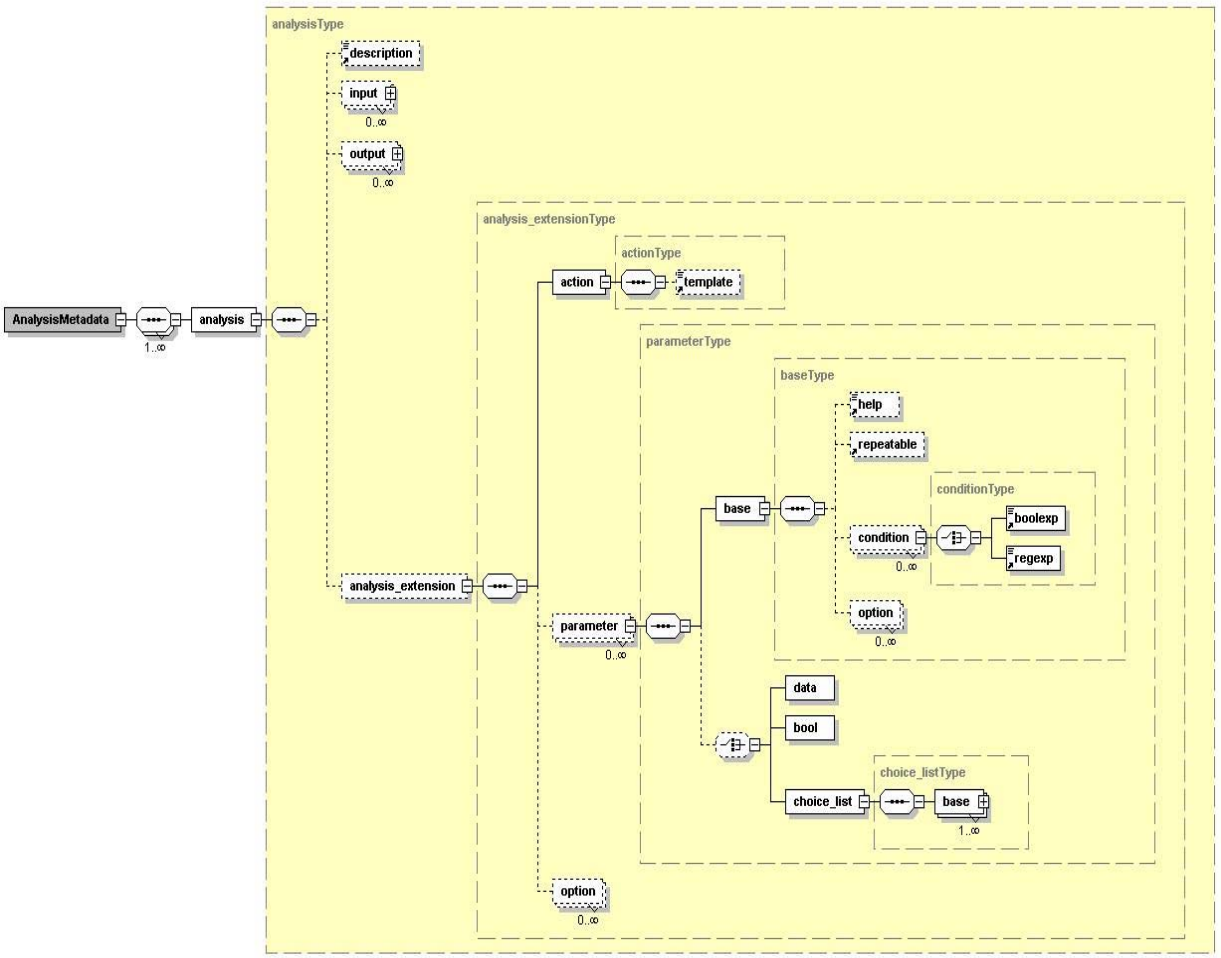


Figure 8.2 - Extended Metadata Model

8.2.3.1 (Extended) Analysis_metadata

All extended metadata are attached to the `Analysis_metadata` entity.

Action

Each analysis service should have at least one Action defining what underlying analysis tool to invoke. Its name can be a program name to be executed, or a class name to link to, or something else depending on the nature of the analysis tool. The `template` suggests how to invoke it. Its content again depends on the underlying analysis tool: it can be, for example, a template for creating a command line, or a signature of a method that has to be called.

The `Action_type` allows further qualifying how to use and how to control the underlying analysis tool:

- `REGULAR`

  It indicates a “normal” action, such as a program invocation. If the analysis service has this action type, then all input data must be available in the time of execution. If there are more actions associated with an analysis service, they should be either `REGULAR` or `DAEMON`, but not mixed.

- `DAEMON`

  It indicates that the underlying analysis tool is capable to accept input data even after it had been started. This action type can be used to simulate an interactive behavior.

- `MONITOR`

  This is an action monitoring the underlying analysis tool. It is usually used to produce `server-push` notifications. The actions of this type can be freely mixed with other types.

Option

This is a placeholder for extensions. Each analysis (and also its subparts) may have a list of properties attached. These properties are just ordinary `name/value` pairs. Each option can be qualified as:

- `NORMAL`

  A regular option that is supposed to be shared with (visible to) clients.

- `PRIVATE`

  An option that is used by the service implementation and is not supposed to be propagated to clients.

- `STYLE`

  A property serving as a hint to the advanced clients for building GUIs.

Parameter

The name “parameter” is a legacy referring to the command-line tools. But its content is quite generic. Each parameter describes a piece of data that is either entering given analysis tool, or that is produced by the tool. In fact each parameter extends either `Minimal_Input_Spec` or `Minimal_Output_Spec` entity. Typically a parameter describes:

- What is the type and format of the entering data. It can describe a small piece of data, such as a command-line option, or a real data input, such as an input file.

- How to present this piece of data to the underlying analysis tool. For example, a command-line tool can get input data on its command line, or it can read them from its standard input.
- What are the constraints for the values of these data and how they depend on values of the other parameters.
- What is the type and format of the producing data, and how they can be obtained from the underlying analysis tool (e.g., in a file or by reading analysis' standard or standard error output).

Each parameter has a unique name that attaches it to the corresponding `Minimal_Input_Spec` or `Minimal_Output_Spec` entity. A longer human-readable description resides in `help`. If `visible` is false, the clients should not be allowed to send any value for this parameter. The template defines how to format the whole parameter (usually using some kind of variable substitution mechanism).

Condition

Whether to use or not to use a parameter may depend on the run-time values of other parameters. This dependency is expressed by a list of conditions, each of them identifiable by a `name`, and containing a Boolean (`boolean_expr`) or regular (`regular_expr`) expression. The language/format used for any of these expressions usually uses some kind of variable substitution mechanism (being able to refer to other parameters and conditions by their names). Note that the language/format for expressions is not part of this specification. A set of conditions is evaluated as a logical intersection of results of their expressions.

Repeatable

Usually, in the run-time, a parameter gets none or one value. But they may be cases that a parameter run-time value can be repeated. For example, a command-line tool may expect on its command line several files, but an unknown number of them in advance. The attributes min and max defines how many times it can be repeated; value -1 indicates an infinite number.

Parameters can be of several kinds:

Standard

This is the most common parameter. It does not add any other attributes to the base `Parameter` entity.

Data

This kind of parameter is used for the "main" or "big" input and output data - in a traditional way, usually expressed as files or data sockets. The `Data` kind is the only parameter that can correspond to a `Minimal_Output_Spec` entity (all other parameters always represent only the input values). The `IOType` specifies if its value is an input (`INPUT`, `STDIN`), or an output (`OUTPUT`, `STDOUT`, `STDERR`) and how it is sent to/from the analysis tool.

Bool

This parameter represents a Boolean input. If its run-time value is true (or it is missing but its default value is true), then the `tag` is used to form the parameter.

Choice_list

This parameter groups together one or more `Bool` kind parameters, and it may separate them using its `separator`. The allowed number of `Bools` is given by the `Group_type`.

8.3 Accessing Analysis Tools

A life cycle of an invocation of an analysis tool consists of a creation of a job, feeding it with input data, executing it, waiting for its completion (or interrupting it earlier) and fetching resulting data. It can be accompanied by listening to the changes of the execution status.

Each analysis tool is represented by an `Analysis_service` object. When a service is executed (using a set of given input data) it creates a `Job` object that represents a single execution of the underlying analysis tool. Each `Analysis_service` object can create many jobs during its lifetime, each of them with a new set of input data.

The main goal of the `Analysis_service` and `Job` interfaces is to allow:

- Finding all information about an analysis tool.
- Feeding the analysis tool with input data in an appropriate format.
- Executing the tool either in a synchronous (blocking) way or in an asynchronous way.
- Receiving notifications about the tool status.
- Fetching resulting data produced by the tool in an appropriate time.

Note that this specification does not include how and from where to get `Analysis_service` objects. Their creation and/or discovery are not within the scope of this specification. However, the platform specific parts include optional comments on a simple discovery of the `Analysis_service` objects.

8.3.1 Input and Output Data

Each `Analysis_service` has one or more inputs. Every input has its descriptive part and a real value. The descriptive part (such as type of the real value) is constant (it does not depend on the current real value) and is available from the analysis metadata. The real value is represented by an `Input_data` object - one per each input data.

```
Input_data {
    string  name;
    object  value;
    void    append (object appended_value);
}
```

The name identifies uniquely each data input. The allowed names are defined in the analysis metadata (by `Minimal_Input_Spec` entity) and they can be retrieved using method **`get_input_spec()`**. The `values` are any objects whose types are again defined in the analysis metadata and who are also obtainable using **`get_input_spec()`**. The implementation is allowed to limit the number of supported types for data inputs.

The method `append` allows sending data in smaller chunks. Its parameter `appended_value` will be concatenated with the attribute value. The precise semantics of the concatenation depends on the type of `value`. The client can append data as long as it does not start the tool. After that moment an implementation may refuse to accept additional inputs (or may not if it represents an interactive tool).

Each `Analysis_service` has one or more results (outputs). Every result has its descriptive part and a real value. The descriptive part (such as type of the real value) is constant (it does not depend on the current real value) and is available from the analysis metadata (by `Minimal_Output_Spec` entity). The real value is represented by an `Output_data` object - one per each result.

```
Output_data {
    string      name;
    object      value;
    object      get_chunk (integer requested_size);
    boolean     reset();
}
```

The `name` uniquely identifies each result. The allowed names are defined in the analysis metadata and they can be retrieved using method **`get_result_spec()`**. The `values` are any objects whose types are again defined in the analysis metadata and who are also obtainable using **`get_result_spec()`**. The implementation is allowed to limit the number of supported types for results.

The method `get_chunk()` allows requesting only part of the resulting data. The parameter `requested_size` is a maximum length (in bytes) of the returned value. If the returned value is empty (the definition of the emptiness depends on the specific platform), no other data chunks are available.

However, after calling the `reset()` method, and if the call was successful, which was indicated by returning the `true` value, the same result is reachable again.

Note that independently on how the result data are obtained (either by calling methods returning the whole `Output_data` object, or objects - as described elsewhere in this document, or by getting data chunks by calling `get_chunk()` method), they may not be complete. The implementation may return only intermediate results. The only way to find that the results are complete and will never change again is to ask for the execution status and get back status code, indicating that the execution has finished.

8.3.1.1 Special output names

There are two pre-defined names indicating results with specific contents:

- `report`
- `detailed_status`

An implementation is allowed not to provide these specific results but it should not provide its regular results under these names.

The `report` result contains provenance data about the analysis invocation. Its type is `string`.

The `detailed_status` contains a number indicating success or failure of the analysis invocation in its simplest way. Its type is `integer`. The intended purpose of this result is to allow an easy checking, especially when the analysis is used in the workflow systems. The success is indicated by any of the following numbers:

- 0

  Traditionally on the UNIX systems, a zero indicates a successful exit status.

- 200

  Number 200 is common for the HTTP-based applications, where again it indicates a success.

8.3.2 Controlling Analysis Execution

The main strength of the `Analysis_service` object is to allow to start and to control an invocation of the underlying analysis tool. There are several methods to achieve it in several different ways:

```
Analysis_service {
    Output_data[] run_and_wait_for (Input_data[] inputs);
    Job create_job (Input_data[] inputs);
    Job create_and_run (Input_data[] inputs);
    Job create_and_run_notifiable (
        Input_data[] inputs,
        Notification_descriptor descriptor
    );
    …
}
```

The underlying analysis tool may be invoked synchronously or asynchronously. Any asynchronous invocation requires creating a `Job` object that can be used for further inquires regarding the underlying analysis. Obviously, such manner needs a state-full implementation.

Output_data[] run_and_wait_for (Input_data[] inputs)

This is a blocking, synchronous method invoking the underlying analysis tool with the given input data, waiting until the analysis tool finishes its job and returning back all available results. It is a convenient combination of several asynchronous methods described below.

This is designed to allow a full state-less server implementation. Note that this method keeps an open connection with the server until the job is finished, which may be for quite a long time. Therefore, depending on the underlying platform-specific protocol and the time-outs of other components involved (e.g., proxy servers), it may be less reliable for longer times. Consequently, an implementation may refuse to implement this method without losing compliancy with this specification, assuming it raises proper exception as defined in platform-specific models.

Note that by using this method there is no way to get additional information usually passed by a notification channel (e.g., events happening during the job life-cycle). A service implementation may consider passing some of those as an additional result (e.g., termination status/code).

Job create_job (Input_data[] inputs)

It creates an instance of a job with given input data but does not start it. The `Job` has its own methods for starting and controlling an invocation of the underlying analysis tool.

Job create_and_run (Input_data[] inputs)

It creates an instance of a job with given input data and it executes it. It is a convenient method combining the `create_job` and `Job.run()` methods.

```
Job create_and_run_notifiable (
    Input_data[] inputs,
    Notification_descriptor descriptor
)
```

It does the same thing as the `create_and_run()` method with addition of the channel notification negotiation as described elsewhere in this document.

8.3.2.1 Appending the input data

The methods creating a `Job` fill it with the input data. Each `Input_data` object has the ability to append chunks of data to itself (method `append()`). This allows building large sets of input data without exceeding memory available for clients or/and for the data transport.

Although the platform independent model does not constrain this ability in any way, it is very likely that the platform specific solutions will restrict it only for the asynchronous calls (and not, for example, for `run_and_wait_for()` method).

Another issue with the data appending is how to know that the input data are complete. This relates to the type of the analysis service, as defined in the analysis metadata. The rules are:

- If the analysis service is described as a *regular* service (see the metadata chapter), then the data appending should be refused after the analysis has been started. Which means that data appending is, for this kind of service, only allowed for jobs created by the `create_job()` method (and not by methods `create_and_run()` or `create_and_run_notifiable()` because these methods also immediately start the job).
- If the analysis service is described as a *daemon* service, then the data appending is allowed until the job finishes. This allows mimicking an interactive process: a client fills some data, calls to start a job, gets results back (job is still running), and feeds the same job by appending new input data.

8.3.2.2 Job

A `Job` is an object representing one single invocation of an analysis tool with one set of input data.

```
Job {
    run();
    void run_notifiable (Notification_descriptor descriptor);
    wait_for();
    terminate();
    Output_data[] get_results();
    Output_data[] get_some_results (string[] result_names);
    destroy();
    Job_status get_status();
    Timestamp get_created();
    Timestamp get_started();
    Timestamp get_ended();
    integer get_elapsed();
    Event get_last_event();
    Notification_descriptor get_notification_descriptor();
}
```

Having an instance of a `Job` one can invoke the following `Job`'s methods:

void run()

It starts an underlying analysis and returns immediately, without waiting for its completion. It uses input data that were sent (and possibly validated) when this job has been created. This method can be run only once (which means that if more invocations with the same input data are wanted, then more jobs have to be created).

void run_notifiable (NotificationDescriptor descriptor)

It does the same thing as the `run()` method with addition of the notification negotiation as described elsewhere in this document.

void wait_for()

It blocks the caller until the underlying running analysis is completed. The implementation raises an exception if this method is called on a job that has not yet been started (e.g., by the `run()` method).

Because the blocking may take a long time the same consequences as described in the `run_and_wait_for()` method should be considered.

void terminate()

It terminates the running job. The implementation raises an exception if the underlying job has not yet been started, but it does not raise any exception if it has finished in the meantime.

Output_data[] get_results()

It returns all available results from the underlying analysis tool. The results may not be complete unless the job has finished. After that moment, this method always should return the same data. Incomplete results may be those that do not contain some results at all, or that have only intermediate data for some or all results.

Note that the `Output_data` objects have its own method how to return resulting data in the smaller chunks, in order to protect the clients and the data transport layer from the exceeding available memory.

Output_data[] get_some_results (string[] result_names)

It returns only named results from the underlying analysis tool. The same note about completeness as for `get_results()` applies.

Job_status get_status()

enum Job_status {
    CREATED
    RUNNING
    COMPLETED
    TERMINATED_BY_REQUEST
    TERMINATED_BY_ERROR
}

The returned status indicates the current state of the job. The enumeration above indicates the minimum that any implementation should understand/provide:

- CREATED indicates a job that has been created but not yet executed.
- RUNNING indicates a job that has been started but has not finished yet. Note that depending on the implementation it does not necessarily mean that the underlying analysis tool is really running - it may be just waiting in a queuing system.
- COMPLETED indicates that a job has finished successfully without any external intervention.
- TERMINATED_BY_REQUEST indicates that a job has been completed after a request from a caller (using method

`terminate()`). It does not say anything about the success or failure of the finished job.

- TERMINATED_BY_ERROR indicates that a job was completed and has concluded with a failure.

void destroy()

It allows freeing all resources attached to this `Job` instance, including removing the resulting data of the underlying analysis tool. All subsequent method calls on the same `Job` instance will raise an exception.

Timestamp get_created()
Timestamp get_started()
Timestamp get_ended()

```
Timestamp {
    long seconds;
}
```

The methods above return various time characteristics. The `seconds` are counted from the beginning of the UNIX epoch (1.1.1970). They return -1 if the particular time characteristic is not (yet) available.

integer get_elapsed()

This method returns milliseconds indicating how long the job execution took, completely or so far. It returns -1 if the time is not available.

For the `Job`'s remaining methods dealing with notification see the next chapter.

8.3.3 Notifications

A running job changes its state and the user may be interested in being notified about these changes. A prominent change in the job lifecycle is, of course, when the job terminates (either by an error or by the user request or by its own nature). The interfaces for controlling an analysis tool provide two ways how to pass information about job changes:

- Client-poll notification
- Server-push notification, represented by a notification channel

The *client-poll* is mandatory for being compliant with this specification, the *server-push* not because for some platform-specific implementations it may be difficult to achieve it.

Both mechanisms differ in a way *how* the events are being exchanged but both share the same format of the events themselves.

8.3.3.1 Notification events

The notification events are all derived from the base object `Event`:

```
Event {
    string id;
    string message;
}
```

Each event is identifiable by its `id` and carries a `message`. Because events from several analysis services notifying about several jobs can be mixed the `id` must be reasonably unique.

The `Event` objects, disregarding how they are sent to clients, and disregarding what platform-specific model is used, are expressed as short XML strings.

The syntax describing the events is simple and extendible in order to include additional provider-specific events. The DTD and XSD are in the accompanied files:

- **`AnalysisEvent.dtd`**
- **`AnalysisEvent.xsd`**

The XSD file was generated for convenience - if there are any discrepancies between these two files, the DTD must be considered as the normative one.

There is also a convenient diagram showing the structure of an analysis event in a file **`AnalysisEvents.jpg`**.

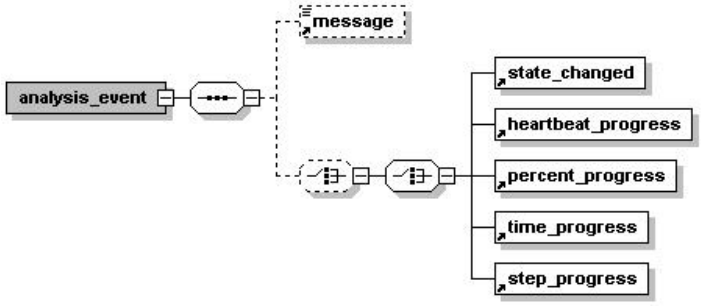


Figure 8.3 - Analysis Events

There are five defined types of analysis events. They all “inherit” from the base, general event.

General (base) event

A general event (and all other events) may have a single message string and an identifier. The message gives some free-form text description of the current progress, and the identifier may be used to filter events when they are sent together with events representing other jobs. An example:

```
<?xml version = "1.0"?>
<!DOCTYPE analysis_event SYSTEM "AnalysisEvent.dtd">
<analysis_event timestamp="today">
    <message>This is a general analysis event.</message>
</analysis_event>
```

Heartbeat event

Heartbeat event indicates that the reported job is still alive and either running or waiting for being executed. An example:

```
<?xml version = "1.0"?>
<!DOCTYPE analysis_event SYSTEM "AnalysisEvent.dtd">
<analysis_event timestamp="today">
    <message>This is a HEARTBEAT analysis event.</message>
    <heartbeat_progress/>
</analysis_event>
```

Percent progress event

Percent progress event provides information regarding the relative amount of work completed by job in terms of percentage complete. The percentage parameter must be greater or equal to 0 and less than or equal to 100. An example:

```
<?xml version = "1.0"?>
<!DOCTYPE analysis_event SYSTEM "AnalysisEvent.dtd">
<analysis_event timestamp="today">
    <message>This is a PERCENT PROGRESS analysis event.</message>
    <percent_progress percentage="52"/>
</analysis_event>
```

Job State changed event

Job State changed event indicates that a job's status has changed. The elements `previous_state` and `new_state` are the stringified versions of the `Job_status` enumeration. An example:

```
<?xml version = "1.0"?>
<!DOCTYPE analysis_event SYSTEM "AnalysisEvent.dtd">
<analysis_event timestamp="today">
    <message>This is a STATUS CHANGED analysis event.</message>
    <state_changed previous_state="created" new_state="running"/>
</analysis_event>
```

Step progress event

Step progress event indicates the total number of steps to be executed by this job and the number of steps completed so far. Multiple Step progress events dealing with the same job must have the same total number of steps. The `steps_completed` parameter must be less than or equal to the `total_steps` parameter. An example:

```
<?xml version = "1.0"?>
<!DOCTYPE analysis_event SYSTEM "AnalysisEvent.dtd">
<analysis_event timestamp="today">
    <message>This is a STEP PROGRESS analysis event.</message>
    <step_progress total_steps="10" steps_completed="5"/>
</analysis_event>
```

Time progress event

Time progress event indicates the estimated number of seconds before the job is completed. There is no requirement that the estimated completion time decreases. An example:

```
<?xml version = "1.0"?>
<!DOCTYPE analysis_event SYSTEM "AnalysisEvent.dtd">
<analysis_event timestamp="today">
    <message>This is a TIME PROGRESS analysis event.</message>
    <time_progress remaining="324"/>
</analysis_event>
```

Provider-defined events

Here is an example of a new event defined by a service provider. The extension is defined by an internal DTD that extends the original one:

```
<?xml version = "1.0"?>
<!DOCTYPE analysis_event SYSTEM "AnalysisEvent.dtd" [
<!ENTITY % event_body_template "(my_progress | state_changed | heartbeat_progress |
    percent_progress | time_progress | step_progress)">
<!ELEMENT my_progress EMPTY>
<!ATTLIST my_progress
```

```
        more CDATA #REQUIRED
        less CDATA #REQUIRED>
]>
<analysis_event timestamp="today">
        <message>This is a USER-DEFINED analysis event.</message>
        <my_progress more="yes" less="no"/>
</analysis_event>
```

Because jobs are allowed to generate such service-provider extended events, the clients should be always written in a way that they ignore events they do not understand.

8.3.3.2 Client-poll notification

The `Job`'s method `get_last_event()` represents a *client-poll* action:

Event get_last_event()

It returns the last event that happened to the given job. This polling does not guarantee that the client gets all events - some of them may be lost between two polling intervals.

8.3.3.3 Notification channel

The notification channel mechanism is a more complex (and more powerful) method of notification. It uses another service, or even set of services, to transport event messages (which may involve things like setting an expiration time, secure channels, or postponed and re-tried deliveries etc.). This specification allows to use existing standards for the notification, or to use the homemade but simple ones. What mechanism is actually used depends on the result of the client-server negotiation.

Usually, it is the server (service-provider's program) who knows about a suitable notification channel. The client who wishes to be notified must ask the server to get it - it is a *per* job request (which does not, however, preclude the server to use the same notification channel for more jobs and more clients). Here is a pseudo-code to achieve that:

```
Job job = service.create_job (inputs);
Notification_descriptor nd = job.get_notification_descriptor();
// use the 'nd' (if it is not empty):
//    - subscribe itself to it (if it expects a subscription)
//    - perhaps start a separate thread waiting
//    for server-push calls
job.run();
```

However, a client may wish to negotiate to use its own notification channel (e.g., because it is guaranteed to be used only by this client, or because it uses a special quality of service, or because it prefers to use a different notification protocol - such as an email notification). The proposed notification descriptor can be passed in the `runNotifiable(jobId,myDescriptor)` or `createAndRunNotifiable(inputs,myDescriptor)`, as shown in this pseudo-code:

```
Job job = service.createJob (inputs);
job.run (myNotificationDescriptor);
```

The server may refuse to use the suggested notification channel (e.g., if it does not understand the notification protocol defined in the notification descriptor).

```
Notification_descriptor {
    string notification_channel_id;
    Protocol[] protocols;
}
```

All details about a notification channel are defined in a *notification descriptor*, which is an XML string whose DTD and XSD are in the accompanied files:

`NotificationDescriptor.dtd`

`NotificationDescriptor.xsd`

The XSD file was generated for convenience - if there are any discrepancies between these two files, the DTD should be considered as the normative one.

There is also a convenient diagram showing the structure of a notification descriptor in a file **`NotificationDescriptor.jpg`**.

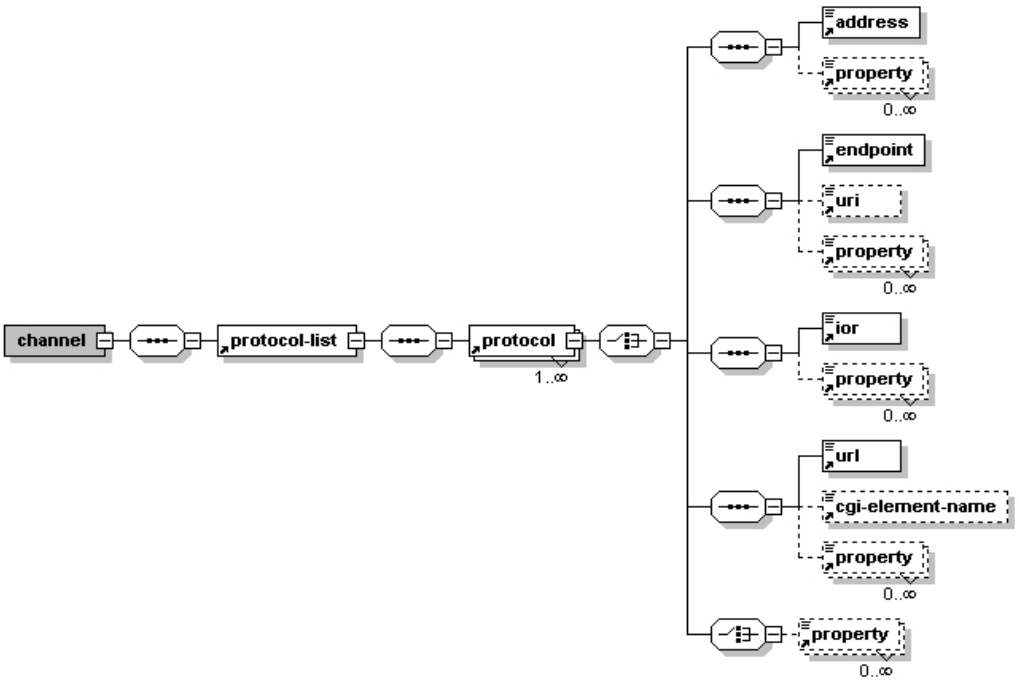


Figure 8.4 - Notification Descriptor

A notification descriptor describes a notification *channel* that can be represented by one or more *protocols*. When a descriptor is created by a client, the protocols indicate what mechanisms are understood by the client - the client is supposed to use any of them. When a descriptor is created by a server, the protocols indicate what mechanisms are supported and used by the server. The producer of this channel (server) is supposed to push events to all of them.

Each channel has an *identifier* that can be useful when the same channel is shared by more jobs. The both sides can save some initialization efforts if they see that the same channel is used again.

Each protocol is recognized by its *type*.

How to use individual protocols depends on their types. For example, some protocols may require subscriptions. The notification descriptor gives opportunity to use existing standards - but their description is out of the scope of this specification. Here is only a brief description of more spread protocols - and what specific properties are expected to appear in the notification descriptors.

Email protocol (type = smtp)

The events are sent by email to given email addresses, defined by one or more mandatory members `address`. Other properties are optional. The recognized property names are:

- `subject`

  A string defining an email subject header. This may be useful for automatic filtering of events.

- `minimal_interval`

  A minimal number of seconds that should elapse before other event email is sent. It protects email boxes to be easily overwhelmed.

Web Services Notification (type = ws-full)

The events are sent using a standard for the Web Service Notification [1]. A notification service is defined by its endpoint, accompanied optionally by its `uri` (namespace).

CORBA Event Service (type = corba-event)

The events are available using a standard CORBA Event service [2]. The mandatory member is an `ior` defining an inter-operable object reference of an Event Channel.

CORBA Notification Service (type = corba-notification)

The events are available using a standard CORBA Notification service [3]. The mandatory member is an `ior` defining an inter-operable object reference of a Notification Channel.

Java RMI (type = rmi)

The events are delivered from Java to Java using RMI calls. The mandatory properties define the place where to send events:

- `registry-host`
- `registry-port`
- `object-binding`

HTTP GET/POST (type = http)

The events are uploaded to a given `url`. The URL expects to be a program (cgi-bin, servlet, php, etc.) that gets events under the name `cgi-element-name` using CGI-BIN conventions.

8.3.4 Accessing Analysis Metadata

Every `Analysis_service` is self-describing by having methods publishing analysis metadata.

Analysis_service {
Analysis_metadata describe()
Minimal_Input_Spec[] get_input_spec()

Minimal_Output_Spec[] get_output_spec()
Minimal_Analysis_Spec get_analysis_spec()
…
}

Analysis_metadata describe()

It returns a complete set of metadata describing this analysis service. The returned metadata must comply at least with the minimal metadata model but can contain also extended (optional) metadata.

Minimal_Input_Spec[] get_input_spec()

It returns information about all possible data inputs. Each element of the returned array describes one data input. Note that it does not contain the input data themselves (as never with metadata). Also note that this is a convenient method retrieving the same data (but only part of it) as method describe().

The returned metadata represent the minimal metadata model for inputs.

Minimal_Output_Spec[] get_output_spec()

It returns information about all possible results. Each element of the returned array describes one result. Note that it does not contain the results themselves. Also note that this is a convenient method retrieving the same data (but only part of it) as method `describe()`.

The returned metadata represent the minimal metadata model for outputs.

Minimal_Analysis_Spec get_analysis_spec()

It returns basic information about this analysis service. Note that this is a convenient method retrieving the same data (but only part of it) as method `describe()`.

The returned metadata represent the minimal metadata model for analysis services.

8.4 Error Codes

Methods of the services described in the previous chapters may raise exceptions in order to stipulate that a fault state occurred. This chapter lists all possible error codes that the implementation should use in such situations. The codes will be used in the platform-specific way, as defined in the platform-specific sections later in this document. The names (in the table below) are only explanatory and may be used in the platform-specific models in the language-specific manner.

Note that some error codes make sense only within some platform-specific models (they simply cannot occur in others).

Code	Name	Description
Error codes dealing with analysis data		
200	UNKNOWN_NAME	Setting input data under a non-existing name. Or asking for a result using an unknown name.
201	INPUTS_INVALID	Input data are invalid, they do not match with their definitions, or with their dependency conditions.

202	INPUT_NOT_ACCEPTED	Used when a client tries to send input data to a job created in a previous call but the server does not any more accept input data.
Error codes dealing with analysis execution		
300	NOT_RUNNABLE	The same job has already been executed, or the data that had been set previously do not exist or are not accessible anymore.
301	NOT_RUNNING	A job has not yet been started. Note that this exception *is not* raised when the job has been already finished.
302	NOT_TERMINATED	A job is not interruptible for some reason.
Error codes dealing with analysis metadata		
400	NO_METADATA_AVAILABLE	There are no metadata available.
Error codes dealing with notification		
500	PROTOCOLS_UNACCEPTED	Used when a server does not agree on using any of the proposed notification protocols.
General error codes		
600	INTERNAL_PROCESSING_ERROR	A generic catch-all for errors not specifically mentioned elsewhere in this list.
601	COMMUNICATION_FAILURE	A generic network failure.
602	UNKNOWN_STATE	Used when a network call expects to find an existing state but failed. An example is an unknown handler representing a Job (unknown Job_ID, typical for Web Services platform).
603	NOT_IMPLEMENTED	A requested method is not implemented. Note that the method in question must exist (otherwise it may be caught already by the underlying protocol and reported differently) - but it has no implementation.

The implementation may extend the set of error codes in order to include implementation-specific codes. If it does so it should use numbers above 20 in each of the groups, or any number above 700. In other words, the free codes are: 221-299, 321-399, 421-499, 521-599, 621-699, 701-above).

9 Platform Specific Models

The previous chapters define a platform independent model of services related to the Life Sciences Analysis Engine. The real implementations, however, are expected to live in a more specific environment. This chapter shows two middleware specific models, one for Java and one for Web Services.

9.1 Java

The platform specific model for Java is expressed in a set of Java interfaces and two Java classes, one defining an exception, one defining a job status. All the interfaces, all the methods and error codes in LSAEEXception are normative. The shown implementation of the LSAEException and JobStatus, however, are not normative (they can be implemented differently).

In all cases the LSAEException shall be raised either as a result of an exceptional condition as described in the platform independent model, or of a Java specific exceptional condition (such as an inability to read an input stream). The implementations are encouraged to define their own, more specific and more expressive exceptions inherited from the LSAEException.

The package name org.omg.lsae is assumed in all definitions.

The Java specific model was derived from the platform independent model manually. It mirrors very closely the platform independent model with the following differences:

- The Java specific name conventions for method and parameter names are used.
- While full metadata are expressed as an XML string (as in any other platform-specific model) the subsets of metadata are conveniently available as arrays of Java data type Map. The keys in the returned map correspond with the XML tag and/or attribute names as defined in the minimal metadata model.
- Notification events are expressed as XML strings as defined in the platform independent model.

The normative files org/omg/lsae/*.java are in the accompanied document.

9.2 Web Services

The platform specific model for Web Services derived its architecture according to the Web Service definition as suggested by the W3C document “Web Services Glossary” [4]:

A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL).

The platform specific model for Web Services is defined with a binding for

- SOAP over HTTP

The model is expressed by port type, binding, and other parts as defined in the accompanied file LSAE.wsdl. This normative WSDL (Web Services Description Language) definition was derived from the platform independent model partly manually and partly automatically, in two steps:

1. The platform independent model was first manually converted into a Java interface AnalysisService_WS, and its implementation class AnalysisService_WS_Impl.java. Both these intermediate files are also attached. (Note that these two files are only intermediate steps for generating WSDL - they are not connected in any way to the Java-

based platform-specific model that is explained elsewhere in this document.)

The principles applied during the manual conversion guarantee that the resulting data types are suitable for the Web Services architecture in many programming languages:

- Used data types are simple in order to avoid as much as possible the need for implementing specific data mappings.
- The state handlers are part of the method signatures in order to be less dependent on the session management in any particular programming language and/or Web Services toolkit.

2. The manually created intermediate files served as source for the automatic conversion into the WSDL definition. It was achieved by Java2WSDL, a tool available from the Apache/Axis toolkit for Web services, version 1.1 [7] with the following command-line arguments:

java org.apache.axis.wsdl.Java2WSDL	
-o LSAE.wsdl	Indicates the name of the output WSDL file.
-P AnalysisService	Indicates the name of the portType element.
-b AnalysisServiceSoapBinding	Indicates the name of the binding element.
-n http://www.omg.org/LSAE/2004/Standard/WSDL	Indicates the name of the target namespace of the WSDL.
--PkgToNS org.omg.lsae=http://www.omg.org/LSAE/2004/Standard/WSDL	Indicates the mapping of a package to a namespace.
-w Interface	Generates a WSDL containing the interface constructs (no service element).
-i org.omg.lsae.AnalysisService_WS_Impl	Indicates an implementation class. Although the class used here is an empty implementation the debug information in the class is used to obtain the method parameter names, which are used to set the WSDL part names.
org.omg.lsae.AnalysisService_WS	The source Java interface, class-of-portType.

9.2.1 Port type (methods)

WSDL has a concept known as a “portType.” A portType is analogous to a CORBA interface or even a Java interface. It is a set of method signatures describing the types of the input and output parameters. It is defined in the normative file LSAE.wsdl.

Because this platform specific interface is derived from the general model described earlier all definitions are valid also here with the following exceptions expressing the specificity of the Web Services platform:

- All operations (26) belong to a single “class,” the Analysis Service that represents a Web Service implementing this specification.
- There is an additional convenient operation getCharacteristics that combines together all operations bringing time characteristics of a job.
- While full metadata are expressed as an XML string (as in any other platform-specific model) the subsets of metadata

are conveniently available as arrays of Java data type Map. The keys in the returned map correspond with the XML tag and/or attribute names as defined in the minimal metadata model.

- Notification events are expressed as XML strings as defined in the platform independent model.
- Slightly different naming conventions for methods and parameter names are used.

9.2.2 Reporting Errors

According to the SOAP specification the errors are carried in a SOAP Fault element.

The <faultstring> should minimally contain a human readable error description. Here is an example with a fault response to createJob with invalid input data:

```
<faultstring>201 : Input invalid (query sequence missing)</faultstring>
```

The <detail> should contain two elements[1], an <errorcode> carrying the error code, and a <description> with the free text. For example:

```
<detail>
   <errorcode>201</errorcode>
   <description>Input invalid (query sequence missing)</description>
 </detail>
```

1. While the SOAP specification asks for an additional and fully qualified element that would include both <errorcode> and <description>, the WS-I specification does not. The WS-I is used here as a main reference in order to remove such ambiguities.

A Accompanied Files

This part of the specification is a set of the accompanied files (document number formal/05-12-03). Some of these files are normative and some of them contain examples and convenient images. If there is a discrepancy between the contents of the normative files and this document, then the files take precedence.

Normative accompanied files

- LSAE-XMI-<version>.xml - an XMI representation of the platform independent model. It includes only and all mandatory parts.
- LSAE.wsdl - a file containing WSDL for the platform specific model for Web Services.
- org/omg/lsae/*.java - a set of Java interfaces for the platform specific Java model (except the files AnalysisService_WS.java and AnalysisService_WS_Impl that are not normative; they are not even part of the Java-based platform-specific model).
- AnalysisEvent.dtd - definition of notification events.
- NotificationDescriptor.dtd - definition of the notification protocol negotiation.
- MinimalAnalysisMetadata.dtd - definition of mandatory metadata.

Non-normative accompanied files

- LSAE.mdl - a Rational Rose file with a diagram of the platform independent model. The diagram contains all parts, mandatory and optional.
- LSAE-XMI-full-<version>.xml - an XMI representation of the platform independent model, including also optional parts. It is here for convenience in order to reproduce the full model.
- AnalysisEvent.xsd - a convenient document generated from AnalysisEvent.dtd.
- NotificationDescriptor.xsd - a convenient document generated from NotificationDescriptor.dtd.
- MinimalAnalysisMetadata.xsd - a convenient document generated from MinimalAnalysisMetadata.dtd.
- ExtendedAnalysisMetadata.dtd - definition of the optional metadata.
- ExtendedAnalysisMetadata.xsd - a convenient document generated from ExtendedAnalysisMetadata.dtd.
- LSAE_all.jpg, *Package.jpg - diagrams exported from LSAE.mdl providing better resolution than the same images included in this document.
- *.jpg - other illustrative images.

INDEX