



Unstructured Information Management Architecture (UIMA) Version 1.0

Working Draft 04

21 May 2008

Specification URIs:

This Version:

[http://docs.oasis-open.org/\[tc-short-name\] / \[additional path/filename\] .html](http://docs.oasis-open.org/[tc-short-name] / [additional path/filename] .html)
[http://docs.oasis-open.org/\[tc-short-name\] / \[additional path/filename\] .doc](http://docs.oasis-open.org/[tc-short-name] / [additional path/filename] .doc)
[http://docs.oasis-open.org/\[tc-short-name\] / \[additional path/filename\] .pdf](http://docs.oasis-open.org/[tc-short-name] / [additional path/filename] .pdf)

Previous Version:

N/A

Latest Version:

[http://docs.oasis-open.org/\[tc-short-name\] / \[additional path/filename\] .html](http://docs.oasis-open.org/[tc-short-name] / [additional path/filename] .html)
[http://docs.oasis-open.org/\[tc-short-name\] / \[additional path/filename\] .doc](http://docs.oasis-open.org/[tc-short-name] / [additional path/filename] .doc)
[http://docs.oasis-open.org/\[tc-short-name\] / \[additional path/filename\] .pdf](http://docs.oasis-open.org/[tc-short-name] / [additional path/filename] .pdf)

Latest Approved Version:

N/A

Technical Committee:

OASIS Unstructured Information Management Architecture (UIMA) TC

Chair(s):

David Ferrucci, IBM

Editor(s):

Adam Lally, IBM
Karin Verspoor, University of Colorado Denver
Eric Nyberg, Carnegie Mellon University

Related work:

This specification is related to:

- OASIS Unstructured Operation Markup Language (UOML). The UIMA specification, however, is independent of any particular model for representing or manipulating unstructured content.

Declared XML Namespace(s):

<http://docs.oasis-open.org/uima.ecore>
<http://docs.oasis-open.org/uima/base.ecore>
<http://docs.oasis-open.org/uima/peMetadata.ecore>
<http://docs.oasis-open.org/uima/pe.ecore>
<http://docs.oasis-open.org/uima/peService>

Abstract:

Unstructured information may be defined as the direct product of human communication. Examples include natural language documents, email, speech, images and video. The UIMA specification defines platform-independent data representations and interfaces for software

components or services called *analytics*, which analyze unstructured information and assign semantics to regions of that unstructured information.

Status:

This draft has not yet been approved by the OASIS UIMA TC.

Technical Committee members should send comments on this specification to the Technical Committee's email list. Others should send comments to the Technical Committee by using the "Send A Comment" button on the Technical Committee's web page at <http://www.oasis-open.org/committees/uima/>.

For information on whether any patents have been disclosed that may be essential to implementing this specification, and any offers of patent licensing terms, please refer to the Intellectual Property Rights section of the Technical Committee web page (<http://www.oasis-open.org/committees/uima/ipr.php>).

Notices

Copyright © OASIS® 2008. All Rights Reserved.

All capitalized terms in the following text have the meanings assigned to them in the OASIS Intellectual Property Rights Policy (the "OASIS IPR Policy"). The full Policy may be found at the OASIS website.

This document and translations of it may be copied and furnished to others, and derivative works that comment on or otherwise explain it or assist in its implementation may be prepared, copied, published, and distributed, in whole or in part, without restriction of any kind, provided that the above copyright notice and this section are included on all such copies and derivative works. However, this document itself may not be modified in any way, including by removing the copyright notice or references to OASIS, except as needed for the purpose of developing any document or deliverable produced by an OASIS Technical Committee (in which case the rules applicable to copyrights, as set forth in the OASIS IPR Policy, must be followed) or as required to translate it into languages other than English.

The limited permissions granted above are perpetual and will not be revoked by OASIS or its successors or assigns.

This document and the information contained herein is provided on an "AS IS" basis and OASIS DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY OWNERSHIP RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

OASIS requests that any OASIS Party or any other party that believes it has patent claims that would necessarily be infringed by implementations of this OASIS Committee Specification or OASIS Standard, to notify OASIS TC Administrator and provide an indication of its willingness to grant patent licenses to such patent claims in a manner consistent with the IPR Mode of the OASIS Technical Committee that produced this specification.

OASIS invites any party to contact the OASIS TC Administrator if it is aware of a claim of ownership of any patent claims that would necessarily be infringed by implementations of this specification by a patent holder that is not willing to provide a license to such patent claims in a manner consistent with the IPR Mode of the OASIS Technical Committee that produced this specification. OASIS may include such claims on its website, but disclaims any obligation to do so.

OASIS takes no position regarding the validity or scope of any intellectual property or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; neither does it represent that it has made any effort to identify any such rights. Information on OASIS' procedures with respect to rights in any document or deliverable produced by an OASIS Technical Committee can be found on the OASIS website. Copies of claims of rights made available for publication and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementers or users of this OASIS Committee Specification or OASIS Standard, can be obtained from the OASIS TC Administrator. OASIS makes no representation that any information or list of intellectual property rights will at any time be complete, or that any claims in such list are, in fact, Essential Claims.

The names "OASIS", [insert specific trademarked names and abbreviations here] are trademarks of OASIS, the owner and developer of this specification, and should be used only to refer to the organization and its official outputs. OASIS welcomes reference to, and implementation and use of, specifications, while reserving the right to enforce its marks against misleading uses. Please see <http://www.oasis-open.org/who/trademark.php> for above guidance.

Table of Contents

1	Introduction	7
1.1	Terminology	8
1.2	Normative References	8
1.3	Non-Normative References	9
2	Basic Concepts and Terms.....	10
3	Elements of the UIMA Specification	12
3.1	Common Analysis Structure (CAS).....	12
3.2	Type System Model	12
3.3	Base Type System	13
3.4	Abstract Interfaces	14
3.5	Behavioral Metadata	14
3.6	Processing Element Metadata	15
3.7	WSDL Service Descriptions	15
4	Full UIMA Specification.....	17
4.1	The Common Analysis Structure (CAS)	17
4.1.1	Basic Structure: Objects and Slots	17
4.1.2	Relationship to Type System	17
4.1.3	The XMI CAS Representation	18
4.1.4	Formal Specification	18
4.2	The Type System Model	19
4.2.1	Ecore as the UIMA Type System Model.....	19
4.2.2	Formal Specification	19
4.3	Base Type System	20
4.3.1	Primitive Types	20
4.3.2	Annotation and Sofa Base Type System.....	20
4.3.3	View Base Type System.....	22
4.3.4	Source Document Information	24
4.3.5	Formal Specification	24
4.4	Abstract Interfaces	24
4.4.1	Processing Element.....	24
4.4.2	Analytic	25
4.4.3	Analyzer	25
4.4.4	CAS Multiplier	25
4.4.5	Flow Controller.....	26
4.4.6	Formal Specification	28
4.5	Behavioral Metadata	31
4.5.1	Behavioral Metadata UML	31
4.5.2	Behavioral Metadata Elements and XML Representation.....	32
4.5.3	Formal Semantics for Behavioral Metadata	32
4.5.4	Formal Specification	34
4.6	Processing Element Metadata	37
4.6.1	Elements of PE Metadata	37
4.6.2	Formal Specification	40

4.7 Service WSDL Descriptions	40
4.7.1 Overview of the WSDL Definition	40
4.7.2 Delta Responses	44
4.7.3 Formal Specification	44
A. Acknowledgements	45
B. Examples (Not Normative).....	46
B.1 XMI CAS Example	46
B.1.1 XMI Tag	46
B.1.2 Objects	46
B.1.3 Attributes (Primitive Features).....	47
B.1.4 References (Object-Valued Features)	48
B.1.5 Multi-valued Features.....	48
B.1.6 Linking an XMI Document to its Ecore Type System.....	49
B.1.7 XMI Extensions	49
B.2 Ecore Example	50
B.2.1 An Introduction to Ecore	50
B.2.2 Differences between Ecore and EMOF	51
B.2.3 Example Ecore Model	52
B.3 Base Type System Examples.....	53
B.3.1 Sofa Reference	53
B.3.2 References to Regions of Sofas	54
B.3.3 Options for Extending Annotation Type System	54
B.3.4 An Example of Annotation Model Extension.....	55
B.3.5 Example Extension of Source Document Information	56
B.4 Abstract Interfaces Examples	57
B.4.1 Analyzer Example	57
B.4.2 CAS Multiplier Example.....	57
B.5 Behavioral Metadata Examples	58
B.5.1 Type Naming Conventions	58
B.5.2 XML Syntax for Behavioral Metadata Elements.....	61
B.5.3 Views	62
B.5.4 Specifying Which Features Are Modified	63
B.5.5 Specifying Preconditions, Postconditions, and Projection Conditions	63
B.6 Processing Element Metadata Example.....	64
B.7 SOAP Service Example.....	65
C. Formal Specification Artifacts	67
C.1 XMI XML Schema.....	67
C.2 Ecore XML Schema.....	70
C.3 Base Type System Ecore Model	75
C.4 PE Metadata and Behavioral Metadata Ecore Model	76
C.5 PE Metadata and Behavioral Metadata XML Schema.....	79
C.6 PE Service WSDL Definition	82
C.7 PE Service XML Schema (uima.peServiceXML.xsd).....	92
D. Revision History	96

1 Introduction

Unstructured information may be defined as the direct product of human communication. Examples include natural language documents, email, speech, images and video. It is information that was not specifically encoded for machines to process but rather authored by humans for humans to understand. We say it is “unstructured” because it lacks explicit semantics (“structure”) required for applications to interpret the information as intended by the human author or required by the end-user application.

Unstructured information may be contrasted with the information in classic relational databases where the intended interpretation for every field data is explicitly encoded in the database by column headings. Consider information encoded in XML as another example. In an XML document some of the data is wrapped by tags which provide explicit semantic information about how that data should be interpreted. An XML document or a relational database may be considered semi-structured in practice, because the content of some chunk of data, a blob of text in a text field labeled “description” for example, may be of interest to an application but remain without any explicit tagging—that is, without any explicit semantics or structure.

Unstructured information represents the largest, most current and fastest growing source of knowledge available to businesses and governments worldwide. The web is just the tip of the iceberg. Consider for example the droves of corporate, scientific, social and technical documentation ranging from best practices, research reports, medical abstracts, problem reports, customer communications, contracts, emails and voice mails. Beyond these, consider the growing number of broadcasts containing audio, video and speech. These mounds of natural language, speech and video artifacts often contain nuggets of knowledge critical for analyzing and solving problems, detecting threats, realizing important trends and relationships, creating new opportunities or preventing disasters.

For unstructured information to be processed by traditional applications that rely on specific structure, it must be first analyzed to assign application-specific semantics to the unstructured content. Another way to say this is that the unstructured information must become “structured” where the added structure explicitly provides the semantics required by target applications to interpret the data.

An example of assigning semantics includes wrapping regions of text in a text document with appropriate XML tags that might identify the names of organizations or products. Another example may extract elements of a document and insert them in the appropriate fields of a relational database or use them to create instances of concepts in a knowledgebase. Another example may analyze a voice stream and tag it with the information explicitly identifying the speaker.

A simple analysis on documents may, for example, scan each token in each document of a collection to identify names of organizations. It may insert a tag wrapping and identify every found occurrence of an organization name and output the XML that explicitly annotates each with the appropriate tag. An application that manages a database of organizations may now use the structured information produced by the document analysis to populate a relational database.

In general, we refer to the act of assigning semantics to a region of some unstructured content (e.g., a document) as “analysis”. A software component or service that performs the analysis is an “analytic”.

Analytics are typically reused and combined together in different flows to perform application-specific aggregate analyses.

49 While different platform-specific, software frameworks have been developed in support of building and
50 integrating component analytics (e.g., Apache UIMA, Gate, Catalyst, Tipster, Mallet, Talent, Open-NLP,
51 etc.), no clear standard has emerged for enabling the interoperability of analytics across platforms,
52 frameworks and modalities (text, audio, video, etc.)

53

54 The UIMA specification defines platform-independent data representations and interfaces for text & multi-
55 modal analytics. The principal objective of the UIMA specification is to support interoperability among
56 *analytics*. This objective is subdivided into the following four design goals:

57

- 58 1. **Data Representation.** Support the common representation of *artifacts* and *artifact metadata*
59 (analysis results) independently of *artifact modality* and *domain model*.
- 60
- 61 2. **Data Modeling and Interchange.** Support the platform-independent interchange of *analysis data*
62 in a form that facilitates a formal modeling approach and alignment with existing programming
63 systems and standards.
- 64
- 65 3. **Discovery, Reuse and Composition.** Support the discovery, reuse and composition of
66 independently-developed *analytics*.
- 67
- 68 4. **Service-Level Interoperability.** Support concrete interoperability of independently developed
69 *analytics* based on a common service description and associated SOAP bindings.

70

71 The text of this specification is normative with the exception of the Introduction and Appendices.

72 1.1 Terminology

73 The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD
74 NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described
75 in [RFC2119].

76 1.2 Normative References

- 77 [RFC2119] S. Bradner, *Key words for use in RFCs to Indicate Requirement Levels*,
78 <http://www.ietf.org/rfc/rfc2119.txt>, IETF RFC 2119, March 1997.
- 79 [MOF1] Object Management Group. Meta Object Facility (MOF) 2.0 Core Specification.
80 <http://www.omg.org/docs/ptc/04-10-15.pdf>
- 81 [OCL1] Object Management Group. Object Constraining Language Version 2.0.
82 <http://www.omg.org/technology/documents/formal/ocl.htm>
- 83 [OSGi1] OSGi Alliance. OSGi Service Platform Core Specification, Release 4, Version 4.1.
84 Available from <http://www.osgi.org>.
- 85 [SOAP1] W3C. SOAP Version 1.2 Part 1: Messaging Framework (Second Edition).
86 <http://www.w3.org/TR/soap12-part1/>
- 87 [UML1] Object Management Group. Unified Modeling Language (UML), version 2.1.2.
88 <http://www.omg.org/technology/documents/formal/uml.htm>
- 89 [XMI1] Object Management Group. XML Metadata Interchange (XMI) Specification, Version 2.0.
90 <http://www.omg.org/docs/formal/03-05-02.pdf>
- 91 [XML1] W3C. Extensible Markup Language (XML) 1.0 (Fourth Edition).
92 <http://www.w3.org/TR/REC-xml>
- 93 [XML2] W3C. Namespaces in XML 1.0 (Second Edition). <http://www.w3.org/TR/REC-xml-names/>

- 94 **[XMLS1]** XML Schema Part 1: Structures Second Edition. [http://www.w3.org/TR/2004/REC-](http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/structures.html)
95 [xmlschema-1-20041028/structures.html](http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/structures.html)
96 **[XMLS2]** XML Schema Part 2: Datatypes Second Edition. [http://www.w3.org/TR/2004/REC-](http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/datatypes.html)
97 [xmlschema-2-20041028/datatypes.html](http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/datatypes.html).
98

99 **1.3 Non-Normative References**

- 100 **[BPEL1]** http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel
101 **[EcoreEMOF1]** <http://dev.eclipse.org/newslists/news.eclipse.tools.emf/msg04197.html>
102
103 **[EMF1]** The Eclipse Modeling Framework (EMF) Overview.
104 <http://help.eclipse.org/help33/index.jsp?topic=/org.eclipse.emf.doc/references/overview/EMF.html>
105 **[EMF2]** Budinsky et al. Eclipse Modeling Framework. Addison-Wesley. 2004.
106 **[EMF3]** Budinsky et al. Eclipse Modeling Framework, Chapter 2, Section 2.3
107 <http://www.awprofessional.com/content/images/0131425420/samplechapter/budinskych02.pdf>
108 **[KLT1]** David Ferrucci, William Murdock, Chris Welty, “Overview of Component Services for
109 Knowledge Integration in UIMA (a.k.a. SUKI)” IBM Research Report RC24074
110 **[XMI2]** Grose et al. Mastering XMI. Java Programming with XMI, XML, and UML. John Wiley &
111 Sons, Inc. 2002

2 Basic Concepts and Terms

112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157

This specification defines and uses the following terms:

Unstructured Information is typically the direct product of human communications. Examples include natural language documents, email, speech, images and video. It is information that was not encoded for machines to understand but rather authored for humans to understand. We say it is “unstructured” because it lacks explicit semantics (“structure”) required for computer programs to interpret the information as intended by the human author or required by the application.

Artifact refers to an application-level unit of information that is subject to analysis by some application. Examples include a text document, a segment of speech or video, a collection of documents, and a stream of any of the above. Artifacts are physically encoded in one or more ways. For example, one way to encode a text document might be as a Unicode string.

Artifact Modality refers to mode of communication the artifact represents, for example, text, video or voice.

Artifact Metadata refers to structured data elements recorded to describe entire artifacts or parts of artifacts. A piece of artifact metadata might indicate, for example, the part of the document that represents its title or the region of video that contains a human face. Another example of metadata might indicate the topic of a document while yet another may tag or annotate occurrences of person names in a document etc. Artifact metadata is logically distinct from the artifact, in that the artifact is the data being analyzed and the artifact metadata is the result of the analysis – it is data about the artifact.

Domain Model refers to a conceptualization of a system, often cast in a formal modeling language. In this specification we use it to refer to any model which describes the structure of artifact metadata. A domain model provides a formal definition of the types of data elements that may constitute artifact metadata. For example, if some artifact metadata represents the organizations detected in a text document (the artifact) then the type Organization and its properties and relationship to other types may be defined in a domain model which the artifact metadata instantiates.

Analysis Data is used to refer to the logical union of an artifact and its metadata.

Analysis Operations are abstract functions that perform some analysis on artifacts and/or their metadata and produce some result. The results may be the addition or modification to artifact metadata and/or the generation of one or more artifacts. An example is an “Annotation” operation which may be defined by the type of artifact metadata it produces to describe or annotate an artifact. Analysis operations may be ultimately bound to software implementations that perform the operations. Implementations may be realized in a variety of software approaches, for example web-services or Java classes.

An **Analytic** is a software object or network service that performs an Analysis Operation.

A **Flow Controller** is a component or service that decides the workflow between a set of analytics.

A **Processing Element (PE)** is either an Analytic or a Flow Controller. PE is the most general type of component/service that developers may implement.

158 **Processing Element Metadata (PE Metadata)** is data that describes a Processing Element (PE) by
159 providing information used for discovering, combining, or reusing the PE for the development of UIM
160 applications. PE Metadata would include Behavioral Metadata for the operation which the PE implements.
161

3 Elements of the UIMA Specification

In this section we provide an overview of the seven elements of the UIMA standard. The full specification for each element will be defined in Section 4.

3.1 Common Analysis Structure (CAS)

The Common Analysis Structure or CAS is the common data structure shared by all UIMA analytics to represent the unstructured information being analyzed (the **artifact**) as well as the metadata produced by the analysis workflow (the **artifact metadata**).

The CAS represents an essential element of the UIMA specification in support of interoperability since it provides the common foundation for sharing data and results across analytics.

The CAS is an Object Graph where Objects are instances of Classes and Classes are Types in a **type system** (see next section).

There are two fundamental types of objects in a CAS:

- **Sofa**, or subject of analysis, which holds the artifact;
- **Annotation**, a type of artifact metadata that points to a region within a Sofa and “annotates” (labels) the designated region in the artifact.

The Sofa and Annotation types are defined as part of the UIMA Base Type System (see Section 3.3).

The CAS provides a domain neutral, object-based representation scheme that is aligned with UML [UML1]. UIMA defines an XML representation of analysis data using the XML Metadata Interchange (XMI) specification [XMI1][XMI2].

The CAS representation can easily be elaborated for specific domains of analysis by defining domain-specific types; interoperability can be achieved across programming languages and operating systems through the use of the CAS representation and its associated type system definition.

3.2 Type System Model

To support the design goal of data modeling and interchange, UIMA requires that a CAS conform to a user-defined schema, called a **type system**.

A type system is a collection of inter-related **type** definitions. Each type defines the structure of any object that is an instance of that type. For example, Person and Organization may be types defined as part of a type system. Each type definition declares the attributes of the type and describes valid fillers for its attributes. For example lastName, age, emergencyContact and employer may be attributes of the Person type. The type system may further specify that the lastName must be filled with exactly one string value, age exactly one integer value, emergencyContact exactly one instance of the same Person type and employer zero or more instances of the Organization type.

The **artifact metadata** in a CAS is represented by an object model. Every object in a CAS must be associated with a Type. The UIMA Type-System language therefore is a declarative language for defining object models.

205 Type Systems are user-defined. UIMA does not specify a particular set of types that developers must use.
206 Developers define type systems to suit their application's requirements. A goal for the UIMA community,
207 however, would be to develop a common set of type-systems for different domains or industry verticals.
208 These common type systems can significantly reduce the efforts involved in integrating independently
209 developed analytics. These may be directly derived from related standards efforts around common tag
210 sets for legal information or common ontologies for biological data, for example.

211

212 Another UIMA design goal is to support the composition of independently developed **analytics**. The
213 behavior of analytics may be specified in terms of type definitions expressed in a type system language.
214 For example an analytic must define the types it requires in an input CAS and those that it may produce
215 as output. This is described as part of the analytic's Behavioral Metadata (See Section 3.5). For example,
216 an analytic may declare that given a plain text document it produces instances of Person annotations
217 where Person is defined as a particular type in a type system.

218

219 The UIMA Type System Model is designed to provide the following features:

- 220 • Object-Oriented. Type systems defined with the UIMA Type System Model are isomorphic to classes in
221 object-oriented representations such as UML, and are easily mapped or compiled into deployment data
222 structures in a particular implementation framework.
- 223 • Inheritance. Types can extend other types, thereby inheriting the features of their parent type.
- 224 • Optional and Required Features. The features associated with types can be optional or required,
225 depending on the needs of the application.
- 226 • Single and Multi-Valued Features with Range Constraints. The features associated with types can be
227 single-valued or multi-valued, depending on the needs of the application. The legal range of values for
228 a feature (its range constraint) may be specified as part of the feature definition.
- 229 • Aligned with UML standards and Tooling. The UIMA Type System model can be directly expressed
230 using existing UML modeling standards, and is designed to take advantage of existing tooling for UML
231 modeling.

232

233 Rather than invent a language for defining the UIMA Type System Model, we have explored standard
234 modeling languages.

235

236 The OMG has defined representation schemes for describing object models including UML and its
237 subsets (modeling languages with increasingly lower levels of expressivity). These include MOF and
238 EMOF (the essential MOF) [[MOF1](#)].

239

240 Ecore is the modeling language of the Eclipse Modeling Framework (EMF) [[EMF1](#)]. It affords the
241 equivalent modeling semantics provided by EMOF with some minor syntactic differences – see Section
242 B.2.2.

243

244 UIMA adopts Ecore as the type system representation, due to the alignment with standards and the
245 availability of EMF tooling.

246 **3.3 Base Type System**

247 The UIMA Base Type System is a standard definition of commonly-used, domain-independent types. It
248 establishes a basic level of interoperability among applications.

249

250 The most significant part of the Base Type System is the *Annotation and Sofa (Subject of Analysis) Type*
251 *System*. A general and motivating UIMA use case is one where analytics label or *annotate* regions of
252 unstructured content. A fundamental approach to representing annotations is referred to as the “stand-
253 off” annotation model. In a “stand-off” annotation model, annotations are represented as objects of a

254 domain model that “point into” or reference elements of the unstructured content (e.g., document or video
255 stream) rather than as inserted tags that affect and/or are constrained by the original form of the content.

256

257 In UIMA, a CAS stores the artifact (i.e., the unstructured content that is the subject of the analysis) and
258 the artifact metadata (i.e., structured data elements that describe the artifact). The metadata generated by
259 an analytic may include a set of annotations that label regions of the artifact with respect to some domain
260 model (e.g., persons, organizations, events, times, opinions, etc). These annotations are logically and
261 physical distinct from the subject of analysis, so UIMA adopts the “stand-off” model for annotations.

262

263 In UIMA the original content is not affected in the analysis process. Rather, an object graph is produced
264 that *stands off* from and annotates the content. Stand-off annotations in UIMA allow for multiple content
265 interpretations of graph complexity to be produced, co-exist, overlap and be retracted without affecting
266 the original content representation. The object model representing the stand-off annotations may be used
267 to produce different representations of the analysis results. A common form for capturing document
268 metadata for example is as in-line XML. An analytic in a UIM application, for example, can generate from
269 the UIMA representation an in-line XML document that conforms to some particular domain model or
270 markup language. Alternatively it can produce an XMI or RDF document.

271

272 The Base Type System also includes the following:

- 273 • Primitive Types (defined by Ecore)
- 274 • Views (Specific collections of objects in a CAS)
- 275 • Source Document Information (Records information about the original source of unstructured
276 information in the CAS)

277 3.4 Abstract Interfaces

278 The UIMA Abstract Interfaces define the standard component types and operations that UIMA developers
279 can implement. The supertype of all UIMA components is called the *Processing Element (PE)*. There are
280 two main subtypes:

- 281 • An *Analytic* is a component that performs analysis of CASes. There are two subtypes:
 - 282 ○ An *Analyzer* processes a CAS and possibly updates it contents. This is the most
283 common type of UIMA component.
 - 284 ○ A *CAS Multiplier* processes a CAS and possibly creates new CASes. This is useful for
285 example to implement a “segmenter” Analytic that takes an input CAS and divides it into
286 pieces, outputting each piece as a new CAS.
- 287 • A *Flow Controller* determines the route CASes take through multiple Analytcs.

288

289 The abstract definitions in this section lay the foundation for the concrete service specification described
290 in Section 3.7.

291 3.5 Behavioral Metadata

292 The Behavioral Metadata of an analytic declaratively describes what the analytic does; for example, what
293 types of CASs it can process, what elements in a CAS it analyzes, and what sorts of effects it may have
294 on CAS contents as a result of its application.

295

296 Behavioral Metadata is designed to achieve the following goals:

- 297 1. **Discovery:** Enable both human developers and automated processes to search a repository and
298 locate components that provide a particular function (i.e., works on certain input, produces certain
299 output)

300

- 301 2. **Composition:** Support composition either by a human developer or an automated process.
302 a. Analytics should be able to declare what they do in enough detail to assist manual
303 and/or automated processes in considering their role in an application or in the
304 composition of aggregate analytics.
305 b. Through their Behavioral Metadata, Analytics should be able to declare enough detail
306 as to enable an application or aggregate to detect “invalid” compositions/workflows
307 (e.g., a workflow where it can be determined that one of the Analytic’s preconditions
308 can never be satisfied by the preceding Analytic).
309
- 310 3. **Efficiency:** Facilitate efficient sharing of CAS content among cooperating analytics. If analytics
311 declare which elements of the CAS (e.g., *views*) they need to receive and which elements they do not
312 need to receive, the CAS can be filtered or split prior to sending it to target analytics, to achieve
313 transport and parallelization efficiencies respectively.
314

315 Behavioral Metadata breaks down into the following categories:

- 316 • **Analyzes:** Types of objects (Sofas) that the analytic intends to produce annotations over.
 - 317 • **Required Inputs:** Types of objects that must be present in the CAS for the analytic to operate.
 - 318 • **Optional Inputs:** Types of objects that the analytic would consult if they were present in the CAS.
 - 319 • **Creates:** Types of objects that the analytic may create.
 - 320 • **Modifies:** Types of objects that the analytic may modify.
 - 321 • **Deletes:** Types of objects that the analytic may delete.
- 322

323 Note that analytics are not required to declare behavioral metadata. If an analytic does not provide
324 behavioral metadata, then an application using the analytic cannot assume anything about the operations
325 that the analytic will perform on a CAS.

326 3.6 Processing Element Metadata

327 All UIMA Processing Elements (PEs) must publish **processing element metadata**, which describes the
328 analytic to support discovery and composition. This section of the spec defines the structure of this
329 metadata and provides an XML schema in which PEs must publish this metadata.

330

331 The PE Metadata is subdivided into the following parts:

332

- 333 1. **Identification Information.** Identifies the PE. It includes for example a symbolic/unique name, a
334 descriptive name, vendor and version information.
- 335 2. **Configuration Parameters.** Declares the names of parameters used by the PE to affect its
336 behavior, as well as the parameters’ default values.
- 337 3. **Behavioral Metadata.** Describes the PEs input requirements and the operations that the PE
338 may perform, as described in Section 3.5.
- 339 4. **Type System.** Defines types used by the PE and referenced from the behavioral specification.
- 340 5. **Extensions.** Allows the PE metadata to contain additional elements, the contents of which are
341 not defined by the UIMA specification. This can be used by framework implementations to
342 extend the PE metadata with additional information that may be meaningful only to that
343 framework.

344 3.7 WSDL Service Descriptions

345 This specification element facilitates interoperability by specifying a WSDL [[WSDL1](#)] description of the
346 UIMA interfaces and a binding to a concrete SOAP interface that compliant frameworks and services
347 MUST implement.

348

349 This SOAP interface implements the Abstract Interfaces described in Section 3.4. The use of SOAP
350 facilitates standard use of web services as a CAS transport.
351
352

353 4 Full UIMA Specification

354 4.1 The Common Analysis Structure (CAS)

355 4.1.1 Basic Structure: Objects and Slots

356 At the most basic level a CAS contains an object graph – a collection of objects that may point to or
357 cross-reference each other. Objects are defined by a set of properties which may have values. Values
358 can be primitive types like numbers or strings or can refer to other objects in the same CAS.

359

360 This approach allows UIMA to adopt general object-oriented modeling and programming standards for
361 representing and manipulating artifacts and artifact metadata.

362

363 UIMA uses the Unified Modeling Language (UML) [UML1] to represent the structure and content of a
364 CAS.

365

366 In UML an **object** is a data structure that has 0 or more slots. We can think of a slot as representing an
367 object's properties and values. Formally a **Slot** in UML is a (feature, value) pair. Features in UML
368 represent an object's properties. A slot represents an assignment of one or more values to a feature.
369 Values can be either primitives (strings or various numeric types) or references to other objects.

370

371 UML uses the notion of classes to represent the required structure of objects. Classes define the slots
372 that objects must have. We refer to a set of classes as a **type system**.

373 4.1.2 Relationship to Type System

374 Every object in a CAS is an instance of a class defined in a UIMA **type system**.

375

376 A type system defines a set of classes. A class may have multiple features. Features may either be
377 attributes or references.

378

379 All features define their type. The type of an attribute is a primitive data type. The type of a reference is a
380 class. Features also have a cardinality (defined by a lower bound and an upper bound), which define how
381 many values they may take. We sometimes refer to features with an upper bound greater than one as
382 multi-valued features.

383

384 An object has one slot for each feature defined by its class.

385

386 Slots for attributes take primitive values; slots for references take objects as values. In general a slot may
387 take multiple values; the number of allowed values is defined by the lower bound and upper bound of the
388 feature.

389

390 The metamodel describing how a CAS relates to a type system is diagrammed in Figure 1.

391

392 Note that some UIMA components may manipulate a CAS without knowledge of its type system. A
393 common example is a CAS Store, which might allow the storage and retrieval of any CAS regardless of
394 what its type system might be.

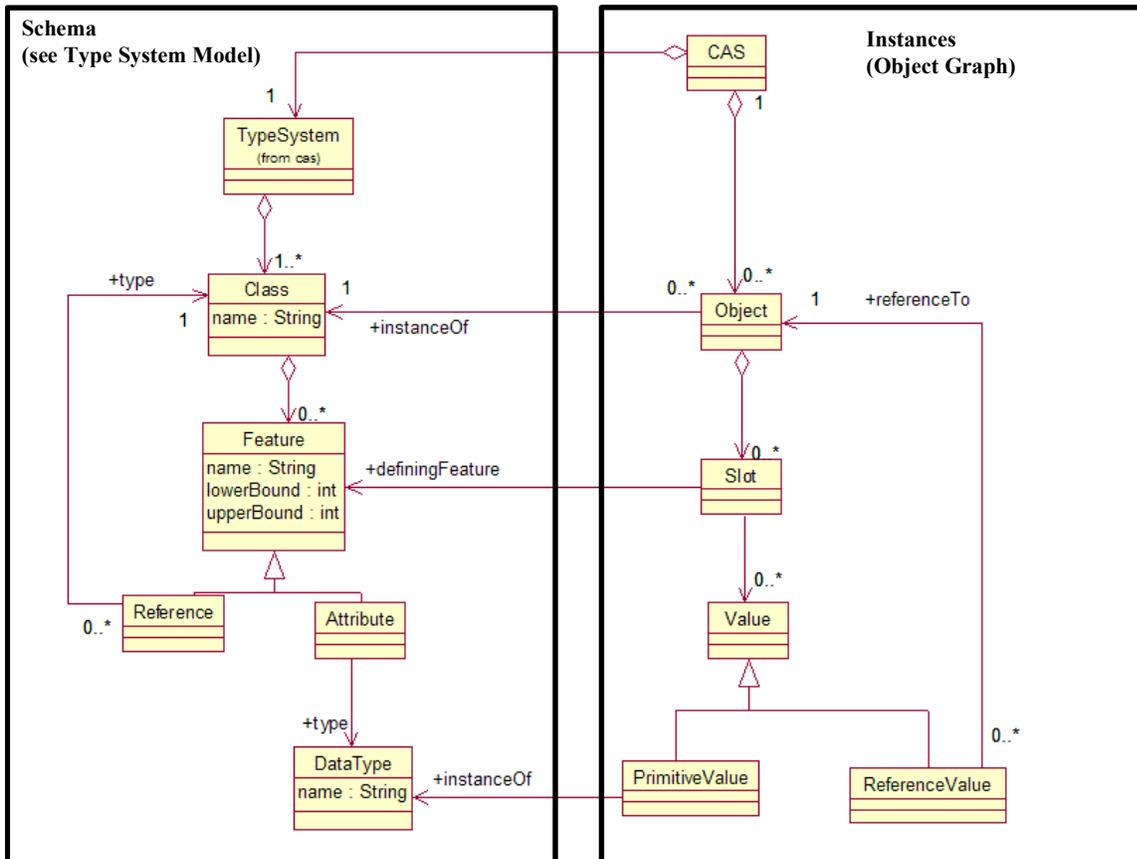


Figure 1: CAS Specification UML

396
397
398

4.1.3 The XMI CAS Representation

400 A UIMA CAS is represented as an XML document using the XMI (XML Metadata Interchange) standard
401 [XMI1, XMI2]. XMI is an OMG standard for expressing object graphs in XML.

402

403 XMI was chosen because it is an established standard, aligned with the object-graph representation of
404 the CAS, aligned with UML and with object-oriented programming, and supported by tooling such as the
405 Eclipse Modeling Framework [EMF1].

4.1.4 Formal Specification

4.1.4.1 Structure

408 UIMA CAS XML MUST be a valid XMI document as defined by the XMI Specification [XMI1].

409

410 This implies that UIMA CAS XML MUST be a valid instance of the XML Schema for XMI, listed in
411 Appendix C.1.

4.1.4.2 Constraints

413 If the root element of the XML CAS contains an xsi:schemaLocation attribute, the CAS is said to be linked
414 to an Ecore Type System. The xsi:schemaLocation attribute defines a mapping from namespace URI to

415 physical URI as defined by the XML Schema specification [XMLS1]. Each of these physical URIs MUST
416 be a valid Ecore document as defined by the XML Schema for Ecore, presented in Appendix C.2.

417

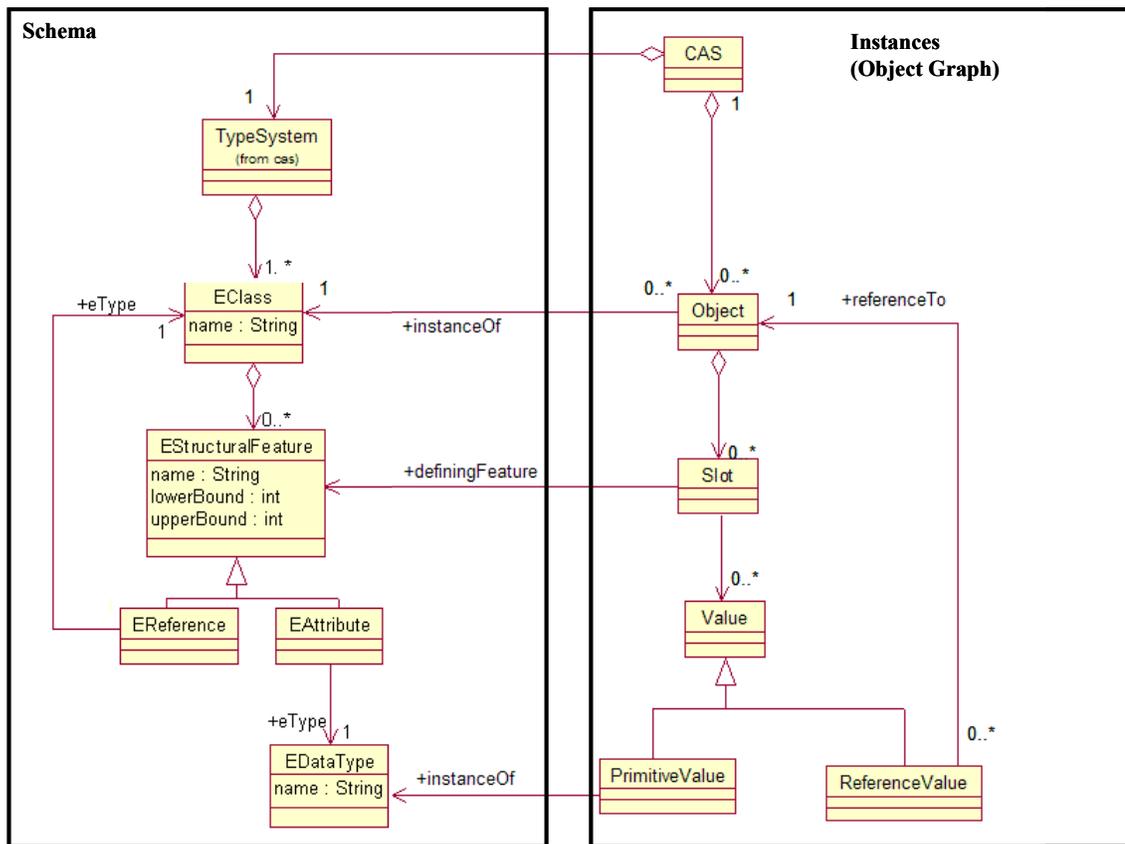
418 A CAS that is linked to an Ecore Type System MUST be valid with respect to that Ecore Type System, as
419 defined in Section 4.2.2.2.

420 4.2 The Type System Model

421 4.2.1 Ecore as the UIMA Type System Model

422 A UIMA Type System is represented using Ecore. Figure 2 shows how Ecore is used to define the
423 schema for a CAS.

424



425

426

Figure 2: Ecore defines schema for CAS

427

428 For an introduction to Ecore and an example of a UIMA Type System represented in Ecore, see Appendix
429 B.2.

430 4.2.2 Formal Specification

431 4.2.2.1 Structure

432 UIMA Type System XML MUST be a valid Ecore/XMI document as defined by Ecore and the XMI
433 Specification [XMI1].

434

435 This implies that UIMA Type System XML MUST be a valid instance of the XML Schema for Ecore, given
436 in Section C.2.

437 **4.2.2.2 Semantics**

438 A CAS is valid with respect to an Ecore type system if each object in the CAS is a valid instance of its
439 corresponding class (EClass) in the type system, as defined by XMI [XMI1], UML [UML1] and MOF
440 [MOF1].

441 **4.3 Base Type System**

442 The XML namespace for types defined in the UIMA base model is `http://docs.oasis-`
443 `open.org/uima/base.ecore`. (With the exception of types defined as part of Ecore, listed in Section
444 4.3.1, whose namespace is defined by Ecore.).

445

446 Examples showing how the Base Type System is used in UIMA examples can be found in Appendix B.3.

447 **4.3.1 Primitive Types**

448 UIMA uses the following primitive types defined by Ecore, which are analogous to the Java (and Apache
449 UIMA) primitive types:

450

- 451 • EString
- 452 • EBoolean
- 453 • EByte (8 bits)
- 454 • EShort (16 bits)
- 455 • EInt (32 bits)
- 456 • ELong (64 bits)
- 457 • EFloat (32 bits)
- 458 • EDouble (64 bits)

459

460 Also Ecore defines the type EObject, which is defined as the superclass of all non-primitive types
461 (classes).

462 **4.3.2 Annotation and Sofa Base Type System**

463 The Annotation and Sofa Base Type System defines a standard way for Annotations to refer to regions
464 within a Subject of Analysis (Sofa). The UML for the Annotation and Sofa Base Type System is given in
465 Figure 3. The discussion in the following subsections refers to this figure.

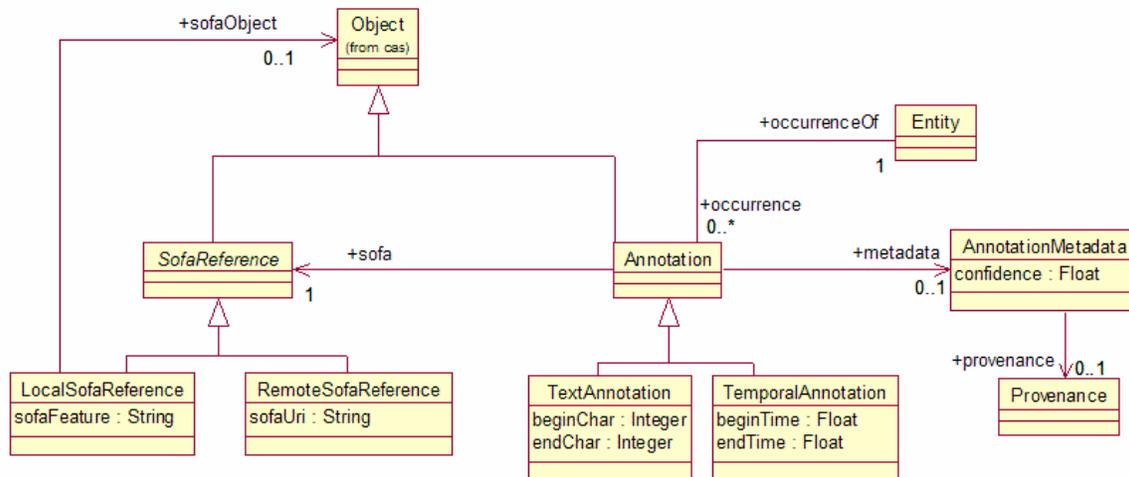


Figure 3: Annotation and Sofa Base Type System UML

466
467
468

4.3.2.1 Annotation and Sofa Reference

470 The UIMA Base Type System defines a standard object type called Annotation for representing stand-off
471 annotations. The Annotation type represents a type of object that is linked to a Subject of Analysis (Sofa).
472

473 The Sofa is the value of a slot in another object. Since a reference directly to a *slot* on an *object* (rather
474 than just an *object* itself) is not a concept directly supported by typical object oriented programming
475 systems or by XMI, UIMA defines a base type called LocalSofaReference for referring to Sofas from
476 annotations. UIMA also defines a RemoteSofaReference type that allows an annotation to refer to a
477 subject of analysis that is not located in the CAS.

4.3.2.2 References to Regions of Sofas

479 An annotation typically points to a region of the artifact data. One of UIMA's design goals is to be
480 independent of modality. For this reason UIMA does not constrain the data type that can function as a
481 subject of analysis and allows for different implementations of the linkage between an annotation and a
482 region of the artifact data.

483
484 The Annotation class has subclasses for each artifact modality, which define how the Annotation refers to
485 a region within the Sofa. The Standard defines subclasses for common modalities – Text and Temporal
486 (audio or video segments). Users may define other subclasses.

487
488 In TextAnnotation, beginChar and endChar refer to Unicode character offsets in the corresponding Sofa
489 string. For TemporalAnnotation, beginTime and endTime are offsets measured in seconds from the start
490 of the Sofa. Note that applications that require a different interpretation of these fields must accept the
491 standard values and handle their own internal mappings.

492
493 Annotations with discontinuous spans are not part of the Base Type System, but could be implemented
494 with a user-defined subclass of the Annotation type.

495 **4.3.2.3 References to Entities**

496 In general, an `Annotation` is an reference to some `Entity` in a domain model. (For example, the text
497 “John Smith” and “he” might refer to the same `Entity`, the person John Smith.) The UIMA Base Type
498 System defines a standard way to encode this information, using the `Annotation` and `Entity` types, and
499 `occurrences/occurrenceOf` features.

500

501 Note that an `Entity` in this context need not be a physical object. For example, `Event` and `Relation` are
502 also considered kinds of `Entity`.

503 **4.3.2.4 Additional Annotation Metadata**

504 In many applications, it will be important to capture metadata about each annotation. In the Base Type
505 System, we introduce an `AnnotationMetadata` class to capture this information. This class provides
506 fields for *confidence*, a float indicating how confident the annotation engine that produced the annotation
507 was in that annotation, and *provenance*, a `Provenance` object which stores information about the source
508 of an annotation. Users may subclass `AnnotationMetadata` and `Provenance` as needed to capture
509 additional application-specific information about annotations.

510 **4.3.3 View Base Type System**

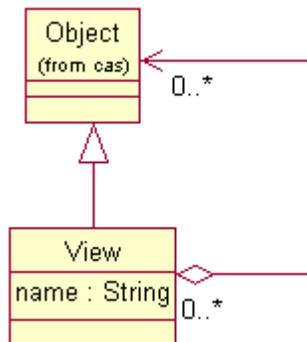
511 A `View`, depicted in Figure 4, is a named collection of *objects* in a CAS. In general a view can represent
512 any subset of the *objects* in the CAS for any purpose. It is intended however that `Views` represent
513 different perspectives of the artifact represented by the CAS. Each `View` is intended to partition the
514 artifact metadata to capture a specific perspective.

515

516 For example, given a CAS representing a document, one `View` may capture the metadata describing an
517 English translation of the document while another may capture the metadata describing a French
518 translation of the document.

519 In another example, given a CAS representing a document, one view many contain an analysis produced
520 using company-confidential data another may produce an analysis using generally available data.

521



522

523

Figure 4: View Type

524

525 UIMA does not require the use of `Views`. However, our experiences developing Apache UIMA suggest
526 that it is a useful design pattern to organize the metadata in a complex CAS by partitioning it into `Views`.
527 Individual analytics may then declare that they require certain `Views` as input or produce certain `Views` as
528 output.

529

530 Any application-specific type system could define a *class* that represents a named collection of *objects*
531 and then refer to that *class* in an analytic’s behavioral specification. However, since it is a common design

532 pattern we define a standard View class to facilitate interoperability between components that operate on
533 such collections of objects.

534

535 The members of a view are those objects explicitly asserted to be contained in the View. Referring to the
536 UML in Figure 4, we mean that there is an explicit reference from the View to the member object.
537 Members of a view may have references to other objects that are not members of the same View. A
538 consequence of this is that we cannot in general "export" the members of a View to form a new self-
539 contained CAS, as there could be dangling references. We define the **reference closure of a view** to
540 mean the collection of objects that includes all of the members of the view but also contains all other
541 objects referenced either directly or indirectly from the members of the view.

542 4.3.3.1 Anchored View

543 A common and intended use for a View is to contain metadata that is associated with a specific
544 interpretation or perspective of an artifact. An application, for example, may produce an analysis of both
545 the XML tagged view of a document and the de-tagged view of the document.

546

547 AnchoredView is as a subtype of View that has a named association with exactly one particular object via
548 the standard feature sofa.

549

550 An AnchoredView requires that all Annotation objects that are members of the AnchoredView have their
551 sofa feature refer to the same SofaReference that is referred to by the View's sofa feature.

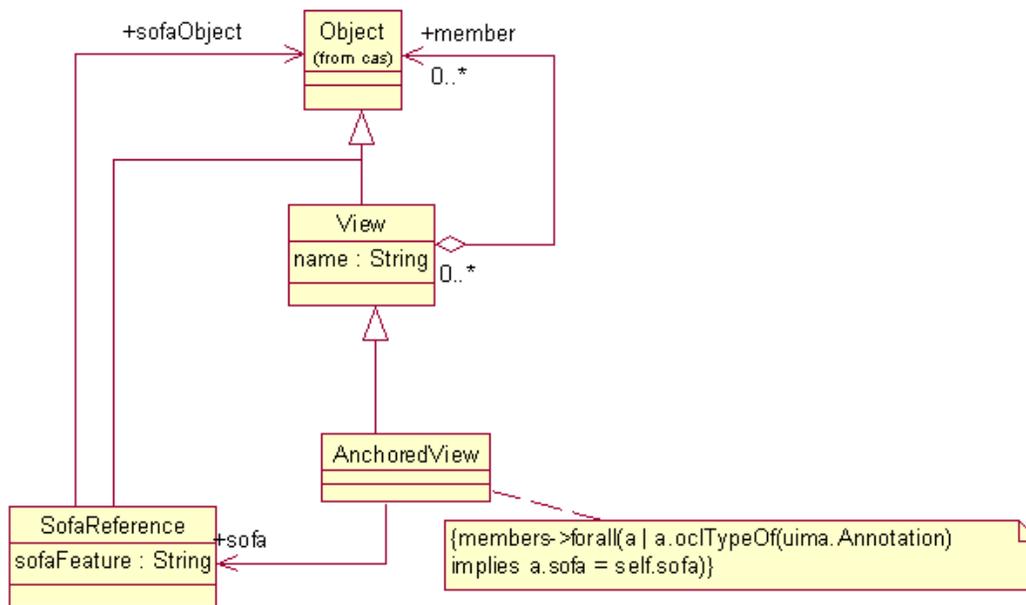
552

553 Simply put, all annotations in an AnchoredView annotate the same subject of analysis.

554

555 Figure 5 shows a UML diagram for the AnchoredView type, including an OCL constraint
556 expression[OCL1] specifying the restriction on the sofa feature of its member annotations.

557



558

559

560

Figure 5: Anchored View Type

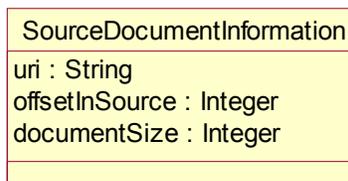
561 The concept of an AnchoredView addresses common use cases. For example, an analytic written to
562 analyze the detagged representation of a document will likely only be able to interpret Annotations that
563 label and therefore refer to regions in that detagged representation. Other Annotations, for example
564 whose offsets referred back to the XML tagged representation or some other subject of analysis would
565 not be correctly interpreted since they point into and describe content the analytic is unaware of.

566

567 If a chain of analytics are intended to all analyze the same representation of the artifact, they can all
568 declare that AnchoredView as a precondition in their Behavioral Specification (see Section 4.5 Behavioral
569 Metadata). With AnchoredViews, all the analytics in the chain can simply assume that all regional
570 references of all Annotations that are members of the AnchoredView refer to the AnchoredView's sofa.
571 This saves them the trouble of filtering Annotations to ensure they all refer to a particular sofa.

572 4.3.4 Source Document Information

573 Often it is useful to record in a CAS some information about the original source of the unstructured data
574 contained in that CAS. In many cases, this could just be a URL (to a local file or a web page) where the
575 source data can be found.



576

577 **Figure 6: Source Document Information UML**

578 Figure 6 contains the specification of a SourceDocumentInformation type included in the BaseType
579 System that can be stored in a CAS and used to capture this information. Here, the offsetInSource and
580 documentSize attributes must be byte offsets into the source, since that source may be in any modality.

581 4.3.5 Formal Specification

582 The Base Type System is formally defined by the Ecore model in Appendix C.3. UIMA services and
583 applications SHOULD use the Base Type System to facilitate interoperability with other UIMA services
584 and applications. The XML namespace <http://docs.oasis-open.org/uima/base.ecore> is reserved
585 for use by the Base Type System Ecore model, and user-defined Type Systems (such as those
586 referenced in PE metadata as discussed in Section 4.6.1.3) MUST NOT define their own type definitions
587 in this namespace.

588 4.4 Abstract Interfaces

589 The UIMA specification defines two fundamental types of Processing Elements (PEs) that developers
590 may implement: *Analytics* and *Flow Controllers*. Refer to Figure 7 for a UML model of the Analytic
591 interfaces and Figure 8 for a UML model of the FlowController interface.

592 4.4.1 Processing Element

593 The base ProcessingElement interface defines the following operations, which are common to all
594 subtypes of ProcessingElement:

- 595 • `getMetadata`, which takes no arguments and returns the *PE Metadata* for the service.
- 596 • `setConfigurationParameters`, which takes a ConfigurationParameterSettings object that
597 contains a set of (name, values) pairs that identify configuration parameters and the values to
598 assign to them.

599

600 After a client calls `setConfigurationParameters`, those parameter settings are applied to all subsequent
601 requests from that client. Note that if the Processing Element service is shared by multiple clients, it
602 needs to keep their configuration parameter settings separate.

603 4.4.2 Analytic

604 An Analytic is a component that performs analysis on CASes. There are two specializations: Analyzer
605 and CasMultiplier. The Analyzer interface supports Analytics that take a CAS as input and output the
606 same CAS, possibly updated. The CasMultiplier interface supports zero or more output CASes per input
607 CAS. This is useful for example to implement a “segmenter” analytic that takes an input CAS and divides
608 it into pieces, outputting each piece as a new CAS.

609 4.4.3 Analyzer

610 The Analyzer interface defines two additional operations:

- 611 • `processCas`, which takes a single CAS plus a list of Sofas to analyze, and returns either an
612 updated CAS, or a set of updates to apply to the CAS.
- 613 • `processCasBatch`, which takes multiple CASes, each with a list of Sofas to analyze, and returns
614 a response that contains, for each of the input CASes: an updated CAS, a set of updates to apply
615 to the CAS, or an exception.

616

617 The `processCasBatch` operation is provided for performance reasons. An Analyzer may not *require*
618 multiple CASes to be passed to it in a batch, and the result of calling `processCasBatch` must be
619 equivalent to that of making several individual calls to `processCas`.

620

621 If an application needs to consider an entire set of CASes in order to make decisions about annotating
622 each individual CAS, it is up to the application to implement this. An example of how to do this would be
623 to use an external resource such as a database, which is populated during one pass and read from
624 during a subsequent pass.

625 4.4.4 CAS Multiplier

626 The CasMultiplier interface can take a CAS as input and produce zero or more additional CASes as
627 output. This is useful for example to implement a “segmenter” analytic that takes an input CAS and
628 divides it into pieces, outputting each piece as a new CAS. The CasMultiplier interface defines the
629 following operations:

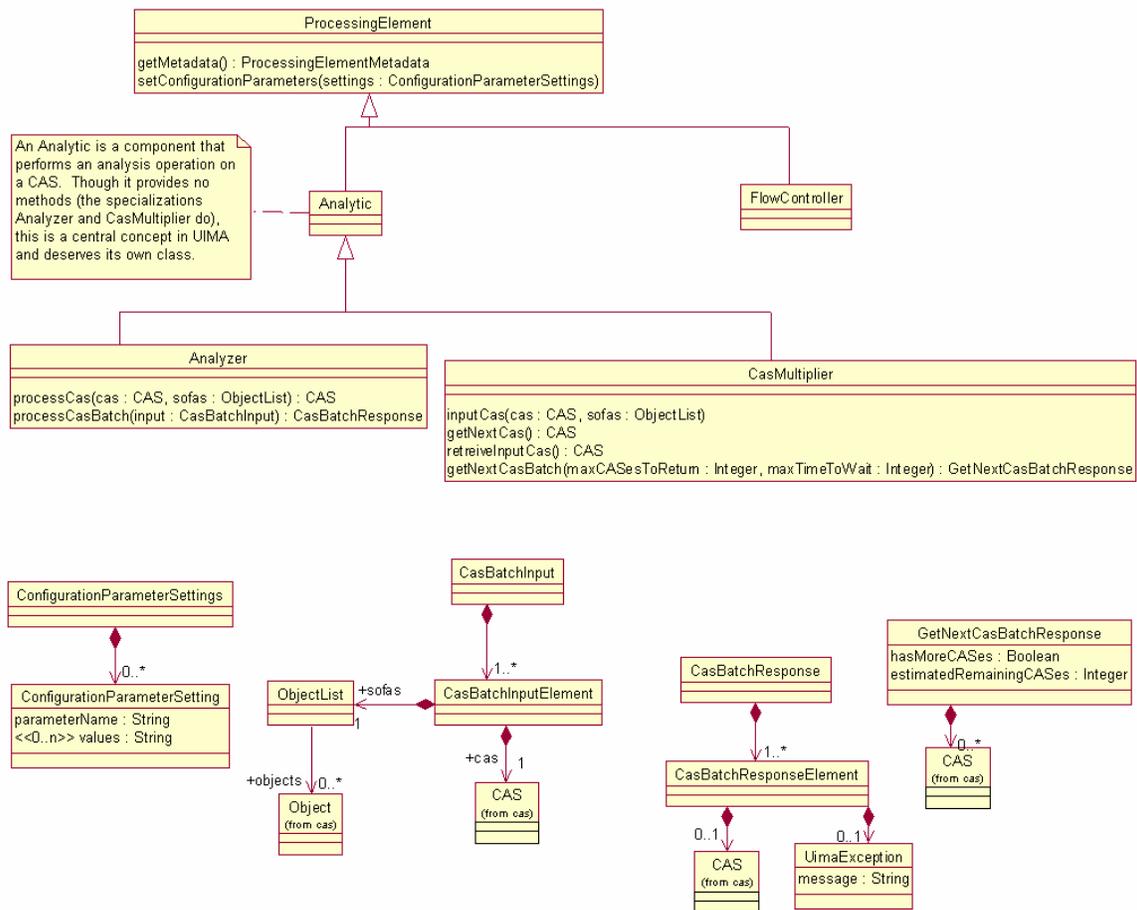
630

- 631 • `inputCas`, which takes a CAS plus a list of Sofas, but returns nothing.
- 632 • `getNextCas`, which takes no input and returns a CAS. This returns the next output CAS. An
633 empty response indicates no more output CASes.
- 634 • `retrieveInputCas`, which takes no arguments and returns the original input CAS, possibly
635 updated.
- 636 • `getNextCasBatch`, which takes a maximum number of CASes to return and a maximum
637 amount of time to wait (in milliseconds), and returns a response that contains: Zero or more
638 CASes (up to the maximum number specified), a Boolean indicating whether any more CASes
639 remain, and an estimate of the number of CASes remaining (if known).

640

641 A CAS Multiplier may also be used to merge multiple input CASes into one output CAS. Upon receiving
642 the first `inputCas` call, the CAS Multiplier would return 0 output CASes and would wait for the next
643 `inputCas` call. It would continue to return 0 output CASes until it has seen some number of input CASes,
644 at which point it would then output the one merged CAS.

645



646
647

Figure 7: Abstract Interfaces UML (Flow Controller Detail Omitted)

648 **4.4.5 Flow Controller**

649 The FlowController interface, shown in Figure 8, defines the operations:

- 650 • `addAvailableAnalytics`, which provides the Flow Controller with access to the Analytic
651 Metadata for all of the Analytics that the Flow Controller may route CASes to. This takes a map
652 from String keys to ProcessingElementMetadata objects. This may be called multiple times, if
653 new analytics are added to the system after the original call is made.
- 654 • `removeAvailableAnalytics`, which takes a set of `Keys` and instructs the Flow Controller to
655 remove some Analytics from consideration as possible destinations.
- 656 • `setAggregateMetadata`, which provides the Flow Controller with Processing Element Metadata
657 that identifies and describes the desired behavior of the entire flow of components that the
658 FlowController is managing. The most common use for this is to specify the desired outputs of
659 the aggregate, so that the Flow Controller can make decisions about which analytics need to be
660 invoked in order to produce those outputs.
- 661 • `getNextDestinations`, which takes a CAS and returns one or more destinations for this CAS.
- 662 • `continueOnFailure`, which can be called by the aggregate/application when a Step issued by
663 the FlowController failed. The FlowController returns true if it can continue, and can change the
664 subsequent flow in any way it chooses based on the knowledge that a failure occurred. The
665 FlowController returns false if it cannot continue.

666

667 The application or aggregate framework containing the FlowController is expected to call
 668 addAvailableAnalytics and setAggregateMetadata before making calls to getNextDestinations. When
 669 getNextDestinations is called, the FlowController implementation uses the available metadata along with
 670 any data in the CAS to choose the next destinations from this set of analytics. The FlowController
 671 responds with a Step object, of which there are three subtypes:

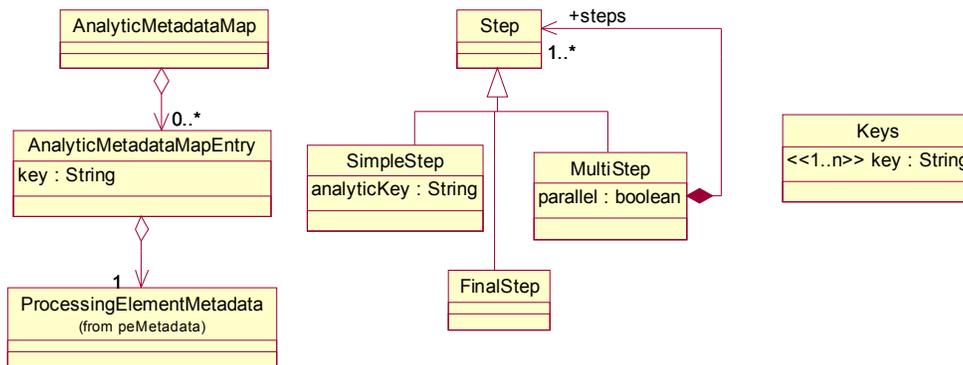
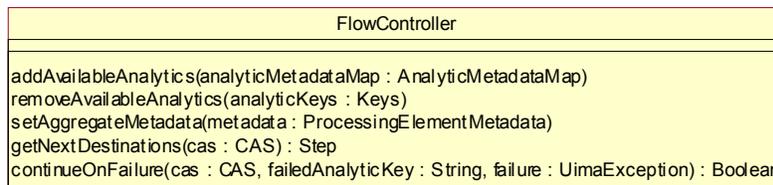
- 672 1. SimpleStep, which identifies a single Analytic to be executed. The Analytic is identified by the
 673 String key that was associated with that Analytic in the AnalyticMetadataMap.
- 674 2. MultiStep, which identifies one more Steps that should be executed next. The MultiStep also
 675 indicates whether these steps must be performed sequentially or whether they may be performed
 676 in parallel.
- 677 3. FinalStep, which indicates that there are no more destinations for this CAS, i.e., that processing
 678 of this CAS has completed.

679
 680 A FlowController, being a subtype of ProcessingElement, may have configuration parameters. For
 681 example, a configuration parameter may refer to a description of the desired flow in some flow language
 682 such as BPEL [BPEL1]. This is one way to create a reusable Flow Controller implementation that can be
 683 applied in many applications or aggregates.

684
 685 Note that the FlowController is not responsible for knowing how to actually invoke a constituent analytic.
 686 Invoking the constituent analytic is the job of the application or aggregate framework that encapsulates
 687 the FlowController. This is an important separation of concerns since applications or frameworks may
 688 use arbitrary protocols to communicate with constituent analytics and it is not reasonable to expect a
 689 reusable FlowController to understand all possible protocols.

690
 691 A Flow Controller may not modify the CAS. However, a concrete implementation of the Flow Controller
 692 interface could provide additional operations on the Flow Controller which allow it to return data. For
 693 example, it could return a Flow data structure to allow the application to get information about the flow
 694 history.

695



696

697

Figure 8: Flow Controller Abstract Interface UML

698 4.4.6 Formal Specification

699 The following subsections specify requirements that a particular type of UIMA service must provide an
700 operation with certain inputs and outputs. For example, a UIMA PE service must implement a
701 `getMetaData` operation that returns standard UIMA PE Metadata. In all cases, the protocol for invoking
702 this operation is not defined by the standard. However, the format in which data is sent to and from the
703 service MUST be the standard UIMA XML representation. Implementations MAY define additional
704 operations that use other formats.

705 4.4.6.1 `ProcessingElement.getMetaData`

706 A UIMA Processing Element (PE) Service MUST implement an operation named `getMetaData`. This
707 operation MUST take zero arguments and MUST return PE Metadata XML as defined in Section 4.6.2. In
708 the following sections, we use the term “this PE Service’s Metadata” to refer to the PE Metadata returned
709 by this operation.

710 4.4.6.2 `ProcessingElement.setConfigurationParameters`

711 A UIMA Processing Element (PE) Service MUST implement an operation named
712 `setConfigurationParameters`. This operation MUST accept one argument, an instance of the
713 `ConfigurationParameterSettings` type defined by the XML Schema in Section C.7.

714

715 The PE Service MUST return an error if the `ConfigurationParameterSettings` object passed to this
716 method contains any of:

- 717 1. a `parameterName` that does not match any of the parameter names declared in this PE Service’s
718 Metadata.
- 719 2. multiple values for a parameter that is not declared as `multiValued` in this PE Service’s Metadata.
- 720 3. a value that is not a valid instance of the type of the parameter as declared in this PE Service’s
721 Metadata. To be a valid instance of the UIMA configuration parameter type, the value must be a
722 valid instance of the corresponding XML Schema datatype in Table 1: Mapping of UIMA
723 Configuration Parameter Types to XML Schema Datatypes, as defined by the XML Schema
724 specification [XMLS2].

725

UIMA Configuration Parameter Type	XML Schema Datatype
String	string
Integer	int
Float	float
Boolean	boolean
ResourceURL	anyURI

726

Table 1: Mapping of UIMA Configuration Parameter Types to XML Schema Datatypes

727

728 After a client calls `setConfigurationParameters`, those parameter settings MUST be applied to all
729 subsequent requests from that client, until such time as a subsequent call to `setConfigurationParameters`
730 specifies new values for the same parameter(s). If the PE service is shared by multiple clients, the PE
731 service MUST provide a way to keep their configuration parameter settings separate.

732

733 **4.4.6.3 Analyzer.processCas**

734 A UIMA Analyzer Service MUST implement an operation named `processCas`. This operation MUST
735 accept two arguments. The first argument is a CAS, represented in XML as defined in Section 4.1.4. The
736 second argument is a list of `xmi:ids` that identify `SofaReference` objects which the Analyzer is expected
737 to analyze. This operation MUST return a valid XML document which is either a valid CAS (as defined in
738 Section 4.1.4) or a description of changes to be applied to the input CAS using the XML differences
739 language defined in [XMI1].

740

741 The output CAS of this operation represents an update of the input CAS. Formally, this means :

- 742 1. All objects in the input CAS must appear in the output CAS, except where an explicit delete or
743 modification was performed by the service (which is only allowed if such operations are declared
744 in the Behavioral Metadata element of this service's PE Metadata).
- 745 2. For the `processCas` operation, an object that appears in both the input CAS and output CAS must
746 have the same value for `xmi:id`.
- 747 3. No newly created object in the output CAS may have the same `xmi:id` as any object in the input
748 CAS.

749

750 The input CAS may contain a reference to its type system (see Section B.1.6). If it does not, then the
751 PE's type system (see Section 4.6.1.3) may provide definitions of the types. If the CAS contains an
752 instance of a type that is not defined in either place, then the PE MUST return that object, unmodified.

753

754 **4.4.6.4 Analyzer.processCasBatch**

755 A UIMA Analyzer Service MUST implement an operation named `processCasBatch`. This operation
756 MUST accept an argument which consists of one or more CASes, each with an associated list of `xmi:ids`
757 that identify `SofaReference` objects in that CAS. This operation MUST return a response that consists of
758 multiple elements, one for each input CAS, where each element is either valid XML document which is
759 either a valid CAS (as defined in Section 4.1.4), a description of changes to be applied to the input CAS
760 using the XML differences language defined in [XMI1], or an exception message.

761

762 The CASes that result from calling `processCasBatch` MUST be identical to the CASes that would result
763 from several individual `processCas` operations, each taking only one of the CASes as input.

764 **4.4.6.5 CasMultiplier.inputCas**

765 A UIMA CAS Multiplier service MUST implement an operation named `inputCas`. This operation MUST
766 accept two arguments. The first argument is a CAS, represented in XML as defined in Section 4.1.4. The
767 second argument is a list of `xmi:ids` that identify `SofaReference` objects which the Analyzer is expected
768 to analyze. This operation returns nothing.

769

770 The CAS that is passed to this operation becomes this CAS Multiplier's *active CAS*.

771 **4.4.6.6 CasMultiplier.getNextCas**

772 A UIMA CAS Multiplier service MUST implement an operation named `getNextCas`. This operation
773 MUST take zero arguments. This operation MUST return a valid CAS as defined in Section 4.1.4, or a
774 result indicating that there are no more CASes available.

775

776 If the client calls `getNextCas` when this CAS Multiplier has no active CAS, then this CAS Multiplier MUST
777 return an error.

778 **4.4.6.7 CasMultiplier.retrieveInputCas**

779 A UIMA CAS Multiplier service MUST implement an operation named `retrieveInputCas`. This operation
780 MUST take zero arguments and MUST return a valid XML document which is either a valid CAS (as
781 defined in Section 4.1.4) or a description of changes to be applied to the CAS Multiplier's active CAS
782 using the XML differences language defined in [XMI1].

783

784 If the client calls `retrieveInputCas` when this CAS Multiplier has no active CAS, then this CAS Multiplier
785 MUST return an error.

786

787 After this method completes, this service no longer has an active CAS, until the client's next call to
788 `inputCas`.

789 **4.4.6.8 CasMultiplier.getNextCasBatch**

790 A UIMA CAS Multiplier service MUST implement an operation named `getNextCasBatch`. This
791 operation MUST take two arguments, both of which are integers. The first argument (named
792 `maxCASesToReturn`) specifies the maximum number of CASes to be returned, and the second argument
793 (named `maxTimeToWait`) indicates the maximum number of milliseconds to wait. This operation MUST
794 return an object with three fields:

- 795 1. Zero or more valid CASes as defined in Section 4.1.4. The number of CASes MUST NOT
796 exceed the value of the `maxCASesToReturn` argument.
- 797 2. a Boolean indicating whether more CAS remain to be retrieved.
- 798 3. An estimated number of remaining CASes. The estimated number of remaining CASes may be
799 set to -1 to indicate an unknown number.

800

801 The call to `getNextCasBatch` SHOULD attempt to complete and return a response in no more than the
802 amount of time specified (in milliseconds) by the `maxTimeToWait` argument.

803

804 If the client calls `getNextCasBatch` when this CAS Multiplier has no active CAS, then this CAS Multiplier
805 MUST return an error.

806

807 CASes returned from `getNextCasBatch` MUST be equivalent to the CASes that would be returned from
808 individual calls to `getNextCas`.

809 **4.4.6.9 FlowController.addAvailableAnalytics**

810 A UIMA Flow Controller service MUST implement an operation named `addAvailableAnalytics`. This
811 operation MUST accept one argument, a Map from String keys to PE Metadata objects. Each of the
812 String keys passed to this operation is added to the set of *available analytic keys* for this Flow Controller
813 service.

814 **4.4.6.10 FlowController.removeAvailableAnalytics**

815 A UIMA Flow Controller service MUST implement an operation named `removeAvailableAnalytics`.
816 This operation MUST accept one argument, which is a collection of one or more String keys. If any of the
817 String keys passed to this operation are not a member of the set of *available analytic keys* for this Flow
818 Controller service, an error MUST be returned. Each of the String keys passed to this operation is
819 removed from the set of *available analytic keys* for this FlowController service.

820 **4.4.6.11 FlowController.setAggregateMetadata**

821 A UIMA Flow Controller service MUST implement an operation named `setAggregateMetadata`. This
822 operation MUST take one argument, which is valid PE Metadata XML as defined in Section 4.6.2.

823

824 There are no formal requirements on what the Flow Controller does with this PE Metadata, but the
825 intention is for the PE Metadata to specify the desired outputs of the workflow, so that the Flow Controller
826 can make decisions about which analytics need to be invoked in order to produce those outputs.

827 **4.4.6.12 FlowController.getNextDestinations**

828 A UIMA Flow Controller service MUST implement an operation named `getNextDestinations`. This
829 operation MUST accept one argument, which is an XML CAS as defined in Section 4.1.4 and MUST
830 return an instance of the `Step` type defined by the XML Schema in Section C.7.

831

832 The different types of Step objects are defined in the UML diagram in Figure 8 and XML schema in
833 Appendix C.7. Their intending meanings are documented in section 4.4.5.

834

835 Each `analyticKey` field of a Step object returned from the `getNextDestinations` operation MUST be a
836 member of the set of *active analytic* keys of this Flow Controller service.

837 **4.4.6.13 FlowController.continueOnFailure**

838 A UIMA FlowController service MUST define an operation named `continueOnFailure`. This operation
839 MUST accept three arguments as follows. The first argument is an XML CAS as defined in Section 4.1.4.
840 The second argument is a String key. The third argument is an instance of the `UimaException` type
841 defined in the XML schema in Section C.7.

842

843 If the String key is not a member of the set of *active analytic keys* of this Flow Controller, then an error
844 must be returned.

845

846 This method is intended to be called by the client when there was a failure in executing a Step issued by
847 the FlowController. The client is expected to pass the CAS that failed, the analytic key from the Step
848 object that was being executed, and the exception that occurred.

849

850 Given that the above assumptions hold, the `continueOnFailure` operation SHOULD return true if a further
851 call to `getNextDestinations` would succeed, and false if a further call to `getNextDestinations` would fail.

852

853 **4.5 Behavioral Metadata**

854 **4.5.1 Behavioral Metadata UML**

855 The following UML diagram defines the UIMA Behavioral Metadata representation:

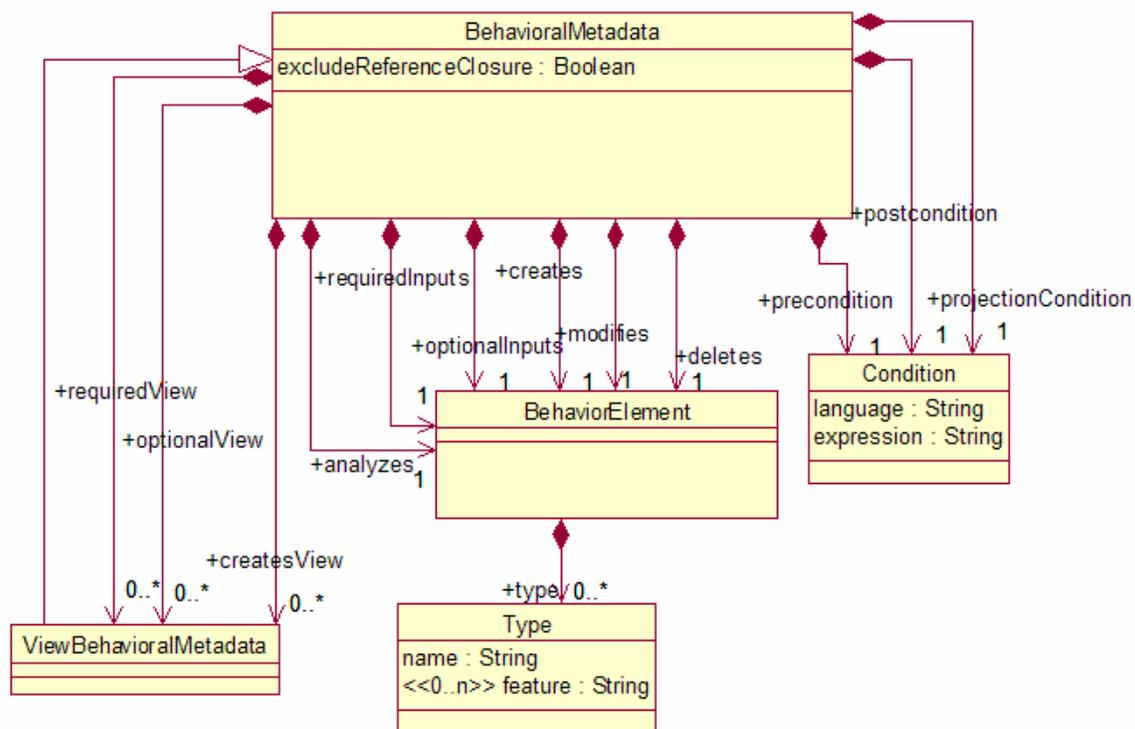


Figure 9: Behavioral Metadata UML

856
857
858

4.5.2 Behavioral Metadata Elements and XML Representation

Behavioral Metadata breaks down into the following categories:

- 861 • **Analyzes:** Types of objects (Sofas) that the analytic intends to produce annotations over.
- 862 • **Required Inputs:** Types of objects that must be present in the CAS for the analytic to operate.
- 863 • **Optional Inputs:** Types of objects that the analytic would consult if they were present in the CAS.
- 864 • **Creates:** Types of objects that the analytic may create.
- 865 • **Modifies:** Types of objects that the analytic may modify.
- 866 • **Deletes:** Types of objects that the analytic may delete.

867

868 The representation of these elements in XML is defined by the BehavioralMetadata element definition in
869 the XML schema given in Appendix C.5. For examples and discussion, see Appendix B.5.

4.5.3 Formal Semantics for Behavioral Metadata

871 All Behavioral Metadata elements may be mapped to three kinds of expressions in a formal language: a
872 **Precondition**, a **Postcondition**, and a **Projection Condition**.

873

874 A *Precondition* is a predicate that qualifies CASs that the analytic considers valid input. More precisely
875 the analytic's behavior would be considered unspecified for any CAS that did not satisfy the pre-condition.
876 The pre-condition may be used by a framework or application to filter or skip CASs routed to an analytic
877 whose pre-condition is not satisfied by the CASs. A human assembler or automated composition process
878 can interpret the pre-conditions to determine if the analytic is suitable for playing a role in some aggregate
879 composition.

880

881 A *Postcondition* is a predicate that is declared to be true of any CAS after having been processed by the
882 analytic, assuming that the CAS satisfied the precondition when it was input to the analytic.

883

884 For example, if the pre-condition requires that valid input CASs contain People, Places and
885 Organizations, but the Postconditions of the previously run Analytic asserts that the CAS will not contain
886 all of these objects, then the composition is clearly invalid.

887

888 A *Projection Condition* is a predicate that is evaluated over a CAS and which evaluates to a subset of the
889 objects in the CAS. This is the set of objects that the Analytic declares that it will consider to perform its
890 function.

891

892 The following is a high-level description of the mapping from Behavioral Metadata Elements to
893 preconditions, postconditions, and projection conditions. For a precise definition of the mapping, see
894 Section 4.5.4.3.

895

896 An `analyzes` or `requiredInputs` predicate translates into a precondition that all input CASes contain the
897 objects that satisfy the predicates.

898

899 A `deletes` predicate translates into a postcondition that for each object O in the input CAS, if O does not
900 satisfy the `deletes` predicate, then O is present in the output CAS.

901

902 A `modifies` predicate translates into a postcondition that for each object O in the input CAS, if O does not
903 satisfy the `modifies` predicate, and if O is present in the output CAS (i.e. it was not deleted), then O has
904 the same values for all of its slots.

905

906 For views, we add the additional constraint that objects are members of that View (and therefore
907 annotations refer to the View's sofa). For example:

```
908 <requiredView sofaType="org.example:TextDocument">
```

```
909   <requiredInputs>
```

```
910     <type>org.example:Token</type>
```

```
911   </requiredInputs>
```

```
912 </requiredView>
```

913

914 This translates into a precondition that the input CAS must contain an anchored view V where V is linked
915 to a Sofa of type TextDocument and V.members contains at least one object of type Token.

916

917 Finally, the projection condition is formed from a disjunction of the “analyzes,” “required inputs,” and
918 “optional inputs” predicates, so that any object which satisfies any of these predicates will satisfy the
919 projection condition.

920

921 UIMA does not mandate a particular expression language for representing these conditions.
922 Implementations are free to use any language they wish. However, to ensure a standard interpretation of
923 the standard UIMA Behavior Elements, the UIMA specification defines how the Behavior Elements map
924 to preconditions, postconditions, and projection conditions in the Object Constraint Language [OCL1], an
925 OMG standard. See Section 4.5.4.3 for details.

926 4.5.4 Formal Specification

927 4.5.4.1 Structure

928 *UIMA Behavioral Metadata XML* is a part of *UIMA Processing Element Metadata XML*. Its structure is
929 defined by the definitions of the BehavioralMetadata class in the Ecore model in C.3.

930

931 This implies that UIMA Behavioral Metadata XML must be a valid instance of the BehavioralMetadata
932 element definition in the XML schema given in Section C.5.

933 4.5.4.2 Constraints

934 Field values must satisfy the following constraints:

935 4.5.4.2.1 Type

- 936 • name must be a valid QName (Qualified Name) as defined by the Namespaces for XML specification
937 [XML2]. The namespace of this QName must match the namespace URI of an EPackage defined in an
938 Ecore model referenced by the PE's *TypeSystemReference*. The local part of the QName must match
939 the name of an EClass within that EPackage.
- 940 • Values for the `feature` attribute must not be specified unless the Type is contained in a `modifies`
941 element.
- 942 • Each value of feature must be a valid UnprefixedName as specified in [XML2], and must match the
943 name of an EStructuralFeature in the EClass corresponding to the value of the name field as described
944 in the previous bullet.

945 4.5.4.2.2 Condition

- 946 • language must be one of:
 - 947 ○ The exact string OCL. If the value of the language field is OCL, then the value of the
948 expression field must be a valid OCL expression as defined by [OCL1].
 - 949 ○ A user-defined language, which must be a String containing the '.' Character (for example
950 "org.example.MyLanguage"). Strings not containing the '.' are reserved by the UIMA
951 standard and may be defined at a later date.

952 4.5.4.3 Semantics

953 To give a formal meaning to the *analyzes*, *required inputs*, *optional inputs*, *creates*, *modifies*, and *deletes*
954 expressions, UIMA defines how these map into formal preconditions, postconditions, and projection
955 conditions in the Object Constraint Language [OCL1], an OMG standard.

956

957 The UIMA specification defines this mapping in order to ensure a standard interpretation of UIMA
958 Behavioral Metadata Elements. There is no requirement on any implementation to evaluate or enforce
959 these expressions. Implementations are free to use other languages for expressing and/or processing
960 preconditions, postconditions, and projection conditions.

961 4.5.4.3.1 Mapping to OCL Precondition

962 An OCL precondition is formed from the `analyzes`, `requiredInputs`, and `requiredView`
963 BehavioralMetadata elements as follows.

964

965 In these OCL expressions the keyword `input` refers to the collection of objects in the CAS when it is input
966 to the analytic.

967

968 For each type *T* in an `analyzes` or `requiredInputs` element, produce the OCL expression:

```
969 input->exists(p | p.oclKindOf(T))
```

970

971 For each `requiredView` element that contains `analyzes` or `requiredInputs` elements with types *T*₁, *T*₂,
972 ..., *T*_{*n*}, produce the OCL expression:

```
973 input->exists(v | ViewExpression and v.members->exists(p | p.oclKindOf(T2))  
974 and ... and v.members(exists(p | p.oclKindOf(Tn)))
```

975 There may be zero `analyzes` or `requiredInputs` elements, in which case there will be no `v.members`
976 clauses in the OCL expression.

977

978 In the above we define `ViewExpression` as follows:

979 If the `requiredView` element has no value for its `sofaType` slot, then `ViewExpression` is:

```
980 v.oclKindOf(uima::cas::View)
```

981 If the `requiredView` has a `sofaType` slot with value then `ViewExpression` is defined as:

```
982 v.oclKindOf(uima::cas::AnchoredView) and v.sofa.sofaObject.oclKindOf(S)
```

983

984 The final precondition expression for the analytic is the conjunction of all the expressions generated from
985 the productions defined in this section, as well as any explicitly declared precondition as defined in
986 Section B.5.5.

987 4.5.4.3.2 Mapping to OCL Postcondition

988 In these OCL expressions the keyword `input` refers to the collection of objects in the CAS when it was
989 input to the analytic, and the keyword `result` refers to the collection of objects in the CAS at the end of
990 the analytic's processing. Also note that the suffix `@pre` applied to any attribute references the value of
991 that attribute at the start of the analytic's operation.

992

993 For types *T*₁, *T*₂, ... *T*_{*n*} specified in `creates` elements, produce the OCL expression:

```
994 result->forAll(p | input->includes(p) or p.oclKindOf(T1) or p.oclKindOf(T2) or  
995 ... or p.oclKindOf(Tn))
```

996

997 For types *T*₁, *T*₂, ... *T*_{*n*} specified in `deletes` elements, produce the OCL expression:

```
998 input->forAll(p | result->includes(p) or p.oclKindOf(T1) or p.oclKindOf(T2) or  
999 ... or p.oclKindOf(Tn))
```

1000

1001 For each `modifies` element specifying type *T* with features *F*={*F*₁, *F*₂, ...*F*_{*n*}}, for each feature *g* defined
1002 on type *T* where *g*∉*F*, produce the OCL expression:

```
1003 result->forAll(p | (input->includes(p) and p.oclKindOf(T)) implies p.g =  
1004 p.g@pre)
```

1005

1006 For each `createsView`, `requiredView` or `optionalView` containing `creates` elements with types
1007 *T*₁,*T*₂,...,*T*_{*n*}, produce the OCL expression:

```
1008 result->forAll(v | (ViewExpression) implies v.members->forAll(p |  
1009 v.members@pre->includes(p) or p.oclKindOf(T1) or p.oclKindOf(T2) or ... or  
1010 p.oclKindOf(Tn))
```

1011 where `ViewExpression` is as defined in Section 4.5.4.3.1.

1012
1013 For each `requiredView` or `optionalView` containing `deletes` elements with types `T1,T2,...,Tn`, produce
1014 the OCL expression:

```
1015 result->forAll(v | (ViewExpression) implies v.members@pre->forAll(p |  
1016 v.members->includes(p) or p.oclKindOf(T1) or p.oclKindOf(T2) or ... or  
1017 p.oclKindOf(Tn))
```

1018 where `ViewExpression` is as defined in Section 4.5.4.3.1.

1019
1020 Within each `requiredView` or `optionalView`, for each `modifies` element specifying type `T` with features
1021 `F={F1, F2, ...Fn}`, for each feature `g` defined on type `T` where `g∉F`, produce the OCL expression:

```
1022 result->forAll(v | (ViewExpression) implies v.members->forAll(p |  
1023 (v.members@pre->includes(p) and p.oclKindOf(T)) implies p.g = p.g@pre))
```

1024 where `ViewExpression` is as defined in Section 4.5.4.3.1.

1025
1026 The final postcondition expression for the analytic is the conjunction of all the expressions generated from
1027 the productions defined in this section, as well as any explicitly declared postcondition as defined in
1028 Section B.5.5.

1029 4.5.4.3.3 Mapping to OCL Projection Condition

1030 In these OCL expressions the keyword `input` refers to the collection of objects in the entire CAS when it
1031 is about to be delivered to the analytic. The OCL expression evaluates to a collection of objects that the
1032 analytic declares it will consider while performing its operation. When an application or framework calls
1033 this analytic, it MUST deliver to the analytic all objects in this collection.

1034
1035 If the `excludeReferenceClosure` attribute of the `BehavioralMetadata` is set to `false` (or omitted), then the
1036 application or framework MUST also deliver all objects that are referenced (directly or indirectly) from any
1037 object in the collection resulting from evaluation of the projection condition.

1038
1039 For types `T1, T2, ... Tn` specified in `analyzes`, `requiredInputs`, or `optionalInputs` elements, produce
1040 the OCL expression:

```
1041 input->select(p | p.oclKindOf(T1) or p.oclKindOf(T2) or ... or  
1042 p.oclKindOf(Tn))
```

1043
1044 For each `requiredView` or `optionalView` produce the OCL expression:

```
1045 input->select(v | ViewExpression)
```

1046 where `ViewExpression` is as defined in Section 4.5.4.3.1.

1047
1048 If the `requiredView` or `optionalView` contains types `T1, T2,...Tn` specified in `analyzes`,
1049 `requiredInputs`, or `optionalInputs` elements, produce the OCL expression:

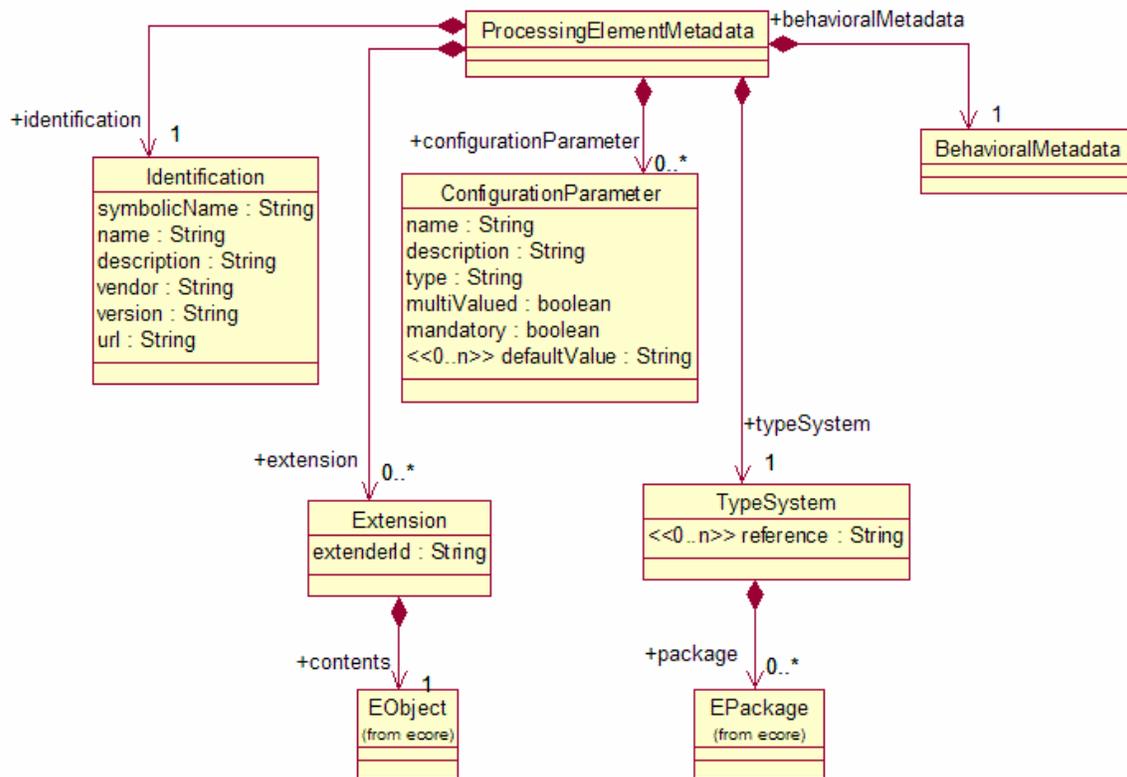
```
1050 input->select(v | ViewExpression)->collect(v.members()->select(p |  
1051 p.oclKindOf(T1) or p.oclKindOf(T2) or ... or p.oclKindOf(Tn)))
```

1052

1053 The final projection condition expression for the analytic is the result of the OCL `union` operator applied
 1054 consecutively to all of the expressions generated from the productions defined in this section, as well as
 1055 any explicitly declared projection condition as defined in Section B.5.5.
 1056

1057 4.6 Processing Element Metadata

1058 Figure 10 is a UML model for the PE metadata. We describe each subpart of the PE metadata in detail in
 1059 the following sections.
 1060



1061
 1062 **Figure 10: Processing Element Metadata UML Model**
 1063

1064 4.6.1 Elements of PE Metadata

1065 4.6.1.1 Identification Information

1066 The Identification Information section of the descriptor defines a small set of properties that developers
 1067 should fill in with information that describes their PE. The main objectives of this information are to:

- 1068 1. Provide human-readable information about the analytic to assist developers in understanding
- 1069 what the purpose of each PE is.
- 1070 2. Facilitate the development of repositories of PEs.

1071

1072 The following properties are included:

- 1073 1. Symbolic Name: A unique name (such as a Java-style dotted name) for this PE.

- 1074 2. Name: A human-readable name for the PE. Not necessarily unique.
1075 3. Description: A textual description of the PE.
1076 4. Version: A version number. This is necessary for PE repositories that need to distinguish
1077 different versions of the same component. The syntax of a version number is as defined in
1078 [OSGi1]: up to four dot-separated components where the first three must be numeric but the
1079 fourth may be alphanumeric. For example 1.2.3.4 and 1.2.3.abc are valid version numbers but
1080 1.2.abc is not.
1081 5. Vendor: The provider of the component.
1082 6. URL: website providing information about the component and possibly allowing download of the
1083 component
1084

1085 4.6.1.2 Configuration Parameters

1086 PEs may be configured to operate in different ways. UIMA provides a standard way for PEs to declare
1087 configuration parameters so that application developers are aware of the options that are available to
1088 them.

1089

1090 UIMA provides a standard interface for setting the values of parameters; see Section 4.4 Abstract
1091 Interfaces.

1092

1093 For each configuration parameter we should allow the PE developer to specify:

1094

- 1095 1. The name of the parameter
1096 2. A description for the parameter
1097 3. The type of value that the parameter may take
1098 4. Whether the parameter accepts multiple values or only one
1099 5. Whether the parameter is mandatory
1100 6. A default value or values for the parameter

1101

1102 One common use of configuration parameters is to refer to external resource data, such as files
1103 containing patterns or statistical models. Frameworks such as Apache UIMA may wish to provide
1104 additional support for such parameters, such as resolution of relative URLs (using classpath/datapath)
1105 and/or caching of shared data. It is therefore important for the UIMA configuration parameter schema to
1106 be expressive enough to distinguish parameters that represent resource locations from parameters that
1107 are just arbitrary strings.

1108

1109 The type of a parameter must be one of the following:

- 1110 • String
1111 • Integer (32-bit)
1112 • Float (32-bit)
1113 • Boolean
1114 • ResourceURL

1115

1116 The ResourceURL satisfies the requirement to explicitly identify parameters that represent resource
1117 locations.

1118

1119 Note that parameters may take multiple values so it is not necessary to have explicit parameter types
1120 such as StringArray, IntegerArray, etc.

1121

1122 As a best practice, analytics SHOULD NOT declare configuration settings that would affect their
1123 Behavioral Metadata. UIMA does not provide any mechanism to keep the behavioral specification in sync
1124 with the different configurations.

1125 **4.6.1.3 Type System**

1126 There are two ways that PE metadata may provide type system information: It can either include it or refer
1127 to it. This specification is only concerned with the format of that reference or inclusion. For the actual
1128 definition of the type system, we have adopted the Ecore/XML representation. See Section 4.2 for details.

1129
1130 If reference is chosen as the way to provide the type system information, then the `reference` field of the
1131 `TypeSystem` object must be set to a valid URI (or multiple URIs). URIs are used as references by many
1132 web-based standards (e.g., RDF), and they are also used within Ecore. Thus we use a URI to refer to the
1133 type system. To achieve interoperability across frameworks, each URI should be a URL which resolves
1134 to a location where Ecore/XML type system data is located.

1135
1136 If embedding is chosen as the way to provide the type system information, then the `package` reference of
1137 the `TypeSystem` object must be set to one or more `EPackages`, where an `EPackage` contains
1138 subpackages and/or classes as defined by Ecore.

1139
1140 The role of this type system is to provide definitions of the types referenced in the PE's behavioral
1141 metadata. It is important to note that this is not a restriction on the CASes that may be input to the PE (if
1142 that is desired, it can be expressed using a precondition in the behavioral specification). If the input CAS
1143 contains instances of types that are not defined by the PE's type system, then the CAS itself may indicate
1144 a URI where definitions of these types may be found (see B.1.6 Linking an XML Document to its Ecore
1145 Type System). Also, some PE's may be capable of processing CASes without being aware of the type
1146 system at all.

1147
1148 Some analytics may be capable of operating on any types. These analytics need not refer to any specific
1149 type system and in their behavioral metadata may declare that they analyze or inspect instances of the
1150 most general type (`EObject` in Ecore).

1151 **4.6.1.4 Behavioral Metadata**

1152 The Behavioral Metadata is discussed in detail in 4.5.

1153 **4.6.1.5 Extensions**

1154 Extension objects allow a framework implementation to extend the PE metadata descriptor with additional
1155 elements, which other frameworks may not necessarily respect. For example Apache UIMA defines an
1156 element `fsIndexCollection` that defines the CAS indexes that the component uses. Other frameworks
1157 could ignore that.

1158
1159 This extensibility is enabled by the Extension class in Figure 10. The Extension class defines two
1160 *features*, `extenderId` and `contents`.

1161
1162 The `extenderId` *feature* identifies the framework implementation that added the extension, which allows
1163 framework implementations to ignore extensions that they were not meant to process.

1164
1165 The `contents` *feature* can contain any `EObject`. (`EObject` is the superclass of all classes in Ecore.) To add
1166 an extension, a framework must provide an Ecore model that defines the structure of the extension.

1167 **4.6.2 Formal Specification**

1168 **4.6.2.1 Structure**

1169 *UIMA Processing Element Metadata XML* must be a valid XML document that is an instance of the UIMA
1170 Processing Element Metadata Ecore model given in Section C.3.

1171

1172 This implies that UIMA Processing Element Metadata XML must be a valid instance of the UIMA
1173 Processing Element Metadata XML schema given in Section C.5.

1174 **4.6.2.2 Constraints**

1175 Field values must satisfy the following constraints

1176

1177 **Identification Information:**

- 1178 • symbolicName must be a valid symbolic-name as defined by the OSGi specification [OSGi1].
- 1179 • version must be a valid version as defined by the OSGi specification [OSGi1].
- 1180 • url must be a valid URL as defined by [URL1].

1181

1182 **Configuration Parameter**

- 1183 • name must be a valid Name as defined by the XML specification [XML1].
- 1184 • type must be one of {String, Integer, Float, Boolean, ResourceURL}

1185

1186 **Type System Reference**

- 1187 • uri must be a syntactically valid URI as defined by [URI1] It is application defined to check the reference
1188 validity of the URI and handle errors related to dereferencing the URI.

1189

1190 **Extensions**

- 1191 • extenderId must be a valid Name as defined by the XML specification [XML1].

1192

1193 **4.7 Service WSDL Descriptions**

1194 In this section we describe the UIMA Service WSDL descriptions at a high level. The formal WSDL
1195 document is given in Section C.6.

1196

1197 **4.7.1 Overview of the WSDL Definition**

1198 Before discussing the elements of the UIMA WSDL definition, as a convenience to the reader we first
1199 provide an overview of WSDL excerpted from the WSDL Specification.

1200

Excerpt from WSDL W3C Note [<http://www.w3.org/TR/wsdl>]

As communications protocols and message formats are standardized in the web community, it becomes increasingly possible and important to be able to describe the communications in some structured way. WSDL addresses this need by defining an XML grammar for describing network services as collections of communication endpoints capable of exchanging messages. WSDL service definitions provide documentation for distributed systems and serve as a recipe for automating the details involved in applications communication.

A WSDL document defines services as collections of network endpoints, or ports. In WSDL, the abstract definition of endpoints and messages is separated from their concrete network deployment or data format bindings. This allows the reuse of abstract definitions: messages, which are abstract descriptions of the data being exchanged, and port types which are abstract collections of operations. The concrete protocol and data format specifications for a particular port type constitutes a reusable binding. A port is defined by associating a network address with a reusable binding, and a collection of ports define a service. Hence, a WSDL document uses the following elements in the definition of network services:

- Types – a container for data type definitions using some type system (such as XSD).
- Message – an abstract, typed definition of the data being communicated.
- Operation – an abstract description of an action supported by the service.
- Port Type – an abstract set of operations supported by one or more endpoints.
- Binding – a concrete protocol and data format specification for a particular port type.
- Port – a single endpoint defined as a combination of a binding and a network address.
- Service – a collection of related endpoints.

1201
1202

1203 **4.7.1.1 Types**

1204 Type Definitions for the UIMA WSDL service are defined using XML schema. These draw from other
1205 elements of the specification. For example the `ProcessingElementMetadata` type, which is returned
1206 from the `getMetadata` operation, is defined by the PE Metadata specification element.

1207

1208 **4.7.1.2 Messages**

1209 Messages are used to define the structure of the request and response of the various operations
1210 supported by the service. Operations are described in the next section.

1211

1212 Messages refer to the XML schema defined under the `<wsdl:types>` element. So wherever a message
1213 includes a CAS (for example the `processCasRequest` and `processCasResponse`, we indicate that the
1214 type of the data is `xmi:XMI` (a type defined by `XMI.xsd`), and where the message consists of PE metadata
1215 (the `getMetadataResponse`), we indicate that the type of the data is `uima:ProcessingElementMetadata` (a
1216 type defined by `UimaDescriptorSchema.xsd`).

1217

1218 The messages defined by the UIMA WSDL service definition are:

1219 For ALL PEs:

- 1220 • getMetadataRequest – takes no arguments
- 1221 • getMetadataResponse – returns ProcessingElementMetadata
- 1222 • setConfigurationParametersRequest – takes one argument: ConfigurationParameterSettings
- 1223 • setConfigurationParameterResponse – returns nothing

1224

1225 For Analyzers:

- 1226 • processCasRequest – takes two arguments – a CAS and a list of Sofas (object IDs) to process
- 1227 • processCasResponse – returns a CAS
- 1228 • processCasBatchRequest – takes one argument, an Object that includes multiple CASes, each with an associated list of Sofas (object IDs) to process
- 1229
- 1230 • processCasResponse – returns a list of elements, each of which is a CAS or an exception message

1231

1232 For CAS Multipliers:

- 1233 • inputCasRequest – takes two arguments – a CAS and a list of Sofas (object IDs) to process
- 1234 • inputCasResponse – returns nothing
- 1235 • getNextCasRequest – takes no arguments
- 1236 • getNextCasResponse – returns a CAS
- 1237 • retrievalInputCasRequest – takes no arguments
- 1238 • retrievalInputCasResponse – returns a CAS
- 1239 • getNextCasBatchRequest – takes two arguments, an integer that specifies the maximum number of CASes to return and an integer which specifies the maximum number of milliseconds to wait
- 1240
- 1241 • getNextCasBatchResponse – returns an object with three fields: a list of zero or more CASes, a Boolean indicating whether any CASes remain to be retrieved, and an integer indicating the estimated number of remaining CASes (-1 if not known).
- 1242
- 1243

1244

1245 For Flow Controllers:

- 1246 • addAvailableAnalyticsRequest – takes one argument, a Map from String keys to PE Metadata objects.
- 1247 • addAvailableAnalyticsResponse – returns nothing
- 1248 • removeAvailableAnalyticsRequest – takes one argument, a collection of one or more String keys
- 1249 • removeAvailableAnalyticsResponse – returns nothing
- 1250 • setAggregateMetadataRequest – takes one argument – a ProcessingElementMetadata
- 1251 • setAggregateMetadataResponse – returns nothing
- 1252 • getNextDestinationsRequest – takes one argument, a CAS
- 1253 • getNextDestinationsResponse – returns a Step object
- 1254 • continueOnFailureRequest – takes three arguments, a CAS, a String key, and a UimaException
- 1255 • continueOnFailureResponse – returns a Boolean

1256

1257 **4.7.1.3 Port Types and Operations**

1258 A *port type* is a collection of *operations*, where each operation is an action that can be performed by the
1259 service. We define a separate port type for each of the three interfaces defined in Section 4.4 Abstract
1260 Interfaces.

1261

1262 The port types and their operations defined by the UIMA WSDL definition are as follows. Each operation
1263 refers to its input and output message, defined in the previous section. Operations also have fault
1264 messages, returned in the case of an error.

- 1265
- 1266 • **Analyzer Port Type**
- 1267 • getMetadata
 - 1268 • setConfigurationParameters
 - 1269 • processCas
 - 1270 • processCasBatch

- 1271
- 1272 • **CasMultiplier Port Type**
- 1273 • getMetadata
 - 1274 • setConfigurationParameters
 - 1275 • inputCas
 - 1276 • getNextCas
 - 1277 • retrieveInputCas
 - 1278 • getNextCasBatch

- 1279
- 1280 **FlowController Port Type**
- 1281 • getMetadata
 - 1282 • setConfigurationParameters
 - 1283 • addAvailableAnalytics
 - 1284 • removeAvailableAnalytics
 - 1285 • setAggregateMetadata
 - 1286 • getNextDestinations
 - 1287 • continueOnFailure

1288 4.7.1.4 SOAP Bindings

1289 For each port type, we define a binding to the SOAP protocol. There are a few configuration choices to
1290 be made:

- 1291
- 1292 In `<wsdlsoap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>`:
- 1293 • The style attribute defines that our operation is an RPC, meaning that our XML messages contain
1294 parameters and return values. The alternative is "document" style, which is used for services that
1295 logically send and receive XML documents without a parameter structure. This has an effect on
1296 how the body of the SOAP message is constructed.
 - 1297 • The transport operation defines that this binding uses the HTTP protocol (the SOAP spec allows
1298 other protocols, such as FTP or SMTP, but HTTP is by far the most common)

1299 For each parameter (message part) in each abstract operation, we have a `<wsdlsoap:body use="literal"/>`
1300 element:

- 1301 • The use of the `<wsdlsoap:body>` tag indicates that this parameter is sent in the body of the SOAP
1302 message. Alternatively we could use `<wsdlsoap:header>` to choose to send parameters in the
1303 SOAP header. This is an arbitrary choice, but a good rule of thumb is that the data being
1304 processed by the service should be sent in the body, and "control information" (i.e., *how* the
1305 message should be processed) can be sent in the header.
- 1306 • The `use="literal"` attribute states that the content of the message must *exactly* conform to the
1307 XML Schema defined earlier in the WSDL definitions. The other option is "encoded", which treats
1308 the XML Schema as an abstract type definition and applies SOAP encoding rules to determine
1309 the exact XML syntax of the messages. The "encoded" style makes more sense if you are
1310 starting from an abstract object model and you want to let the SOAP rules determine your XML
1311 syntax. In our case, we already know what XML syntax we want (e.g., XML), so the "literal" style
1312 is more appropriate.

1313

1314 **4.7.2 Delta Responses**

1315 If an Analytic makes only a small number of changes to its input CAS, it will be more efficient if the service
1316 response specifies the “deltas” rather than repeating the entire CAS. UIMA supports this by using the
1317 XMI standard way to specify differences between object graphs [XMI1]. An example of such a delta
1318 response is given in the next section.

1319 **4.7.3 Formal Specification**

1320 A *UIMA SOAP Service* must conform to the WSDL document given in Section C.6 and must implement at
1321 least one of the portTypes and corresponding SOAP bindings defined in that WSDL document, as defined
1322 in [WSDL1] and [SOAP1].

1323

1324 A *UIMA Analyzer SOAP Service* must implement the Analyzer portType and the AnalyzerSoapBinding.

1325

1326 A *UIMA CAS Multiplier SOAP Service* must implement the CasMultiplier portType and the
1327 CasMultiplierSoapBinding.

1328

1329

A. Acknowledgements

1330 The following individuals have participated in the creation of this specification and are gratefully
1331 acknowledged:

1332 **Participants:**

1333 Eric Nyberg, Carnegie Mellon Univeristy

1334 Carl Mattocks, CheckMi

1335 Alex Rankov, EMC Corporation

1336 David Ferrucci, IBM

1337 Thilo Goetz, IBM

1338 Thomas Hampp-Bahnmueller, IBM

1339 Adam Lally, IBM

1340 Clifford Thompson, Individual

1341 Karin Verspoor, University of Colorado Denver

1342 Christopher Chute, Mayo Clinic College of Medicine

1343 Vinod Kaggal, Mayo Clinic College of Medicine

1344 Adrian Miley, Miley Watts LLP

1345 Loretta Auvil, National Center for Supercomputing Applications

1346 Duane Sears Smith, National Center for Supercomputing Applications

1347 Pascal Coupet, Temis

1348 Tim Miller, Thomson

1349 Yoshinobu Kano, Tsujii Laboratory, The University of Tokyo

1350 Ngan Nguyen, Tsujii Laboratory, The University of Tokyo

1351 Scott Piao, University of Manchester

1352 Hamish Cunningam, University of Sheffield

1353 Ian Roberts, University of Sheffield

1354

1355

1356 B. Examples (Not Normative)

1357 B.1 XMI CAS Example

1358 This section describes how the CAS is represented in XMI, by way of an example. This is not normative.
1359 The exact specification for XMI is defined by the OMG XMI standard [XMI1].

1360 B.1.1 XMI Tag

1361 The outermost tag is typically `<xmi:XMI>` (this is just a convention; the XMI spec allows this tag to be
1362 arbitrary). The outermost tag must, however, include an XMI version number and XML namespace
1363 attribute:

```
1364  
1365     <xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI">  
1366     <!-- CAS Contents here -->  
1367     </xmi:XMI>
```

1368
1369 XML namespaces [XML1] are used throughout. The xmi namespace prefix is typically used to identify
1370 elements and attributes that are defined by the XMI specification.

1371
1372 The XMI document will also define one namespace prefix for each CAS namespace, as described in the
1373 next section.

1374

1375 B.1.2 Objects

1376 Each *Object* in the CAS is represented as an XML element. The name of the element is the name of the
1377 object's *class*. The XML namespace of the element identifies the *package* that contains that *class*.

1378

1379 For example consider the following XMI document:

```
1380     <xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"  
1381     xmlns:myorg="http://org/myorg.ecore">  
1382     ...  
1383     <myorg:Person xmi:id="1"/>  
1384     ...  
1385     </xmi:XMI>
```

1386

1387 This XMI document contains an object whose class is named *Person*. The *Person* class is in the
1388 package with URI `http://org/myorg.ecore`. Note that the use of the `http` scheme is a common convention,
1389 and does not imply any HTTP communication. The `.ecore` suffix is due to the fact that the recommended
1390 type system definition for a package is an ECore model.

1391

1392 Note that the order in which Objects are listed in the XMI is not important, and components that process
1393 XMI are not required to maintain this order.

1394

1395 The xmi:id attribute can be used to refer to an object from elsewhere in the XMI document. It is not
1396 required if the object is never referenced. If an xmi:id is provided, it must be unique among all xmi:ids on
1397 all objects in this CAS.

1398

1399 All namespace prefixes (e.g., myorg) in this example must be bound to URIs using the
1400 "xmlns..." attribute, as defined by the XML namespaces specification [XMLS1].

1401

1402 **B.1.3 Attributes (Primitive Features)**

1403 *Attributes* (that is, *features* whose values are of primitive types, for example, strings, integers and other
1404 numeric types – see Base Type System for details) can be mapped either to XML attributes or XML
1405 elements.

1406

1407 For example, an *object* of *class* Person, with slots:

1408

1409 begin = 14

1410 end = 25

1411 name = "Fred Center"

1412

1413 could be mapped to the attribute serialization as follows:

1414

```
1415 <xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"  
1416     xmlns:myorg="http://org/myorg.ecore">  
1417     ...  
1418     <myorg:Person xmi:id="1" begin="14" end="25" name="Fred Center"/>  
1419     ...  
1420 </xmi:XMI>
```

1421

1422 or alternatively to an element serialization as follows:

1423

```
1424 <xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"  
1425     xmlns:myorg="http://org/myorg.ecore">  
1426     ...  
1427     <myorg:Person xmi:id="1">  
1428         <begin>14</begin>  
1429         <end>25</end>  
1430         <name>Fred Center</name>  
1431     </myorg:Person>  
1432     ...  
1433 </xmi:XMI>
```

1434

1435 UIMA framework components that process XMI are required to support both. Mixing the two styles is
1436 allowed; some *features* can be represented as attributes and others as elements.

1437 B.1.4 References (Object-Valued Features)

1438 *Features* that are references to other *objects* are serialized as ID references.

1439

1440 If we add to the previous CAS example an Object of Class Organization, with *feature* myCEO that is a
1441 reference to the Person object, the serialization would be:

1442

```
1443 <xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"  
1444     xmlns:myorg="http://org/myorg.ecore">  
1445     ...  
1446     <myorg:Person xmi:id="1" begin="14" end="25" name="Fred Center"/>  
1447     <myorg:Organization xmi:id="2" myCEO="1"/>  
1448     ...  
1449 </xmi:XMI>
```

1450

1451 As with primitive-valued *features*, it is permitted to use an element rather than an attribute, and UIMA
1452 framework components that process XMI are required to support both representations. However, the XMI
1453 spec defines a slightly different syntax for this as is illustrated in this example:

1454

```
1455 <myorg:Organization xmi:id="2">  
1456   <myCEO href="#1"/>  
1457 </myorg:Organization>
```

1458

1459 Note that in the attribute representation, a reference *feature* is indistinguishable from an integer-valued
1460 *feature*, so the meaning cannot be determined without prior knowledge of the type system. The element
1461 representation is unambiguous.

1462 B.1.5 Multi-valued Features

1463 *Features* may have multiple values. Consider the example where the *object* of *class* Baz has a *feature*
1464 myIntArray whose value is {2,4,6}. This can be mapped to:

1465

```
1466 <myorg:Baz xmi:id="3" myIntArray="2 4 6"/>
```

1467

1468 or:

1469

```
1470 <myorg:Baz xmi:id="3">  
1471   <myIntArray>2</myIntArray>  
1472   <myIntArray>4</myIntArray>  
1473   <myIntArray>6</myIntArray>  
1474 </myorg:Baz>
```

1475

1476 Note that string arrays whose elements contain embedded spaces must use the latter mapping.

1477

1478 Multi-valued *references* serialized in a similar way. For example a *reference* that refers to the elements
1479 with xmi:ids "13" and "42" could be serialized as:

1480
1481 `<myorg:Baz xmi:id="3" myRefFeature="13 42"/>`

1482
1483 or:

1484
1485 `<myorg:Baz xmi:id="3">`
1486 `<myRefFeature href="#13"/>`
1487 `<myRefFeature href="#42"/>`
1488 `</myorg:Baz>`

1489
1490 Note that the order in which the elements of a multi-valued feature are listed *is* meaningful, and
1491 components that process XML documents must maintain this order.
1492

1493 **B.1.6 Linking an XML Document to its Ecore Type System**

1494 The structure of a CAS is defined by a UIMA type system, which is represented by an Ecore model (see
1495 Section 4.2).

1496
1497 If the CAS Type System has been saved to an Ecore file, it is possible to store a link from an XML
1498 document to that Ecore type system. This is done using an `xsi:schemaLocation` attribute on the root XML
1499 element.

1500
1501 The `xsi:schemaLocation` attribute is a space-separated list that represents a mapping from the
1502 namespace URI (e.g., `http://org/myorg.ecore`) to the physical URI of the `.ecore` file containing the type
1503 system for that namespace. For example:

1504
1505 `xsi:schemaLocation="http://org/myorg.ecore file:/c:/typesystems/myorg.ecore"`

1506
1507 would indicate that the definition for the `org.myorg` CAS types is contained in the file
1508 `c:/typesystems/myorg.ecore`. You can specify a different mapping for each of your CAS namespaces. For
1509 details see [EMF2].

1510 **B.1.7 XML Extensions**

1511 XML defines an extension mechanism that can be used to record information that you may not want to
1512 include in your type system. This can be used for system-level data that is not part of your domain
1513 model, for example. The syntax is:

1514
1515 `<xmi:Extension extenderId="NAME">`
1516 `<!-- arbitrary content can go inside the Extension element -->`
1517 `</xmi:Extension>`

1518
1519 The `extenderId` attribute allows a particular "extender" (e.g., a UIMA framework implementation) to record
1520 metadata that's relevant only within that framework, without confusing other frameworks that may want to
1521 process the same CAS.

1522

1523 **B.2 Ecore Example**

1524 **B.2.1 An Introduction to Ecore**

1525 Ecore is well described by Budinsky et al. in the book *Eclipse Modeling Framework* [EMF2]. Some brief
1526 introduction to Ecore can be found in a chapter of that book available online [EMF3]. As a convenience to
1527 the reader we include an excerpt from that chapter:

Excerpt from Budinsky et al. *Eclipse Modeling Framework*

Ecore is a metamodel - a model for defining other models. Ecore uses very similar terminology to UML, but it is a small and simplified subset of full UML.

The following diagram illustrates the "Ecore Kernel", a simplified subset of the Ecore model.

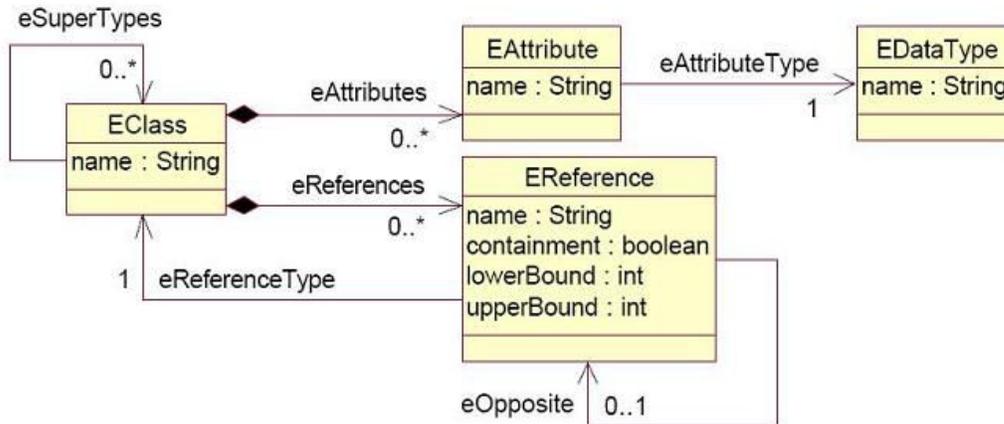


Figure 11: The Ecore Kernel

This model defines four types of objects, that is, four classes:

- **EClass** models classes themselves. Classes are identified by name and can contain a number of attributes and references. To support inheritance, a class can refer to a number of other classes as its supertypes.
- **EAttribute** models attributes, the components of an object's data. They are identified by name, and they have a type.
- **EDataType** models the types of attributes, representing primitive and object data types that are defined in Java, but not in EMF. Data types are also identified by name.
- **EReference** is used in modeling associations between classes; it models one end of the association. Like attributes, references are identified by name and have a type. However, this type must be the EClass at the other end of the association. If the association is navigable in the opposite direction, there will be another corresponding reference. A reference specifies lower and upper bounds on its multiplicity. Finally, a reference can be used to represent a stronger type of association, called containment; the reference specifies whether to enforce containment semantics.

1528

1529

1530 B.2.2 Differences between Ecore and EMOF

1531 The primary differences between Ecore and EMOF are:

- 1532 • EMOF does not use the ‘E’ prefix for its metamodel elements. For example EMOF uses the terms
1533 *Class* and *Data Type* rather than Ecore’s *EClass* and *EDataType*.
- 1534 • EMOF uses a single concept *Property* that subsumes both *EAttribute* and *EReference*.

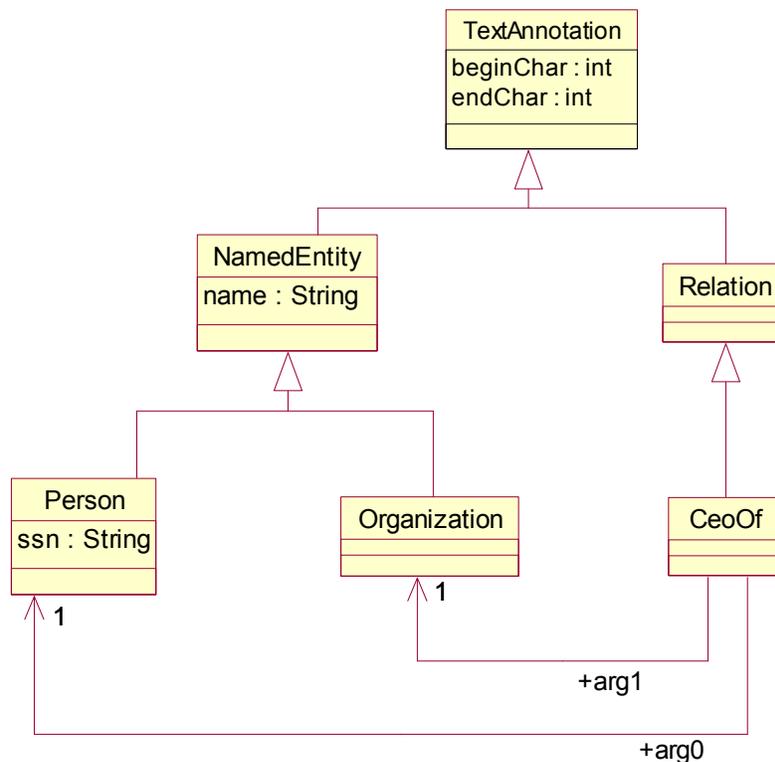
1535

1536 For a detailed mapping of Ecore terms to EMOF terms see [EcoreEMOF1].

1537 B.2.3 Example Ecore Model

1538 Figure 12 shows a simple example of an object model in UML. This model describes two types of Named
1539 Entities: Person and Organization. They may participate in a CeoOf relation (i.e., a Person is the CEO of
1540 an Organization). The NamedEntity and Relation types are subtypes of TextAnnotation (a standard UIMA
1541 base type, see 4.3), so they will inherit beginChar and endChar features that specify their location in a
1542 text document.

1543



1544

1545

Figure 12: Example UML Model

1546

1547 XMI [XMI1] is an XML format for representing object graphs. EMF tools may be used to automatically
1548 convert this to an Ecore model and generate an XML rendering of the model using XMI:

1549

```

1550 <?xml version="1.0" encoding="UTF-8"?>
1551 <ecore:EPackage xmi:version="2.0"
1552   xmlns:xmi="http://www.omg.org/XMI"
1553   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
1554   xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore"
1555   name="org" nsURI="http://org.ecore" nsPrefix="org">
  
```

```

1556     <eSubpackages name="example" nsURI="http://org/example.ecore"
1557 nsPrefix="org.example">
1558     <eClassifiers xsi:type="ecore:EClass" name="NamedEntity"
1559 eSuperTypes="ecore:EClass http://docs.oasis-
1560 open.org/uima.ecore#//base/TextAnnotation">
1561     <eStructuralFeatures xsi:type="ecore:EAttribute" name="name"
1562 eType="ecore:EDatatype http://www.eclipse.org/emf/2002/Ecore#//EString"/>
1563     </eClassifiers>
1564     <eClassifiers xsi:type="ecore:EClass" name="Relation"
1565 eSuperTypes="ecore:EClass http://docs.oasis-
1566 open.org/uima.ecore#//base/TextAnnotation"/>
1567     <eClassifiers xsi:type="ecore:EClass" name="Person"
1568 eSuperTypes="#//example/NamedEntity">
1569     <eStructuralFeatures xsi:type="ecore:EAttribute" name="ssn"
1570 eType="ecore:EDatatype http://www.eclipse.org/emf/2002/Ecore#//EString"/>
1571     </eClassifiers>
1572     <eClassifiers xsi:type="ecore:EClass" name="CeoOf"
1573 eSuperTypes="#//example/Relation">
1574     <eStructuralFeatures xsi:type="ecore:EReference" name="arg0"
1575 lowerBound="1"
1576     eType="#//example/Person"/>
1577     <eStructuralFeatures xsi:type="ecore:EReference" name="arg1"
1578 lowerBound="1"
1579     eType="#//example/Organization"/>
1580     </eClassifiers>
1581     <eClassifiers xsi:type="ecore:EClass" name="TextDocument">
1582     <eStructuralFeatures xsi:type="ecore:EAttribute" name="text"
1583 eType="ecore:EDatatype http://www.eclipse.org/emf/2002/Ecore#//EString"/>
1584     </eClassifiers>
1585     <eClassifiers xsi:type="ecore:EClass" name="Organization"
1586 eSuperTypes="#//example/NamedEntity"/>
1587     </eSubpackages>
1588 </ecore:EPackage>

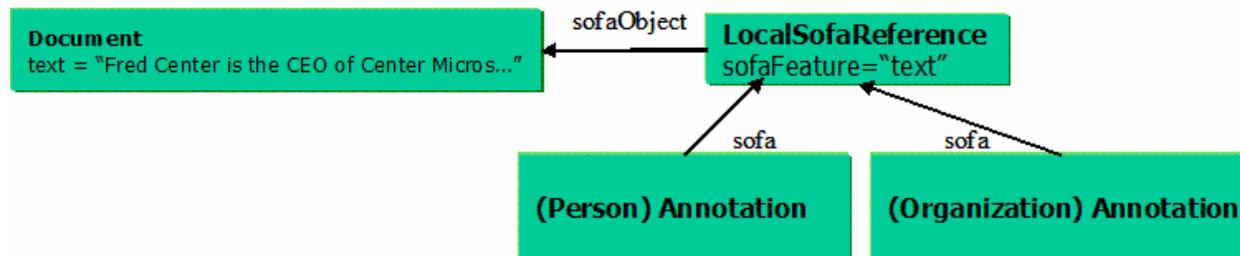
```

1589 This XML document is a valid representation of a UIMA Type System.

1592 B.3 Base Type System Examples

1593 B.3.1 Sofa Reference

1594 Figure 13 illustrates an example of an annotation referring to its subject of analysis (Sofa).



1595
1596 **Figure 13: Annotation and Subject of Analysis**

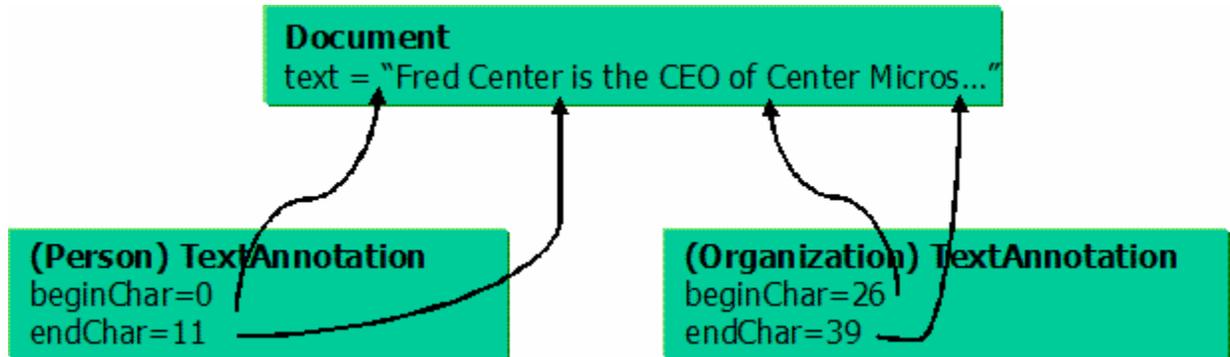
1597 The CAS contains an *object* of class Document with a *slot* text containing the string value, "Fred Center is
1598 the CEO of Center Micros."

1599

1600 Two annotations, a Person annotation and an Organization annotation, refer to that string value. The
1601 method of indicating a subrange of characters within the text string is shown in the next example. For
1602 now, note that the `LocalSofaReference` object is used to indicate which object, and *which field (slot)*
1603 *within that object*, serves as the Subject of Analysis (Sofa).
1604

1605 B.3.2 References to Regions of Sofas

1606 Figure 14 extends the previous example by showing how the `TextAnnotation` subtype of `Annotation` is
1607 used to specify a range of character offsets to which the annotation applies.



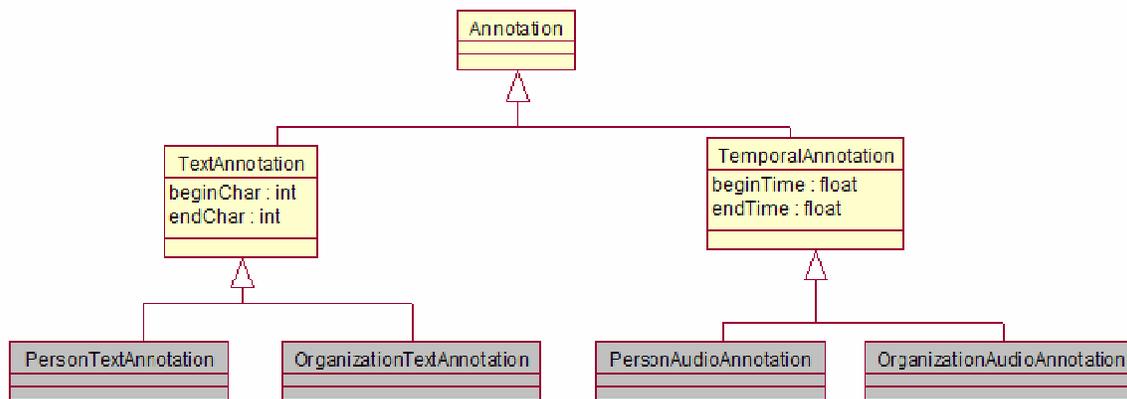
1608

1609 Figure 14: References from Annotations to Regions of the Sofa

1610 B.3.3 Options for Extending Annotation Type System

1611 The standard types in the UIMA Base Type system are very high level. Users will likely wish to extend
1612 these base types, for instance to capture the semantics of specific kinds of annotations. There are two
1613 options for implementing these extensions. The choice of the extension model for the annotation type
1614 system is up to the user and depends on application-specific needs or preferences.
1615

1616 The first option is to subclass the Annotation types, as in Figure 15. In this model, the Annotation subtype
1617 for each modality will be independently subclassed according to the annotation types found in that
1618 modality. One advantage of this approach is that all subtype classes remain subtypes of Annotation.
1619 However, a disadvantage is that types that are annotations of the same semantic class, but for different
1620 modalities, are not grouped together in the type system. We see in the figure that an annotation of a
1621 reference to a Person or an Organization would have a distinct type depending on the nature of the Sofa
1622 the reference occurred in.

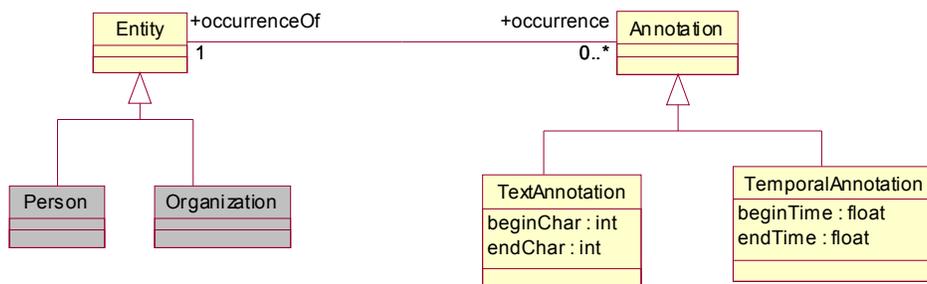


1623

1624 Figure 15: Extending the base type system through subclassing.

1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636

The second option, shown in Figure 16, is to create an Entity type that subsumes the relevant semantic classes, and associate the Annotation with the appropriate Entity type. In this model, an Annotation is viewed as an occurrence of an Entity reference in a particular modality. The advantage of this approach is that all annotations corresponding to a particular Entity type (e.g. Person or Organization), regardless of the modality they are expressed in, will have the same occurrence value and can thus be easily grouped together. It does, however, push the semantic information about the annotation into an associated type that needs to be investigated rather than being immediately available in the type of the Annotation object. In other words, it introduces a level of indirection for accessing the semantic information about the Annotation. However, an additional advantage of this approach is that it allows for multiple Annotations to be associated with a single Entity, so that for instance multiple distinct references to a person in a text can be linked to a single Entity object representing that person.



1637
1638
1639

Figure 16: Associate Annotation with Entity type

B.3.4 An Example of Annotation Model Extension

The Base Type System is intended to specify only the top-level classes for the Annotation system used in an application. Users will need to extend these classes in order to meet the particular needs of their applications. An example of how an application might extend the base type system comes from examining the redesign of IBM's Knowledge Level Types [KLT1] in terms of the standard. The current model in KLT appears in Figure 17. It uses the Annotation class, but subclasses it with its own EntityAnnotation, models coreference with a reified HasOccurrence link, and captures provenance through a *componentId* attribute.

1648
1649
1650

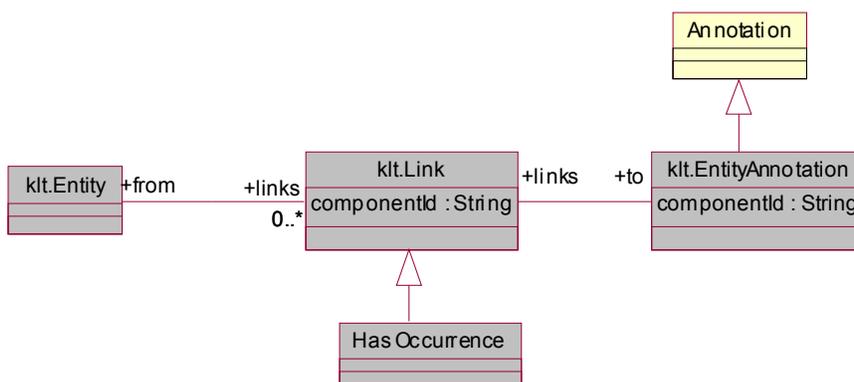
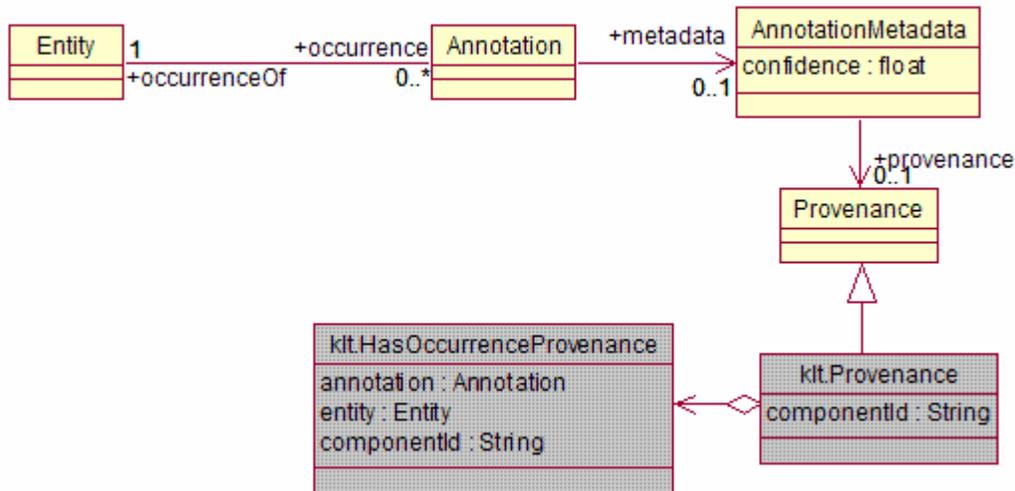


Figure 17: IBM's Knowledge Level Types

Using the standard base type system, this type system could be refactored as in Figure 18. This refactoring uses the standard definitions of Annotation and Entity. The `klt.Link` type, which was used to represent a HasOccurrence link between Entity and Annotation, is replaced by the direct

1654 occurrence/occurrenceOf features in the standard base type system. Provenance on the occurrence link
 1655 is captured using a subclass of the Provenance type.

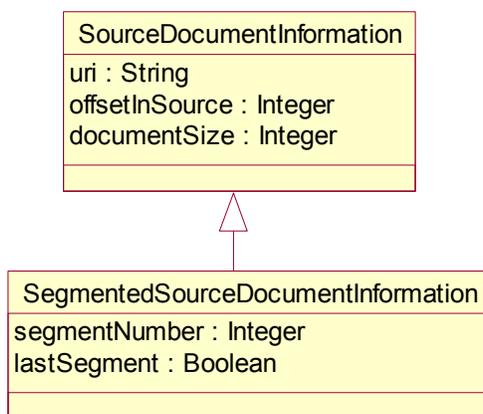


1656
 1657
 1658

Figure 18: Refactoring of KLT using the standard base type system.

1659 B.3.5 Example Extension of Source Document Information

1660 If an application needs to process multiple segments of an artifact and later merger the results, then
 1661 additional offset information may also be needed on each segment. While not a standard part of the
 1662 specification, a representative extension to the `SourceDocumentInformation` type to capture such
 1663 information is shown in Figure 19. This `SegmentedSourceDocumentInformation` type adds features
 1664 to track information about the segment of the source document the CAS corresponds to. Specifically, it
 1665 adds an Integer `segmentNumber` to capture the segment number of this segment, and a Boolean
 1666 `lastSegment` that is true when this segment is the last segment derived from the source document.



1667
 1668
 1669

Figure 19: Segmented Source Document Information UML

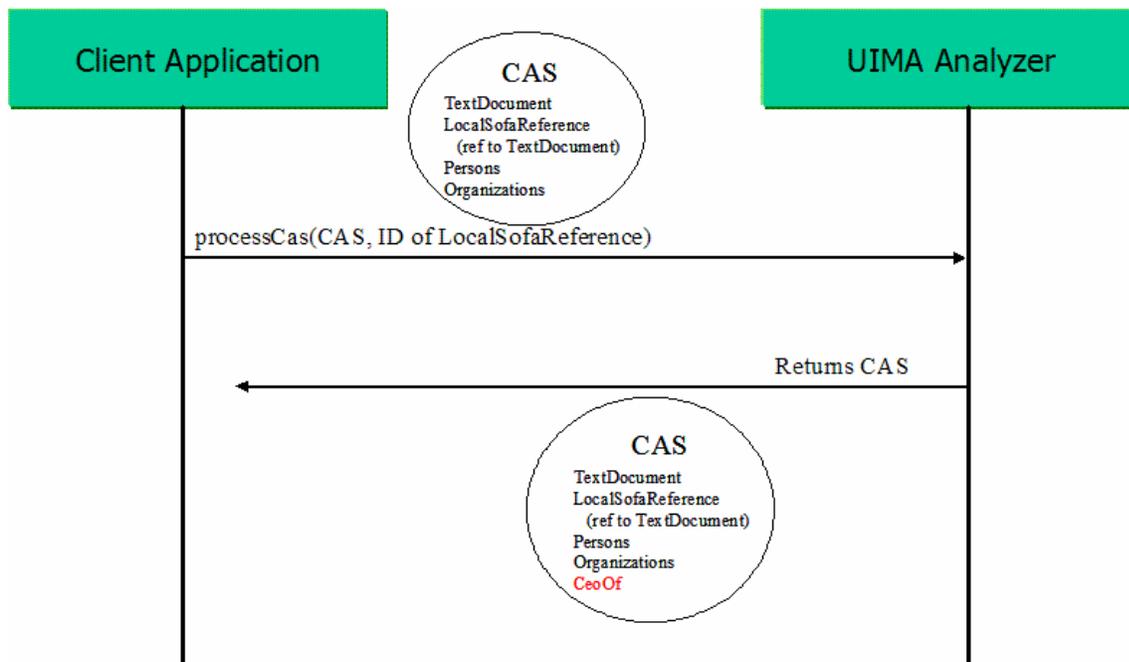
1670 B.4 Abstract Interfaces Examples

1671 B.4.1 Analyzer Example

1672 The sequence diagram in Figure 20 illustrates how a client interacts with a UIMA Analyzer service. In this
1673 example the Analyzer is a “CEO Relation Detector,” which given a text document with Person and
1674 Organization annotations, can find occurrences of CeoOf relationships between them.

1675
1676 The example shows that the client calls the `processCas(cas, sofas)` operation. The first argument is
1677 the CAS to be processed (in XMI format). It contains a `TextDocument`, a `LocalSofaReference` (see
1678 Section 4.3.2.1) that points to a text field in that `TextDocument`, and `Person` and `Organization`
1679 annotations that annotate regions in the `TextDocument`. The second argument is the `xmi:id` of the
1680 `LocalSofaReference` object, indicating that this object should be considered the subject of analysis (Sofa)
1681 for this operation.

1682
1683 The response from the `processCas` operation is a CAS (in XMI format), which in addition to the objects in
1684 the input CAS, also contains `CeoOf` annotations.



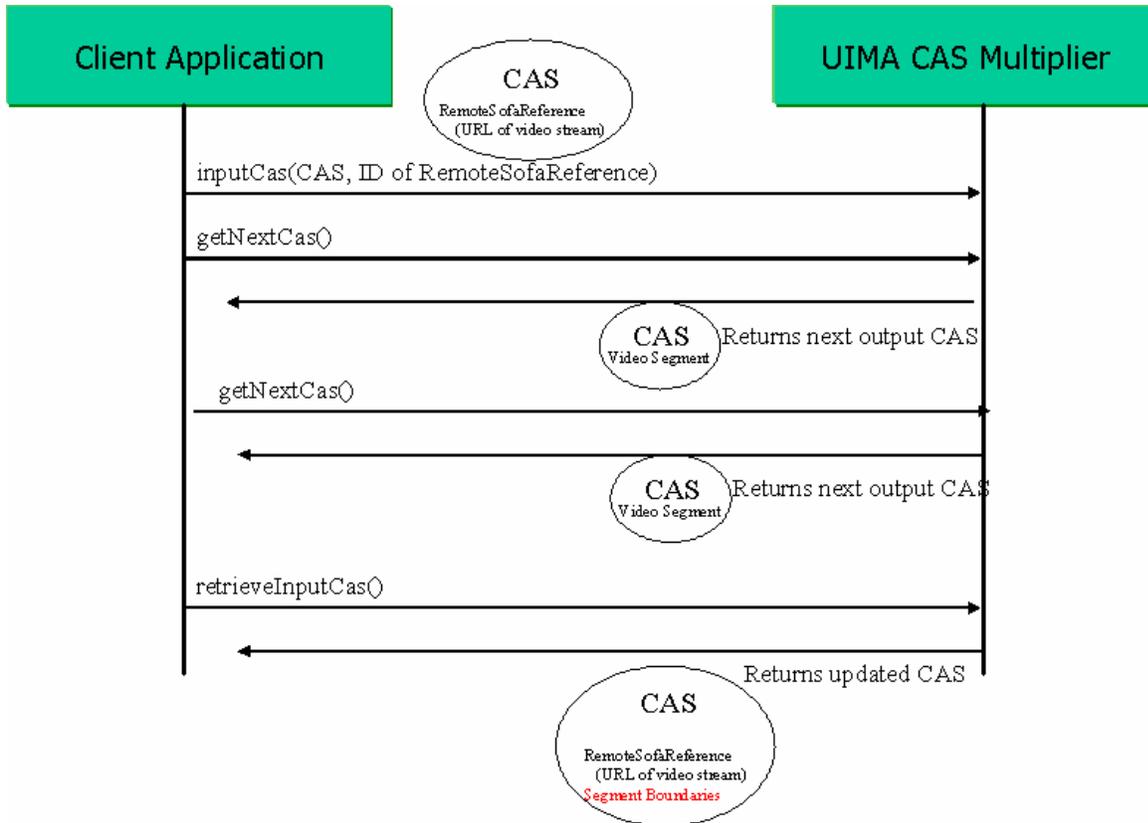
1685
1686 **Figure 20: Analyzer Sequence Diagram**

1687 B.4.2 CAS Multiplier Example

1688 The sequence diagram in Figure 21 illustrates how a client interacts with a UIMA CAS Multiplier service.
1689 In this case the CAS Multiplier is a Video Segmenter, which given a video stream divides it into individual
1690 segments.

1691
1692 The client first calls the `inputCas(cas, sofas)` operation. The first argument is a CAS containing a
1693 reference to the video stream to analyze. Typically a large artifact such as a video stream is represented
1694 in the CAS as a reference (using the `RemoteSofaReference` base type introduced in section 4.3.2.1),
1695 rather than included directly in the CAS as is typically done with a text document. The second argument
1696 to `inputCas` is the `xmi:id` of the `RemoteSofaReference` object, so that the service knows that this is the
1697 subject of analysis for this operation.

1698
 1699 The client then calls the `getNextCas` operation. This returns a CAS containing the data for the first
 1700 segment (or possibly, a reference to it). The client repeatedly calls `getNextCas` to obtain each
 1701 successive segment. Eventually, `getNextCas` returns null to indicate there are no more segments.
 1702
 1703 Finally, the client calls the `retrieveInputCas` operation. This returns the original CAS, with additional
 1704 information added. In this example, the Video Segmenter adds information to the original CAS indicating
 1705 at what time offsets each of the segment boundaries were detected. Any other information from the
 1706 individual segment CASes could also be merged back into the original CAS.
 1707



1708
 1709 **Figure 21: CAS Multiplier Sequence Diagram**
 1710

1711 B.5 Behavioral Metadata Examples

1712 For each of the Behavioral Metadata Elements (analyzes, required inputs, optional inputs, creates,
 1713 modifies, and deletes), there will be a corresponding XML element. For each element a list of type
 1714 names is declared.

1715
 1716 To address some common situations where an analytic operates on a *view* (a collection of objects all
 1717 referring to the same subject of analysis), we also provide a simple way for behavioral metadata to refer
 1718 to views.

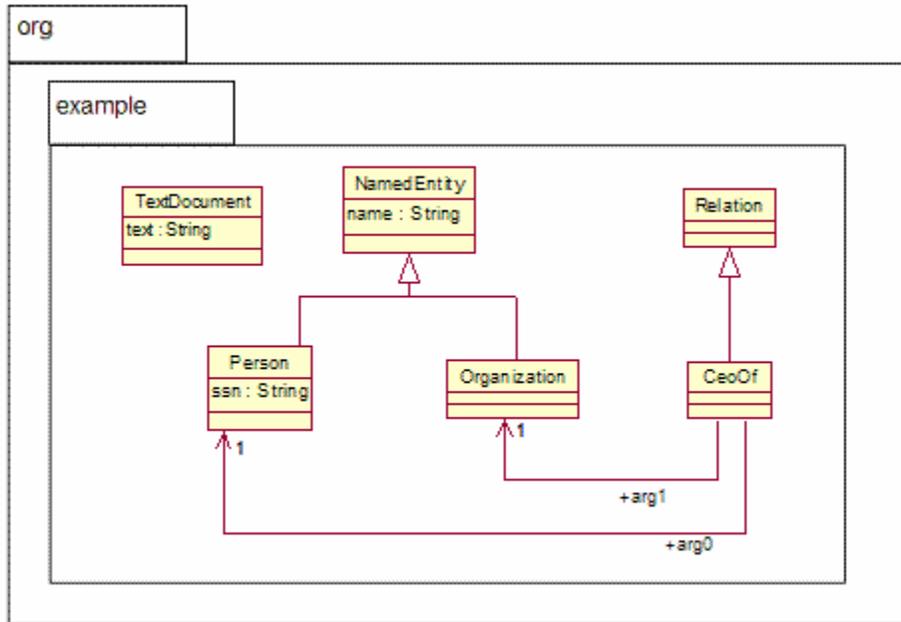
1719 B.5.1 Type Naming Conventions

1720 In the XML behavioral metadata, type names are represented in the same way as in Ecore and XML.

1721
1722
1723
1724
1725
1726
1727
1728
1729

In UML (and Ecore), a *Package* is a collection of classes and/or other packages. All classes must be contained in a package.

Figure 1 is a UML diagram of an example type system. It depicts a Package “org” containing a Package “example” containing several classes.



1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742

Figure 22: Example Type System UML Model

In the Ecore model, each package is assigned (by the developer) three identifiers: a *name*, a *namespace URI*, and a *namespace prefix*. The *name* is a simple string that must be unique within the containing package (top-level package names must be globally unique). The namespace URI and namespace prefix are standard concepts in the XML namespaces spec [2] are used to refer to that package in XML, including the behavioral metadata as well as the XMI CAS. An example is given below.

Figure 23 shows the relevant parts of the Ecore definition for this type system. Some details have been omitted (marked with an ellipsis) to show only the parts where packages and namespaces are concerned, and only a subset of the classes in the diagram are shown.

```

<ecore:EPackage ... name="org"
  nsURI="http://docs.oasis-open.org/uima/org.ecore"
  nsPrefix="org">

  <eSubpackages name="example" nsURI="http://docs.oasis-
open.org/uima/org/example.ecore"
  nsPrefix="org.example">
    <eClassifiers xsi:type="ecore:EClass" name="NamedEntity">
      ...
    </eClassifiers>
    <eClassifiers xsi:type="ecore:EClass" name="Person"
eSuperTypes="#//example/NamedEntity"/>

```

Figure 23: Partial Ecore Representation of Example Type System

1743

1744

1745

1746 In this example, the namespace URI for the nested “example” project is `http://docs.oasis-`
1747 `open.org/uima/org/example.ecore`¹, and the corresponding prefix is `org.example`. It is
1748 important to note that the URI and prefix are arbitrarily determined by the type system developer and
1749 there is no required mapping from the package names “org” and “example” to the URI and prefix. In the
1750 above example, the namespace prefix could have been set to “foo” and it would be completely valid.

1751

1752 Now, to refer to a type name within the behavioral metadata XML, we use the namespace URI and prefix
1753 in the normal XML namespaces way, for example:

1754

```

1755 <behavioralMetadata xmlns:org.example="http://docs.oasis-
1756 open.org/uima/org/example.ecore">
1757   ...
1758   <type name="org.example:Person"/>
1759   ...
1760 </behavioralMetadata>

```

1761

1762 The “xmlns” attribute declares that the prefix “org.example” is bound to the URI `http://docs.oasis-`
1763 `open.org/uima/org/example.ecore`. Then, each time we want to refer to a type in that package, we use
1764 the prefix “org.example.”

1765

¹ The use of the “http” scheme is a common XML namespace convention and does not imply that any actual http communication is occurring.

1766 Technically, the XML document does not have to use the same namespace prefix as what is in the Ecore
1767 model. It is only a guideline. The namespace URI is what matters. For example, the above XML is
1768 completely equivalent to the following

```
1769  
1770 <behavioralMetadata xmlns:foo="http://docs.oasis-  
1771 open.org/uima/org/example.ecore">  
1772     ...  
1773     <type name="foo:Person"/>  
1774     ...  
1775 </behavioralMetadata>
```

1776
1777 This is because the namespace URI is a globally unique identifier for the package, but the namespace
1778 prefix need only be unique within the current XML document. For more information on XML namespace
1779 syntax, see [XML1].

1780
1781 The above discussion centered on the representation of type names in XML. When specifying
1782 preconditions, postconditions, and projection conditions (see Section B.5.5), the Object Constraint
1783 Language (OCL) [OCL1] may be used. There is a different representation of type names needed within
1784 OCL expressions. Since OCL is not primarily XML-based, it does not use the XML namespace URIs or
1785 prefixes to refer to packages. Instead, OCL expressions refer directly to the simple package names
1786 separated by double colons, as in “org::example::Person”. For more information see [OCL1].

1787 **B.5.2 XML Syntax for Behavioral Metadata Elements**

1788 The following example is the behavioral metadata for an analytic that analyzes a Sofa of type
1789 `TextDocument`, requires objects of type `Person`, and will inspect objects of type `Organization` if they are
1790 present. It may create objects of type `CeoOf`.

```
1791  
1792 <behavioralMetadata xmlns:org.example="http://docs.oasis-  
1793 open.org/uima/org/example.ecore" excludeReferenceClosure="true">  
1794     <analyzes>  
1795         <type name="org.example:TextDocument"/>  
1796     </analyzes>  
1797     <requiredInputs>  
1798         <type name="org.example:Person"/>  
1799     </requiredInputs>  
1800     <optionalInputs>  
1801         <type name="org.example:Organization"/>  
1802     </optionalInputs>  
1803     <creates>  
1804         <type name="org.example:CeoOf"/>  
1805     </creates>  
1806 </behavioralMetadata>
```

1807
1808 Note that the inheritance hierarchy declared in the type system is respected. So for example a CAS
1809 containing objects of type `GovernmentOfficial` and `Country` would be valid input to this analytic,
1810 assuming that the type system declared these to be subtypes of `org.example:Person` and
1811 `org.example:Place`, respectively.

1812
1813 The `excludeReferenceClosure` attribute on the Behavioral Metadata element, when set to true,
1814 indicates that objects that are referenced from optional/required inputs of this analytic will not be
1815 guaranteed to be included in the CAS passed to the analytic. This attribute defaults to false.

1816
1817 For example, assume in this example the `Person` object had an employer feature of type `Company`. With
1818 `excludeReferenceClosure` set to true, the caller of this analytic is not required to include `Company`
1819 objects in the CAS that is delivered to this analytic. If `Company` objects are filtered then the employer
1820 feature would become null. If `excludeReferenceClosure` were not set, then `Company` objects would be
1821 guaranteed to be included in the CAS.

1822 **B.5.3 Views**

1823 Behavioral Metadata may refer to a View, where a View may collect all annotations referring to a
1824 particular Sofa.

```
1825  
1826 <behavioralMetadata xmlns:org.example="http://docs.oasis-  
1827 open.org/uima/org/example.ecore">  
1828   <requiredView sofaType="org.example:TextDocument">  
1829     <requiredInputs>  
1830       <type name="org.example:Token"/>  
1831     </requiredInputs>  
1832     <creates>  
1833       <type name="org.example:Person"/>  
1834     </creates>  
1835   </requiredView>  
1836   <optionalView sofaType="org.example:RawAudio">  
1837     <requiredInputs>  
1838       <type name="org.example:SpeakerBoundary"/>  
1839     </requiredInputs>  
1840     <creates>  
1841       <type name="org.example:AudioPerson"/>  
1842     </creates>  
1843   </optionalView>  
1844 </behavioralMetadata>
```

1845
1846 This example requires a `TextDocument` Sofa and optionally accepts a `RawAudio` Sofa. It has different
1847 input and output types for the different Sofas.

1848
1849 As with an optional input, an “optional view” is one that the analytic would consider if it were present in the
1850 CAS. Views that do not satisfy the required view or optional view expressions might not be delivered to
1851 the analytic.

1852
1853 The meaning of an `optionalView` having a `requiredInput` is that a view not containing the required input
1854 types is not considered to satisfy the `optionalView` expression and might not be delivered to the analytic.

1855

1856 An analytic can also declare that it creates a View along with an associated Sofa and annotations. For
1857 example, this Analytic transcribes audio to text, and also outputs Person annotations over that text:

```
1858  
1859 <behavioralMetadata xmlns:org.example="http://docs.oasis-  
1860 open.org/uima/org/example.ecore">  
1861   <requiredView sofaType="org.example:RawAudio">  
1862     <requiredInputs>  
1863       <type name="org.example:SpeakerBoundary"/>  
1864     </requiredInputs>  
1865   </requiredView>  
1866   <createsView sofaType="org.example:TextDocument">  
1867     <creates>  
1868       <type name="org.example:Person"/>  
1869     </creates>  
1870   </createsView>  
1871 </behavioralMetadata>
```

1872 **B.5.4 Specifying Which Features Are Modified**

1873 For the “modifies” predicate we allow an additional piece of information: the names of the features that
1874 may be modified. This is primarily to support discovery. For example:

```
1875  
1876 <behavioralMetadata xmlns:org.example="http://docs.oasis-  
1877 open.org/uima/org/example.ecore">  
1878   <requiredInputs>  
1879     <type name="org.example:Person"/>  
1880   </requiredInputs>  
1881   <modifies>  
1882     <type name="org.example:Person">  
1883       <feature name="ssn"/>  
1884     </type>  
1885   </modifies>  
1886 </behavioralMetadata>
```

1887
1888 This Analytic inputs `Person` objects and updates their `ssn` features.
1889

1890 **B.5.5 Specifying Preconditions, Postconditions, and Projection Conditions**

1891 Although we expect it to be rare, analytic developers may declare preconditions, postconditions, and
1892 projection conditions directly. The syntax for this is straightforward:

```
1893  
1894 <behavioralMetadata>  
1895   <precondition language="OCL"  
1896     expression="exists(s | s.oclKindOf(org::example::Sofa) and  
1897     s.mimeTypeMajor = 'audio')"/>  
1898   <postcondition language="OCL"
```

```
1899     expr="exists (p | p.oclKindOf(org::example::Sofa) and s.mimeTypeMajor =
1900 'text')"/>
1901     <projectionCondition language="OCL"
1902     expr=" select (p | p.oclKindOf(org::example::NamedEntity)) "/>
1903 </behavioralMetadata>
```

1904
1905 UIMA does not define what language must be used for expression these conditions. OCL is just one
1906 example.

1907
1908 Preconditions and postconditions are expressions that evaluate to a Boolean value. Projection conditions
1909 are expressions that evaluate to a collection of objects.

1910
1911 Behavioral Metadata can include these conditions as well as the other elements (analyzes,
1912 requiredInputs, etc.). In that case, the overall precondition and postcondition of the analytic are a
1913 combination of the user-specified conditions and the conditions derived from the other behavioral
1914 metadata elements as described in the next section. (For precondition and postcondition it is a
1915 conjunction; for projection condition it is a union.)

1916

1917 **B.6 Processing Element Metadata Example**

1918 The following XML fragment is an example of Processing Element Metadata for a “CeoOf Relation
1919 Detector” analytic.

```
1920 <pemd:ProcessingElementMetadata xmi:version="2.0"
1921 xmlns:xmi="http://www.omg.org/XMI" xmlns:pemd="http://docs.oasis-
1922 open.org/uima/pemetadata.ecore">
1923   <identification
1924     symbolicName="org.oasis-open.uima.example.CeoRelationAnnotator"
1925     name="Ceo Relation Annotator"
1926     description="Detects CeoOf relationships between Persons and
1927 Organizations in a text document."
1928     vendor="OASIS"
1929     version="1.0.0"/>
1930
1931   <configurationParameter
1932     name="PatternFile"
1933     description="Location of external file containing patterns that
1934 indicate a CeoOf relation in text."
1935     type="ResourceURL">
1936     <defaultValue>myResources/ceoPatterns.dat</defaultValue>
1937   </configurationParameter>
1938
1939   <typeSystem
1940     reference="http://docs.oasis-
1941 open.org/uima/types/exampleTypeSystem.ecore"/>
1942
1943   <behavioralMetadata>
1944     <analyzes>
1945       <type name="org.example:Document"/>
1946     </analyzes>
1947     <requiredInputs>
1948       <type name="org.example:Person"/>
1949       <type name="org.example:Organization"/>
1950     </requiredInputs>
```

```

1951     <creates>
1952         <type name="org.example:CeoOf"/>
1953     </creates>
1954 </behavioralMetadata>
1955
1956     <extension extenderId="org.apache.uima">
1957         ...
1958     </extension>
1959 </pemd:ProcessingElementMetadata>

```

1960 B.7 SOAP Service Example

1961 Returning to our example of the CEO Relation Detector analytic, this section gives examples of SOAP
 1962 messages used to send a CAS to and from the analytic.

1963

1964 The processCas request message is shown here:

```

1965 <soapenv:Envelope...>
1966   <soapenv:Body>
1967     <processCas xmlns="">
1968       <cas xmi:version="2.0" ... >
1969         <org.example:Document xmi:id="1"
1970           text="Fred Center is the CEO of Center Micros."/>
1971         <base:LocalSofaReference xmi:id="2" sofaObject="1"
1972           sofaFeature="text"/>
1973         <org.example:Person xmi:id="3" sofa="2" begin="0" end="11"/>
1974         <org.example:Organization xmi:id="4" sofa="2" begin="26" end="39"/>
1975       </cas>
1976       <sofas objects="1"/>
1977     </processCas>
1978   </soapenv:Body>
1979 </soapenv:Envelope>

```

1980 This message is simply an XMI CAS wrapped in an appropriate SOAP envelope, indicating which
 1981 operation is being invoked (processCas).

1982

1983 The processCas response message returned from the service is shown here:

1984

```

1985 <soapenv:Envelope...>
1986   <soapenv:Body>
1987     <processCas xmlns="">
1988       <cas xmi:version="2.0" ... >
1989         <org.example:Document xmi:id="1"
1990           text="Fred Center is the CEO of Center Micros."/>
1991         <base:SofaReference xmi:id="2" sofaObject="1" sofaFeature="text"/>
1992         <org.example:Person xmi:id="3" sofa="2" begin="0" end="11"/>
1993         <org.example:Organization xmi:id="4" sofa="2" begin="26" end="39"/>
1994         <org.example:CeoOf xmi:id="5" sofa="2" begin="0" end="31" arg0="3"
1995         arg1="4"/>
1996       </cas>
1997     </processCas>
1998   </soapenv:Body>
1999 </soapenv:Envelope>

```

2000 Again this is just an XMI CAS wrapped in a SOAP envelope. Note that the “CeoOf” object has been
 2001 added to the CAS.

2002

2003 Alternatively, the service could have responded with a “delta” using the XMI differences language. Here
2004 is an example:

```
2005 <soapenv:Envelope...>  
2006   <soapenv:Body>  
2007     <processCas xmlns="">  
2008       <cas xmi:version="2.0" ... >  
2009         <xmi:Difference>  
2010           <target href="input.xmi"/>  
2011           <xmi:Add addition="5">  
2012             </xmi:Difference>  
2013             <org.example:CeoOf xmi:id="5" sofa="2" begin="0" end="31" arg0="3"  
2014 arg1="4"/>  
2015           </cas>  
2016         </processCas>  
2017       </soapenv:Body>  
2018     </soapenv:Envelope>
```

2019
2020 Note that the `target` element is defined in the XMI specification to hold an href to the original XMI file to
2021 which these differences will get applied. In UIMA we don't really have a URI for that - it is just the input to
2022 the Process CAS Request. The example conventionally uses `input.xmi` for this URI.

2023

C. Formal Specification Artifacts

2024 This section includes artifacts such as Ecore models and XML Schemata, which formally define elements
2025 of the UIMA specification.

2026 C.1 XMI XML Schema

2027 This XML schema is defined by the XMI specification [XMI1] and repeated here for completeness:

2028

2029 `<?xml version="1.0" encoding="UTF-8"?>`

2030 `<xsd:schema xmlns:xmi="http://www.omg.org/XMI"`

2031 `xmlns:xsd="http://www.w3.org/2001/XMLSchema"`

2032 `targetNamespace="http://www.omg.org/XMI">`

2033 `<xsd:attribute name="id" type="xsd:ID"/>`

2034 `<xsd:attributeGroup name="IdentityAttribs">`

2035 `<xsd:attribute form="qualified" name="label" type="xsd:string"`

2036 `use="optional"/>`

2037 `<xsd:attribute form="qualified" name="uuid" type="xsd:string"`

2038 `use="optional"/>`

2039 `</xsd:attributeGroup>`

2040 `<xsd:attributeGroup name="LinkAttribs">`

2041 `<xsd:attribute name="href" type="xsd:string" use="optional"/>`

2042 `<xsd:attribute form="qualified" name="idref" type="xsd:IDREF"`

2043 `use="optional"/>`

2044 `</xsd:attributeGroup>`

2045 `<xsd:attributeGroup name="ObjectAttribs">`

2046 `<xsd:attributeGroup ref="xmi:IdentityAttribs"/>`

2047 `<xsd:attributeGroup ref="xmi:LinkAttribs"/>`

2048 `<xsd:attribute fixed="2.0" form="qualified" name="version"`

2049 `type="xsd:string" use="optional"/>`

2050 `<xsd:attribute form="qualified" name="type" type="xsd:QName"`

2051 `use="optional"/>`

2052 `</xsd:attributeGroup>`

2053 `<xsd:complexType name="XMI">`

2054 `<xsd:choice maxOccurs="unbounded" minOccurs="0">`

2055 `<xsd:any processContents="strict"/>`

2056 `</xsd:choice>`

2057 `<xsd:attributeGroup ref="xmi:IdentityAttribs"/>`

2058 `<xsd:attributeGroup ref="xmi:LinkAttribs"/>`

2059 `<xsd:attribute form="qualified" name="type" type="xsd:QName"`

2060 `use="optional"/>`

2061 `<xsd:attribute fixed="2.0" form="qualified" name="version"`

2062 `type="xsd:string" use="required"/>`

```

2063 </xsd:complexType>
2064 <xsd:element name="XMI" type="xmi:XMI"/>
2065 <xsd:complexType name="PackageReference">
2066   <xsd:choice maxOccurs="unbounded" minOccurs="0">
2067     <xsd:element name="name" type="xsd:string"/>
2068     <xsd:element name="version" type="xsd:string"/>
2069   </xsd:choice>
2070   <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
2071   <xsd:attribute name="name" type="xsd:string" use="optional"/>
2072 </xsd:complexType>
2073 <xsd:element name="PackageReference"
2074   type="xmi:PackageReference"/>
2075 <xsd:complexType name="Model">
2076   <xsd:complexContent>
2077     <xsd:extension base="xmi:PackageReference"/>
2078   </xsd:complexContent>
2079 </xsd:complexType>
2080 <xsd:element name="Model" type="xmi:Model"/>
2081 <xsd:complexType name="Import">
2082   <xsd:complexContent>
2083     <xsd:extension base="xmi:PackageReference"/>
2084   </xsd:complexContent>
2085 </xsd:complexType>
2086 <xsd:element name="Import" type="xmi:Import"/>
2087 <xsd:complexType name="MetaModel">
2088   <xsd:complexContent>
2089     <xsd:extension base="xmi:PackageReference"/>
2090   </xsd:complexContent>
2091 </xsd:complexType>
2092 <xsd:element name="MetaModel" type="xmi:MetaModel"/>
2093 <xsd:complexType name="Documentation">
2094   <xsd:choice maxOccurs="unbounded" minOccurs="0">
2095     <xsd:element name="contact" type="xsd:string"/>
2096     <xsd:element name="exporter" type="xsd:string"/>
2097     <xsd:element name="exporterVersion" type="xsd:string"/>
2098     <xsd:element name="longDescription" type="xsd:string"/>
2099     <xsd:element name="shortDescription" type="xsd:string"/>
2100     <xsd:element name="notice" type="xsd:string"/>
2101     <xsd:element name="owner" type="xsd:string"/>
2102   </xsd:choice>
2103   <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
2104   <xsd:attribute name="contact" type="xsd:string" use="optional"/>
2105   <xsd:attribute name="exporter" type="xsd:string"

```

```

2106         use="optional"/>
2107     <xsd:attribute name="exporterVersion" type="xsd:string"
2108         use="optional"/>
2109     <xsd:attribute name="longDescription" type="xsd:string"
2110         use="optional"/>
2111     <xsd:attribute name="shortDescription" type="xsd:string"
2112         use="optional"/>
2113     <xsd:attribute name="notice" type="xsd:string" use="optional"/>
2114     <xsd:attribute name="owner" type="xsd:string" use="optional"/>
2115 </xsd:complexType>
2116 <xsd:element name="Documentation" type="xmi:Documentation"/>
2117 <xsd:complexType name="Extension">
2118     <xsd:choice maxOccurs="unbounded" minOccurs="0">
2119         <xsd:any processContents="lax"/>
2120     </xsd:choice>
2121     <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
2122     <xsd:attribute name="extender" type="xsd:string"
2123         use="optional"/>
2124     <xsd:attribute name="extenderID" type="xsd:string"
2125         use="optional"/>
2126 </xsd:complexType>
2127 <xsd:element name="Extension" type="xmi:Extension"/>
2128 <xsd:complexType name="Difference">
2129     <xsd:choice maxOccurs="unbounded" minOccurs="0">
2130         <xsd:element name="target">
2131             <xsd:complexType>
2132                 <xsd:choice maxOccurs="unbounded" minOccurs="0">
2133                     <xsd:any processContents="skip"/>
2134                 </xsd:choice>
2135                 <xsd:anyAttribute processContents="skip"/>
2136             </xsd:complexType>
2137         </xsd:element>
2138         <xsd:element name="difference" type="xmi:Difference"/>
2139         <xsd:element name="container" type="xmi:Difference"/>
2140     </xsd:choice>
2141     <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
2142     <xsd:attribute name="target" type="xsd:IDREFS" use="optional"/>
2143     <xsd:attribute name="container" type="xsd:IDREFS"
2144         use="optional"/>
2145 </xsd:complexType>
2146 <xsd:element name="Difference" type="xmi:Difference"/>
2147 <xsd:complexType name="Add">
2148     <xsd:complexContent>

```

```

2149     <xsd:extension base="xmi:Difference">
2150         <xsd:attribute name="position" type="xsd:string"
2151             use="optional"/>
2152         <xsd:attribute name="addition" type="xsd:IDREFS"
2153             use="optional"/>
2154     </xsd:extension>
2155 </xsd:complexContent>
2156 </xsd:complexType>
2157 <xsd:element name="Add" type="xmi:Add"/>
2158 <xsd:complexType name="Replace">
2159     <xsd:complexContent>
2160         <xsd:extension base="xmi:Difference">
2161             <xsd:attribute name="position" type="xsd:string"
2162                 use="optional"/>
2163             <xsd:attribute name="replacement" type="xsd:IDREFS"
2164                 use="optional"/>
2165         </xsd:extension>
2166     </xsd:complexContent>
2167 </xsd:complexType>
2168 <xsd:element name="Replace" type="xmi:Replace"/>
2169 <xsd:complexType name="Delete">
2170     <xsd:complexContent>
2171         <xsd:extension base="xmi:Difference"/>
2172     </xsd:complexContent>
2173 </xsd:complexType>
2174 <xsd:element name="Delete" type="xmi:Delete"/>
2175 <xsd:complexType name="Any">
2176     <xsd:choice maxOccurs="unbounded" minOccurs="0">
2177         <xsd:any processContents="skip"/>
2178     </xsd:choice>
2179     <xsd:anyAttribute processContents="skip"/>
2180 </xsd:complexType>
2181 </xsd:schema>

```

2182 C.2 Ecore XML Schema

2183 This XML schema is defined by Ecore [EMF1] and repeated here for completeness:

```

2184 <?xml version="1.0" encoding="UTF-8"?>
2185 <xsd:schema xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore"
2186     xmlns:xmi="http://www.omg.org/XMI"
2187     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
2188     targetNamespace="http://www.eclipse.org/emf/2002/Ecore">
2189     <xsd:import namespace="http://www.omg.org/XMI" schemaLocation="XMI.xsd"/>
2190     <xsd:complexType name="EAttribute">
2191         <xsd:complexContent>

```

```

2192     <xsd:extension base="ecore:EStructuralFeature">
2193         <xsd:attribute name="id" type="xsd:boolean"/>
2194     </xsd:extension>
2195 </xsd:complexContent>
2196 </xsd:complexType>
2197 <xsd:element name="EAttribute" type="ecore:EAttribute"/>
2198 <xsd:complexType name="EAnnotation">
2199     <xsd:complexContent>
2200         <xsd:extension base="ecore:EModelElement">
2201             <xsd:choice maxOccurs="unbounded" minOccurs="0">
2202                 <xsd:element name="details" type="ecore:EStringToStringMapEntry"/>
2203                 <xsd:element name="contents" type="ecore:EObject"/>
2204                 <xsd:element name="references" type="ecore:EObject"/>
2205             </xsd:choice>
2206             <xsd:attribute name="source" type="xsd:string"/>
2207             <xsd:attribute name="references" type="xsd:string"/>
2208         </xsd:extension>
2209     </xsd:complexContent>
2210 </xsd:complexType>
2211 <xsd:element name="EAnnotation" type="ecore:EAnnotation"/>
2212 <xsd:complexType name="EClass">
2213     <xsd:complexContent>
2214         <xsd:extension base="ecore:EClassifier">
2215             <xsd:choice maxOccurs="unbounded" minOccurs="0">
2216                 <xsd:element name="eSuperTypes" type="ecore:EClass"/>
2217                 <xsd:element name="eOperations" type="ecore:EOperation"/>
2218                 <xsd:element name="eStructuralFeatures"
2219 type="ecore:EStructuralFeature"/>
2220             </xsd:choice>
2221             <xsd:attribute name="abstract" type="xsd:boolean"/>
2222             <xsd:attribute name="interface" type="xsd:boolean"/>
2223             <xsd:attribute name="eSuperTypes" type="xsd:string"/>
2224         </xsd:extension>
2225     </xsd:complexContent>
2226 </xsd:complexType>
2227 <xsd:element name="EClass" type="ecore:EClass"/>
2228 <xsd:complexType abstract="true" name="EClassifier">
2229     <xsd:complexContent>
2230         <xsd:extension base="ecore:ENamedElement">
2231             <xsd:attribute name="instanceClassName" type="xsd:string"/>
2232         </xsd:extension>
2233     </xsd:complexContent>
2234 </xsd:complexType>

```

```

2235 <xsd:element name="EClassifier" type="ecore:EClassifier"/>
2236 <xsd:complexType name="EDatatype">
2237   <xsd:complexContent>
2238     <xsd:extension base="ecore:EClassifier">
2239       <xsd:attribute name="serializable" type="xsd:boolean"/>
2240     </xsd:extension>
2241   </xsd:complexContent>
2242 </xsd:complexType>
2243 <xsd:element name="EDatatype" type="ecore:EDatatype"/>
2244 <xsd:complexType name="EEnum">
2245   <xsd:complexContent>
2246     <xsd:extension base="ecore:EDatatype">
2247       <xsd:choice maxOccurs="unbounded" minOccurs="0">
2248         <xsd:element name="eLiterals" type="ecore:EEnumLiteral"/>
2249       </xsd:choice>
2250     </xsd:extension>
2251   </xsd:complexContent>
2252 </xsd:complexType>
2253 <xsd:element name="EEnum" type="ecore:EEnum"/>
2254 <xsd:complexType name="EEnumLiteral">
2255   <xsd:complexContent>
2256     <xsd:extension base="ecore:ENamedElement">
2257       <xsd:attribute name="value" type="xsd:int"/>
2258       <xsd:attribute name="literal" type="xsd:string"/>
2259     </xsd:extension>
2260   </xsd:complexContent>
2261 </xsd:complexType>
2262 <xsd:element name="EEnumLiteral" type="ecore:EEnumLiteral"/>
2263 <xsd:complexType name="EFactory">
2264   <xsd:complexContent>
2265     <xsd:extension base="ecore:EModelElement"/>
2266   </xsd:complexContent>
2267 </xsd:complexType>
2268 <xsd:element name="EFactory" type="ecore:EFactory"/>
2269 <xsd:complexType abstract="true" name="EModelElement">
2270   <xsd:complexContent>
2271     <xsd:extension base="ecore:EObject">
2272       <xsd:choice maxOccurs="unbounded" minOccurs="0">
2273         <xsd:element name="eAnnotations" type="ecore:EAnnotation"/>
2274       </xsd:choice>
2275     </xsd:extension>
2276   </xsd:complexContent>
2277 </xsd:complexType>

```

```

2278 <xsd:element name="EModelElement" type="ecore:EModelElement"/>
2279 <xsd:complexType abstract="true" name="ENamedElement">
2280   <xsd:complexContent>
2281     <xsd:extension base="ecore:EModelElement">
2282       <xsd:attribute name="name" type="xsd:string"/>
2283     </xsd:extension>
2284   </xsd:complexContent>
2285 </xsd:complexType>
2286 <xsd:element name="ENamedElement" type="ecore:ENamedElement"/>
2287 <xsd:complexType name="EObject">
2288   <xsd:choice maxOccurs="unbounded" minOccurs="0">
2289     <xsd:element ref="xmi:Extension"/>
2290   </xsd:choice>
2291   <xsd:attribute ref="xmi:id"/>
2292   <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
2293 </xsd:complexType>
2294 <xsd:element name="EObject" type="ecore:EObject"/>
2295 <xsd:complexType name="EOperation">
2296   <xsd:complexContent>
2297     <xsd:extension base="ecore:ETypedElement">
2298       <xsd:choice maxOccurs="unbounded" minOccurs="0">
2299         <xsd:element name="eParameters" type="ecore:EParameter"/>
2300         <xsd:element name="eExceptions" type="ecore:EClassifier"/>
2301       </xsd:choice>
2302       <xsd:attribute name="eExceptions" type="xsd:string"/>
2303     </xsd:extension>
2304   </xsd:complexContent>
2305 </xsd:complexType>
2306 <xsd:element name="EOperation" type="ecore:EOperation"/>
2307 <xsd:complexType name="EPackage">
2308   <xsd:complexContent>
2309     <xsd:extension base="ecore:ENamedElement">
2310       <xsd:choice maxOccurs="unbounded" minOccurs="0">
2311         <xsd:element name="eClassifiers" type="ecore:EClassifier"/>
2312         <xsd:element name="eSubpackages" type="ecore:EPackage"/>
2313       </xsd:choice>
2314       <xsd:attribute name="nsURI" type="xsd:string"/>
2315       <xsd:attribute name="nsPrefix" type="xsd:string"/>
2316     </xsd:extension>
2317   </xsd:complexContent>
2318 </xsd:complexType>
2319 <xsd:element name="EPackage" type="ecore:EPackage"/>
2320 <xsd:complexType name="EParameter">

```

```

2321     <xsd:complexContent>
2322         <xsd:extension base="ecore:ETypedElement"/>
2323     </xsd:complexContent>
2324 </xsd:complexType>
2325 <xsd:element name="EParameter" type="ecore:EParameter"/>
2326 <xsd:complexType name="EReference">
2327     <xsd:complexContent>
2328         <xsd:extension base="ecore:EStructuralFeature">
2329             <xsd:choice maxOccurs="unbounded" minOccurs="0">
2330                 <xsd:element name="eOpposite" type="ecore:EReference"/>
2331             </xsd:choice>
2332             <xsd:attribute name="containment" type="xsd:boolean"/>
2333             <xsd:attribute name="resolveProxies" type="xsd:boolean"/>
2334             <xsd:attribute name="eOpposite" type="xsd:string"/>
2335         </xsd:extension>
2336     </xsd:complexContent>
2337 </xsd:complexType>
2338 <xsd:element name="EReference" type="ecore:EReference"/>
2339 <xsd:complexType abstract="true" name="EStructuralFeature">
2340     <xsd:complexContent>
2341         <xsd:extension base="ecore:ETypedElement">
2342             <xsd:attribute name="changeable" type="xsd:boolean"/>
2343             <xsd:attribute name="volatile" type="xsd:boolean"/>
2344             <xsd:attribute name="transient" type="xsd:boolean"/>
2345             <xsd:attribute name="defaultValueLiteral" type="xsd:string"/>
2346             <xsd:attribute name="unsettable" type="xsd:boolean"/>
2347             <xsd:attribute name="derived" type="xsd:boolean"/>
2348         </xsd:extension>
2349     </xsd:complexContent>
2350 </xsd:complexType>
2351 <xsd:element name="EStructuralFeature" type="ecore:EStructuralFeature"/>
2352 <xsd:complexType abstract="true" name="ETypedElement">
2353     <xsd:complexContent>
2354         <xsd:extension base="ecore:ENamedElement">
2355             <xsd:choice maxOccurs="unbounded" minOccurs="0">
2356                 <xsd:element name="eType" type="ecore:EClassifier"/>
2357             </xsd:choice>
2358             <xsd:attribute name="ordered" type="xsd:boolean"/>
2359             <xsd:attribute name="unique" type="xsd:boolean"/>
2360             <xsd:attribute name="lowerBound" type="xsd:int"/>
2361             <xsd:attribute name="upperBound" type="xsd:int"/>
2362             <xsd:attribute name="eType" type="xsd:string"/>
2363         </xsd:extension>

```

```

2364     </xsd:complexContent>
2365 </xsd:complexType>
2366 <xsd:element name="ETypedElement" type="ecore:ETypedElement"/>
2367 <xsd:complexType name="EStringToStringMapEntry">
2368     <xsd:choice maxOccurs="unbounded" minOccurs="0">
2369         <xsd:element ref="xmi:Extension"/>
2370     </xsd:choice>
2371     <xsd:attribute ref="xmi:id"/>
2372     <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
2373     <xsd:attribute name="key" type="xsd:string"/>
2374     <xsd:attribute name="value" type="xsd:string"/>
2375 </xsd:complexType>
2376 <xsd:element name="EStringToStringMapEntry"
2377 type="ecore:EStringToStringMapEntry"/>
2378 </xsd:schema>
2379

```

2380 C.3 Base Type System Ecore Model

2381 This Ecore model formally defines the UIMA Base Type System.

```

2382 <?xml version="1.0" encoding="UTF-8"?>
2383 <ecore:EPackage xmi:version="2.0"
2384     xmlns:xmi="http://www.omg.org/XMI"
2385     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2386     xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore" name="uima"
2387     nsURI="http://docs.oasis-open.org/uima.ecore" nsPrefix="uima">
2388     <eSubpackages name="base" nsURI="http://docs.oasis-
2389 open.org/uima/base.ecore" nsPrefix="uima.base">
2390         <eClassifiers xsi:type="ecore:EClass" name="Annotation">
2391             <eStructuralFeatures xsi:type="ecore:EReference" name="sofa"
2392 lowerBound="1"
2393             eType="#//base/SofaReference"/>
2394             <eStructuralFeatures xsi:type="ecore:EReference" name="metadata"
2395 eType="#//base/AnnotationMetadata"/>
2396             <eStructuralFeatures xsi:type="ecore:EReference" name="occurrenceOf"
2397 lowerBound="1"
2398             eType="#//base/Entity" eOpposite="#//base/Entity/occurrence"/>
2399         </eClassifiers>
2400         <eClassifiers xsi:type="ecore:EClass" name="SofaReference"
2401 abstract="true"/>
2402         <eClassifiers xsi:type="ecore:EClass" name="LocalSofaReference"
2403 eSuperTypes="#//base/SofaReference">
2404             <eStructuralFeatures xsi:type="ecore:EAttribute" name="sofaFeature"
2405 eType="ecore:EDatatype http://www.eclipse.org/emf/2002/Ecore#//EString"/>
2406             <eStructuralFeatures xsi:type="ecore:EReference" name="sofaObject"
2407 eType="ecore:EClass http://www.eclipse.org/emf/2002/Ecore#//EObject"/>
2408         </eClassifiers>
2409         <eClassifiers xsi:type="ecore:EClass" name="RemoteSofaReference"
2410 eSuperTypes="#//base/SofaReference">
2411             <eStructuralFeatures xsi:type="ecore:EAttribute" name="sofaUri"
2412 eType="ecore:EDatatype http://www.eclipse.org/emf/2002/Ecore#//EString"/>
2413         </eClassifiers>

```

```

2414     <eClassifiers xsi:type="ecore:EClass" name="TextAnnotation"
2415 eSuperTypes="#//base/Annotation">
2416     <eStructuralFeatures xsi:type="ecore:EAttribute" name="beginChar"
2417 eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EInt"/>
2418     <eStructuralFeatures xsi:type="ecore:EAttribute" name="endChar"
2419 eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EInt"/>
2420     </eClassifiers>
2421     <eClassifiers xsi:type="ecore:EClass" name="TemporalAnnotation"
2422 eSuperTypes="#//base/Annotation">
2423     <eStructuralFeatures xsi:type="ecore:EAttribute" name="beginTime"
2424 eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EFloat"/>
2425     <eStructuralFeatures xsi:type="ecore:EAttribute" name="endTime"
2426 eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EFloat"/>
2427     </eClassifiers>
2428     <eClassifiers xsi:type="ecore:EClass" name="AnnotationMetadata">
2429     <eStructuralFeatures xsi:type="ecore:EAttribute" name="confidence"
2430 eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EFloat"/>
2431     <eStructuralFeatures xsi:type="ecore:EReference" name="provenance"
2432 eType="#//base/Provenance"/>
2433     </eClassifiers>
2434     <eClassifiers xsi:type="ecore:EClass" name="Provenance"/>
2435     <eClassifiers xsi:type="ecore:EClass" name="Entity">
2436     <eStructuralFeatures xsi:type="ecore:EReference" name="occurrence"
2437 upperBound="-1"
2438     eType="#//base/Annotation"
2439 eOpposite="#//base/Annotation/occurrenceOf"/>
2440     </eClassifiers>
2441     <eClassifiers xsi:type="ecore:EClass" name="SourceDocumentInformation">
2442     <eStructuralFeatures xsi:type="ecore:EAttribute" name="uri"
2443 eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
2444     <eStructuralFeatures xsi:type="ecore:EAttribute" name="offsetInSource"
2445 eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EInt"/>
2446     <eStructuralFeatures xsi:type="ecore:EAttribute" name="documentSize"
2447 eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EInt"/>
2448     </eClassifiers>
2449     <eClassifiers xsi:type="ecore:EClass" name="AnchoredView"
2450 eSuperTypes="#//base/View">
2451     <eStructuralFeatures xsi:type="ecore:EReference" name="sofa"
2452 upperBound="-1"
2453     eType="#//base/SofaReference"/>
2454     </eClassifiers>
2455     <eClassifiers xsi:type="ecore:EClass" name="View">
2456     <eStructuralFeatures xsi:type="ecore:EReference" name="IndexRepository"
2457 lowerBound="1"/>
2458     <eStructuralFeatures xsi:type="ecore:EReference" name="member"
2459 upperBound="-1"
2460     eType="ecore:EClass
2461 http://www.eclipse.org/emf/2002/Ecore#//EObject"/>
2462     <eStructuralFeatures xsi:type="ecore:EAttribute" name="name"
2463 eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
2464     </eClassifiers>
2465     </eSubpackages>
2466 </ecore:EPackage>

```

2467 **C.4 PE Metadata and Behavioral Metadata Ecore Model**

2468 This Ecore model formally defines the UIMA Processing Element Metadata and Behavioral Metadata.

```
2469 <?xml version="1.0" encoding="UTF-8"?>
```

```

2470 <ecore:EPackage xmi:version="2.0"
2471     xmlns:xmi="http://www.omg.org/XMI"
2472     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2473     xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore" name="uima"
2474     nsURI="http://docs.oasis-open.org/uima.ecore" nsPrefix="uima">
2475     <Subpackages name="peMetadata" nsURI="http://docs.oasis-
2476     open.org/uima/peMetadata.ecore"
2477     nsPrefix="uima.peMetadata">
2478     <eClassifiers xsi:type="ecore:EClass" name="Identification">
2479     <eStructuralFeatures xsi:type="ecore:EAttribute" name="symbolicName"
2480     eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#/EString"/>
2481     <eStructuralFeatures xsi:type="ecore:EAttribute" name="name"
2482     eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#/EString"/>
2483     <eStructuralFeatures xsi:type="ecore:EAttribute" name="description"
2484     eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#/EString"/>
2485     <eStructuralFeatures xsi:type="ecore:EAttribute" name="vendor"
2486     eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#/EString"/>
2487     <eStructuralFeatures xsi:type="ecore:EAttribute" name="version"
2488     eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#/EString"/>
2489     <eStructuralFeatures xsi:type="ecore:EAttribute" name="url"
2490     eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#/EString"/>
2491     </eClassifiers>
2492     <eClassifiers xsi:type="ecore:EClass" name="ConfigurationParameter">
2493     <eStructuralFeatures xsi:type="ecore:EAttribute" name="name"
2494     eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#/EString"/>
2495     <eStructuralFeatures xsi:type="ecore:EAttribute" name="description"
2496     eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#/EString"/>
2497     <eStructuralFeatures xsi:type="ecore:EAttribute" name="type"
2498     eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#/EString"/>
2499     <eStructuralFeatures xsi:type="ecore:EAttribute" name="multiValued"
2500     eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#/EBoolean"/>
2501     <eStructuralFeatures xsi:type="ecore:EAttribute" name="mandatory"
2502     eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#/EBoolean"/>
2503     <eStructuralFeatures xsi:type="ecore:EAttribute" name="defaultValue"
2504     upperBound="-1"
2505     eType="ecore:EDataType
2506     http://www.eclipse.org/emf/2002/Ecore#/EString"/>
2507     </eClassifiers>
2508     <eClassifiers xsi:type="ecore:EClass" name="TypeSystem">
2509     <eStructuralFeatures xsi:type="ecore:EAttribute" name="reference"
2510     upperBound="-1"
2511     eType="ecore:EDataType
2512     http://www.eclipse.org/emf/2002/Ecore#/EString"/>
2513     <eStructuralFeatures xsi:type="ecore:EReference" name="package"
2514     upperBound="-1"
2515     eType="ecore:EClass
2516     http://www.eclipse.org/emf/2002/Ecore#/EPackage" containment="true"/>
2517     </eClassifiers>
2518     <eClassifiers xsi:type="ecore:EClass" name="BehavioralMetadata">
2519     <eStructuralFeatures xsi:type="ecore:EAttribute"
2520     name="excludeReferenceClosure"
2521     eType="ecore:EDataType
2522     http://www.eclipse.org/emf/2002/Ecore#/EBooleanObject"/>
2523     <eStructuralFeatures xsi:type="ecore:EReference" name="analyzes"
2524     lowerBound="1"
2525     eType="#//peMetadata/BehaviorElement" containment="true"/>
2526     <eStructuralFeatures xsi:type="ecore:EReference" name="requiredInputs"
2527     lowerBound="1"

```

```

2528         eType="#//peMetadata/BehaviorElement" containment="true"/>
2529     <eStructuralFeatures xsi:type="ecore:EReference" name="optionalInputs"
2530 lowerBound="1"
2531         eType="#//peMetadata/BehaviorElement" containment="true"/>
2532     <eStructuralFeatures xsi:type="ecore:EReference" name="creates"
2533 lowerBound="1"
2534         eType="#//peMetadata/BehaviorElement" containment="true"/>
2535     <eStructuralFeatures xsi:type="ecore:EReference" name="modifies"
2536 lowerBound="1"
2537         eType="#//peMetadata/BehaviorElement" containment="true"/>
2538     <eStructuralFeatures xsi:type="ecore:EReference" name="deletes"
2539 lowerBound="1"
2540         eType="#//peMetadata/BehaviorElement" containment="true"/>
2541     <eStructuralFeatures xsi:type="ecore:EReference" name="precondition"
2542 lowerBound="1"
2543         eType="#//peMetadata/Condition" containment="true"/>
2544     <eStructuralFeatures xsi:type="ecore:EReference" name="postcondition"
2545 lowerBound="1"
2546         eType="#//peMetadata/Condition" containment="true"/>
2547     <eStructuralFeatures xsi:type="ecore:EReference"
2548 name="projectionCondition"
2549         lowerBound="1" eType="#//peMetadata/Condition" containment="true"/>
2550     <eStructuralFeatures xsi:type="ecore:EReference" name="requiredView"
2551 upperBound="-1"
2552         eType="#//peMetadata/ViewBehavioralMetadata" containment="true"/>
2553     <eStructuralFeatures xsi:type="ecore:EReference" name="optionalView"
2554 upperBound="-1"
2555         eType="#//peMetadata/ViewBehavioralMetadata" containment="true"/>
2556     <eStructuralFeatures xsi:type="ecore:EReference" name="createsView"
2557 upperBound="-1"
2558         eType="#//peMetadata/ViewBehavioralMetadata" containment="true"/>
2559 </eClassifiers>
2560 <eClassifiers xsi:type="ecore:EClass" name="ProcessingElementMetadata">
2561     <eStructuralFeatures xsi:type="ecore:EReference"
2562 name="configurationParameter"
2563         upperBound="-1" eType="#//peMetadata/ConfigurationParameter"
2564 containment="true"/>
2565     <eStructuralFeatures xsi:type="ecore:EReference" name="identification"
2566 lowerBound="1"
2567         eType="#//peMetadata/Identification" containment="true"/>
2568     <eStructuralFeatures xsi:type="ecore:EReference" name="typeSystem"
2569 lowerBound="1"
2570         eType="#//peMetadata/TypeSystem" containment="true"/>
2571     <eStructuralFeatures xsi:type="ecore:EReference"
2572 name="behavioralMetadata" lowerBound="1"
2573         eType="#//peMetadata/BehavioralMetadata" containment="true"/>
2574     <eStructuralFeatures xsi:type="ecore:EReference" name="extension"
2575 upperBound="-1"
2576         eType="#//peMetadata/Extension" containment="true"/>
2577 </eClassifiers>
2578 <eClassifiers xsi:type="ecore:EClass" name="Extension">
2579     <eStructuralFeatures xsi:type="ecore:EAttribute" name="extenderId"
2580 eType="ecore:EDatatype http://www.eclipse.org/emf/2002/Ecore#//EString"/>
2581     <eStructuralFeatures xsi:type="ecore:EReference" name="contents"
2582 lowerBound="1"
2583         eType="ecore:EClass
2584 http://www.eclipse.org/emf/2002/Ecore#//EObject" containment="true"/>
2585 </eClassifiers>

```

```

2586     <eClassifiers xsi:type="ecore:EClass" name="BehaviorElement">
2587       <eStructuralFeatures xsi:type="ecore:EReference" name="type"
2588 upperBound="-1"
2589       eType="#//peMetadata/Type" containment="true"/>
2590     </eClassifiers>
2591     <eClassifiers xsi:type="ecore:EClass" name="Type">
2592       <eStructuralFeatures xsi:type="ecore:EAttribute" name="name"
2593 eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
2594       <eStructuralFeatures xsi:type="ecore:EAttribute" name="feature"
2595 upperBound="-1"
2596       eType="ecore:EDataType
2597 http://www.eclipse.org/emf/2002/Ecore#//EString"/>
2598     </eClassifiers>
2599     <eClassifiers xsi:type="ecore:EClass" name="Condition">
2600       <eStructuralFeatures xsi:type="ecore:EAttribute" name="language"
2601 eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
2602       <eStructuralFeatures xsi:type="ecore:EAttribute" name="expression"
2603 eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
2604     </eClassifiers>
2605     <eClassifiers xsi:type="ecore:EClass" name="ViewBehavioralMetadata"
2606 eSuperTypes="#//peMetadata/BehavioralMetadata"/>
2607   </eSubpackages>
2608 </ecore:EPackage>

```

2609 C.5 PE Metadata and Behavioral Metadata XML Schema

2610 This XML schema was generated from the Ecore model in Appendix C.4 by the Eclipse Modeling
2611 Framework tools.

```

2612 <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2613 <xsd:schema xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore"
2614 xmlns:uima.peMetadata="http://docs.oasis-open.org/uima/peMetadata.ecore"
2615 xmlns:xmi="http://www.omg.org/XMI"
2616 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
2617 targetNamespace="http://docs.oasis-open.org/uima/peMetadata.ecore">
2618   <xsd:import namespace="http://www.eclipse.org/emf/2002/Ecore"
2619 schemaLocation="ecore.xsd"/>
2620   <xsd:import namespace="http://www.omg.org/XMI"
2621 schemaLocation="../../../plugin/org.eclipse.emf.ecore/model/XMI.xsd"/>
2622   <xsd:complexType name="Identification">
2623     <xsd:choice maxOccurs="unbounded" minOccurs="0">
2624       <xsd:element ref="xmi:Extension"/>
2625     </xsd:choice>
2626     <xsd:attribute ref="xmi:id"/>
2627     <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
2628     <xsd:attribute name="symbolicName" type="xsd:string"/>
2629     <xsd:attribute name="name" type="xsd:string"/>
2630     <xsd:attribute name="description" type="xsd:string"/>
2631     <xsd:attribute name="vendor" type="xsd:string"/>
2632     <xsd:attribute name="version" type="xsd:string"/>
2633     <xsd:attribute name="url" type="xsd:string"/>
2634   </xsd:complexType>
2635   <xsd:element name="Identification" type="uima.peMetadata:Identification"/>
2636   <xsd:complexType name="ConfigurationParameter">
2637     <xsd:choice maxOccurs="unbounded" minOccurs="0">
2638       <xsd:element name="defaultValue" nillable="true" type="xsd:string"/>
2639       <xsd:element ref="xmi:Extension"/>
2640     </xsd:choice>

```

```

2641     <xsd:attribute ref="xmi:id"/>
2642     <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
2643     <xsd:attribute name="name" type="xsd:string"/>
2644     <xsd:attribute name="description" type="xsd:string"/>
2645     <xsd:attribute name="type" type="xsd:string"/>
2646     <xsd:attribute name="multiValued" type="xsd:boolean"/>
2647     <xsd:attribute name="mandatory" type="xsd:boolean"/>
2648   </xsd:complexType>
2649   <xsd:element name="ConfigurationParameter"
2650 type="uima.peMetadata:ConfigurationParameter"/>
2651   <xsd:complexType name="TypeSystem">
2652     <xsd:choice maxOccurs="unbounded" minOccurs="0">
2653       <xsd:element name="reference" nillable="true" type="xsd:string"/>
2654       <xsd:element name="package" type="ecore:EPackage"/>
2655       <xsd:element ref="xmi:Extension"/>
2656     </xsd:choice>
2657     <xsd:attribute ref="xmi:id"/>
2658     <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
2659   </xsd:complexType>
2660   <xsd:element name="TypeSystem" type="uima.peMetadata:TypeSystem"/>
2661   <xsd:complexType name="BehavioralMetadata">
2662     <xsd:choice maxOccurs="unbounded" minOccurs="0">
2663       <xsd:element name="analyzes" type="uima.peMetadata:BehaviorElement"/>
2664       <xsd:element name="requiredInputs"
2665 type="uima.peMetadata:BehaviorElement"/>
2666       <xsd:element name="optionalInputs"
2667 type="uima.peMetadata:BehaviorElement"/>
2668       <xsd:element name="creates" type="uima.peMetadata:BehaviorElement"/>
2669       <xsd:element name="modifies" type="uima.peMetadata:BehaviorElement"/>
2670       <xsd:element name="deletes" type="uima.peMetadata:BehaviorElement"/>
2671       <xsd:element name="precondition" type="uima.peMetadata:Condition"/>
2672       <xsd:element name="postcondition" type="uima.peMetadata:Condition"/>
2673       <xsd:element name="projectionCondition"
2674 type="uima.peMetadata:Condition"/>
2675       <xsd:element name="requiredView"
2676 type="uima.peMetadata:ViewBehavioralMetadata"/>
2677       <xsd:element name="optionalView"
2678 type="uima.peMetadata:ViewBehavioralMetadata"/>
2679       <xsd:element name="createsView"
2680 type="uima.peMetadata:ViewBehavioralMetadata"/>
2681       <xsd:element ref="xmi:Extension"/>
2682     </xsd:choice>
2683     <xsd:attribute ref="xmi:id"/>
2684     <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
2685     <xsd:attribute name="excludeReferenceClosure" type="xsd:boolean"/>
2686   </xsd:complexType>
2687   <xsd:element name="BehavioralMetadata"
2688 type="uima.peMetadata:BehavioralMetadata"/>
2689   <xsd:complexType name="ProcessingElementMetadata">
2690     <xsd:choice maxOccurs="unbounded" minOccurs="0">
2691       <xsd:element name="configurationParameter"
2692 type="uima.peMetadata:ConfigurationParameter"/>
2693       <xsd:element name="identification"
2694 type="uima.peMetadata:Identification"/>
2695       <xsd:element name="typeSystem" type="uima.peMetadata:TypeSystem"/>
2696       <xsd:element name="behavioralMetadata"
2697 type="uima.peMetadata:BehavioralMetadata"/>
2698       <xsd:element name="extension" type="uima.peMetadata:Extension"/>

```

```

2699     <xsd:element ref="xmi:Extension"/>
2700   </xsd:choice>
2701   <xsd:attribute ref="xmi:id"/>
2702   <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
2703 </xsd:complexType>
2704 <xsd:element name="ProcessingElementMetadata"
2705 type="uima.peMetadata:ProcessingElementMetadata"/>
2706 <xsd:complexType name="Extension">
2707   <xsd:choice maxOccurs="unbounded" minOccurs="0">
2708     <xsd:element name="contents" type="ecore:EObject"/>
2709     <xsd:element ref="xmi:Extension"/>
2710   </xsd:choice>
2711   <xsd:attribute ref="xmi:id"/>
2712   <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
2713   <xsd:attribute name="extenderId" type="xsd:string"/>
2714 </xsd:complexType>
2715 <xsd:element name="Extension" type="uima.peMetadata:Extension"/>
2716 <xsd:complexType name="BehaviorElement">
2717   <xsd:choice maxOccurs="unbounded" minOccurs="0">
2718     <xsd:element name="type" type="uima.peMetadata:Type"/>
2719     <xsd:element ref="xmi:Extension"/>
2720   </xsd:choice>
2721   <xsd:attribute ref="xmi:id"/>
2722   <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
2723 </xsd:complexType>
2724 <xsd:element name="BehaviorElement"
2725 type="uima.peMetadata:BehaviorElement"/>
2726 <xsd:complexType name="Type">
2727   <xsd:choice maxOccurs="unbounded" minOccurs="0">
2728     <xsd:element name="feature" nillable="true" type="xsd:string"/>
2729     <xsd:element ref="xmi:Extension"/>
2730   </xsd:choice>
2731   <xsd:attribute ref="xmi:id"/>
2732   <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
2733   <xsd:attribute name="name" type="xsd:string"/>
2734 </xsd:complexType>
2735 <xsd:element name="Type" type="uima.peMetadata:Type"/>
2736 <xsd:complexType name="Condition">
2737   <xsd:choice maxOccurs="unbounded" minOccurs="0">
2738     <xsd:element ref="xmi:Extension"/>
2739   </xsd:choice>
2740   <xsd:attribute ref="xmi:id"/>
2741   <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
2742   <xsd:attribute name="language" type="xsd:string"/>
2743   <xsd:attribute name="expression" type="xsd:string"/>
2744 </xsd:complexType>
2745 <xsd:element name="Condition" type="uima.peMetadata:Condition"/>
2746 <xsd:complexType name="ViewBehavioralMetadata">
2747   <xsd:complexContent>
2748     <xsd:extension base="uima.peMetadata:BehavioralMetadata"/>
2749   </xsd:complexContent>
2750 </xsd:complexType>
2751 <xsd:element name="ViewBehavioralMetadata"
2752 type="uima.peMetadata:ViewBehavioralMetadata"/>
2753 </xsd:schema>

```

2754 C.6 PE Service WSDL Definition

2755 This WSDL document formally defines a UIMA SOAP Service.

```
2756 <?xml version="1.0" encoding="UTF-8"?>
2757 <wsdl:definitions
2758   targetNamespace="http://docs.oasis-open.org/uima/peService"
2759   xmlns:service="http://docs.oasis-open.org/uima/peService"
2760   xmlns:pemd="http://docs.oasis-open.org/uima/peMetadata.ecore"
2761   xmlns:pe="http://docs.oasis-open.org/uima/pe.ecore"
2762   xmlns:wSDL="http://schemas.xmlsoap.org/wSDL/"
2763   xmlns:wSDLsoap="http://schemas.xmlsoap.org/wSDL/soap/"
2764   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
2765   xmlns:xmi="http://www.omg.org/XMI">
2766
2767   <wsdl:types>
2768     <!-- Import the PE Metadata Schema Definitions -->
2769     <xsd:import
2770       namespace="http://docs.oasis-open.org/uima/peMetadata.ecore"
2771       schemaLocation="uima.peMetadataXMI.xsd"/>
2772
2773     <!-- Import the XMI schema. -->
2774     <xsd:import namespace="http://www.omg.org/XMI"
2775       schemaLocation="XMI.xsd"/>
2776
2777     <!-- Import other type definitions used as part of the service API. -->
2778     <xsd:import
2779       namespace="http://docs.oasis-open.org/uima/pe.ecore"
2780       schemaLocation="uima.peServiceXMI.xsd"/>
2781   </wsdl:types>
2782
2783   <!-- Define the messages sent to and from the service. -->
2784
2785   <!-- Messages for all UIMA Processing Elements -->
2786   <wsdl:message name="getMetadataRequest">
2787   </wsdl:message>
2788
2789   <wsdl:message name="getMetadataResponse">
2790     <wsdl:part element="metadata"
2791       type="pemd:ProcessingElementMetadata" name="metadata"/>
2792   </wsdl:message>
2793
2794   <wsdl:message name="setConfigurationParametersRequest">
2795     <wsdl:part element="settings"
2796       type="pe:ConfigurationParameterSettings" name="settings"/>
2797   </wsdl:message>
2798
2799   <wsdl:message name="setConfigurationParametersResponse">
2800   </wsdl:message>
2801
2802   <wsdl:message name="uimaFault">
2803     <wsdl:part element="exception" type="pe:UimaException" name="exception"/>
2804   </wsdl:message>
2805
2806
2807   <!-- Messages for the Analyzer interface -->
2808
2809   <wsdl:message name="processCasRequest">
2810     <wsdl:part element="cas" type="xmi:XMI" name="cas"/>
```

```

2811     <wsdl:part element="sofas" type="pe:ObjectList" name="sofas"/>
2812 </wsdl:message>
2813
2814 <wsdl:message name="processCasResponse">
2815     <wsdl:part element="cas" type="xmi:XMI" name="cas"/>
2816 </wsdl:message>
2817
2818 <wsdl:message name="processCasBatchRequest">
2819     <wsdl:part element="casBatchInput" type="pe:CasBatchInput"
2820 name="casBatchInput"/>
2821 </wsdl:message>
2822
2823 <wsdl:message name="processCasBatchResponse">
2824     <wsdl:part element="casBatchResponse" type="pe:CasBatchResponse"
2825 name="casBatchResponse"/>
2826 </wsdl:message>
2827
2828
2829 <!-- Messages for the CasMultiplier interface -->
2830 <wsdl:message name="inputCasRequest">
2831     <wsdl:part element="cas" type="xmi:XMI" name="cas"/>
2832     <wsdl:part element="sofas" type="pe:ObjectList" name="sofas"/>
2833 </wsdl:message>
2834
2835 <wsdl:message name="inputCasResponse">
2836 </wsdl:message>
2837
2838 <wsdl:message name="getNextCasRequest">
2839 </wsdl:message>
2840
2841 <wsdl:message name="getNextCasResponse">
2842     <wsdl:part element="cas" type="xmi:XMI" name="cas"/>
2843 </wsdl:message>
2844
2845 <wsdl:message name="retrieveInputCasRequest">
2846 </wsdl:message>
2847
2848 <wsdl:message name="retrieveInputCasResponse">
2849     <wsdl:part element="cas" type="xmi:XMI" name="cas"/>
2850 </wsdl:message>
2851
2852 <wsdl:message name="getNextCasRequest">
2853     <wsdl:part element="maxCASesToReturn" type="xsd:integer"
2854 name="maxCASesToReturn"/>
2855     <wsdl:part element="timeToWait" type="xsd:integer" name="timeToWait"/>
2856 </wsdl:message>
2857
2858 <wsdl:message name="getNextCasResponse">
2859     <wsdl:part element="cas" type="xmi:XMI" name="cas"/>
2860 </wsdl:message>
2861
2862 <wsdl:message name="getNextCasBatchRequest">
2863     <wsdl:part element="maxCASesToReturn" type="xsd:integer"
2864 name="maxCASesToReturn"/>
2865     <wsdl:part element="timeToWait" type="xsd:integer" name="timeToWait"/>
2866 </wsdl:message>
2867
2868 <wsdl:message name="getNextCasBatchResponse">

```

```

2869     <wsdl:part element="reponse" type="pe:GetNextCasBatchResponse"
2870 name="response"/>
2871 </wsdl:message>
2872
2873 <!-- Messages for the FlowController interface -->
2874
2875 <wsdl:message name="addAvailableAnalyticsRequest">
2876     <wsdl:part element="analyticMetadataMap"
2877         type="pe:AnalyticMetadataMap" name="analyticMetadataMap"/>
2878 </wsdl:message>
2879
2880 <wsdl:message name="addAvailableAnalyticsResponse">
2881 </wsdl:message>
2882
2883 <wsdl:message name="removeAvailableAnalyticsRequest">
2884     <wsdl:part element="analyticKeys" type="pe:Keys"
2885         name="analyticKeys"/>
2886 </wsdl:message>
2887
2888 <wsdl:message name="removeAvailableAnalyticsResponse">
2889 </wsdl:message>
2890
2891 <wsdl:message name="setAggregateMetadataRequest">
2892     <wsdl:part element="metadata"
2893         type="pemd:ProcessingElementMetadata" name="metadata"/>
2894 </wsdl:message>
2895
2896 <wsdl:message name="setAggregateMetadataResponse">
2897 </wsdl:message>
2898
2899 <wsdl:message name="getNextDestinationsRequest">
2900     <wsdl:part element="cas" type="xmi:XMI" name="cas"/>
2901 </wsdl:message>
2902
2903 <wsdl:message name="getNextDestinationsResponse">
2904     <wsdl:part element="step" type="pe:Step" name="step"/>
2905 </wsdl:message>
2906
2907 <wsdl:message name="continueOnFailureRequest">
2908     <wsdl:part element="cas" type="xmi:XMI" name="cas"/>
2909     <wsdl:part element="failedAnalyticKey" type="xsd:string"
2910 name="failedAnalyticKey"/>
2911     <wsdl:part element="failure" type="pe:UimaException" name="failure"/>
2912 </wsdl:message>
2913
2914 <wsdl:message name="continueOnFailureResponse">
2915     <wsdl:part element="continue" type="xsd:boolean" name="continue"/>
2916 </wsdl:message>
2917
2918 <!-- Define a portType for each of the UIMA interfaces -->
2919 <wsdl:portType name="Analyzer">
2920
2921     <wsdl:operation name="getMetadata">
2922         <wsdl:input message="service:getMetadataRequest"
2923             name="getMetadataRequest"/>
2924         <wsdl:output message="service:getMetadataResponse"
2925             name="getMetadataResponse"/>
2926         <wsdl:fault message="service:uimaFault"

```

```

2927     name="uimaFault"/>
2928 </wsdl:operation>
2929
2930 <wsdl:operation name="setConfigurationParameters">
2931   <wsdl:input
2932     message="service:setConfigurationParametersRequest"
2933     name="setConfigurationParametersRequest"/>
2934   <wsdl:output
2935     message="service:setConfigurationParametersResponse"
2936     name="setConfigurationParametersResponse"/>
2937   <wsdl:fault message="service:uimaFault"
2938     name="uimaFault"/>
2939 </wsdl:operation>
2940
2941 <wsdl:operation name="processCas">
2942   <wsdl:input message="service:processCasRequest"
2943     name="processCasRequest"/>
2944   <wsdl:output message="service:processCasResponse"
2945     name="processCasResponse"/>
2946   <wsdl:fault message="service:uimaFault"
2947     name="uimaFault"/>
2948 </wsdl:operation>
2949
2950 <wsdl:operation name="processCasBatch">
2951   <wsdl:input message="service:processCasBatchRequest"
2952     name="processCasBatchRequest"/>
2953   <wsdl:output message="service:processCasBatchResponse"
2954     name="processCasBatchResponse"/>
2955   <wsdl:fault message="service:uimaFault"
2956     name="uimaFault"/>
2957 </wsdl:operation>
2958 </wsdl:portType>
2959
2960 <wsdl:portType name="CasMultiplier">
2961
2962   <wsdl:operation name="getMetadata">
2963     <wsdl:input message="service:getMetadataRequest"
2964       name="getMetadataRequest"/>
2965     <wsdl:output message="service:getMetadataResponse"
2966       name="getMetadataResponse"/>
2967     <wsdl:fault message="service:uimaFault"
2968       name="uimaFault"/>
2969   </wsdl:operation>
2970
2971   <wsdl:operation name="setConfigurationParameters">
2972     <wsdl:input
2973       message="service:setConfigurationParametersRequest"
2974       name="setConfigurationParametersRequest"/>
2975     <wsdl:output
2976       message="service:setConfigurationParametersResponse"
2977       name="setConfigurationParametersResponse"/>
2978     <wsdl:fault message="service:uimaFault"
2979       name="uimaFault"/>
2980   </wsdl:operation>
2981
2982   <wsdl:operation name="inputCas">
2983     <wsdl:input message="service:inputCasRequest"
2984       name="inputCasRequest"/>

```

```

2985     <wsdl:output message="service:inputCasResponse"
2986         name="inputCasResponse"/>
2987     <wsdl:fault message="service:uimaFault"
2988         name="uimaFault"/>
2989 </wsdl:operation>
2990
2991 <wsdl:operation name="getNextCas">
2992     <wsdl:input message="service:getNextCasRequest"
2993         name="getNextCasRequest"/>
2994     <wsdl:output message="service:getNextCasResponse"
2995         name="getNextCasResponse"/>
2996     <wsdl:fault message="service:uimaFault"
2997         name="uimaFault"/>
2998 </wsdl:operation>
2999
3000 <wsdl:operation name="retrieveInputCas">
3001     <wsdl:input message="service:retrieveInputCasRequest"
3002         name="retrieveInputCasRequest"/>
3003     <wsdl:output message="service:retrieveInputCasResponse"
3004         name="retrieveInputCasResponse"/>
3005     <wsdl:fault message="service:uimaFault"
3006         name="uimaFault"/>
3007 </wsdl:operation>
3008
3009 <wsdl:operation name="getNextCasBatch">
3010     <wsdl:input message="service:getNextCasBatchRequest"
3011         name="getNextCasBatchRequest"/>
3012     <wsdl:output message="service:getNextCasBatchResponse"
3013         name="getNextCasBatchResponse"/>
3014     <wsdl:fault message="service:uimaFault"
3015         name="uimaFault"/>
3016 </wsdl:operation>
3017 </wsdl:portType>
3018
3019 <wsdl:portType name="FlowController">
3020
3021     <wsdl:operation name="getMetadata">
3022         <wsdl:input message="service:getMetadataRequest"
3023             name="getMetadataRequest"/>
3024         <wsdl:output message="service:getMetadataResponse"
3025             name="getMetadataResponse"/>
3026         <wsdl:fault message="service:uimaFault"
3027             name="uimaFault"/>
3028     </wsdl:operation>
3029
3030     <wsdl:operation name="setConfigurationParameters">
3031         <wsdl:input
3032             message="service:setConfigurationParametersRequest"
3033             name="setConfigurationParametersRequest"/>
3034         <wsdl:output
3035             message="service:setConfigurationParametersResponse"
3036             name="setConfigurationParametersResponse"/>
3037         <wsdl:fault message="service:uimaFault"
3038             name="uimaFault"/>
3039     </wsdl:operation>
3040
3041     <wsdl:operation name="addAvailableAnalytics">
3042         <wsdl:input message="service:addAvailableAnalyticsRequest"

```

```

3043     name="addAvailableAnalyticsRequest"/>
3044     <wsdl:output message="service:addAvailableAnalyticsResponse"
3045     name="addAvailableAnalyticsResponse"/>
3046     <wsdl:fault message="service:uimaFault"
3047     name="uimaFault"/>
3048 </wsdl:operation>
3049
3050 <wsdl:operation name="removeAvailableAnalytics">
3051     <wsdl:input
3052     message="service:removeAvailableAnalyticsRequest"
3053     name="removeAvailableAnalyticsRequest"/>
3054     <wsdl:output
3055     message="service:removeAvailableAnalyticsResponse"
3056     name="removeAvailableAnalyticsResponse"/>
3057     <wsdl:fault message="service:uimaFault"
3058     name="uimaFault"/>
3059 </wsdl:operation>
3060
3061 <wsdl:operation name="setAggregateMetadata">
3062     <wsdl:input message="service:setAggregateMetadataRequest"
3063     name="setAggregateMetadataRequest"/>
3064     <wsdl:output message="service:setAggregateMetadataResponse"
3065     name="setAggregateMetadataResponse"/>
3066     <wsdl:fault message="service:uimaFault"
3067     name="uimaFault"/>
3068 </wsdl:operation>
3069
3070 <wsdl:operation name="getNextDestinations">
3071     <wsdl:input message="service:getNextDestinationsRequest"
3072     name="getNextDestinationsRequest"/>
3073     <wsdl:output message="service:getNextDestinationsResponse"
3074     name="getNextDestinationsResponse"/>
3075     <wsdl:fault message="service:uimaFault"
3076     name="uimaFault"/>
3077 </wsdl:operation>
3078
3079 <wsdl:operation name="continueOnFailure">
3080     <wsdl:input message="service:continueOnFailureRequest"
3081     name="continueOnFailureRequest"/>
3082     <wsdl:output message="service:continueOnFailureResponse"
3083     name="continueOnFailureResponse"/>
3084     <wsdl:fault message="service:uimaFault"
3085     name="uimaFault"/>
3086 </wsdl:operation>
3087
3088 </wsdl:portType>
3089
3090 <!-- Define a SOAP binding for each portType. -->
3091 <wsdl:binding name="AnalyzerSoapBinding" type="service:Analyzer">
3092
3093     <wsdlsoap:binding style="rpc"
3094     transport="http://schemas.xmlsoap.org/soap/http"/>
3095
3096     <wsdl:operation name="getMetadata">
3097     <wsdlsoap:operation soapAction=""/>
3098
3099     <wsdl:input name="getMetadataRequest">
3100     <wsdlsoap:body use="literal"/>

```

```

3101     </wsdl:input>
3102
3103     <wsdl:output name="getMetadataResponse">
3104         <wsdlsoap:body use="literal"/>
3105     </wsdl:output>
3106 </wsdl:operation>
3107
3108 <wsdl:operation name="setConfigurationParameters">
3109     <wsdlsoap:operation soapAction=""/>
3110
3111     <wsdl:input name="setConfigurationParametersRequest">
3112         <wsdlsoap:body use="literal"/>
3113     </wsdl:input>
3114
3115     <wsdl:output name="setConfigurationParametersResponse">
3116         <wsdlsoap:body use="literal"/>
3117     </wsdl:output>
3118 </wsdl:operation>
3119
3120 <wsdl:operation name="processCas">
3121     <wsdlsoap:operation soapAction=""/>
3122
3123     <wsdl:input name="processCasRequest">
3124         <wsdlsoap:body use="literal"/>
3125     </wsdl:input>
3126
3127     <wsdl:output name="processCasResponse">
3128         <wsdlsoap:body use="literal"/>
3129     </wsdl:output>
3130 </wsdl:operation>
3131
3132 <wsdl:operation name="processCasBatch">
3133     <wsdlsoap:operation soapAction=""/>
3134
3135     <wsdl:input name="processCasBatchRequest">
3136         <wsdlsoap:body use="literal"/>
3137     </wsdl:input>
3138
3139     <wsdl:output name="processCasBatchResponse">
3140         <wsdlsoap:body use="literal"/>
3141     </wsdl:output>
3142 </wsdl:operation>
3143 </wsdl:binding>
3144
3145 <wsdl:binding name="CasMultiplierSoapBinding"
3146     type="service:CasMultiplier">
3147
3148     <wsdlsoap:binding style="rpc"
3149         transport="http://schemas.xmlsoap.org/soap/http"/>
3150
3151     <wsdl:operation name="getMetadata">
3152         <wsdlsoap:operation soapAction=""/>
3153
3154     <wsdl:input name="getMetadataRequest">
3155         <wsdlsoap:body use="literal"/>
3156     </wsdl:input>
3157
3158     <wsdl:output name="getMetadataResponse">

```

```

3159     <wsdlsoap:body use="literal"/>
3160 </wsdl:output>
3161
3162     <wsdl:fault name="uimaFault">
3163         <wsdlsoap:fault use="literal"/>
3164     </wsdl:fault>
3165 </wsdl:operation>
3166
3167 <wsdl:operation name="setConfigurationParameters">
3168     <wsdlsoap:operation soapAction=""/>
3169
3170     <wsdl:input name="setConfigurationParametersRequest">
3171         <wsdlsoap:body use="literal"/>
3172     </wsdl:input>
3173
3174     <wsdl:output name="setConfigurationParametersResponse">
3175         <wsdlsoap:body use="literal"/>
3176     </wsdl:output>
3177
3178     <wsdl:fault name="uimaFault">
3179         <wsdlsoap:fault use="literal"/>
3180     </wsdl:fault>
3181 </wsdl:operation>
3182
3183 <wsdl:operation name="inputCas">
3184     <wsdlsoap:operation soapAction=""/>
3185
3186     <wsdl:input name="inputCasRequest">
3187         <wsdlsoap:body use="literal"/>
3188     </wsdl:input>
3189
3190     <wsdl:output name="inputCasResponse">
3191         <wsdlsoap:body use="literal"/>
3192     </wsdl:output>
3193
3194     <wsdl:fault name="uimaFault">
3195         <wsdlsoap:fault use="literal"/>
3196     </wsdl:fault>
3197 </wsdl:operation>
3198
3199 <wsdl:operation name="getNextCas">
3200     <wsdlsoap:operation soapAction=""/>
3201
3202     <wsdl:input name="getNextCasRequest">
3203         <wsdlsoap:body use="literal"/>
3204     </wsdl:input>
3205
3206     <wsdl:output name="getNextCasResponse">
3207         <wsdlsoap:body use="literal"/>
3208     </wsdl:output>
3209
3210     <wsdl:fault name="uimaFault">
3211         <wsdlsoap:fault use="literal"/>
3212     </wsdl:fault>
3213 </wsdl:operation>
3214
3215 <wsdl:operation name="retrieveInputCas">
3216     <wsdlsoap:operation soapAction=""/>

```

```

3217
3218     <wsdl:input name="retrieveInputCasRequest">
3219         <wsdlsoap:body use="literal"/>
3220     </wsdl:input>
3221
3222     <wsdl:output name="retrieveInputCasResponse">
3223         <wsdlsoap:body use="literal"/>
3224     </wsdl:output>
3225
3226     <wsdl:fault name="uimaFault">
3227         <wsdlsoap:fault use="literal"/>
3228     </wsdl:fault>
3229 </wsdl:operation>
3230
3231 <wsdl:operation name="getNextCasBatch">
3232     <wsdlsoap:operation soapAction=""/>
3233
3234     <wsdl:input name="getNextCasBatchRequest">
3235         <wsdlsoap:body use="literal"/>
3236     </wsdl:input>
3237
3238     <wsdl:output name="getNextCasBatchResponse">
3239         <wsdlsoap:body use="literal"/>
3240     </wsdl:output>
3241
3242     <wsdl:fault name="uimaFault">
3243         <wsdlsoap:fault use="literal"/>
3244     </wsdl:fault>
3245 </wsdl:operation>
3246 </wsdl:binding>
3247
3248 <wsdl:binding name="FlowControllerSoapBinding"
3249     type="service:FlowController">
3250
3251     <wsdlsoap:binding style="rpc"
3252         transport="http://schemas.xmlsoap.org/soap/http"/>
3253
3254     <wsdl:operation name="getMetadata">
3255         <wsdlsoap:operation soapAction=""/>
3256
3257         <wsdl:input name="getMetadataRequest">
3258             <wsdlsoap:body use="literal"/>
3259         </wsdl:input>
3260
3261         <wsdl:output name="getMetadataResponse">
3262             <wsdlsoap:body use="literal"/>
3263         </wsdl:output>
3264
3265         <wsdl:fault name="uimaFault">
3266             <wsdlsoap:fault use="literal"/>
3267         </wsdl:fault>
3268     </wsdl:operation>
3269
3270     <wsdl:operation name="setConfigurationParameters">
3271         <wsdlsoap:operation soapAction=""/>
3272
3273         <wsdl:input name="setConfigurationParametersRequest">
3274             <wsdlsoap:body use="literal"/>

```

```
3275     </wsdl:input>
3276
3277     <wsdl:output name="setConfigurationParametersResponse">
3278         <wsdlsoap:body use="literal"/>
3279     </wsdl:output>
3280
3281     <wsdl:fault name="uimaFault">
3282         <wsdlsoap:fault use="literal"/>
3283     </wsdl:fault>
3284 </wsdl:operation>
3285
3286 <wsdl:operation name="addAvailableAnalytics">
3287     <wsdlsoap:operation soapAction=""/>
3288
3289     <wsdl:input name="addAvailableAnalyticsRequest">
3290         <wsdlsoap:body use="literal"/>
3291     </wsdl:input>
3292
3293     <wsdl:output name="addAvailableAnalyticsResponse">
3294         <wsdlsoap:body use="literal"/>
3295     </wsdl:output>
3296
3297     <wsdl:fault name="uimaFault">
3298         <wsdlsoap:fault use="literal"/>
3299     </wsdl:fault>
3300 </wsdl:operation>
3301
3302 <wsdl:operation name="removeAvailableAnalytics">
3303     <wsdlsoap:operation soapAction=""/>
3304
3305     <wsdl:input name="removeAvailableAnalyticsRequest">
3306         <wsdlsoap:body use="literal"/>
3307     </wsdl:input>
3308
3309     <wsdl:output name="removeAvailableAnalyticsResponse">
3310         <wsdlsoap:body use="literal"/>
3311     </wsdl:output>
3312
3313     <wsdl:fault name="uimaFault">
3314         <wsdlsoap:fault use="literal"/>
3315     </wsdl:fault>
3316 </wsdl:operation>
3317
3318 <wsdl:operation name="setAggregateMetadata">
3319     <wsdlsoap:operation soapAction=""/>
3320
3321     <wsdl:input name="setAggregateMetadataRequest">
3322         <wsdlsoap:body use="literal"/>
3323     </wsdl:input>
3324
3325     <wsdl:output name="setAggregateMetadataResponse">
3326         <wsdlsoap:body use="literal"/>
3327     </wsdl:output>
3328
3329     <wsdl:fault name="uimaFault">
3330         <wsdlsoap:fault use="literal"/>
3331     </wsdl:fault>
3332 </wsdl:operation>
```

```

3333
3334     <wsdl:operation name="getNextDestinations">
3335         <wsdlsoap:operation soapAction=""/>
3336
3337         <wsdl:input name="getNextDestinationsRequest">
3338             <wsdlsoap:body use="literal"/>
3339         </wsdl:input>
3340
3341         <wsdl:output name="getNextDestinationsResponse">
3342             <wsdlsoap:body use="literal"/>
3343         </wsdl:output>
3344
3345         <wsdl:fault name="uimaFault">
3346             <wsdlsoap:fault use="literal"/>
3347         </wsdl:fault>
3348     </wsdl:operation>
3349
3350     <wsdl:operation name="continueOnFailure">
3351         <wsdlsoap:operation soapAction=""/>
3352
3353         <wsdl:input name="continueOnFailureRequest">
3354             <wsdlsoap:body use="literal"/>
3355         </wsdl:input>
3356
3357         <wsdl:output name="continueOnFailureResponse">
3358             <wsdlsoap:body use="literal"/>
3359         </wsdl:output>
3360
3361         <wsdl:fault name="uimaFault">
3362             <wsdlsoap:fault use="literal"/>
3363         </wsdl:fault>
3364     </wsdl:operation>
3365 </wsdl:binding>
3366 </wsdl:definitions>
3367

```

3368 **C.7 PE Service XML Schema (uima.peServiceXMI.xsd)**

3369 This XML schema is referenced from the WSDL definition in Appendix C.6

```

3370 <?xml version="1.0" encoding="UTF-8" standalone="no"?>
3371 <xsd:schema xmlns:uima.pe="http://docs.oasis-open.org/uima/pe.ecore"
3372 xmlns:uima.peMetadata="http://docs.oasis-open.org/uima/peMetadata.ecore"
3373 xmlns:xmi="http://www.omg.org/XMI"
3374 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
3375 targetNamespace="http://docs.oasis-open.org/uima/pe.ecore">
3376     <xsd:import namespace="http://docs.oasis-open.org/uima/peMetadata.ecore"
3377 schemaLocation="uima.peMetadataXMI.xsd"/>
3378     <xsd:import namespace="http://www.omg.org/XMI" schemaLocation="XMI.xsd"/>
3379     <xsd:complexType name="AnalyticMetadataMap">
3380         <xsd:choice maxOccurs="unbounded" minOccurs="0">
3381             <xsd:element name="AnalyticMetadataMapEntry"
3382 type="uima.pe:AnalyticMetadataMapEntry"/>
3383             <xsd:element ref="xmi:Extension"/>
3384         </xsd:choice>
3385         <xsd:attribute ref="xmi:id"/>
3386         <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
3387         <xsd:attribute name="AnalyticMetadataMapEntry" type="xsd:string"/>

```

```

3388     </xsd:complexType>
3389     <xsd:element name="AnalyticMetadataMap"
3390 type="uima.pe:AnalyticMetadataMap"/>
3391     <xsd:complexType name="AnalyticMetadataMapEntry">
3392     <xsd:choice maxOccurs="unbounded" minOccurs="0">
3393     <xsd:element name="ProcessingElementMetadata"
3394 type="uima.peMetadata:ProcessingElementMetadata"/>
3395     <xsd:element ref="xmi:Extension"/>
3396     </xsd:choice>
3397     <xsd:attribute ref="xmi:id"/>
3398     <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
3399     <xsd:attribute name="key" type="xsd:string"/>
3400     <xsd:attribute name="ProcessingElementMetadata" type="xsd:string"/>
3401     </xsd:complexType>
3402     <xsd:element name="AnalyticMetadataMapEntry"
3403 type="uima.pe:AnalyticMetadataMapEntry"/>
3404     <xsd:complexType name="Step">
3405     <xsd:choice maxOccurs="unbounded" minOccurs="0">
3406     <xsd:element ref="xmi:Extension"/>
3407     </xsd:choice>
3408     <xsd:attribute ref="xmi:id"/>
3409     <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
3410     </xsd:complexType>
3411     <xsd:element name="Step" type="uima.pe:Step"/>
3412     <xsd:complexType name="SimpleStep">
3413     <xsd:complexContent>
3414     <xsd:extension base="uima.pe:Step">
3415     <xsd:attribute name="analyticKey" type="xsd:string"/>
3416     </xsd:extension>
3417     </xsd:complexContent>
3418     </xsd:complexType>
3419     <xsd:element name="SimpleStep" type="uima.pe:SimpleStep"/>
3420     <xsd:complexType name="MultiStep">
3421     <xsd:complexContent>
3422     <xsd:extension base="uima.pe:Step">
3423     <xsd:choice maxOccurs="unbounded" minOccurs="0">
3424     <xsd:element name="steps" type="uima.pe:Step"/>
3425     </xsd:choice>
3426     <xsd:attribute name="parallel" type="xsd:boolean"/>
3427     </xsd:extension>
3428     </xsd:complexContent>
3429     </xsd:complexType>
3430     <xsd:element name="MultiStep" type="uima.pe:MultiStep"/>
3431     <xsd:complexType name="FinalStep">
3432     <xsd:complexContent>
3433     <xsd:extension base="uima.pe:Step"/>
3434     </xsd:complexContent>
3435     </xsd:complexType>
3436     <xsd:element name="FinalStep" type="uima.pe:FinalStep"/>
3437     <xsd:complexType name="Keys">
3438     <xsd:choice maxOccurs="unbounded" minOccurs="0">
3439     <xsd:element name="key" nillable="true" type="xsd:string"/>
3440     <xsd:element ref="xmi:Extension"/>
3441     </xsd:choice>
3442     <xsd:attribute ref="xmi:id"/>
3443     <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
3444     </xsd:complexType>
3445     <xsd:element name="Keys" type="uima.pe:Keys"/>

```

```

3446 <xsd:complexType name="ObjectList">
3447   <xsd:choice maxOccurs="unbounded" minOccurs="0">
3448     <xsd:element name="objects" type="xmi:Any"/>
3449     <xsd:element ref="xmi:Extension"/>
3450   </xsd:choice>
3451   <xsd:attribute ref="xmi:id"/>
3452   <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
3453   <xsd:attribute name="objects" type="xsd:string"/>
3454 </xsd:complexType>
3455 <xsd:element name="ObjectList" type="uima.pe:ObjectList"/>
3456 <xsd:complexType name="UimaException">
3457   <xsd:choice maxOccurs="unbounded" minOccurs="0">
3458     <xsd:element ref="xmi:Extension"/>
3459   </xsd:choice>
3460   <xsd:attribute ref="xmi:id"/>
3461   <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
3462   <xsd:attribute name="message" type="xsd:string"/>
3463 </xsd:complexType>
3464 <xsd:element name="UimaException" type="uima.pe:UimaException"/>
3465 <xsd:complexType name="ConfigurationParameterSettings">
3466   <xsd:choice maxOccurs="unbounded" minOccurs="0">
3467     <xsd:element name="ConfigurationParameterSetting"
3468 type="uima.pe:ConfigurationParameterSetting"/>
3469     <xsd:element ref="xmi:Extension"/>
3470   </xsd:choice>
3471   <xsd:attribute ref="xmi:id"/>
3472   <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
3473 </xsd:complexType>
3474 <xsd:element name="ConfigurationParameterSettings"
3475 type="uima.pe:ConfigurationParameterSettings"/>
3476 <xsd:complexType name="ConfigurationParameterSetting">
3477   <xsd:choice maxOccurs="unbounded" minOccurs="0">
3478     <xsd:element name="values" nillable="true" type="xsd:string"/>
3479     <xsd:element ref="xmi:Extension"/>
3480   </xsd:choice>
3481   <xsd:attribute ref="xmi:id"/>
3482   <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
3483   <xsd:attribute name="parameterName" type="xsd:string"/>
3484 </xsd:complexType>
3485 <xsd:element name="ConfigurationParameterSetting"
3486 type="uima.pe:ConfigurationParameterSetting"/>
3487 <xsd:complexType name="CasBatchInput">
3488   <xsd:choice maxOccurs="unbounded" minOccurs="0">
3489     <xsd:element name="CasBatchInputElement"
3490 type="uima.pe:CasBatchInputElement"/>
3491     <xsd:element ref="xmi:Extension"/>
3492   </xsd:choice>
3493   <xsd:attribute ref="xmi:id"/>
3494   <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
3495 </xsd:complexType>
3496 <xsd:element name="CasBatchInput" type="uima.pe:CasBatchInput"/>
3497 <xsd:complexType name="CasBatchInputElement">
3498   <xsd:choice maxOccurs="unbounded" minOccurs="0">
3499     <xsd:element name="cas" type="xmi:Any"/>
3500     <xsd:element name="sofas" type="uima.pe:ObjectList"/>
3501     <xsd:element ref="xmi:Extension"/>
3502   </xsd:choice>
3503   <xsd:attribute ref="xmi:id"/>

```

```

3504     <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
3505   </xsd:complexType>
3506   <xsd:element name="CasBatchInputElement"
3507 type="uima.pe:CasBatchInputElement"/>
3508   <xsd:complexType name="CasBatchResponse">
3509     <xsd:choice maxOccurs="unbounded" minOccurs="0">
3510       <xsd:element name="CasBatchResponseElement"
3511 type="uima.pe:CasBatchResponseElement"/>
3512       <xsd:element ref="xmi:Extension"/>
3513     </xsd:choice>
3514     <xsd:attribute ref="xmi:id"/>
3515     <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
3516   </xsd:complexType>
3517 <xsd:element name="CasBatchResponse" type="uima.pe:CasBatchResponse"/>
3518 <xsd:complexType name="CasBatchResponseElement">
3519   <xsd:choice maxOccurs="unbounded" minOccurs="0">
3520     <xsd:element name="CAS" type="xmi:Any"/>
3521     <xsd:element name="UimaException" type="uima.pe:UimaException"/>
3522     <xsd:element ref="xmi:Extension"/>
3523   </xsd:choice>
3524   <xsd:attribute ref="xmi:id"/>
3525   <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
3526 </xsd:complexType>
3527 <xsd:element name="CasBatchResponseElement"
3528 type="uima.pe:CasBatchResponseElement"/>
3529 <xsd:complexType name="GetNextCasBatchResponse">
3530   <xsd:choice maxOccurs="unbounded" minOccurs="0">
3531     <xsd:element name="CAS" type="xmi:Any"/>
3532     <xsd:element ref="xmi:Extension"/>
3533   </xsd:choice>
3534   <xsd:attribute ref="xmi:id"/>
3535   <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
3536   <xsd:attribute name="hasMoreCASes" type="xsd:boolean"/>
3537   <xsd:attribute name="estimatedRemainingCASes" type="xsd:int"/>
3538 </xsd:complexType>
3539 <xsd:element name="GetNextCasBatchResponse"
3540 type="uima.pe:GetNextCasBatchResponse"/>
3541 </xsd:schema>
3542

```

3543

D. Revision History

3544 [optional; should not be included in OASIS Standards]

3545

Revision	Date	Editor	Changes Made
1	11 March 2008	Adam Lally	First spec revision in OASIS template
2	10 April 2008	Adam Lally	Integrated Section 3.3 text from Karin. Rewrote Abstract Interface Complainece points to require standard XMLdata representation. Expanded Section 4.5.4 Behavioral Metadata Formal Specification, to include mapping to OCL. Other cleanup to sections 3.5 and 3.7.
3	24 April 2008	Adam Lally	Integrated Section 1 text from Dave, Section 3.1 and 3.2 text from Eric, additional Section 3.3 updates from Karin, and section 3.6 text from Thomas. Also fixed some UML diagrams in these sections. Added processCasBatch and getNextCasBatch operations to Abstract Interfaces so they would be in sync with the WSDL spec. Added 3.2.4.2 to reference XMI, UML, and MOF for definition of an object being a valid instance of a class. Fixed OCL in 3.5.8.3.1.
4	21 May 2008	Adam Lally	Major reorganization. Section 3 now contains an expanded overview of each spec element. Section 4 is the full specification of each element. Appendix B is the examples. Fixed many errors and typos found by Karin and myself. Updated all the Formal Specification Artifacts in Appendix C. Added Related Work, Abstract, and Acknowledgments sections. Added Karin and Eric to list of editors. Added a note that Discontiguous annotations

are not defined by standard but can be implemented by a user-defined subtype of Annotation (section 4.3.2.2).

Added a note that the Entity type subsumes Events and Relations (section 4.3.2.3).

3546