



Unstructured Information Management Architecture (UIMA) Version 1.0

Committee Specification 01

14 November 2008

Specification URIs:

This Version:

<http://docs.oasis-open.org/uima/v1.0/cs01/uima-spec-cs-01.html>
<http://docs.oasis-open.org/uima/v1.0/cs01/uima-spec-cs-01.doc> (Authoritative)
<http://docs.oasis-open.org/uima/v1.0/cs01/uima-spec-cs-01.pdf>

Previous Version:

<http://docs.oasis-open.org/uima/v1.0/cd01/uima-spec-cd-01.html>
<http://docs.oasis-open.org/uima/v1.0/cd01/uima-spec-cd-01.doc> (Authoritative)
<http://docs.oasis-open.org/uima/v1.0/cd01/uima-spec-cd-01.pdf>

Latest Version:

<http://docs.oasis-open.org/uima/v1.0/uima-v1.0.html>
<http://docs.oasis-open.org/uima/v1.0/uima-v1.0.doc>
<http://docs.oasis-open.org/uima/v1.0/uima-v1.0.pdf>

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/ns/base.ecore>
<http://docs.oasis-open.org/uima/ns/peMetadata.ecore>
<http://docs.oasis-open.org/uima/ns/pe.ecore>
<http://docs.oasis-open.org/uima/ns/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 document was last revised or approved by the UIMA TC on the above date. The level of approval is also listed above. Check the "Latest Approved Version" location noted above for possible later revisions of this document.

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" and "UIMA" 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.....	6
1.1	Terminology	7
1.2	Normative References	7
1.3	Non-Normative References	8
2	Basic Concepts and Terms	9
3	Elements of the UIMA Specification	11
3.1	Common Analysis Structure (CAS)	11
3.2	Type System Model	11
3.3	Base Type System.....	13
3.4	Abstract Interfaces	13
3.5	Behavioral Metadata.....	14
3.6	Processing Element Metadata.....	15
3.7	WSDL Service Descriptions.....	16
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	CAS Formal Specification	18
4.2	The Type System Model.....	19
4.2.1	Ecore as the UIMA Type System Model	19
4.2.2	Type System Model 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	Base Type System Formal Specification	25
4.4	Abstract Interfaces.....	25
4.4.1	Abstract Interfaces URL	25
4.4.2	Abstract Interfaces Formal Specification.....	26
4.5	Behavioral Metadata.....	30
4.5.1	Behavioral Metadata UML.....	30
4.5.2	Behavioral Metadata Elements and XML Representation	31
4.5.3	Formal Semantics for Behavioral Metadata	31
4.5.4	Behavioral Metadata Formal Specification.....	33
4.6	Processing Element Metadata.....	36
4.6.1	Elements of PE Metadata.....	36
4.6.2	Processing Element Metadata Formal Specification.....	39
4.7	Service WSDL Descriptions.....	39
4.7.1	Overview of the WSDL Definition	39
4.7.2	Delta Responses	43
4.7.3	Service WSDL Formal Specification	43

5	Conformance	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.....	59
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	Base Type System XML Schema	76
C.5	PE Metadata Ecore Model	79
C.6	PE Metadata XML Schema.....	82
C.7	PE Service WSDL Definition	84
C.8	PE Service Data Types XML Schema (uima.peServiceXML.xsd).....	95

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 including 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 applications that rely on specific semantics, 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 correctly.

An example of assigning semantics includes labeling regions of text in a text document with appropriate XML tags that, for example, 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 or identifying a person or a type of physical object in a series of video frames.

In general, we refer to a segment of unstructured content (e.g., a document, a video etc.) as an **artifact** and we refer to the act of assigning semantics to a region of an artifact as **analysis**. A software component or service that performs the analysis is referred to as an **analytic**. The results of the analysis of an artifact by an analytic are referred to as **artifact metadata**.

Analytics are typically reused and combined together in different flows to perform application-specific aggregate analyses. For example, in the analysis of a document the first analytic may simply identify and label the distinct tokens or words in the document. The next analytic might identify parts of speech, the third might use the output of the previous two to more accurately identify instances of persons, organizations and the relationships between them

49 While different platform-specific, software frameworks have been developed with varying features in
50 support of building and integrating component analytics (e.g., Apache UIMA, Gate, Catalyst, Tipster,
51 Mallet, Talent, Open-NLP, LingPipe etc.), no clear standard has emerged for enabling the interoperability
52 of analytics across platforms, frameworks and modalities (text, audio, video, etc.) Significant advances in
53 the field of unstructured information analysis require that it is easier to combine best-of-breed analytics
54 across these dimensions.

55

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

59

60 1. **Data Representation.** Support the common representation of *artifacts* and *artifact metadata*
61 independently of *artifact modality* and *domain model* and in a way that is independent of the
62 original representation of the artifact.

63

64 2. **Data Modeling and Interchange.** Support the platform-independent interchange of *analysis data*
65 (*artifact and its metadata*) in a form that facilitates a formal modeling approach and alignment with
66 existing programming systems and standards.

67

68 3. **Discovery, Reuse and Composition.** Support the discovery, reuse and composition of
69 independently-developed *analytics*.

70

71 4. **Service-Level Interoperability.** Support concrete interoperability of independently developed
72 *analytics* based on a common service description and associated SOAP bindings.

73

74 The text of this specification is normative with the exception of the Introduction and Examples (Appendix
75 B).

76 1.1 Terminology

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

80 1.2 Normative References

81 [RFC2119] S. Bradner, *Key words for use in RFCs to Indicate Requirement Levels*,
82 <http://www.ietf.org/rfc/rfc2119.txt>, IETF RFC 2119, March 1997.

83 [MOF1] Object Management Group. Meta Object Facility (MOF) 2.0 Core Specification.
84 <http://www.omg.org/docs/ptc/04-10-15.pdf>

85 [OCL1] Object Management Group. Object Constraint Language Version 2.0.
86 <http://www.omg.org/technology/documents/formal/ocl.htm>

87 [OSGi1] OSGi Alliance. OSGi Service Platform Core Specification, Release 4, Version 4.1.
88 Available from <http://www.osgi.org>.

89 [SOAP1] W3C. SOAP Version 1.2 Part 1: Messaging Framework (Second Edition).
90 <http://www.w3.org/TR/soap12-part1/>

91 [UML1] Object Management Group. Unified Modeling Language (UML), version 2.1.2.
92 <http://www.omg.org/technology/documents/formal/uml.htm>

93 [XMI1] Object Management Group. XML Metadata Interchange (XMI) Specification, Version 2.0.
94 <http://www.omg.org/docs/formal/03-05-02.pdf>

- 95 **[XML1]** W3C. Extensible Markup Language (XML) 1.0 (Fourth Edition).
96 <http://www.w3.org/TR/REC-xml>
- 97 **[XML2]** W3C. Namespaces in XML 1.0 (Second Edition). <http://www.w3.org/TR/REC-xml-names/>
- 98 **[XMLS1]** XML Schema Part 1: Structures Second Edition. <http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/structures.html>
- 99
- 100 **[XMLS2]** XML Schema Part 2: Datatypes Second Edition. <http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/datatypes.html>
- 101
- 102

103 **1.3 Non-Normative References**

- 104 **[BPEL1]** http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel
- 105 **[EcoreEMOF1]** <http://dev.eclipse.org/newslists/news.eclipse.tools.emf/msg04197.html>
- 106
- 107 **[EMF1]** The Eclipse Modeling Framework (EMF) Overview.
108 <http://help.eclipse.org/help33/index.jsp?topic=/org.eclipse.emf.doc//references/overview/EMF.html>
- 109
- 110 **[EMF2]** Budinsky et al. Eclipse Modeling Framework. Addison-Wesley. 2004.
- 111 **[EMF3]** Budinsky et al. Eclipse Modeling Framework, Chapter 2, Section 2.3
112 <http://www.awprofessional.com/content/images/0131425420/samplechapter/budinskych02.pdf>
- 113
- 114 **[KLT1]** David Ferrucci, William Murdock, Chris Welty, “Overview of Component Services for
115 Knowledge Integration in UIMA (a.k.a. SUKI)” IBM Research Report RC24074
- 116 **[XMI2]** Grose et al. Mastering XMI. Java Programming with XMI, XML, and UML. John Wiley &
117 Sons, Inc. 2002

118 2 Basic Concepts and Terms

119 This specification defines and uses the following terms:

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

125

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

130

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

133

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

140

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

147

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

149

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

156

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

158

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

160

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

163

164 **Processing Element Metadata (PE Metadata)** is data that describes a Processing Element (PE) by
165 providing information used for discovering, combining, or reusing the PE for the development of UIM
166 applications. PE Metadata would include Behavioral Metadata for the operation which the PE implements.
167

168 3 Elements of the UIMA Specification

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

171 3.1 Common Analysis Structure (CAS)

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

175
176 The CAS represents an essential element of the UIMA specification in support of interoperability since it
177 provides the common foundation for sharing data and results across analytics.

178
179 The CAS is an Object Graph where Objects are instances of Classes and Classes are Types in a **type**
180 **system** (see next section).

181
182 A general and motivating UIMA use case is one where analytics label or *annotate* regions of unstructured
183 content. A fundamental approach to representing annotations is referred to as the “stand-off” annotation
184 model. In a “stand-off” annotation model, annotations are represented as objects of a domain model that
185 “point into” or reference elements of the unstructured content (e.g., document or video stream) rather than
186 as inserted tags that affect and/or are constrained by the original form of the content.

187
188 To support the stand-off annotation model, UIMA defines two fundamental types of objects in a CAS:

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

192 The Sofa and Annotation types are formally defined as part of the UIMA Base Type System (see Section
193 3.3).

194
195 The CAS provides a domain neutral, object-based representation scheme that is aligned with UML
196 [UML1]. UIMA defines an XML representation of analysis data using the XML Metadata Interchange
197 (XMI) specification [XMI1][XMI2].

198
199 The CAS representation can easily be elaborated for specific domains of analysis by defining domain-
200 specific types; interoperability can be achieved across programming languages and operating systems
201 through the use of the CAS representation and its associated type system definition.

202
203 For the full CAS specification, see Section 4.1.

204 3.2 Type System Model

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

207
208 A type system is a collection of inter-related **type** definitions. Each type defines the structure of any object
209 that is an instance of that type. For example, Person and Organization may be types defined as part of a

210 type system. Each type definition declares the attributes of the type and describes valid fillers for its
211 attributes. For example lastName, age, emergencyContact and employer may be attributes of the Person
212 type. The type system may further specify that the lastName must be filled with exactly one string value,
213 age exactly one integer value, emergencyContact exactly one instance of the same Person type and
214 employer zero or more instances of the Organization type.

215

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

219

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

226

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

233

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

- 235 • **Object-Oriented.** Type systems defined with the UIMA Type System Model are isomorphic to classes
236 in object-oriented representations such as UML, and are easily mapped or compiled into deployment
237 data structures in a particular implementation framework.
- 238 • **Inheritance.** Types can extend other types, thereby inheriting the features of their parent type.
- 239 • **Optional and Required Features.** The features associated with types can be optional or required,
240 depending on the needs of the application.
- 241 • **Single and Multi-Valued Features with Range Constraints.** The features associated with types can
242 be single-valued or multi-valued, depending on the needs of the application. The legal range of values
243 for a feature (its range constraint) may be specified as part of the feature definition.
- 244 • **Alignment with UML standards and Tooling.** The UIMA Type System model can be directly
245 expressed using existing UML modeling standards, and is designed to take advantage of existing
246 tooling for UML modeling.

247

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

250

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

254

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

258

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

261

262 For the full Type System Model specification, see Section 4.2.

263 **3.3 Base Type System**

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

266

267 The most significant part of the Base Type System is the *Annotation and Sofa (Subject of Analysis) Type*
268 *System*. In UIMA, a CAS stores the artifact (i.e., the unstructured content that is the subject of the
269 analysis) and the artifact metadata (i.e., structured data elements that describe the artifact). The
270 metadata generated by an analytic may include a set of annotations that label regions of the artifact with
271 respect to some domain model (e.g., persons, organizations, events, times, opinions, etc). These
272 annotations are logically and physical distinct from the subject of analysis, so this model is referred to as
273 the “*stand-off*” model for annotations.

274

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

283

284 The Base Type System also includes the following:

- 285 • Primitive Types (defined by Ecore)
- 286 • Views (Specific collections of objects in a CAS)
- 287 • Source Document Information (Records information about the original source of unstructured
288 information in the CAS)

289

290 For the full Base Type System specification, see Section 4.3.

291 **3.4 Abstract Interfaces**

292 The UIMA Abstract Interfaces define the standard component types and operations that UIMA services
293 implement. The abstract definitions in this section lay the foundation for the concrete service specification
294 described in Section 3.7.

295

296 All types of UIMA services operate on the Common Analysis Structure (CAS). As defined in Section 3.1,
297 the CAS is the common data structure that represents the unstructured information being analyzed as
298 well as the metadata produced by the analysis workflow.

299

300 The supertype of all UIMA components is called the *Processing Element (PE)*. The ProcessingElement
301 interface defines the following operations, which are common to all subtypes of ProcessingElement:

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

306

307 An *Analytic* is a subtype of PE that performs analysis of CASes. There are two subtypes, *Analyzer* and
308 *CAS Multiplier*.

309

310 An *Analyzer* processes a CAS and possibly updates its contents. This is the most common type of UIMA
311 component. The *Analyzer* interface defines the operations:

- 312 • `processCas`, which takes a single CAS plus a list of Sofas to analyze, and returns either an
313 updated CAS, or a set of updates to apply to the CAS.
- 314 • `processCasBatch`, which takes multiple CASes, each with a list of Sofas to analyze, and returns
315 a response that contains, for each of the input CASes: an updated CAS, a set of updates to apply
316 to the CAS, or an exception.

317

318 A *CAS Multiplier* processes a CAS and possibly creates new CASes. This is useful for example to
319 implement a “segmenter” *Analytic* that takes an input CAS and divides it into pieces, outputting each
320 piece as a new CAS. A *CAS multiplier* can also be used to merge information from multiple CASes into
321 one output CAS. The *CAS Multiplier* interface defines the following operations:

- 322 • `inputCas`, which takes a CAS plus a list of Sofas, but returns nothing.
- 323 • `getNextCas`, which takes no input and returns a CAS. This returns the next output CAS. An
324 empty response indicates no more output CASes.
- 325 • `retrieveInputCas`, which takes no arguments and returns the original input CAS, possibly
326 updated.
- 327 • `getNextCasBatch`, which takes a maximum number of CASes to return and a maximum amount
328 of time to wait (in milliseconds), and returns a response that contains: Zero or more
329 CASes (up to the maximum number specified), a Boolean indicating whether any more CASes
330 remain, and an estimate of the number of CASes remaining (if known).

331

332 A *Flow Controller* is a subtype of PE that determines the route CASes take through multiple *Analytics*.
333 The *Flow Controller* interface defines the following operations:

- 334 • `addAvailableAnalytics`, which provides the *Flow Controller* with access to the *Analytic*
335 *Metadata* for all of the *Analytics* that the *Flow Controller* may route CASes to. This takes a map
336 from String keys to *ProcessingElementMetadata* objects. This may be called multiple times, if
337 new *analytics* are added to the system after the original call is made.
- 338 • `removeAvailableAnalytics`, which takes a set of *Keys* and instructs the *Flow Controller* to
339 remove some *Analytics* from consideration as possible destinations.
- 340 • `setAggregateMetadata`, which provides the *Flow Controller* with *Processing Element Metadata*
341 that identifies and describes the desired behavior of the entire flow of components that the
342 *FlowController* is managing. The most common use for this is to specify the desired outputs of
343 the aggregate, so that the *Flow Controller* can make decisions about which *analytics* need to be
344 invoked in order to produce those outputs.
- 345 • `getNextDestinations`, which takes a CAS and returns one or more destinations for this CAS.
- 346 • `continueOnFailure`, which can be called by the aggregate/application when a *Step* issued by
347 the *FlowController* failed. The *FlowController* returns true if it can continue, and can change the
348 subsequent flow in any way it chooses based on the knowledge that a failure occurred. The
349 *FlowController* returns false if it cannot continue.

350

351 For the full *Abstract Interfaces* specification, see Section 4.4.

352 **3.5 Behavioral Metadata**

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

356

357 Behavioral Metadata is designed to achieve the following goals:

- 358 1. **Discovery:** Enable both human developers and automated processes to search a repository and
359 locate components that provide a particular function (i.e., works on certain input, produces certain
360 output)
- 361
- 362 2. **Composition:** Support composition either by a human developer or an automated process.
363 a. Analytics should be able to declare what they do in enough detail to assist manual
364 and/or automated processes in considering their role in an application or in the
365 composition of aggregate analytics.
366 b. Through their Behavioral Metadata, Analytics should be able to declare enough detail
367 as to enable an application or aggregate to detect “invalid” compositions/workflows
368 (e.g., a workflow where it can be determined that one of the Analytic’s preconditions
369 can never be satisfied by the preceding Analytic).
- 370
- 371 3. **Efficiency:** Facilitate efficient sharing of CAS content among cooperating analytics. If analytics
372 declare which elements of the CAS (e.g., *views*) they need to receive and which elements they do not
373 need to receive, the CAS can be filtered or split prior to sending it to target analytics, to achieve
374 transport and parallelization efficiencies respectively.

375

376 Behavioral Metadata breaks down into the following categories:

- 377 • **Analyzes:** Types of objects (Sofas) that the analytic intends to produce annotations over.
378 • **Required Inputs:** Types of objects that must be present in the CAS for the analytic to operate.
379 • **Optional Inputs:** Types of objects that the analytic would consult if they were present in the CAS.
380 • **Creates:** Types of objects that the analytic may create.
381 • **Modifies:** Types of objects that the analytic may modify.
382 • **Deletes:** Types of objects that the analytic may delete.

383

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

387

388 For the full Behavioral Metadata specification, see Section 4.5.

389 3.6 Processing Element Metadata

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

393

394 The PE Metadata is subdivided into the following parts:

395

- 396 1. **Identification Information.** Identifies the PE. It includes for example a symbolic/unique name, a
397 descriptive name, vendor and version information.
- 398 2. **Configuration Parameters.** Declares the names of parameters used by the PE to affect its
399 behavior, as well as the parameters’ default values.
- 400 3. **Behavioral Metadata.** Describes the PEs input requirements and the operations that the PE
401 may perform, as described in Section 3.5.
- 402 4. **Type System.** Defines types used by the PE and referenced from the behavioral specification.
- 403 5. **Extensions.** Allows the PE metadata to contain additional elements, the contents of which are
404 not defined by the UIMA specification. This can be used by framework implementations to

405 extend the PE metadata with additional information that may be meaningful only to that
406 framework.
407

408 For the full Processing Element Metadata specification, see Section 4.6.

409 **3.7 WSDL Service Descriptions**

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

413

414 This SOAP interface implements the Abstract Interfaces described in Section 3.4. The use of SOAP
415 facilitates standard use of web services as a CAS transport.

416

417 For the full WSDL Service Descriptions specification, see Section 4.7.

418

419 4 Full UIMA Specification

420 4.1 The Common Analysis Structure (CAS)

421 4.1.1 Basic Structure: Objects and Slots

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

425

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

428

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

431

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

436

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

439 4.1.2 Relationship to Type System

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

441

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

444

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

449

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

451

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

455

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

457

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

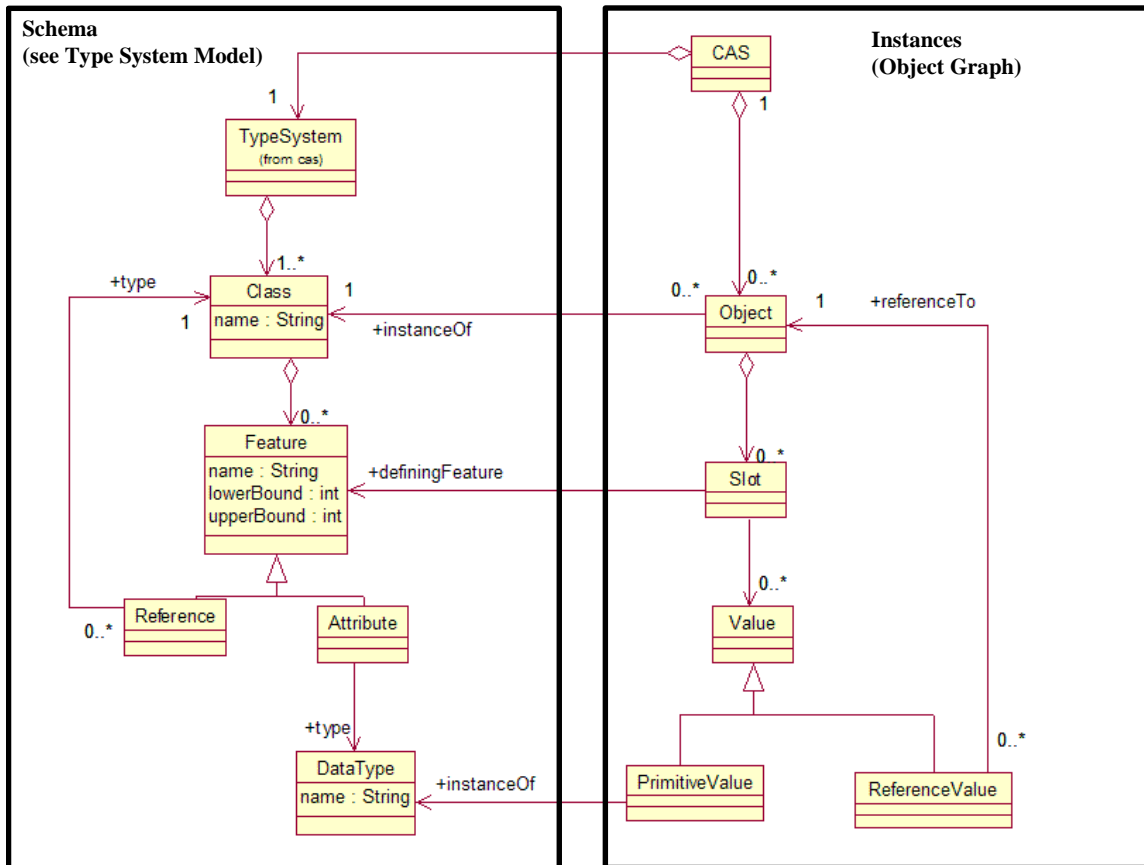


Figure 1: CAS Specification UML

462
463
464

4.1.3 The XMI CAS Representation

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

468

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

4.1.4 CAS Formal Specification

4.1.4.1 Structure

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

475

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

4.1.4.2 Constraints

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

480

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

483

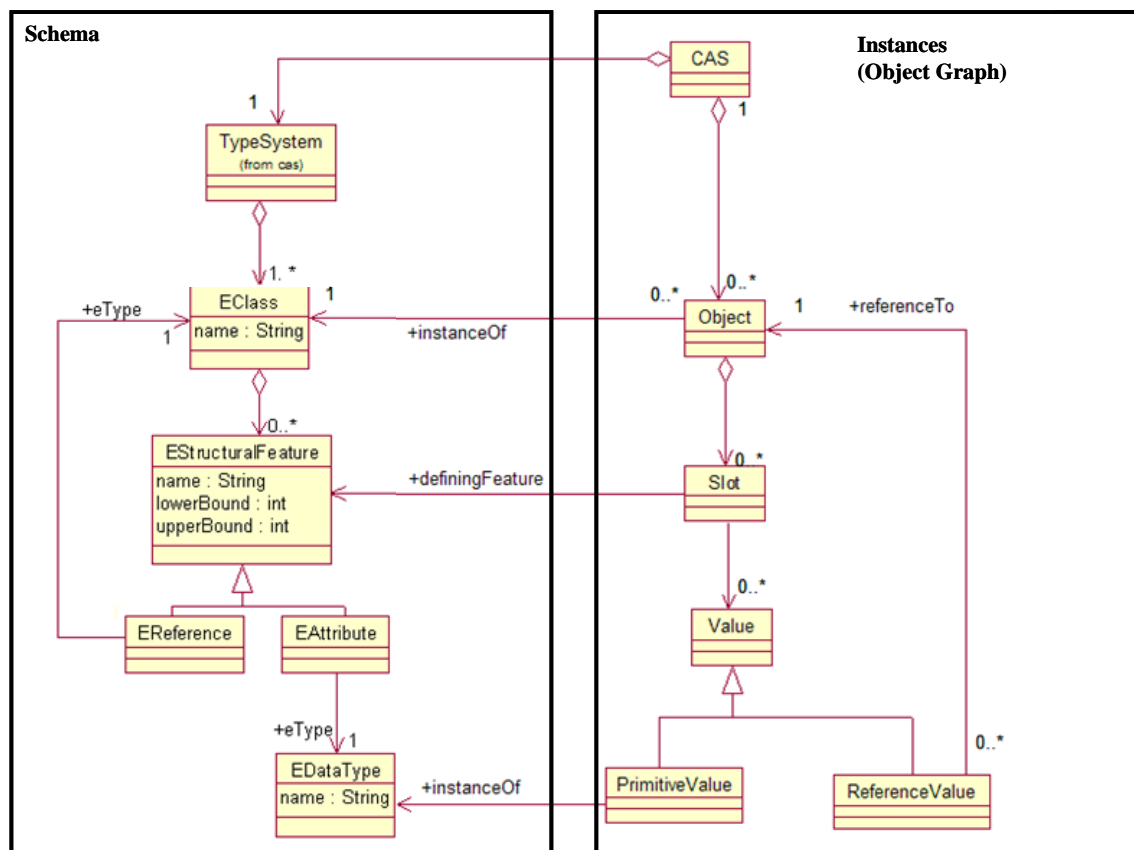
484 A CAS that is linked to an Ecore Type System MUST be valid with respect to that Ecore Type System, as
485 defined in Section 4.2.2.2.

486 4.2 The Type System Model

487 4.2.1 Ecore as the UIMA Type System Model

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

490



491

492

Figure 2: Ecore defines schema for CAS

493

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

496 4.2.2 Type System Model Formal Specification

497 4.2.2.1 Structure

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

500

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

503 **4.2.2.2 Semantics**

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

507 **4.3 Base Type System**

508 The XML namespace for types defined in the UIMA base model is `http://docs.oasis-`
509 `open.org/uima/ns/base.ecore`. (With the exception of types defined as part of Ecore, listed in Section
510 4.3.1, whose namespace is defined by Ecore.).

511

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

513 **4.3.1 Primitive Types**

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

516

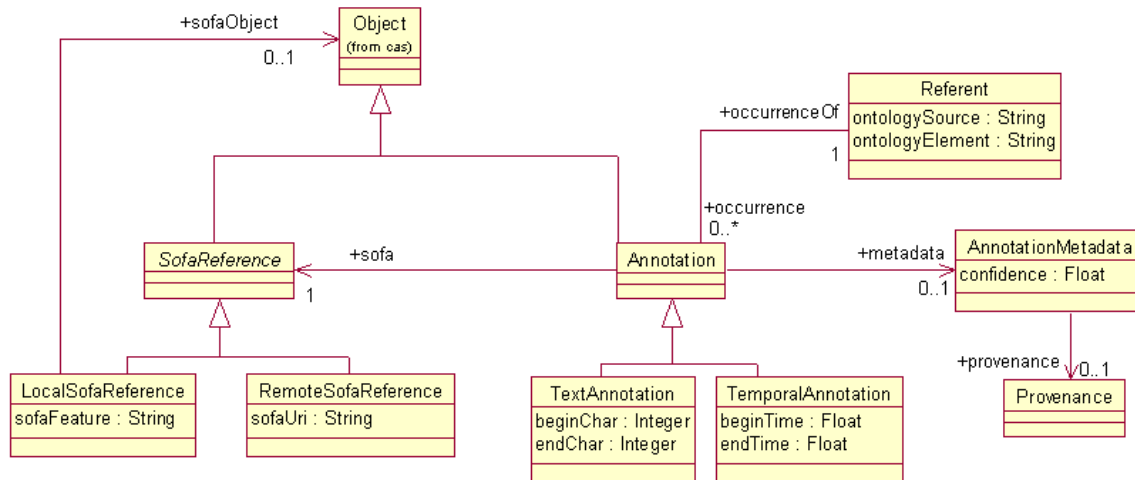
- 517 • EString
- 518 • EBoolean
- 519 • EByte (8 bits)
- 520 • EShort (16 bits)
- 521 • EInt (32 bits)
- 522 • ELong (64 bits)
- 523 • EFloat (32 bits)
- 524 • EDouble (64 bits)

525

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

528 **4.3.2 Annotation and Sofa Base Type System**

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



532
533
534

Figure 3: Annotation and Sofa Base Type System UML

535 **4.3.2.1 Annotation and Sofa Reference**

536 The UIMA Base Type System defines a standard object type called Annotation for representing stand-off
537 annotations. The Annotation type represents a type of object that is linked to a Subject of Analysis (Sofa).
538

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

544 **4.3.2.2 References to Regions of Sofas**

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

549
550 The Annotation class has subclasses for each artifact modality, which define how the Annotation refers to
551 a region within the Sofa. The Standard defines subclasses for common modalities – Text and Temporal
552 (audio or video segments). Users may define other subclasses.

553
554 In TextAnnotation, beginChar and endChar refer to Unicode character offsets in the corresponding Sofa
555 string. For TemporalAnnotation, beginTime and endTime are offsets measured in seconds from the start
556 of the Sofa. Note that applications that require a different interpretation of these fields must accept the
557 standard values and handle their own internal mappings.

558
559 Annotations with discontinuous spans are not part of the Base Type System, but could be implemented
560 with a user-defined subclass of the Annotation type.

561 4.3.2.3 Referents

562 In general, an `Annotation` is an reference to some element in a domain ontology. (For example, the text
563 “John Smith” and “he” might refer to the same person John Smith.) The UIMA Base Type System defines
564 a standard way to encode this information, using the `Annotation` and `Referent` types, and
565 `occurrences/occurrenceOf` features.

566
567 The value of the `Annotation`'s `occurrenceOf` feature is the `Referent` object that identifies the domain
568 element to which that `Annotation` refers. All of the `Annotation` objects that refer to the same thing should
569 share the same `Referent` object. The `Referent`'s `occurrences` feature is the inverse relationship,
570 pointing to all of the `Annotation` objects that refer to that `Referent`.

571
572 A `Referent` need not be a physical object. For example, `Event` and `Relation` are also considered kinds of
573 `Referent`.

574
575 The domain ontology can either be defined directly in the CAS type system or in an external ontology
576 system. If the domain ontology is defined directly in the CAS, then domain classes should be subclasses
577 of the `Referent` class. If the domain ontology is defined in an external ontology system, then the feature
578 `Referent.ontologySource` should be used to identify the target ontology and the feature
579 `Referent.ontologyElement` should be used to identify the target element within that ontology. The
580 format of these identifiers is not defined by UIMA.

581 4.3.2.4 Additional Annotation Metadata

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

588 4.3.3 View Base Type System

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

593
594 For example, given a CAS representing a document, one `View` may capture the metadata describing an
595 English translation of the document while another may capture the metadata describing a French
596 translation of the document.

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

599

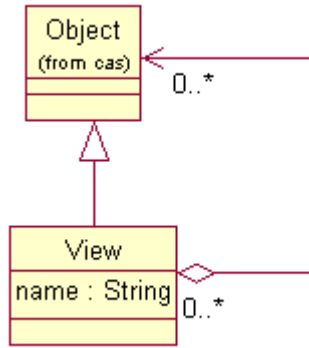


Figure 4: View Type

600
601
602
603
604
605
606
607

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

608 Any application-specific type system could define a *class* that represents a named collection of *objects*
609 and then refer to that *class* in an analytic's behavioral specification. However, since it is a common design
610 pattern we define a standard *View class* to facilitate interoperability between components that operate on
611 such collections of *objects*.

612
613 The members of a view are those *objects* explicitly asserted to be contained in the View. Referring to the
614 UML in Figure 4, we mean that there is an explicit reference from the View to the member *object*.
615 Members of a view may have references to other *objects* that are not members of the same View. A
616 consequence of this is that we cannot in general "export" the members of a View to form a new self-
617 contained CAS, as there could be dangling references. We define the **reference closure of a view**
618 to mean the collection of objects that includes all of the members of the view but also contains all other
619 *objects* referenced either directly or indirectly from the members of the view.

620 4.3.3.1 Anchored View

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

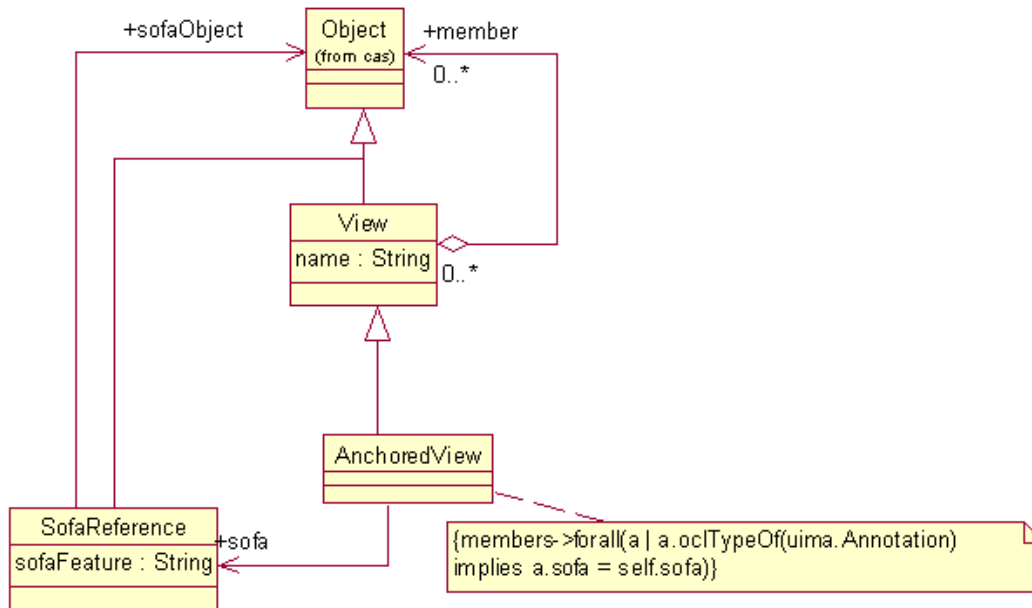
624
625 AnchoredView is as a subtype of View that has a named association with exactly one particular *object* via
626 the standard *feature sofa*.

627
628 An AnchoredView requires that all Annotation *objects* that are members of the AnchoredView have their
629 *sofa feature* refer to the same SofaReference that is referred to by the View's *sofa feature*.

630
631 Simply put, all annotations in an AnchoredView annotate the same subject of analysis.

632
633 Figure 5 shows a UML diagram for the AnchoredView type, including an OCL constraint
634 expression[OCL1] specifying the restriction on the sofa feature of its member annotations.

635



636
637
638

Figure 5: Annotated View Type

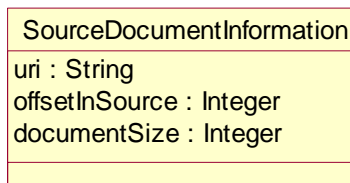
639 The concept of an AnchoredView addresses common use cases. For example, an analytic written to
640 analyze the detagged representation of a document will likely only be able to interpret Annotations that
641 label and therefore refer to regions in that detagged representation. Other Annotations,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

644

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

650 4.3.4 Source Document Information

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



654
655

Figure 6: Source Document Information UML

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

659 **4.3.5 Base Type System Formal Specification**

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

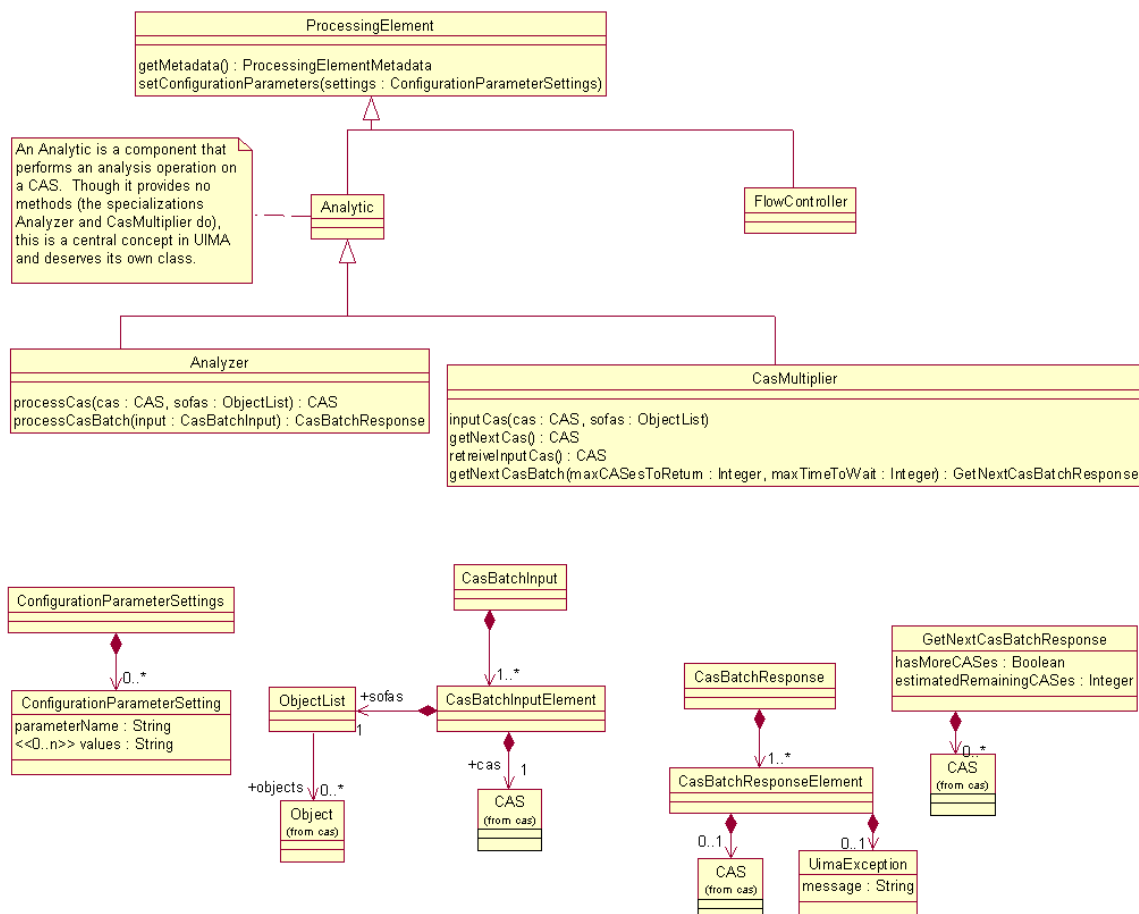
666 **4.4 Abstract Interfaces**

667 **4.4.1 Abstract Interfaces URL**

668 The UIMA specification defines two fundamental types of Processing Elements (PEs) that developers
 669 may implement: *Analytics* and *Flow Controllers*. Refer to Figure 7 for a UML model of the Analytic
 670 interfaces and Figure 8 for a UML model of the FlowController interface. A summary of the operations
 671 defined by each interface is given in Section 3.4.

672 **4.4.1.1 Analytic**

673 An Analytic is a component that performs analysis on CASes. There are two specializations: Analyzer
 674 and CasMultiplier. The Analyzer interface supports Analytics that take a CAS as input and output the
 675 same CAS, possibly updated. The CasMultiplier interface supports zero or more output CASes per input
 676 CAS.



677

678
679

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

680 4.4.1.2 Flow Controller

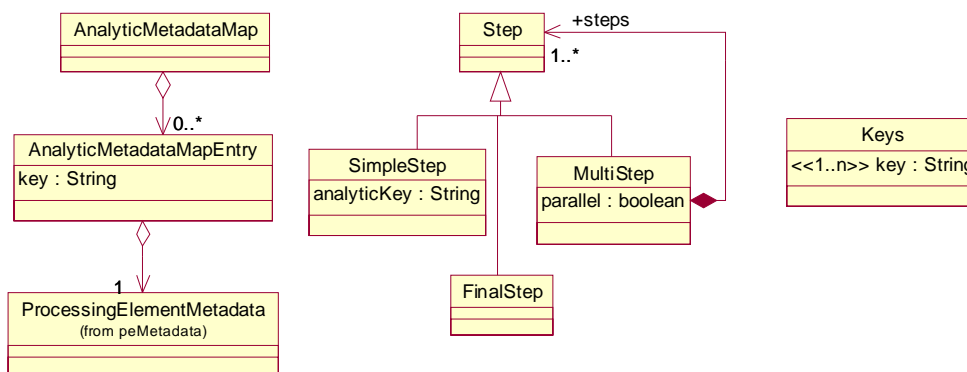
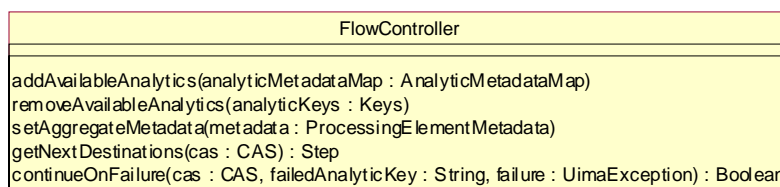
681 A *Flow Controller* is a component that determines the route CASes take through multiple Analytics
682

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

688
689 A FlowController, being a subtype of ProcessingElement, may have configuration parameters. For
690 example, a configuration parameter may refer to a description of the desired flow in some flow language
691 such as BPEL [BPEL1]. This is one way to create a reusable Flow Controller implementation that can be
692 applied in many applications or aggregates.

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

698



699
700

Figure 8: Flow Controller Abstract Interface UML

701 4.4.2 Abstract Interfaces Formal Specification

702 The following subsections specify requirements that a particular type of UIMA service must provide an
703 operation with certain inputs and outputs. For example, a UIMA PE service must implement a

704 getMetaData operation that returns standard UIMA PE Metadata. In all cases, the protocol for invoking
705 this operation is not defined by the standard. However, the format in which data is sent to and from the
706 service MUST be the standard UIMA XML representation. Implementations MAY define additional
707 operations that use other formats.

708 **4.4.2.1 ProcessingElement.getMetaData**

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

713 **4.4.2.2 ProcessingElement.setConfigurationParameters**

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

717
718 The PE Service MUST return an error if the `ConfigurationParameterSettings` object passed to this
719 method contains any of:

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

728

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

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

730

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

735

736 **4.4.2.3 Analyzer.processCas**

737 A UIMA Analyzer Service MUST implement an operation named `processCas`. This operation MUST
738 accept two arguments. The first argument is a CAS, represented in XML as defined in Section 4.1.4. The
739 second argument is a list of `xmi:ids` that identify `SofaReference` objects which the Analyzer is expected

740 to analyze. This operation MUST return a valid XMI document which is either a valid CAS (as defined in
741 Section4.1.4) or a description of changes to be applied to the input CAS using the XMI differences
742 language defined in [XMI1].

743

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

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

752

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

756

757 **4.4.2.4 Analyzer.processCasBatch**

758 A UIMA Analyzer Service MUST implement an operation named `processCasBatch`. This operation
759 MUST accept an argument which consists of one or more CASes, each with an associated list of xmi:ids
760 that identify `SofaReference` objects in that CAS. This operation MUST return a response that consists of
761 multiple elements, one for each input CAS, where each element is either valid XMI document which is
762 either a valid CAS (as defined in Section4.1.4), a description of changes to be applied to the input CAS
763 using the XMI differences language defined in [XMI1], or an exception message.

764

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

767

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

772 **4.4.2.5 CasMultiplier.inputCas**

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

777

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

779 **4.4.2.6 CasMultiplier.getNextCas**

780 A UIMA CAS Multiplier service MUST implement an operation named `getNextCas`. This operation
781 MUST take zero arguments. This operation MUST return a valid CAS as defined in Section4.1.4, or a
782 result indicating that there are no more CASes available.

783

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

786 **4.4.2.7 CasMultiplier.retrieveInputCas**

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

791
792 If the client calls `retrieveInputCas` when this CAS Multiplier has no active CAS, then this CAS Multiplier
793 MUST return an error.

794
795 After this method completes, this service no longer has an active CAS, until the client's next call to
796 `inputCas`.

797 **4.4.2.8 CasMultiplier.getNextCasBatch**

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

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

808
809 The call to `getNextCasBatch` SHOULD attempt to complete and return a response in no more than the
810 amount of time specified (in milliseconds) by the `maxTimeToWait` argument.

811
812 If the client calls `getNextCasBatch` when this CAS Multiplier has no active CAS, then this CAS Multiplier
813 MUST return an error.

814
815 CASes returned from `getNextCasBatch` MUST be equivalent to the CASes that would be returned from
816 individual calls to `getNextCas`.

817 **4.4.2.9 FlowController.addAvailableAnalytics**

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

822 **4.4.2.10 FlowController.removeAvailableAnalytics**

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

828 **4.4.2.11 FlowController.setAggregateMetadata**

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

831

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

835 **4.4.2.12 FlowController.getNextDestinations**

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

839

840 The different types of Step objects are defined in the UML diagram in Figure 8 and XML schema in
841 Appendix C.7. Their intending meanings are as follows:

- 842 • `SimpleStep` identifies a single Analytic to be executed. The Analytic is identified by the String key
843 that was associated with that Analytic in the `AnalyticMetadataMap`.
- 844 • `MultiStep` identifies one more Steps that should be executed next. The `MultiStep` also indicates
845 whether these steps must be performed sequentially or whether they may be performed in parallel.
- 846 • `FinalStep` which indicates that there are no more destinations for this CAS, i.e., that processing of
847 this CAS has completed.

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

850 **4.4.2.13 FlowController.continueOnFailure**

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

855

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

858

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

862

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

865

866 **4.5 Behavioral Metadata**

867 **4.5.1 Behavioral Metadata UML**

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

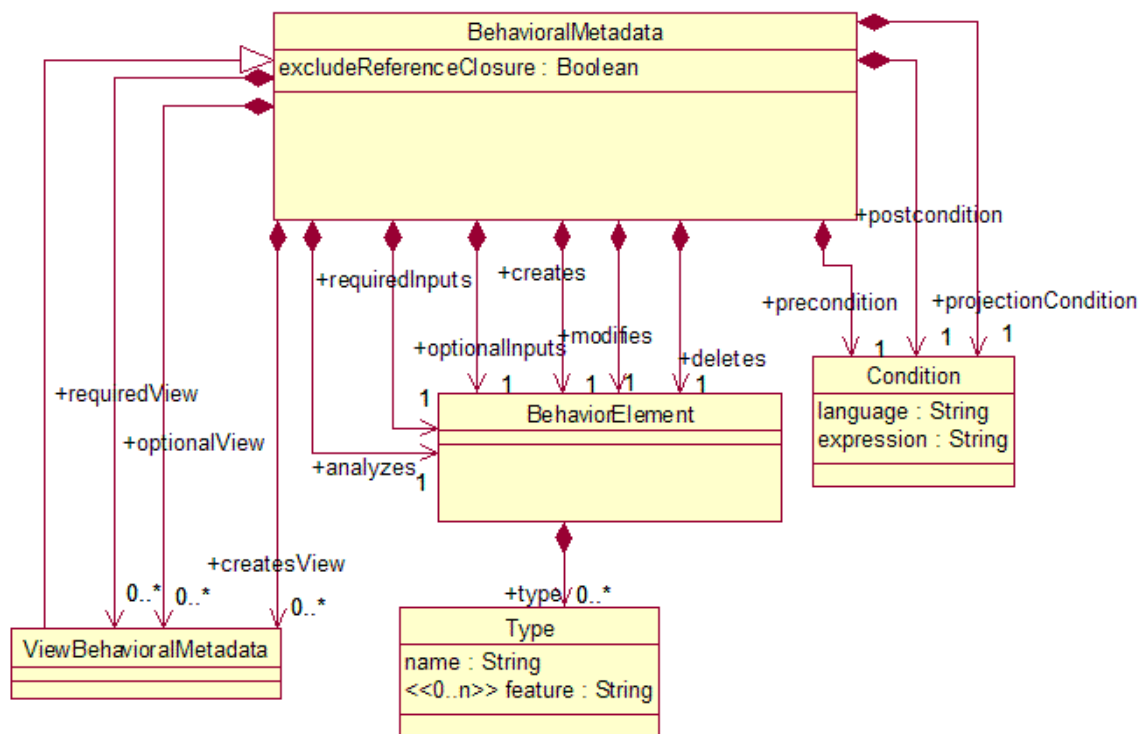


Figure 9: Behavioral Metadata UML

869
870
871

4.5.2 Behavioral Metadata Elements and XML Representation

Behavioral Metadata breaks down into the following categories:

- 874 • **Analyzes:** Types of objects (Sofas) that the analytic intends to produce annotations over.
- 875 • **Required Inputs:** Types of objects that must be present in the CAS for the analytic to operate.
- 876 • **Optional Inputs:** Types of objects that the analytic would consult if they were present in the CAS.
- 877 • **Creates:** Types of objects that the analytic may create.
- 878 • **Modifies:** Types of objects that the analytic may modify.
- 879 • **Deletes:** Types of objects that the analytic may delete.

880

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

4.5.3 Formal Semantics for Behavioral Metadata

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

886

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

893

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

896

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

900

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

904

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

908

909 An *analyzes* or *requiredInputs* predicate translates into a precondition that all input CASes contain the
910 objects that satisfy the predicates.

911

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

914

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

918

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

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

```
922   <requiredInputs>
```

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

```
924   </requiredInputs>
```

```
925 </requiredView>
```

926

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

929

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

933

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

939 4.5.4 Behavioral Metadata Formal Specification

940 4.5.4.1 Structure

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

943

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

946 4.5.4.2 Constraints

947 Field values must satisfy the following constraints:

948 4.5.4.2.1 Type

- 949 • name must be a valid QName (Qualified Name) as defined by the Namespaces for XML specification
950 [XML2]. The namespace of this QName must match the namespace URI of an EPackage defined in an
951 Ecore model referenced by the PE's *TypeSystemReference*. The local part of the QName must match
952 the name of an EClass within that EPackage.
- 953 • Values for the `feature` attribute must not be specified unless the Type is contained in a `modifies`
954 element.
- 955 • Each value of feature must be a valid UnprefixedName as specified in [XML2], and must match the
956 name of an EStructuralFeature in the EClass corresponding to the value of the name field as described
957 in the previous bullet.

958 4.5.4.2.2 Condition

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

965 4.5.4.3 Semantics

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

969

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

974 4.5.4.3.1 Mapping to OCL Precondition

975 An OCL precondition is formed from the *analyzes*, *requiredInputs*, and *requiredView*
976 BehavioralMetadata elements as follows.

977

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

980

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

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

983

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

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

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

990

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

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

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

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

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

996

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

1000 4.5.4.3.2 Mapping to OCL Postcondition

1001 In these OCL expressions the keyword `input` refers to the collection of objects in the CAS when it was
1002 input to the analytic, and the keyword `result` refers to the collection of objects in the CAS at the end of
1003 the analytic's processing. Also note that the suffix `@pre` applied to any attribute references the value of
1004 that attribute at the start of the analytic's operation.

1005

1006 For types *T*₁, *T*₂, ... *T*_{*n*} specified in `creates` elements, produce the OCL expression:

```
1007 result->forAll(p | input->includes(p) or p.oclKindOf(T1) or p.oclKindOf(T2) or  
1008 ... or p.oclKindOf(Tn))
```

1009

1010 For types *T*₁, *T*₂, ... *T*_{*n*} specified in `deletes` elements, produce the OCL expression:

```
1011 input->forAll(p | result->includes(p) or p.oclKindOf(T1) or p.oclKindOf(T2) or  
1012 ... or p.oclKindOf(Tn))
```

1013

1014 For each `modifies` element specifying type *T* with features *F*={*F*₁, *F*₂, ...*F*_{*n*}}, for each feature *g* defined
1015 on type *T* where *g*∉*F*, produce the OCL expression:

```
1016 result->forAll(p | (input->includes(p) and p.oclKindOf(T)) implies p.g =  
1017 p.g@pre)
```

1018

1019 For each `createsView`, `requiredView` or `optionalView` containing `creates` elements with types
1020 *T*₁,*T*₂,...,*T*_{*n*}, produce the OCL expression:

```
1021 result->forAll(v | (ViewExpression) implies v.members->forAll(p |
1022 v.members@pre->includes(p) or p.oclKindOf(T1) or p.oclKindOf(T2) or ... or
1023 p.oclKindOf(Tn))
```

1024 where ViewExpression is as defined in Section 4.5.4.3.1.

1025

1026 For each requiredView or optionalView containing deletes elements with types T1,T2,...,Tn, produce
1027 the OCL expression:

```
1028 result->forAll(v | (ViewExpression) implies v.members@pre->forAll(p |
1029 v.members->includes(p) or p.oclKindOf(T1) or p.oclKindOf(T2) or ... or
1030 p.oclKindOf(Tn))
```

1031 where ViewExpression is as defined in Section 4.5.4.3.1.

1032

1033 Within each requiredView or optionalView, for each modifies element specifying type T with features
1034 F={F1, F2, ...Fn}, for each feature g defined on type T where g∉F, produce the OCL expression:

```
1035 result->forAll(v | (ViewExpression) implies v.members->forAll(p |
1036 (v.members@pre->includes(p) and p.oclKindOf(T)) implies p.g = p.g@pre))
```

1037 where ViewExpression is as defined in Section 4.5.4.3.1.

1038

1039 The final postcondition expression for the analytic is the conjunction of all the expressions generated from
1040 the productions defined in this section, as well as any explicitly declared postcondition as defined in
1041 Section B.5.5.

1042 4.5.4.3.3 Mapping to OCL Projection Condition

1043 In these OCL expressions the keyword input refers to the collection of objects in the entire CAS when it
1044 is about to be delivered to the analytic. The OCL expression evaluates to a collection of objects that the
1045 analytic declares it will consider while performing its operation. When an application or framework calls
1046 this analytic, it MUST deliver to the analytic all objects in this collection.

1047

1048 If the excludeReferenceClosure attribute of the BehavioralMetadata is set to false (or omitted), then the
1049 application or framework MUST also deliver all objects that are referenced (directly or indirectly) from any
1050 object in the collection resulting from evaluation of the projection condition.

1051

1052 For types T1, T2, ... Tn specified in analyzes, requiredInputs, or optionalInputs elements, produce
1053 the OCL expression:

```
1054 input->select(p | p.oclKindOf(T1) or p.oclKindOf(T2) or ... or
1055 p.oclKindOf(Tn))
```

1056

1057 For each requiredView or optionalView produce the OCL expression:

```
1058 input->select(v | ViewExpression)
```

1059 where ViewExpression is as defined in Section 4.5.4.3.1.

1060

1061 If the requiredView or optionalView contains types T1, T2,...Tn specified in analyzes,
1062 requiredInputs, or optionalInputs elements, produce the OCL expression:

```
1063 input->select(v | ViewExpression)->collect(v.members()->select(p |
1064 p.oclKindOf(T1) or p.oclKindOf(T2) or ... or p.oclKindOf(Tn)))
```

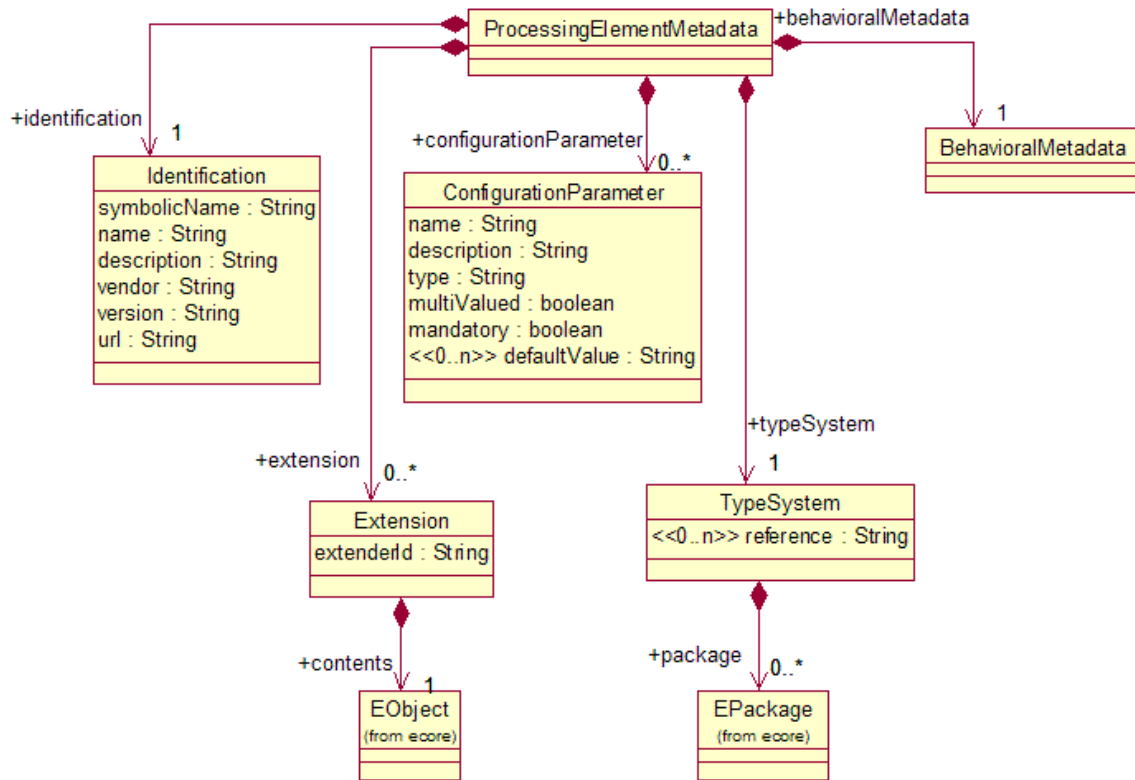
1065
1066
1067
1068
1069

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

1070 4.6 Processing Element Metadata

1071 Figure 10 is a UML model for the PE metadata. We describe each subpart of the PE metadata in detail in
1072 the following sections.

1073



1074
1075
1076

Figure 10: Processing Element Metadata UML Model

1077 4.6.1 Elements of PE Metadata

1078 4.6.1.1 Identification Information

1079 The Identification Information section of the descriptor defines a small set of properties that developers
1080 should fill in with information that describes their PE. The main objectives of this information are to:

- 1081 1. Provide human-readable information about the analytic to assist developers in understanding
- 1082 what the purpose of each PE is.
- 1083 2. Facilitate the development of repositories of PEs.

1084

1085 The following properties are included:

- 1086 1. Symbolic Name: A unique name (such as a Java-style dotted name) for this PE.
1087 2. Name: A human-readable name for the PE. Not necessarily unique.
1088 3. Description: A textual description of the PE.
1089 4. Version: A version number. This is necessary for PE repositories that need to distinguish
1090 different versions of the same component. The syntax of a version number is as defined in
1091 [OSGi1]: up to four dot-separated components where the first three must be numeric but the
1092 fourth may be alphanumeric. For example 1.2.3.4 and 1.2.3.abc are valid version numbers but
1093 1.2.abc is not.
1094 5. Vendor: The provider of the component.
1095 6. URL: website providing information about the component and possibly allowing download of the
1096 component
1097

1098 4.6.1.2 Configuration Parameters

1099 PEs may be configured to operate in different ways. UIMA provides a standard way for PEs to declare
1100 configuration parameters so that application developers are aware of the options that are available to
1101 them.

1102

1103 UIMA provides a standard interface for setting the values of parameters; see Section 4.4 Abstract
1104 Interfaces.

1105

1106 For each configuration parameter we should allow the PE developer to specify:

1107

- 1108 1. The name of the parameter
- 1109 2. A description for the parameter
- 1110 3. The type of value that the parameter may take
- 1111 4. Whether the parameter accepts multiple values or only one
- 1112 5. Whether the parameter is mandatory
- 1113 6. A default value or values for the parameter

1114

1115 One common use of configuration parameters is to refer to external resource data, such as files
1116 containing patterns or statistical models. Frameworks such as Apache UIMA may wish to provide
1117 additional support for such parameters, such as resolution of relative URLs (using classpath/datapath)
1118 and/or caching of shared data. It is therefore important for the UIMA configuration parameter schema to
1119 be expressive enough to distinguish parameters that represent resource locations from parameters that
1120 are just arbitrary strings.

1121

1122 The type of a parameter must be one of the following:

- 1123 • String
- 1124 • Integer (32-bit)
- 1125 • Float (32-bit)
- 1126 • Boolean
- 1127 • ResourceURL

1128

1129 The ResourceURL satisfies the requirement to explicitly identify parameters that represent resource
1130 locations.

1131

1132 Note that parameters may take multiple values so it is not necessary to have explicit parameter types
1133 such as StringArray, IntegerArray, etc.

1134

1135 As a best practice, analytics SHOULD NOT declare configuration settings that would affect their
1136 Behavioral Metadata. UIMA does not provide any mechanism to keep the behavioral specification in sync
1137 with the different configurations.

1138 **4.6.1.3 Type System**

1139 There are two ways that PE metadata may provide type system information: It can either include it or refer
1140 to it. This specification is only concerned with the format of that reference or inclusion. For the actual
1141 definition of the type system, we have adopted the Ecore/XMLI representation. See Section 4.2 for details.

1142
1143 If reference is chosen as the way to provide the type system information, then the `reference` field of the
1144 `TypeSystem` object must be set to a valid URI (or multiple URIs). URIs are used as references by many
1145 web-based standards (e.g., RDF), and they are also used within Ecore. Thus we use a URI to refer to the
1146 type system. To achieve interoperability across frameworks, each URI should be a URL which resolves
1147 to a location where Ecore/XMLI type system data is located.

1148
1149 If embedding is chosen as the way to provide the type system information, then the `package` reference of
1150 the `TypeSystem` object must be set to one or more `EPackages`, where an `EPackage` contains
1151 subpackages and/or classes as defined by Ecore.

1152
1153 The role of this type system is to provide definitions of the types referenced in the PE's behavioral
1154 metadata. It is important to note that this is not a restriction on the CASes that may be input to the PE (if
1155 that is desired, it can be expressed using a precondition in the behavioral specification). If the input CAS
1156 contains instances of types that are not defined by the PE's type system, then the CAS itself may indicate
1157 a URI where definitions of these types may be found (see B.1.6 Linking an XML Document to its Ecore
1158 Type System). Also, some PE's may be capable of processing CASes without being aware of the type
1159 system at all.

1160
1161 Some analytics may be capable of operating on any types. These analytics need not refer to any specific
1162 type system and in their behavioral metadata may declare that they analyze or inspect instances of the
1163 most general type (`EObject` in Ecore).

1164 **4.6.1.4 Behavioral Metadata**

1165 The Behavioral Metadata is discussed in detail in 4.5.

1166 **4.6.1.5 Extensions**

1167 Extension objects allow a framework implementation to extend the PE metadata descriptor with additional
1168 elements, which other frameworks may not necessarily respect. For example Apache UIMA defines an
1169 element `fsIndexCollection` that defines the CAS indexes that the component uses. Other frameworks
1170 could ignore that.

1171
1172 This extensibility is enabled by the Extension class in Figure 10. The Extension class defines two
1173 *features*, `extenderId` and `contents`.

1174
1175 The `extenderId` *feature* identifies the framework implementation that added the extension, which allows
1176 framework implementations to ignore extensions that they were not meant to process.

1177
1178 The `contents` *feature* can contain any `EObject`. (`EObject` is the superclass of all classes in Ecore.) To add
1179 an extension, a framework must provide an Ecore model that defines the structure of the extension.

1180 4.6.2 Processing Element Metadata Formal Specification

1181 4.6.2.1 Structure

1182 *UIMA Processing Element Metadata XML* must be a valid XMI document that is an instance of the UIMA
1183 Processing Element Metadata Ecore model given in Section C.3.

1184

1185 This implies that UIMA Processing Element Metadata XML must be a valid instance of the UIMA
1186 Processing Element Metadata XML schema given in Section C.5.

1187 4.6.2.2 Constraints

1188 Field values must satisfy the following constraints

1189

1190 **Identification Information:**

- 1191 • symbolicName must be a valid symbolic-name as defined by the OSGi specification [OSGi1].
- 1192 • version must be a valid version as defined by the OSGi specification [OSGi1].
- 1193 • url must be a valid URL as defined by [URL1].

1194

1195 **Configuration Parameter**

- 1196 • name must be a valid Name as defined by the XML specification [XML1].
- 1197 • type must be one of {String, Integer, Float, Boolean, ResourceURL}

1198

1199 **Type System Reference**

- 1200 • uri must be a syntactically valid URI as defined by [URI1] It is application defined to check the reference
1201 validity of the URI and handle errors related to dereferencing the URI.

1202

1203 **Extensions**

- 1204 • extenderId must be a valid Name as defined by the XML specification [XML1].

1205

1206 4.7 Service WSDL Descriptions

1207 In this section we describe the UIMA Service WSDL descriptions at a high level. The formal WSDL
1208 document is given in Section C.6.

1209

1210 4.7.1 Overview of the WSDL Definition

1211 Before discussing the elements of the UIMA WSDL definition, as a convenience to the reader we first
1212 provide an overview of WSDL excerpted from the WSDL Specification.

1213

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.

1214
1215

1216 **4.7.1.1 Types**

1217 Type Definitions for the UIMA WSDL service are defined using XML schema. These draw from other
1218 elements of the specification. For example the `ProcessingElementMetadata` type, which is returned
1219 from the `getMetadata` operation, is defined by the PE Metadata specification element.

1220

1221 **4.7.1.2 Messages**

1222 Messages are used to define the structure of the request and response of the various operations
1223 supported by the service. Operations are described in the next section.

1224

1225 Messages refer to the XML schema defined under the `<wsdl:types>` element. So wherever a message
1226 includes a CAS (for example the `processCasRequest` and `processCasResponse`, we indicate that the
1227 type of the data is `xmi:XMI` (a type defined by `XMI.xsd`), and where the message consists of PE metadata
1228 (the `getMetadataResponse`), we indicate that the type of the data is `uima:ProcessingElementMetadata` (a
1229 type defined by `UimaDescriptorSchema.xsd`).

1230

1231 The messages defined by the UIMA WSDL service definition are:

1232 For ALL PEs:

- 1233 • getMetadataRequest – takes no arguments
- 1234 • getMetadataResponse – returns ProcessingElementMetadata
- 1235 • setConfigurationParametersRequest – takes one argument: ConfigurationParameterSettings
- 1236 • setConfigurationParameterResponse – returns nothing

1237

1238 For Analyzers:

- 1239 • processCasRequest – takes two arguments – a CAS and a list of Sofas (object IDs) to process
- 1240 • processCasResponse – returns a CAS
- 1241 • processCasBatchRequest – takes one argument, an Object that includes multiple CASes, each with an associated list of Sofas (object IDs) to process
- 1242
- 1243 • processCasResponse – returns a list of elements, each of which is a CAS or an exception message

1244

1245 For CAS Multipliers:

- 1246 • inputCasRequest – takes two arguments – a CAS and a list of Sofas (object IDs) to process
- 1247 • inputCasResponse – returns nothing
- 1248 • getNextCasRequest – takes no arguments
- 1249 • getNextCasResponse – returns a CAS
- 1250 • retrievalInputCasRequest – takes no arguments
- 1251 • retrievalInputCasResponse – returns a CAS
- 1252 • 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
- 1253
- 1254 • 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).
- 1255
- 1256

1257

1258 For Flow Controllers:

- 1259 • addAvailableAnalyticsRequest – takes one argument, a Map from String keys to PE Metadata objects.
- 1260 • addAvailableAnalyticsResponse – returns nothing
- 1261 • removeAvailableAnalyticsRequest – takes one argument, a collection of one or more String keys
- 1262 • removeAvailableAnalyticsResponse – returns nothing
- 1263 • setAggregateMetadataRequest – takes one argument – a ProcessingElementMetadata
- 1264 • setAggregateMetadataResponse – returns nothing
- 1265 • getNextDestinationsRequest – takes one argument, a CAS
- 1266 • getNextDestinationsResponse – returns a Step object
- 1267 • continueOnFailureRequest – takes three arguments, a CAS, a String key, and a UimaException
- 1268 • continueOnFailureResponse – returns a Boolean

1269

1270 **4.7.1.3 Port Types and Operations**

1271 A *port type* is a collection of *operations*, where each operation is an action that can be performed by the
1272 service. We define a separate port type for each of the three interfaces defined in Section 4.4 Abstract
1273 Interfaces.

1274

1275 The port types and their operations defined by the UIMA WSDL definition are as follows. Each operation
1276 refers to its input and output message, defined in the previous section. Operations also have fault
1277 messages, returned in the case of an error.

- 1278
- 1279 • **Analyzer Port Type**
- 1280 • getMetadata
 - 1281 • setConfigurationParameters
 - 1282 • processCas
 - 1283 • processCasBatch

- 1284
- 1285 • **CasMultiplier Port Type**
- 1286 • getMetadata
 - 1287 • setConfigurationParameters
 - 1288 • inputCas
 - 1289 • getNextCas
 - 1290 • retrievalInputCas
 - 1291 • getNextCasBatch

- 1292
- 1293 **FlowController Port Type**
- 1294 • getMetadata
 - 1295 • setConfigurationParameters
 - 1296 • addAvailableAnalytics
 - 1297 • removeAvailableAnalytics
 - 1298 • setAggregateMetadata
 - 1299 • getNextDestinations
 - 1300 • continueOnFailure

1301 4.7.1.4 SOAP Bindings

1302 For each port type, we define a binding to the SOAP protocol. There are a few configuration choices to
1303 be made:

1304

1305 In `<wsdlsoap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>`:

- 1306 • The style attribute defines that our operation is an RPC, meaning that our XML messages contain
1307 parameters and return values. The alternative is "document" style, which is used for services that
1308 logically send and receive XML documents without a parameter structure. This has an effect on
1309 how the body of the SOAP message is constructed.
- 1310 • The transport operation defines that this binding uses the HTTP protocol (the SOAP spec allows
1311 other protocols, such as FTP or SMTP, but HTTP is by far the most common)

1312 For each parameter (message part) in each abstract operation, we have a `<wsdlsoap:body use="literal"/>`
1313 element:

- 1314 • The use of the `<wsdlsoap:body>` tag indicates that this parameter is sent in the body of the SOAP
1315 message. Alternatively we could use `<wsdlsoap:header>` to choose to send parameters in the
1316 SOAP header. This is an arbitrary choice, but a good rule of thumb is that the data being
1317 processed by the service should be sent in the body, and "control information" (i.e., *how* the
1318 message should be processed) can be sent in the header.
- 1319 • The `use="literal"` attribute states that the content of the message must *exactly* conform to the
1320 XML Schema defined earlier in the WSDL definitions. The other option is "encoded", which treats
1321 the XML Schema as an abstract type definition and applies SOAP encoding rules to determine
1322 the exact XML syntax of the messages. The "encoded" style makes more sense if you are
1323 starting from an abstract object model and you want to let the SOAP rules determine your XML
1324 syntax. In our case, we already know what XML syntax we want (e.g., XML), so the "literal" style
1325 is more appropriate.

1326

1327 **4.7.2 Delta Responses**

1328 If an Analytic makes only a small number of changes to its input CAS, it will be more efficient if the service
1329 response specifies the “deltas” rather than repeating the entire CAS. UIMA supports this by using the
1330 XMI standard way to specify differences between object graphs [XMI1]. An example of such a delta
1331 response is given in the next section.

1332 **4.7.3 Service WSDL Formal Specification**

1333 A *UIMA SOAP Service* MUST conform to the WSDL document given in Section C.6 and MUST
1334 implement at least one of the portTypes and corresponding SOAP bindings defined in that WSDL
1335 document, as defined in [WSDL1] and [SOAP1].

1336

1337 A *UIMA Analyzer SOAP Service* MUST implement the Analyzer portType and the AnalyzerSoapBinding.

1338

1339 A *UIMA CAS Multiplier SOAP Service* MUST implement the CasMultiplier portType and the
1340 CasMultiplierSoapBinding.

5 Conformance

1341

1342 An XML document is conforming UIMA CAS XML if it satisfies the conditions defined in Section 4.1.4
1343 CAS Formal Specification.

1344

1345 An XML document is conforming UIMA Type System XML if it satisfies the conditions defined in Section
1346 4.2.2 Type System Model Formal Specification.

1347

1348 An XML document is conforming UIMA Behavioral Metadata XML if it satisfies the conditions defined in
1349 Section 4.5.4 Behavioral Metadata Formal Specification.

1350

1351 An XML document is conforming UIMA Processing Element Metadata XML if it satisfies the conditions
1352 defined in Section 4.6.2 Processing Element Metadata Formal Specification.

1353

1354 An implementation SHOULD use the Base Type System as defined in Section 4.3.5 Base Type System
1355 Formal Specification.

1356

1357 An implementation is a conforming UIMA Processing Element (PE) Service if it satisfies the conditions
1358 defined in Section 4.4.2 Abstract Interfaces Formal Specification.

1359

1360 An implementation is a conforming UIMA SOAP Service if it satisfied the conditions defined in Section
1361 4.7.3 Service WSDL Formal Specification.

1362

1363 An implementation shall be a UIMA Processing Element (PE) Service or a UIMA SOAP Service.

1364

A. Acknowledgements

1365 The following individuals have participated in the creation of this specification and are gratefully
1366 acknowledged:

1367 **Participants:**

1368 Eric Nyberg, Carnegie Mellon University
1369 Carl Mattocks, CheckMi
1370 Alex Rankov, EMC Corporation
1371 David Ferrucci, IBM
1372 Thilo Goetz, IBM
1373 Thomas Hampp-Bahnmueller, IBM
1374 Adam Lally, IBM
1375 Clifford Thompson, Individual
1376 Karin Verspoor, University of Colorado Denver
1377 Christopher Chute, Mayo Clinic College of Medicine
1378 Vinod Kaggal, Mayo Clinic College of Medicine
1379 Adrian Miley, Miley Watts LLP
1380 Loretta Auvil, National Center for Supercomputing Applications
1381 Duane Sears Smith, National Center for Supercomputing Applications
1382 Pascal Coupet, Temis
1383 Tim Miller, Thomson
1384 Yoshinobu Kano, Tsujii Laboratory, The University of Tokyo
1385 Ngan Nguyen, Tsujii Laboratory, The University of Tokyo
1386 Scott Piao, University of Manchester
1387 Hamish Cunningam, University of Sheffield
1388 Ian Roberts, University of Sheffield
1389
1390

1391 B. Examples (Not Normative)

1392 B.1 XMI CAS Example

1393 This section describes how the CAS is represented in XMI, by way of an example. This is not normative.
1394 The exact specification for XMI is defined by the OMG XMI standard [XMI1].

1395 B.1.1 XMI Tag

1396 The outermost tag is typically <xmi:XMI> (this is just a convention; the XMI spec allows this tag to be
1397 arbitrary). The outermost tag must, however, include an XMI version number and XML namespace
1398 attribute:

```
1399  
1400 <xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI">  
1401 <!-- CAS Contents here -->  
1402 </xmi:XMI>
```

1403
1404 XML namespaces [XML1] are used throughout. The xmi namespace prefix is typically used to identify
1405 elements and attributes that are defined by the XMI specification.

1406
1407 The XMI document will also define one namespace prefix for each CAS namespace, as described in the
1408 next section.

1409

1410 B.1.2 Objects

1411 Each *Object* in the CAS is represented as an XML element. The name of the element is the name of the
1412 object's *class*. The XML namespace of the element identifies the *package* that contains that *class*.

1413

1414 For example consider the following XMI document:

```
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"/>  
1419   ...  
1420 </xmi:XMI>
```

1421

1422 This XMI document contains an object whose class is named Person. The Person class is in the
1423 package with URI http://org/myorg.ecore. Note that the use of the http scheme is a common convention,
1424 and does not imply any HTTP communication. The .ecore suffix is due to the fact that the recommended
1425 type system definition for a package is an ECore model.

1426

1427 Note that the order in which Objects are listed in the XMI is not important, and components that process
1428 XMI are not required to maintain this order.

1429

1430 The xmi:id attribute can be used to refer to an object from elsewhere in the XMI document. It is not
1431 required if the object is never referenced. If an xmi:id is provided, it must be unique among all xmi:ids on
1432 all objects in this CAS.

1433

1434 All namespace prefixes (e.g., myorg) in this example must be bound to URIs using the
1435 "xmlns..." attribute, as defined by the XML namespaces specification [XMLS1].

1436

1437 **B.1.3 Attributes (Primitive Features)**

1438 *Attributes* (that is, *features* whose values are of primitive types, for example, strings, integers and other
1439 numeric types – see Base Type System for details) can be mapped either to XML attributes or XML
1440 elements.

1441

1442 For example, an *object* of *class* Person, with slots:

1443

1444 begin = 14

1445 end = 25

1446 name = "Fred Center"

1447

1448 could be mapped to the attribute serialization as follows:

1449

```
1450 <xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI "  
1451     xmlns:myorg="http://org/myorg.ecore">  
1452     ...  
1453     <myorg:Person xmi:id="1" begin="14" end="25" name="Fred Center"/>  
1454     ...  
1455 </xmi:XMI>
```

1456

1457 or alternatively to an element serialization as follows:

1458

```
1459 <xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI "  
1460     xmlns:myorg="http://org/myorg.ecore">  
1461     ...  
1462     <myorg:Person xmi:id="1">  
1463         <begin>14</begin>  
1464         <end>25</end>  
1465         <name>Fred Center</name>  
1466     </myorg:Person>  
1467     ...  
1468 </xmi:XMI>
```

1469

1470 UIMA framework components that process XMI are required to support both. Mixing the two styles is
1471 allowed; some *features* can be represented as attributes and others as elements.

1472 B.1.4 References (Object-Valued Features)

1473 *Features* that are references to other *objects* are serialized as ID references.

1474

1475 If we add to the previous CAS example an Object of Class Organization, with *feature* myCEO that is a
1476 reference to the Person object, the serialization would be:

1477

```
1478 <xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"  
1479   xmlns:myorg="http://org/myorg.ecore">  
1480   ...  
1481   <myorg:Person xmi:id="1" begin="14" end="25" name="Fred Center"/>  
1482   <myorg:Organization xmi:id="2" myCEO="1"/>  
1483   ...  
1484 </xmi:XMI>
```

1485

1486 As with primitive-valued *features*, it is permitted to use an element rather than an attribute, and UIMA
1487 framework components that process XML are required to support both representations. However, the XML
1488 spec defines a slightly different syntax for this as is illustrated in this example:

1489

```
1490 <myorg:Organization xmi:id="2">  
1491   <myCEO href="#1"/>  
1492 </myorg:Organization>
```

1493

1494 Note that in the attribute representation, a reference *feature* is indistinguishable from an integer-valued
1495 *feature*, so the meaning cannot be determined without prior knowledge of the type system. The element
1496 representation is unambiguous.

1497 B.1.5 Multi-valued Features

1498 *Features* may have multiple values. Consider the example where the *object* of class Baz has a *feature*
1499 myIntArray whose value is {2,4,6}. This can be mapped to:

1500

```
1501 <myorg:Baz xmi:id="3" myIntArray="2 4 6"/>
```

1502

1503 or:

1504

```
1505 <myorg:Baz xmi:id="3">  
1506   <myIntArray>2</myIntArray>  
1507   <myIntArray>4</myIntArray>  
1508   <myIntArray>6</myIntArray>  
1509 </myorg:Baz>
```

1510

1511 Note that string arrays whose elements contain embedded spaces must use the latter mapping.

1512

1513 Multi-valued *references* serialized in a similar way. For example a *reference* that refers to the elements
1514 with xmi:ids "13" and "42" could be serialized as:

1515
1516 `<myorg:Baz xmi:id="3" myRefFeature="13 42"/>`

1517
1518 or:

1519
1520 `<myorg:Baz xmi:id="3">`
1521 `<myRefFeature href="#13"/>`
1522 `<myRefFeature href="#42"/>`
1523 `</myorg:Baz>`

1524
1525 Note that the order in which the elements of a multi-valued feature are listed *is* meaningful, and
1526 components that process XML documents must maintain this order.
1527

1528 **B.1.6 Linking an XML Document to its Ecore Type System**

1529 The structure of a CAS is defined by a UIMA type system, which is represented by an Ecore model (see
1530 Section 4.2).

1531
1532 If the CAS Type System has been saved to an Ecore file, it is possible to store a link from an XML
1533 document to that Ecore type system. This is done using an `xsi:schemaLocation` attribute on the root XML
1534 element.

1535
1536 The `xsi:schemaLocation` attribute is a space-separated list that represents a mapping from the
1537 namespace URI (e.g., `http://org/myorg.ecore`) to the physical URI of the `.ecore` file containing the type
1538 system for that namespace. For example:

1539
1540 `xsi:schemaLocation="http://org/myorg.ecore file:c:/typesystems/myorg.ecore"`

1541
1542 would indicate that the definition for the `org.myorg` CAS types is contained in the file
1543 `c:/typesystems/myorg.ecore`. You can specify a different mapping for each of your CAS namespaces. For
1544 details see [EMF2].

1545 **B.1.7 XML Extensions**

1546 XML defines an extension mechanism that can be used to record information that you may not want to
1547 include in your type system. This can be used for system-level data that is not part of your domain
1548 model, for example. The syntax is:

1549
1550 `<xmi:Extension extenderId="NAME">`
1551 `<!-- arbitrary content can go inside the Extension element -->`
1552 `</xmi:Extension>`

1553
1554 The `extenderId` attribute allows a particular "extender" (e.g., a UIMA framework implementation) to record
1555 metadata that's relevant only within that framework, without confusing other frameworks that may want to
1556 process the same CAS.

1557

1558 **B.2 Ecore Example**

1559 **B.2.1 An Introduction to Ecore**

1560 Ecore is well described by Budinsky et al. in the book *Eclipse Modeling Framework* [EMF2]. Some brief
1561 introduction to Ecore can be found in a chapter of that book available online [EMF3]. As a convenience to
1562 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.

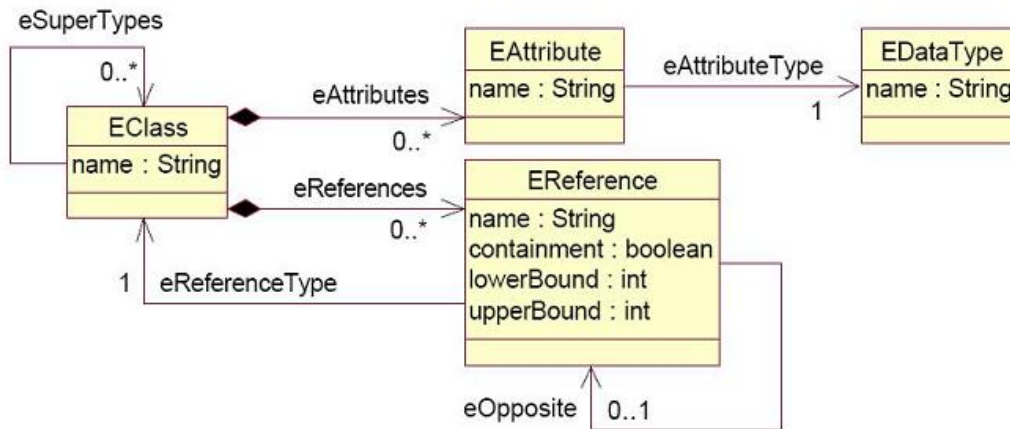


Figure 11: The Ecore Kernel

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.

1563

1564

1565 B.2.2 Differences between Ecore and EMOF

1566 The primary differences between Ecore and EMOF are:

- 1567 • EMOF does not use the 'E' prefix for its metamodel elements. For example EMOF uses the terms
- 1568 *Class* and *Data Type* rather than Ecore's *EClass* and *EDataType*.
- 1569 • EMOF uses a single concept *Property* that subsumes both *EAttribute* and *EReference*.

1570

1571 For a detailed mapping of Ecore terms to EMOF terms see [EcoreEMOF1].

1572 B.2.3 Example Ecore Model

1573 Figure 12 shows a simple example of an object model in UML. This model describes two types of Named

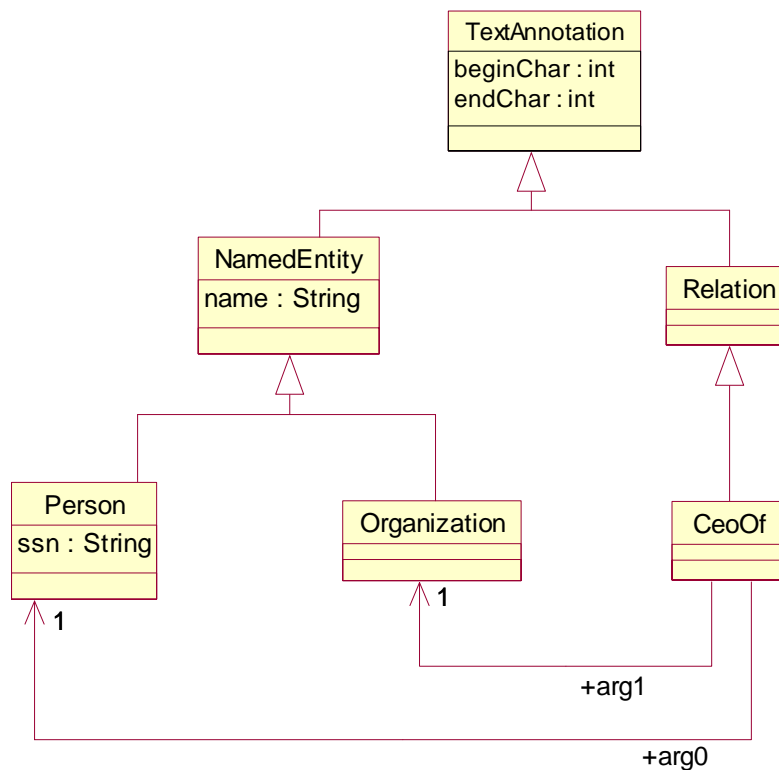
1574 Entities: Person and Organization. They may participate in a CeoOf relation (i.e., a Person is the CEO of

1575 an Organization). The NamedEntity and Relation types are subtypes of TextAnnotation (a standard UIMA

1576 base type, see 4.3), so they will inherit beginChar and endChar features that specify their location within a

1577 text document.

1578



1579

1580

Figure 12: Example UML Model

1581

1582 XMI [XMI1] is an XML format for representing object graphs. EMF tools may be used to automatically

1583 convert this to an Ecore model and generate an XML rendering of the model using XMI:

1584

```

1585 <?xml version="1.0" encoding="UTF-8"?>
1586 <ecore:EPackage xmi:version="2.0"
1587   xmlns:xmi="http://www.omg.org/XMI"
1588   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
1589   xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore"
1590   name="org" nsURI="http://org.ecore" nsPrefix="org">
  
```

```

1591     <eSubpackages name="example" nsURI="http://org/example.ecore"
1592 nsPrefix="org.example">
1593     <eClassifiers xsi:type="ecore:EClass" name="NamedEntity"
1594 eSuperTypes="ecore:EClass http://docs.oasis-
1595 open.org/uima/ns/uima.ecore#//base/TextAnnotation">
1596     <eStructuralFeatures xsi:type="ecore:EAttribute" name="name"
1597 eType="ecore:EDatatype http://www.eclipse.org/emf/2002/Ecore#//EString"/>
1598     </eClassifiers>
1599     <eClassifiers xsi:type="ecore:EClass" name="Relation"
1600 eSuperTypes="ecore:EClass http://docs.oasis-
1601 open.org/uima/ns/uima.ecore#//base/TextAnnotation"/>
1602     <eClassifiers xsi:type="ecore:EClass" name="Person"
1603 eSuperTypes="#//example/NamedEntity">
1604     <eStructuralFeatures xsi:type="ecore:EAttribute" name="ssn"
1605 eType="ecore:EDatatype http://www.eclipse.org/emf/2002/Ecore#//EString"/>
1606     </eClassifiers>
1607     <eClassifiers xsi:type="ecore:EClass" name="CeoOf"
1608 eSuperTypes="#//example/Relation">
1609     <eStructuralFeatures xsi:type="ecore:EReference" name="arg0"
1610 lowerBound="1"
1611     eType="#//example/Person"/>
1612     <eStructuralFeatures xsi:type="ecore:EReference" name="arg1"
1613 lowerBound="1"
1614     eType="#//example/Organization"/>
1615     </eClassifiers>
1616     <eClassifiers xsi:type="ecore:EClass" name="TextDocument">
1617     <eStructuralFeatures xsi:type="ecore:EAttribute" name="text"
1618 eType="ecore:EDatatype http://www.eclipse.org/emf/2002/Ecore#//EString"/>
1619     </eClassifiers>
1620     <eClassifiers xsi:type="ecore:EClass" name="Organization"
1621 eSuperTypes="#//example/NamedEntity"/>
1622     </eSubpackages>
1623 </ecore:EPackage>

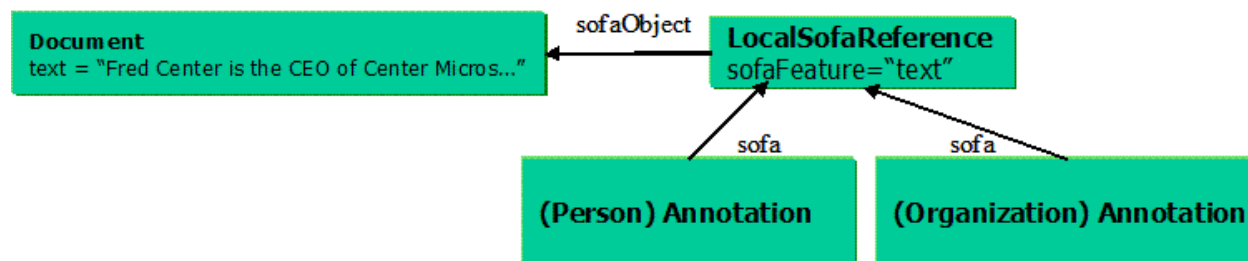
```

1624
1625 This XML document is a valid representation of a UIMA Type System.
1626

1627 B.3 Base Type System Examples

1628 B.3.1 Sofa Reference

1629 Figure 13 illustrates an example of an annotation referring to its subject of analysis (Sofa).



1630
1631 **Figure 13: Annotation and Subject of Analysis**

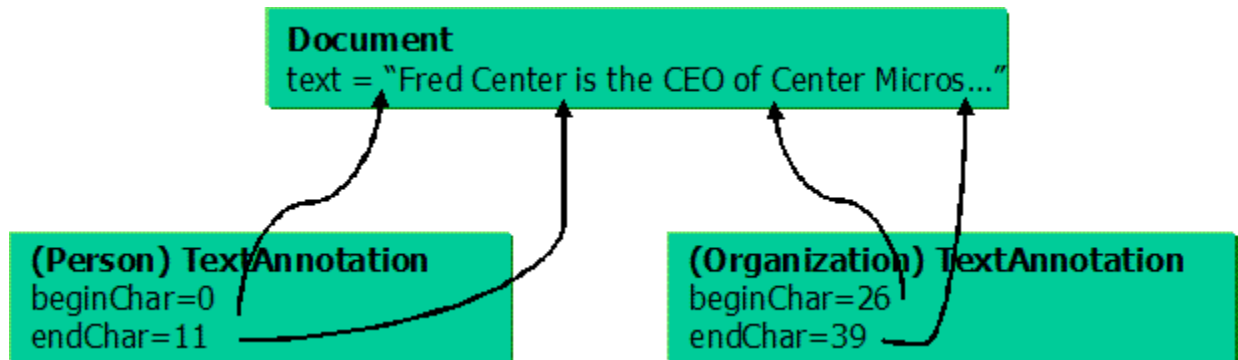
1632 The CAS contains an *object* of class Document with a *slot* text containing the string value, "Fred Center is
1633 the CEO of Center Micros."

1634
1635
1636
1637
1638
1639

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

B.3.2 References to Regions of Sofas

1640
1641
1642 Figure 14 extends the previous example by showing how the `TextAnnotation` subtype of `Annotation` is used to specify a range of character offsets to which the annotation applies.



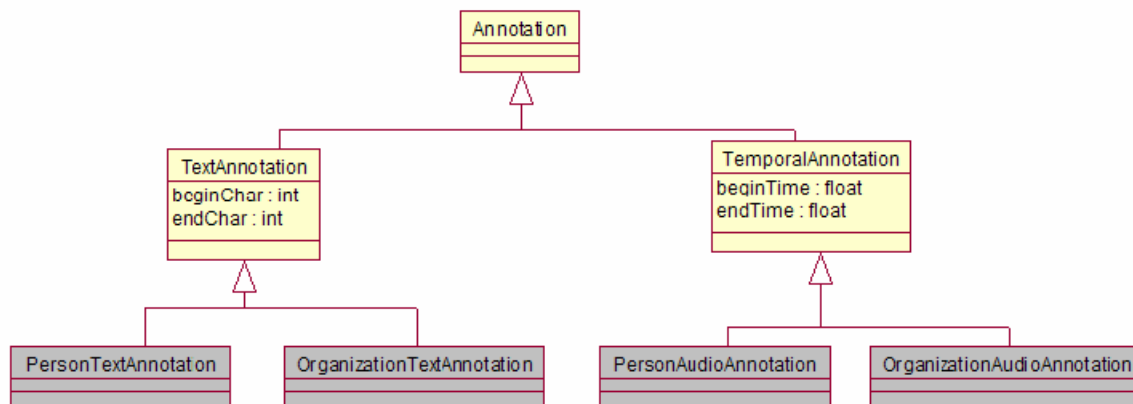
1643
1644

Figure 14: References from Annotations to Regions of the Sofa

B.3.3 Options for Extending Annotation Type System

1645
1646
1647
1648
1649
1650 The standard types in the UIMA Base Type system are very high level. Users will likely wish to extend these base types, for instance to capture the semantics of specific kinds of annotations. There are two options for implementing these extensions. The choice of the extension model for the annotation type system is up to the user and depends on application-specific needs or preferences.

1651
1652
1653
1654
1655
1656
1657 The first option is to subclass the `Annotation` types, as in Figure 15. In this model, the `Annotation` subtype for each modality will be independently subclassed according to the annotation types found in that modality. One advantage of this approach is that all subtype classes remain subtypes of `Annotation`. However, a disadvantage is that types that are annotations of the same semantic class, but for different modalities, are not grouped together in the type system. We see in the figure that an annotation of a reference to a Person or an Organization would have a distinct type depending on the nature of the Sofa the reference occurred in.

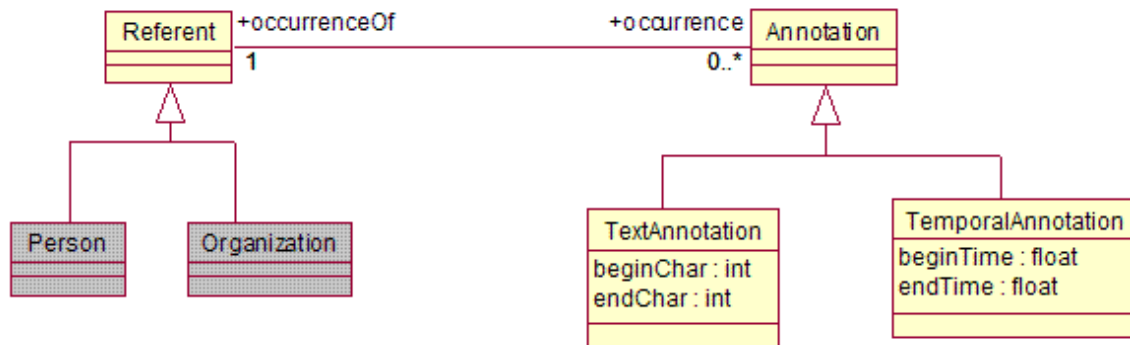


1658
1659

Figure 15: Extending the base type system through subclassing.

1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671

The second option, shown in Figure 16, is to create subtypes of Referent that subsumes the relevant semantic classes, and associate the Annotation with the appropriate Referent type. In this model, an Annotation is viewed as a reference to a Referent in a particular modality. The advantage of this approach is that all annotations corresponding to a particular Referent 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 Referent, so that for instance multiple distinct references to a person in a text can be linked to a single Referent object representing that person.

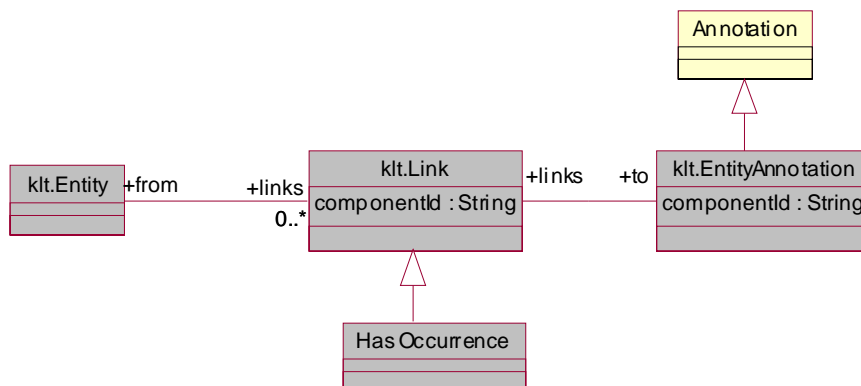


1672
1673
1674

Figure 16: Associate Annotation with Referent type

1675 B.3.4 An Example of Annotation Model Extension

1676 The Base Type System is intended to specify only the top-level classes for the Annotation system used in
1677 an application. Users will need to extend these classes in order to meet the particular needs of their
1678 applications. An example of how an application might extend the base type system comes from
1679 examining the redesign of IBM's Knowledge Level Types [KLT1] in terms of the standard. The current
1680 model in KLT appears in Figure 17. It uses the Annotation class, but subclasses it with its own
1681 EntityAnnotation, models coreference with a reified HasOccurrence link, and captures provenance
1682 through a *componentId* attribute.

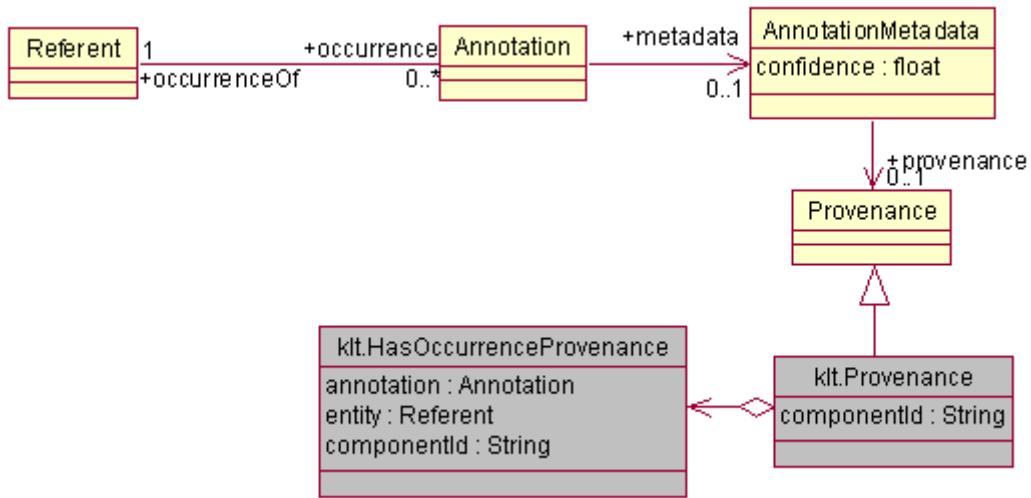


1683
1684
1685

Figure 17: IBM's Knowledge Level Types

1686 Using the standard base type system, this type system could be refactored as in Figure 18. This
1687 refactoring uses the standard definitions of Annotation and Referent. The klt.Link type, which was

1688 used to represent a HasOccurrence link between Entity and Annotation, is replaced by the direct
 1689 occurrence/occurrenceOf features in the standard base type system. Provenance on the occurrence link
 1690 is captured using a subclass of the Provenance type.

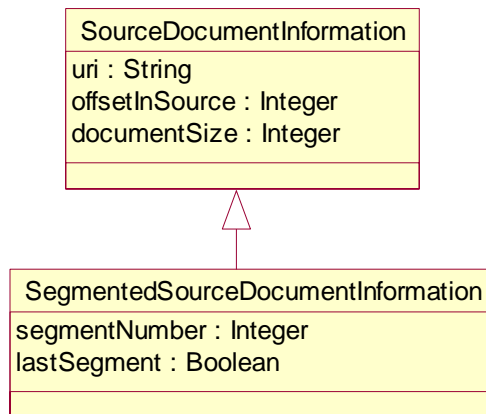


1691
 1692
 1693

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

1694 **B.3.5 Example Extension of Source Document Information**

1695 If an application needs to process multiple segments of an artifact and later merger the results, then
 1696 additional offset information may also be needed on each segment. While not a standard part of the
 1697 specification, a representative extension to the SourceDocumentInformation type to capture such
 1698 information is shown in Figure 19. This SegmentedSourceDocumentInformation type adds features
 1699 to track information about the segment of the source document the CAS corresponds to. Specifically, it
 1700 adds an Integer segmentNumber to capture the segment number of this segment, and a Boolean
 1701 lastSegment that is true when this segment is the last segment derived from the source document.



1702
 1703
 1704

Figure 19: Segmented Source Document Information UML

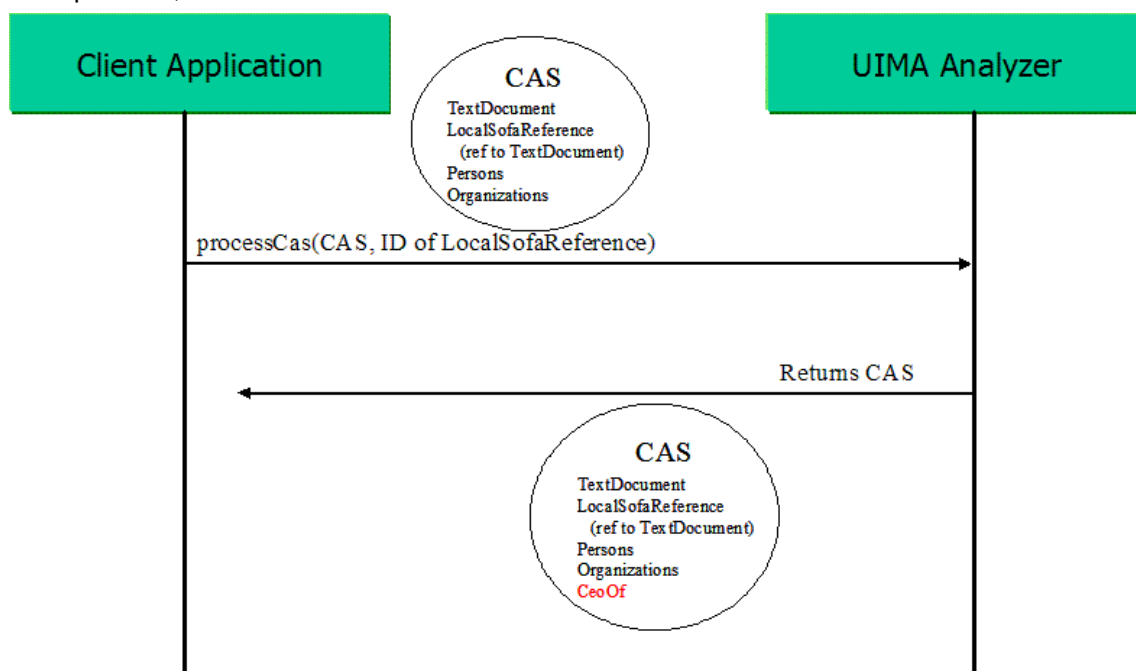
1705 B.4 Abstract Interfaces Examples

1706 B.4.1 Analyzer Example

1707 The sequence diagram in Figure 20 illustrates how a client interacts with a UIMA Analyzer service. In this
1708 example the Analyzer is a “CEO Relation Detector,” which given a text document with Person and
1709 Organization annotations, can find occurrences of CeoOf relationships between them.

1710
1711 The example shows that the client calls the `processCas(cas, sofas)` operation. The first argument is
1712 the CAS to be processed (in XMI format). It contains a `TextDocument`, a `LocalSofaReference` (see
1713 Section 4.3.2.1) that points to a text field in that `TextDocument`, and `Person` and `Organization` annotations
1714 that annotate regions in the `TextDocument`. The second argument is the `xmi:id` of the
1715 `LocalSofaReference` object, indicating that this object should be considered the subject of analysis (Sofa)
1716 for this operation.

1717
1718 The response from the `processCas` operation is a CAS (in XMI format), which in addition to the objects in
1719 the input CAS, also contains `CeoOf` annotations.



1720
1721 **Figure 20: Analyzer Sequence Diagram**

1722 B.4.2 CAS Multiplier Example

1723 The sequence diagram in Figure 21 illustrates how a client interacts with a UIMA CAS Multiplier service.
1724 In this case the CAS Multiplier is a Video Segmenter, which given a video stream divides it into individual
1725 segments.

1726
1727 The client first calls the `inputCas(cas, sofas)` operation. The first argument is a CAS containing a
1728 reference to the video stream to analyze. Typically a large artifact such as a video stream is represented
1729 in the CAS as a reference (using the `RemoteSofaReference` base type introduced in section 4.3.2.1),
1730 rather than included directly in the CAS as is typically done with a text document. The second argument
1731 to `inputCas` is the `xmi:id` of the `RemoteSofaReference` object, so that the service knows that this is the
1732 subject of analysis for this operation.

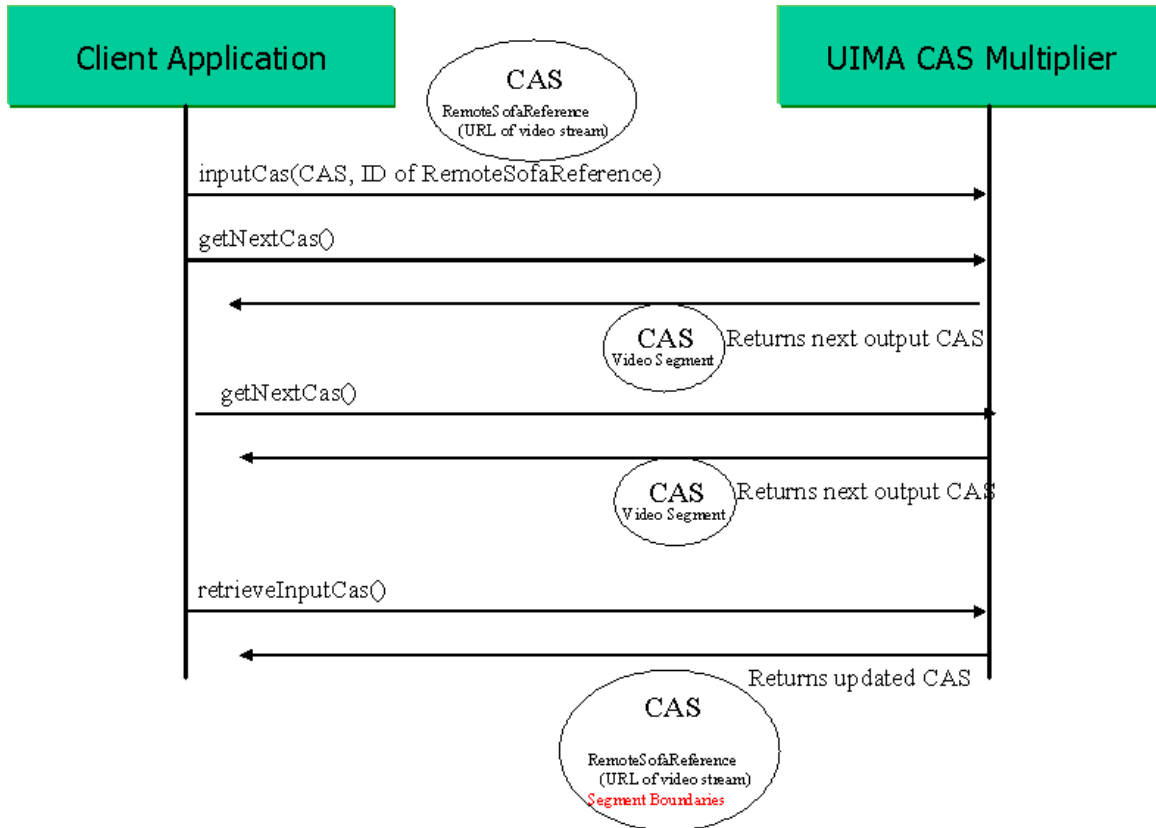
1733

1734 The client then calls the getNextCas operation. This returns a CAS containing the data for the first
1735 segment (or possibly, a reference to it). The client repeatedly calls getNextCas to obtain each
1736 successive segment. Eventually, getNextCas returns null to indicate there are no more segments.

1737

1738 Finally, the client calls the retrieveInputCas operation. This returns the original CAS, with additional
1739 information added. In this example, the Video Segmenter adds information to the original CAS indicating
1740 at what time offsets each of the segment boundaries were detected. Any other information from the
1741 individual segment CASes could also be merged back into the original CAS.

1742



1743

1744

Figure 21: CAS Multiplier Sequence Diagram

1745

1746 Note that a CAS Multiplier may also be used to merge multiple input CASes into one output CAS. Upon
1747 receiving the first inputCas call, the CAS Multiplier would return 0 output CASes and would wait for the
1748 next inputCas call. It would continue to return 0 output CASes until it has seen some number of input
1749 CASes, at which point it would then output the one merged CAS.

1750 B.5 Behavioral Metadata Examples

1751 For each of the Behavioral Metadata Elements (analyzes, required inputs, optional inputs, creates,
1752 modifies, and deletes), there will be a corresponding XML element. For each element a list of type
1753 names is declared.

1754

1755 To address some common situations where an analytic operates on a view (a collection of objects all
1756 referring to the same subject of analysis), we also provide a simple way for behavioral metadata to refer
1757 to views.

1758 **B.5.1 Type Naming Conventions**

1759 In the XML behavioral metadata, type names are represented in the same way as in Ecore and XMI.

1760

1761 In UML (and Ecore), a *Package* is a collection of classes and/or other packages. All classes must be
1762 contained in a package.

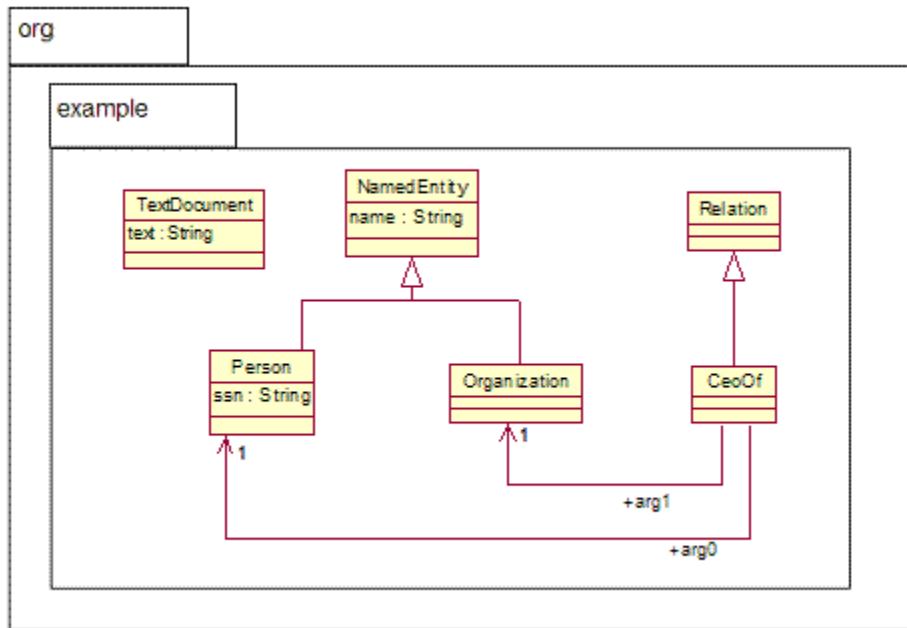
1763

1764 Figure 1 is a UML diagram of an example type system. It depicts a Package “org” containing a Package
1765 “example” containing several classes.

1766

1767

1768



1769

1770

Figure 22: Example Type System UML Model

1771

1772 In the Ecore model, each package is assigned (by the developer) three identifiers: a *name*, a *namespace*
1773 *URI*, and a *namespace prefix*. The *name* is a simple string that must be unique within the containing
1774 package (top-level package names must be globally unique). The namespace URI and namespace prefix
1775 are standard concepts in the XML namespaces spec [2] are used to refer to that package in XML,
1776 including the behavioral metadata as well as the XMI CAS. An example is given below.

1777

1778 Figure 23 shows the relevant parts of the Ecore definition for this type system. Some details have been
1779 omitted (marked with an ellipsis) to show only the parts where packages and namespaces are concerned,
1780 and only a subset of the classes in the diagram are shown.

1781

```

<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

1782

1783

1784

1785 In this example, the namespace URI for the nested “example” project is `http://docs.oasis-`
 1786 `open.org/uima/org/example.ecore`¹, and the corresponding prefix is `org.example`. It is
 1787 important to note that the URI and prefix are arbitrarily determined by the type system developer and
 1788 there is no required mapping from the package names “org” and “example” to the URI and prefix. In the
 1789 above example, the namespace prefix could have been set to “foo” and it would be completely valid.

1790

1791 Now, to refer to a type name within the behavioral metadata XML, we use the namespace URI and prefix
 1792 in the normal XML namespaces way, for example:

1793

```

<behavioralMetadata xmlns:org.example="http://docs.oasis-
open.org/uima/org/example.ecore">
  ...
  <type name="org.example:Person"/>
  ...
</behavioralMetadata>

```

1800

1801 The “xmlns” attribute declares that the prefix “org.example” is bound to the URI `http://docs.oasis-`
 1802 `open.org/uima/org/example.ecore`. Then, each time we want to refer to a type in that package, we use
 1803 the prefix “org.example:”

1804

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

1805 Technically, the XML document does not have to use the same namespace prefix as what is in the Ecore
1806 model. It is only a guideline. The namespace URI is what matters. For example, the above XML is
1807 completely equivalent to the following

```
1808  
1809 <behavioralMetadata xmlns:foo="http://docs.oasis-  
1810 open.org/uima/org/example.ecore">  
1811     ...  
1812     <type name="foo:Person"/>  
1813     ...  
1814 </behavioralMetadata>
```

1815
1816 This is because the namespace URI is a globally unique identifier for the package, but the namespace
1817 prefix need only be unique within the current XML document. For more information on XML namespace
1818 syntax, see [XML1].

1819
1820 The above discussion centered on the representation of type names in XML. When specifying
1821 preconditions, postconditions, and projection conditions (see Section B.5.5), the Object Constraint
1822 Language (OCL) [OCL1] may be used. There is a different representation of type names needed within
1823 OCL expressions. Since OCL is not primarily XML-based, it does not use the XML namespace URIs or
1824 prefixes to refer to packages. Instead, OCL expressions refer directly to the simple package names
1825 separated by double colons, as in “org::example::Person”. For more information see [OCL1].

1826 **B.5.2 XML Syntax for Behavioral Metadata Elements**

1827 The following example is the behavioral metadata for an analytic that analyzes a Sofa of type
1828 `TextDocument`, requires objects of type `Person`, and will inspect objects of type `Organization` if they are
1829 present. It may create objects of type `CeoOf`.

```
1830  
1831 <behavioralMetadata xmlns:org.example="http://docs.oasis-  
1832 open.org/uima/org/example.ecore" excludeReferenceClosure="true">  
1833     <analyzes>  
1834         <type name="org.example:TextDocument"/>  
1835     </analyzes>  
1836     <requiredInputs>  
1837         <type name="org.example:Person"/>  
1838     </requiredInputs>  
1839     <optionalInputs>  
1840         <type name="org.example:Organization"/>  
1841     </optionalInputs>  
1842     <creates>  
1843         <type name="org.example:CeoOf"/>  
1844     </creates>  
1845 </behavioralMetadata>
```

1846
1847 Note that the inheritance hierarchy declared in the type system is respected. So for example a CAS
1848 containing objects of type `GovernmentOfficial` and `Country` would be valid input to this analytic,
1849 assuming that the type system declared these to be subtypes of `org.example:Person` and
1850 `org.example:Place`, respectively.

1851

1852 The `excludeReferenceClosure` attribute on the Behavioral Metadata element, when set to true,
1853 indicates that objects that are referenced from optional/required inputs of this analytic will not be
1854 guaranteed to be included in the CAS passed to the analytic. This attribute defaults to false.

1855

1856 For example, assume in this example the `Person` object had an employer feature of type `Company`. With
1857 `excludeReferenceClosure` set to true, the caller of this analytic is not required to include `Company`
1858 objects in the CAS that is delivered to this analytic. If `Company` objects are filtered then the employer
1859 feature would become null. If `excludeReferenceClosure` were not set, then `Company` objects would be
1860 guaranteed to be included in the CAS.

1861 **B.5.3 Views**

1862 Behavioral Metadata may refer to a View, where a View may collect all annotations referring to a
1863 particular Sofa.

1864

```
1865 <behavioralMetadata xmlns:org.example="http://docs.oasis-  
1866 open.org/uima/org/example.ecore">
```

```
1867   <requiredView sofaType="org.example:TextDocument">
```

```
1868     <requiredInputs>
```

```
1869       <type name="org.example:Token"/>
```

```
1870     </requiredInputs>
```

```
1871     <creates>
```

```
1872       <type name="org.example:Person"/>
```

```
1873     </creates>
```

```
1874   </requiredView>
```

```
1875   <optionalView sofaType="org.example:RawAudio">
```

```
1876     <requiredInputs>
```

```
1877       <type name="org.example:SpeakerBoundary"/>
```

```
1878     </requiredInputs>
```

```
1879     <creates>
```

```
1880       <type name="org.example:AudioPerson"/>
```

```
1881     </creates>
```

```
1882   </optionalView>
```

```
1883 </behavioralMetadata>
```

1884

1885 This example requires a `TextDocument` Sofa and optionally accepts a `RawAudio` Sofa. It has different
1886 input and output types for the different Sofas.

1887

1888 As with an optional input, an “optional view” is one that the analytic would consider if it were present in the
1889 CAS. Views that do not satisfy the required view or optional view expressions might not be delivered to
1890 the analytic.

1891

1892 The meaning of an `optionalView` having a `requiredInput` is that a view not containing the required input
1893 types is not considered to satisfy the `optionalView` expression and might not be delivered to the analytic.

1894

1895 An analytic can also declare that it creates a View along with an associated Sofa and annotations. For
1896 example, this Analytic transcribes audio to text, and also outputs Person annotations over that text:

```
1897  
1898 <behavioralMetadata xmlns:org.example="http://docs.oasis-  
1899 open.org/uima/org/example.ecore">  
1900   <requiredView sofaType="org.example:RawAudio">  
1901     <requiredInputs>  
1902       <type name="org.example:SpeakerBoundary"/>  
1903     </requiredInputs>  
1904   </requiredView>  
1905   <createsView sofaType="org.example:TextDocument">  
1906     <creates>  
1907       <type name="org.example:Person"/>  
1908     </creates>  
1909   </createsView>  
1910 </behavioralMetadata>
```

1911 **B.5.4 Specifying Which Features Are Modified**

1912 For the “modifies” predicate we allow an additional piece of information: the names of the features that
1913 may be modified. This is primarily to support discovery. For example:

```
1914  
1915 <behavioralMetadata xmlns:org.example="http://docs.oasis-  
1916 open.org/uima/org/example.ecore">  
1917   <requiredInputs>  
1918     <type name="org.example:Person"/>  
1919   </requiredInputs>  
1920   <modifies>  
1921     <type name="org.example:Person">  
1922       <feature name="ssn"/>  
1923     </type>  
1924   </modifies>  
1925 </behavioralMetadata>
```

1926
1927 This Analytic inputs `Person` objects and updates their `ssn` features.
1928

1929 **B.5.5 Specifying Preconditions, Postconditions, and Projection Conditions**

1930 Although we expect it to be rare, analytic developers may declare preconditions, postconditions, and
1931 projection conditions directly. The syntax for this is straightforward:

```
1932 <behavioralMetadata>  
1933   <precondition language="OCL"  
1934     expression="exists(s | s.oclKindOf(org::example::Sofa) and  
1935 s.mimeTypeMajor = 'audio')"/>  
1936   <postcondition language="OCL"
```

```

1938         expr="exists(p | p.oclKindOf(org::example::Sofa) and s.mimeTypeMajor =
1939 'text')"/>
1940     <projectionCondition language="OCL"
1941         expr=" select(p | p.oclKindOf(org::example::NamedEntity))"/>
1942 </behavioralMetadata>

```

1943

1944 UIMA does not define what language must be used for expression these conditions. OCL is just one
1945 example.

1946

1947 Preconditions and postconditions are expressions that evaluate to a Boolean value. Projection conditions
1948 are expressions that evaluate to a collection of objects.

1949

1950 Behavioral Metadata can include these conditions as well as the other elements (analyzes,
1951 requiredInputs, etc.). In that case, the overall precondition and postcondition of the analytic are a
1952 combination of the user-specified conditions and the conditions derived from the other behavioral
1953 metadata elements as described in the next section. (For precondition and postcondition it is a
1954 conjunction; for projection condition it is a union.)

1955

1956 **B.6 Processing Element Metadata Example**

1957 The following XML fragment is an example of Processing Element Metadata for a “CeoOf Relation
1958 Detector” analytic.

```

1959 <pemd:ProcessingElementMetadata xmi:version="2.0"
1960 xmlns:xmi="http://www.omg.org/XMI" xmlns:pemd="http://docs.oasis-
1961 open.org/uima/ns/peMetadata.ecore">
1962     <identification
1963         symbolicName="org.oasis-open.uima.example.CeoRelationAnnotator"
1964         name="Ceo Relation Annotator"
1965         description="Detects CeoOf relationships between Persons and
1966 Organizations in a text document."
1967         vendor="OASIS"
1968         version="1.0.0"/>
1969
1970     <configurationParameter
1971         name="PatternFile"
1972         description="Location of external file containing patterns that
1973 indicate a CeoOf relation in text."
1974         type="ResourceURL">
1975         <defaultValue>myResources/ceoPatterns.dat</defaultValue>
1976     </configurationParameter>
1977
1978     <typeSystem
1979         reference="http://docs.oasis-
1980 open.org/uima/types/exampleTypeSystem.ecore"/>
1981
1982     <behavioralMetadata>
1983         <analyzes>
1984             <type name="org.example:Document"/>
1985         </analyzes>
1986         <requiredInputs>
1987             <type name="org.example:Person"/>
1988             <type name="org.example:Organization"/>
1989         </requiredInputs>

```



```

1990     <creates>
1991         <type name="org.example:CeoOf" />
1992     </creates>
1993 </behavioralMetadata>
1994
1995     <extension extenderId="org.apache.uima">
1996         ...
1997     </extension>
1998 </pemd:ProcessingElementMetadata>

```

1999 B.7 SOAP Service Example

2000 Returning to our example of the CEO Relation Detector analytic, this section gives examples of SOAP
 2001 messages used to send a CAS to and from the analytic.

2002

2003 The processCas request message is shown here:

```

2004 <soapenv:Envelope...>
2005   <soapenv:Body>
2006     <processCas xmlns="">
2007       <cas xmi:version="2.0" ... >
2008         <org.example:Document xmi:id="1"
2009           text="Fred Center is the CEO of Center Micros." />
2010         <base:LocalSofaReference xmi:id="2" sofaObject="1"
2011         sofaFeature="text" />
2012         <org.example:Person xmi:id="3" sofa="2" begin="0" end="11" />
2013         <org.example:Organization xmi:id="4" sofa="2" begin="26" end="39" />
2014       </cas>
2015       <sofas objects="1" />
2016     </processCas>
2017   </soapenv:Body>
2018 </soapenv:Envelope>

```

2019 This message is simply an XMI CAS wrapped in an appropriate SOAP envelope, indicating which
 2020 operation is being invoked (processCas).

2021

2022 The processCas response message returned from the service is shown here:

2023

```

2024 <soapenv:Envelope...>
2025   <soapenv:Body>
2026     <processCas xmlns="">
2027       <cas xmi:version="2.0" ... >
2028         <org.example:Document xmi:id="1"
2029           text="Fred Center is the CEO of Center Micros." />
2030         <base:SofaReference xmi:id="2" sofaObject="1" sofaFeature="text" />
2031         <org.example:Person xmi:id="3" sofa="2" begin="0" end="11" />
2032         <org.example:Organization xmi:id="4" sofa="2" begin="26" end="39" />
2033         <org.example:CeoOf xmi:id="5" sofa="2" begin="0" end="31" arg0="3"
2034         arg1="4" />
2035       </cas>
2036     </processCas>
2037   </soapenv:Body>
2038 </soapenv:Envelope>

```

2039 Again this is just an XMI CAS wrapped in a SOAP envelope. Note that the "CeoOf" object has been
 2040 added to the CAS.

2041

2042 Alternatively, the service could have responded with a “delta” using the XMI differences language. Here
2043 is an example:

```
2044 <soapenv:Envelope...>  
2045   <soapenv:Body>  
2046     <processCas xmlns="">  
2047       <cas xmi:version="2.0" ... >  
2048         <xmi:Difference>  
2049           <target href="input.xmi"/>  
2050           <xmi:Add addition="5">  
2051             </xmi:Difference>  
2052             <org.example:CeoOf xmi:id="5" sofa="2" begin="0" end="31" arg0="3"  
2053 arg1="4"/>  
2054           </cas>  
2055         </processCas>  
2056       </soapenv:Body>  
2057     </soapenv:Envelope>
```

2058
2059 Note that the `target` element is defined in the XMI specification to hold an href to the original XMI file to
2060 which these differences will get applied. In UIMA we don't really have a URI for that - it is just the input to
2061 the Process CAS Request. The example conventionally uses `input.xmi` for this URI.

2062

C. Formal Specification Artifacts

2063 This section includes artifacts such as Ecore models and XML Schemata, which formally define elements
2064 of the UIMA specification.

2065 C.1 XMI XML Schema

2066 This XML schema is defined by the XMI specification [XMI1] and repeated here for completeness:

2067

```
2068 <?xml version="1.0" encoding="UTF-8"?>
2069 <xsd:schema xmlns:xmi="http://www.omg.org/XMI"
2070   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
2071   targetNamespace="http://www.omg.org/XMI">
2072   <xsd:attribute name="id" type="xsd:ID"/>
2073   <xsd:attributeGroup name="IdentityAttribs">
2074     <xsd:attribute form="qualified" name="label" type="xsd:string"
2075       use="optional"/>
2076     <xsd:attribute form="qualified" name="uuid" type="xsd:string"
2077       use="optional"/>
2078   </xsd:attributeGroup>
2079   <xsd:attributeGroup name="LinkAttribs">
2080     <xsd:attribute name="href" type="xsd:string" use="optional"/>
2081     <xsd:attribute form="qualified" name="idref" type="xsd:IDREF"
2082       use="optional"/>
2083   </xsd:attributeGroup>
2084   <xsd:attributeGroup name="ObjectAttribs">
2085     <xsd:attributeGroup ref="xmi:IdentityAttribs"/>
2086     <xsd:attributeGroup ref="xmi:LinkAttribs"/>
2087     <xsd:attribute fixed="2.0" form="qualified" name="version"
2088       type="xsd:string" use="optional"/>
2089     <xsd:attribute form="qualified" name="type" type="xsd:QName"
2090       use="optional"/>
2091   </xsd:attributeGroup>
2092   <xsd:complexType name="XMI">
2093     <xsd:choice maxOccurs="unbounded" minOccurs="0">
2094       <xsd:any processContents="strict"/>
2095     </xsd:choice>
2096     <xsd:attributeGroup ref="xmi:IdentityAttribs"/>
2097     <xsd:attributeGroup ref="xmi:LinkAttribs"/>
2098     <xsd:attribute form="qualified" name="type" type="xsd:QName"
2099       use="optional"/>
2100     <xsd:attribute fixed="2.0" form="qualified" name="version"
2101       type="xsd:string" use="required"/>
```

```

2102 </xsd:complexType>
2103 <xsd:element name="XMI" type="xmi:XMI"/>
2104 <xsd:complexType name="PackageReference">
2105   <xsd:choice maxOccurs="unbounded" minOccurs="0">
2106     <xsd:element name="name" type="xsd:string"/>
2107     <xsd:element name="version" type="xsd:string"/>
2108   </xsd:choice>
2109   <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
2110   <xsd:attribute name="name" type="xsd:string" use="optional"/>
2111 </xsd:complexType>
2112 <xsd:element name="PackageReference"
2113   type="xmi:PackageReference"/>
2114 <xsd:complexType name="Model">
2115   <xsd:complexContent>
2116     <xsd:extension base="xmi:PackageReference"/>
2117   </xsd:complexContent>
2118 </xsd:complexType>
2119 <xsd:element name="Model" type="xmi:Model"/>
2120 <xsd:complexType name="Import">
2121   <xsd:complexContent>
2122     <xsd:extension base="xmi:PackageReference"/>
2123   </xsd:complexContent>
2124 </xsd:complexType>
2125 <xsd:element name="Import" type="xmi:Import"/>
2126 <xsd:complexType name="MetaModel">
2127   <xsd:complexContent>
2128     <xsd:extension base="xmi:PackageReference"/>
2129   </xsd:complexContent>
2130 </xsd:complexType>
2131 <xsd:element name="MetaModel" type="xmi:MetaModel"/>
2132 <xsd:complexType name="Documentation">
2133   <xsd:choice maxOccurs="unbounded" minOccurs="0">
2134     <xsd:element name="contact" type="xsd:string"/>
2135     <xsd:element name="exporter" type="xsd:string"/>
2136     <xsd:element name="exporterVersion" type="xsd:string"/>
2137     <xsd:element name="longDescription" type="xsd:string"/>
2138     <xsd:element name="shortDescription" type="xsd:string"/>
2139     <xsd:element name="notice" type="xsd:string"/>
2140     <xsd:element name="owner" type="xsd:string"/>
2141   </xsd:choice>
2142   <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
2143   <xsd:attribute name="contact" type="xsd:string" use="optional"/>
2144   <xsd:attribute name="exporter" type="xsd:string"

```

```

2145         use="optional"/>
2146     <xsd:attribute name="exporterVersion" type="xsd:string"
2147         use="optional"/>
2148     <xsd:attribute name="longDescription" type="xsd:string"
2149         use="optional"/>
2150     <xsd:attribute name="shortDescription" type="xsd:string"
2151         use="optional"/>
2152     <xsd:attribute name="notice" type="xsd:string" use="optional"/>
2153     <xsd:attribute name="owner" type="xsd:string" use="optional"/>
2154 </xsd:complexType>
2155 <xsd:element name="Documentation" type="xmi:Documentation"/>
2156 <xsd:complexType name="Extension">
2157     <xsd:choice maxOccurs="unbounded" minOccurs="0">
2158         <xsd:any processContents="lax"/>
2159     </xsd:choice>
2160     <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
2161     <xsd:attribute name="extender" type="xsd:string"
2162         use="optional"/>
2163     <xsd:attribute name="extenderID" type="xsd:string"
2164         use="optional"/>
2165 </xsd:complexType>
2166 <xsd:element name="Extension" type="xmi:Extension"/>
2167 <xsd:complexType name="Difference">
2168     <xsd:choice maxOccurs="unbounded" minOccurs="0">
2169         <xsd:element name="target">
2170             <xsd:complexType>
2171                 <xsd:choice maxOccurs="unbounded" minOccurs="0">
2172                     <xsd:any processContents="skip"/>
2173                 </xsd:choice>
2174                 <xsd:anyAttribute processContents="skip"/>
2175             </xsd:complexType>
2176         </xsd:element>
2177         <xsd:element name="difference" type="xmi:Difference"/>
2178         <xsd:element name="container" type="xmi:Difference"/>
2179     </xsd:choice>
2180     <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
2181     <xsd:attribute name="target" type="xsd:IDREFS" use="optional"/>
2182     <xsd:attribute name="container" type="xsd:IDREFS"
2183         use="optional"/>
2184 </xsd:complexType>
2185 <xsd:element name="Difference" type="xmi:Difference"/>
2186 <xsd:complexType name="Add">
2187     <xsd:complexContent>

```

```

2188     <xsd:extension base="xmi:Difference">
2189         <xsd:attribute name="position" type="xsd:string"
2190             use="optional"/>
2191         <xsd:attribute name="addition" type="xsd:IDREFS"
2192             use="optional"/>
2193     </xsd:extension>
2194 </xsd:complexContent>
2195 </xsd:complexType>
2196 <xsd:element name="Add" type="xmi:Add"/>
2197 <xsd:complexType name="Replace">
2198     <xsd:complexContent>
2199         <xsd:extension base="xmi:Difference">
2200             <xsd:attribute name="position" type="xsd:string"
2201                 use="optional"/>
2202             <xsd:attribute name="replacement" type="xsd:IDREFS"
2203                 use="optional"/>
2204         </xsd:extension>
2205     </xsd:complexContent>
2206 </xsd:complexType>
2207 <xsd:element name="Replace" type="xmi:Replace"/>
2208 <xsd:complexType name="Delete">
2209     <xsd:complexContent>
2210         <xsd:extension base="xmi:Difference"/>
2211     </xsd:complexContent>
2212 </xsd:complexType>
2213 <xsd:element name="Delete" type="xmi:Delete"/>
2214 <xsd:complexType name="Any">
2215     <xsd:choice maxOccurs="unbounded" minOccurs="0">
2216         <xsd:any processContents="skip"/>
2217     </xsd:choice>
2218     <xsd:anyAttribute processContents="skip"/>
2219 </xsd:complexType>
2220 </xsd:schema>

```

2221 C.2 Ecore XML Schema

2222 This XML schema is defined by Ecore [EMF1] and repeated here for completeness:

```

2223 <?xml version="1.0" encoding="UTF-8"?>
2224 <xsd:schema xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore"
2225     xmlns:xmi="http://www.omg.org/XMI"
2226     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
2227     targetNamespace="http://www.eclipse.org/emf/2002/Ecore">
2228     <xsd:import namespace="http://www.omg.org/XMI" schemaLocation="XMI.xsd"/>
2229     <xsd:complexType name="EAttribute">
2230     <xsd:complexContent>

```

```

2231     <xsd:extension base="ecore:EStructuralFeature">
2232         <xsd:attribute name="id" type="xsd:boolean"/>
2233     </xsd:extension>
2234 </xsd:complexContent>
2235 </xsd:complexType>
2236 <xsd:element name="EAttribute" type="ecore:EAttribute"/>
2237 <xsd:complexType name="EAnnotation">
2238     <xsd:complexContent>
2239         <xsd:extension base="ecore:EModelElement">
2240             <xsd:choice maxOccurs="unbounded" minOccurs="0">
2241                 <xsd:element name="details" type="ecore:EStringToStringMapEntry"/>
2242                 <xsd:element name="contents" type="ecore:EObject"/>
2243                 <xsd:element name="references" type="ecore:EObject"/>
2244             </xsd:choice>
2245             <xsd:attribute name="source" type="xsd:string"/>
2246             <xsd:attribute name="references" type="xsd:string"/>
2247         </xsd:extension>
2248     </xsd:complexContent>
2249 </xsd:complexType>
2250 <xsd:element name="EAnnotation" type="ecore:EAnnotation"/>
2251 <xsd:complexType name="EClass">
2252     <xsd:complexContent>
2253         <xsd:extension base="ecore:EClassifier">
2254             <xsd:choice maxOccurs="unbounded" minOccurs="0">
2255                 <xsd:element name="eSuperTypes" type="ecore:EClass"/>
2256                 <xsd:element name="eOperations" type="ecore:EOperation"/>
2257                 <xsd:element name="eStructuralFeatures"
2258 type="ecore:EStructuralFeature"/>
2259             </xsd:choice>
2260             <xsd:attribute name="abstract" type="xsd:boolean"/>
2261             <xsd:attribute name="interface" type="xsd:boolean"/>
2262             <xsd:attribute name="eSuperTypes" type="xsd:string"/>
2263         </xsd:extension>
2264     </xsd:complexContent>
2265 </xsd:complexType>
2266 <xsd:element name="EClass" type="ecore:EClass"/>
2267 <xsd:complexType abstract="true" name="EClassifier">
2268     <xsd:complexContent>
2269         <xsd:extension base="ecore:ENamedElement">
2270             <xsd:attribute name="instanceClassName" type="xsd:string"/>
2271         </xsd:extension>
2272     </xsd:complexContent>
2273 </xsd:complexType>

```

```

2274 <xsd:element name="EClassifier" type="ecore:EClassifier"/>
2275 <xsd:complexType name="EDatatype">
2276   <xsd:complexContent>
2277     <xsd:extension base="ecore:EClassifier">
2278       <xsd:attribute name="serializable" type="xsd:boolean"/>
2279     </xsd:extension>
2280   </xsd:complexContent>
2281 </xsd:complexType>
2282 <xsd:element name="EDatatype" type="ecore:EDatatype"/>
2283 <xsd:complexType name="EEnum">
2284   <xsd:complexContent>
2285     <xsd:extension base="ecore:EDatatype">
2286       <xsd:choice maxOccurs="unbounded" minOccurs="0">
2287         <xsd:element name="eLiterals" type="ecore:EEnumLiteral"/>
2288       </xsd:choice>
2289     </xsd:extension>
2290   </xsd:complexContent>
2291 </xsd:complexType>
2292 <xsd:element name="EEnum" type="ecore:EEnum"/>
2293 <xsd:complexType name="EEnumLiteral">
2294   <xsd:complexContent>
2295     <xsd:extension base="ecore:ENamedElement">
2296       <xsd:attribute name="value" type="xsd:int"/>
2297       <xsd:attribute name="literal" type="xsd:string"/>
2298     </xsd:extension>
2299   </xsd:complexContent>
2300 </xsd:complexType>
2301 <xsd:element name="EEnumLiteral" type="ecore:EEnumLiteral"/>
2302 <xsd:complexType name="EFactory">
2303   <xsd:complexContent>
2304     <xsd:extension base="ecore:EModelElement"/>
2305   </xsd:complexContent>
2306 </xsd:complexType>
2307 <xsd:element name="EFactory" type="ecore:EFactory"/>
2308 <xsd:complexType abstract="true" name="EModelElement">
2309   <xsd:complexContent>
2310     <xsd:extension base="ecore:EObject">
2311       <xsd:choice maxOccurs="unbounded" minOccurs="0">
2312         <xsd:element name="eAnnotations" type="ecore:EAnnotation"/>
2313       </xsd:choice>
2314     </xsd:extension>
2315   </xsd:complexContent>
2316 </xsd:complexType>

```



```

2317 <xsd:element name="EModelElement" type="ecore:EModelElement"/>
2318 <xsd:complexType abstract="true" name="ENamedElement">
2319   <xsd:complexContent>
2320     <xsd:extension base="ecore:EModelElement">
2321       <xsd:attribute name="name" type="xsd:string"/>
2322     </xsd:extension>
2323   </xsd:complexContent>
2324 </xsd:complexType>
2325 <xsd:element name="ENamedElement" type="ecore:ENamedElement"/>
2326 <xsd:complexType name="EObject">
2327   <xsd:choice maxOccurs="unbounded" minOccurs="0">
2328     <xsd:element ref="xmi:Extension"/>
2329   </xsd:choice>
2330   <xsd:attribute ref="xmi:id"/>
2331   <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
2332 </xsd:complexType>
2333 <xsd:element name="EObject" type="ecore:EObject"/>
2334 <xsd:complexType name="EOperation">
2335   <xsd:complexContent>
2336     <xsd:extension base="ecore:ETypedElement">
2337       <xsd:choice maxOccurs="unbounded" minOccurs="0">
2338         <xsd:element name="eParameters" type="ecore:EParameter"/>
2339         <xsd:element name="eExceptions" type="ecore:EClassifier"/>
2340       </xsd:choice>
2341       <xsd:attribute name="eExceptions" type="xsd:string"/>
2342     </xsd:extension>
2343   </xsd:complexContent>
2344 </xsd:complexType>
2345 <xsd:element name="EOperation" type="ecore:EOperation"/>
2346 <xsd:complexType name="EPackage">
2347   <xsd:complexContent>
2348     <xsd:extension base="ecore:ENamedElement">
2349       <xsd:choice maxOccurs="unbounded" minOccurs="0">
2350         <xsd:element name="eClassifiers" type="ecore:EClassifier"/>
2351         <xsd:element name="eSubpackages" type="ecore:EPackage"/>
2352       </xsd:choice>
2353       <xsd:attribute name="nsURI" type="xsd:string"/>
2354       <xsd:attribute name="nsPrefix" type="xsd:string"/>
2355     </xsd:extension>
2356   </xsd:complexContent>
2357 </xsd:complexType>
2358 <xsd:element name="EPackage" type="ecore:EPackage"/>
2359 <xsd:complexType name="EParameter">

```

```

2360     <xsd:complexContent>
2361         <xsd:extension base="ecore:ETypedElement" />
2362     </xsd:complexContent>
2363 </xsd:complexType>
2364 <xsd:element name="EParameter" type="ecore:EParameter" />
2365 <xsd:complexType name="EReference">
2366     <xsd:complexContent>
2367         <xsd:extension base="ecore:EStructuralFeature">
2368             <xsd:choice maxOccurs="unbounded" minOccurs="0">
2369                 <xsd:element name="eOpposite" type="ecore:EReference" />
2370             </xsd:choice>
2371             <xsd:attribute name="containment" type="xsd:boolean" />
2372             <xsd:attribute name="resolveProxies" type="xsd:boolean" />
2373             <xsd:attribute name="eOpposite" type="xsd:string" />
2374         </xsd:extension>
2375     </xsd:complexContent>
2376 </xsd:complexType>
2377 <xsd:element name="EReference" type="ecore:EReference" />
2378 <xsd:complexType abstract="true" name="EStructuralFeature">
2379     <xsd:complexContent>
2380         <xsd:extension base="ecore:ETypedElement">
2381             <xsd:attribute name="changeable" type="xsd:boolean" />
2382             <xsd:attribute name="volatile" type="xsd:boolean" />
2383             <xsd:attribute name="transient" type="xsd:boolean" />
2384             <xsd:attribute name="defaultValueLiteral" type="xsd:string" />
2385             <xsd:attribute name="unsettable" type="xsd:boolean" />
2386             <xsd:attribute name="derived" type="xsd:boolean" />
2387         </xsd:extension>
2388     </xsd:complexContent>
2389 </xsd:complexType>
2390 <xsd:element name="EStructuralFeature" type="ecore:EStructuralFeature" />
2391 <xsd:complexType abstract="true" name="ETypedElement">
2392     <xsd:complexContent>
2393         <xsd:extension base="ecore:ENamedElement">
2394             <xsd:choice maxOccurs="unbounded" minOccurs="0">
2395                 <xsd:element name="eType" type="ecore:EClassifier" />
2396             </xsd:choice>
2397             <xsd:attribute name="ordered" type="xsd:boolean" />
2398             <xsd:attribute name="unique" type="xsd:boolean" />
2399             <xsd:attribute name="lowerBound" type="xsd:int" />
2400             <xsd:attribute name="upperBound" type="xsd:int" />
2401             <xsd:attribute name="eType" type="xsd:string" />
2402         </xsd:extension>

```

```

2403     </xsd:complexContent>
2404 </xsd:complexType>
2405 <xsd:element name="ETypedElement" type="ecore:ETypedElement"/>
2406 <xsd:complexType name="EStringToStringMapEntry">
2407     <xsd:choice maxOccurs="unbounded" minOccurs="0">
2408         <xsd:element ref="xmi:Extension"/>
2409     </xsd:choice>
2410     <xsd:attribute ref="xmi:id"/>
2411     <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
2412     <xsd:attribute name="key" type="xsd:string"/>
2413     <xsd:attribute name="value" type="xsd:string"/>
2414 </xsd:complexType>
2415 <xsd:element name="EStringToStringMapEntry"
2416 type="ecore:EStringToStringMapEntry"/>
2417 </xsd:schema>
2418

```

2419 C.3 Base Type System Ecore Model

2420 This Ecore model formally defines the UIMA Base Type System.

```

2421 <?xml version="1.0" encoding="UTF-8"?>
2422 <ecore:EPackage xmi:version="2.0"
2423     xmlns:xmi="http://www.omg.org/XMI"
2424     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2425     xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore" name="uima"
2426     nsURI="http://docs.oasis-open.org/uima/ns/uima.ecore" nsPrefix="uima">
2427     <eSubpackages name="base" nsURI="http://docs.oasis-
2428 open.org/uima/ns/base.ecore" nsPrefix="uima.base">
2429         <eClassifiers xsi:type="ecore:EClass" name="Annotation">
2430             <eStructuralFeatures xsi:type="ecore:EReference" name="sofa"
2431 lowerBound="1"
2432     eType="#//base/SofaReference"/>
2433             <eStructuralFeatures xsi:type="ecore:EReference" name="metadata"
2434 eType="#//base/AnnotationMetadata"/>
2435             <eStructuralFeatures xsi:type="ecore:EReference" name="occurrenceOf"
2436 lowerBound="1"
2437     eType="#//base/Entity" eOpposite="#//base/Entity/occurrence"/>
2438         </eClassifiers>
2439         <eClassifiers xsi:type="ecore:EClass" name="SofaReference"
2440 abstract="true"/>
2441         <eClassifiers xsi:type="ecore:EClass" name="LocalSofaReference"
2442 eSuperTypes="#//base/SofaReference">
2443             <eStructuralFeatures xsi:type="ecore:EAttribute" name="sofaFeature"
2444 eType="ecore:EDatatype http://www.eclipse.org/emf/2002/Ecore#//EString"/>
2445             <eStructuralFeatures xsi:type="ecore:EReference" name="sofaObject"
2446 eType="ecore:EClass http://www.eclipse.org/emf/2002/Ecore#//EObject"/>
2447         </eClassifiers>
2448         <eClassifiers xsi:type="ecore:EClass" name="RemoteSofaReference"
2449 eSuperTypes="#//base/SofaReference">
2450             <eStructuralFeatures xsi:type="ecore:EAttribute" name="sofaUri"
2451 eType="ecore:EDatatype http://www.eclipse.org/emf/2002/Ecore#//EString"/>
2452         </eClassifiers>

```

```

2453     <eClassifiers xsi:type="ecore:EClass" name="TextAnnotation"
2454 eSuperTypes="#//base/Annotation">
2455     <eStructuralFeatures xsi:type="ecore:EAttribute" name="beginChar"
2456 eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EInt"/>
2457     <eStructuralFeatures xsi:type="ecore:EAttribute" name="endChar"
2458 eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EInt"/>
2459     </eClassifiers>
2460     <eClassifiers xsi:type="ecore:EClass" name="TemporalAnnotation"
2461 eSuperTypes="#//base/Annotation">
2462     <eStructuralFeatures xsi:type="ecore:EAttribute" name="beginTime"
2463 eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EFloat"/>
2464     <eStructuralFeatures xsi:type="ecore:EAttribute" name="endTime"
2465 eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EFloat"/>
2466     </eClassifiers>
2467     <eClassifiers xsi:type="ecore:EClass" name="AnnotationMetadata">
2468     <eStructuralFeatures xsi:type="ecore:EAttribute" name="confidence"
2469 eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EFloat"/>
2470     <eStructuralFeatures xsi:type="ecore:EReference" name="provenance"
2471 eType="#//base/Provenance"/>
2472     </eClassifiers>
2473     <eClassifiers xsi:type="ecore:EClass" name="Provenance"/>
2474     <eClassifiers xsi:type="ecore:EClass" name="Entity">
2475     <eStructuralFeatures xsi:type="ecore:EReference" name="occurrence"
2476 upperBound="-1"
2477     eType="#//base/Annotation"
2478 eOpposite="#//base/Annotation/occurrenceOf"/>
2479     </eClassifiers>
2480     <eClassifiers xsi:type="ecore:EClass" name="SourceDocumentInformation">
2481     <eStructuralFeatures xsi:type="ecore:EAttribute" name="uri"
2482 eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
2483     <eStructuralFeatures xsi:type="ecore:EAttribute" name="offsetInSource"
2484 eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EInt"/>
2485     <eStructuralFeatures xsi:type="ecore:EAttribute" name="documentSize"
2486 eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EInt"/>
2487     </eClassifiers>
2488     <eClassifiers xsi:type="ecore:EClass" name="AnchoredView"
2489 eSuperTypes="#//base/View">
2490     <eStructuralFeatures xsi:type="ecore:EReference" name="sofa"
2491 upperBound="-1"
2492     eType="#//base/SofaReference"/>
2493     </eClassifiers>
2494     <eClassifiers xsi:type="ecore:EClass" name="View">
2495     <eStructuralFeatures xsi:type="ecore:EReference" name="IndexRepository"
2496 lowerBound="1"/>
2497     <eStructuralFeatures xsi:type="ecore:EReference" name="member"
2498 upperBound="-1"
2499     eType="ecore:EClass
2500 http://www.eclipse.org/emf/2002/Ecore#//EObject"/>
2501     <eStructuralFeatures xsi:type="ecore:EAttribute" name="name"
2502 eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
2503     </eClassifiers>
2504     </eSubpackages>
2505 </ecore:EPackage>

```

2506 C.4 Base Type System XML Schema

2507 This XML schema was generated from the Ecore model in Appendix C.3 by the Eclipse Modeling
2508 Framework tools.

```

2509 <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2510 <xsd:schema xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore"
2511 xmlns:uima.base="http://docs.oasis-open.org/uima/ns/base.ecore"
2512 xmlns:xmi="http://www.omg.org/XMI"
2513 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
2514 targetNamespace="http://docs.oasis-open.org/uima/ns/base.ecore">
2515   <xsd:import namespace="http://www.eclipse.org/emf/2002/Ecore"
2516   schemaLocation="ecore.xsd"/>
2517   <xsd:import namespace="http://www.omg.org/XMI"
2518   schemaLocation="../../../../plugin/org.eclipse.emf.ecore/model/XMI.xsd"/>
2519   <xsd:complexType name="Annotation">
2520     <xsd:choice maxOccurs="unbounded" minOccurs="0">
2521       <xsd:element name="sofa" type="uima.base:SofaReference"/>
2522       <xsd:element name="metadata" type="uima.base:AnnotationMetadata"/>
2523       <xsd:element name="occurrenceOf" type="uima.base:Entity"/>
2524       <xsd:element ref="xmi:Extension"/>
2525     </xsd:choice>
2526     <xsd:attribute ref="xmi:id"/>
2527     <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
2528     <xsd:attribute name="sofa" type="xsd:string"/>
2529     <xsd:attribute name="metadata" type="xsd:string"/>
2530     <xsd:attribute name="occurrenceOf" type="xsd:string"/>
2531   </xsd:complexType>
2532   <xsd:element name="Annotation" type="uima.base:Annotation"/>
2533   <xsd:complexType abstract="true" name="SofaReference">
2534     <xsd:choice maxOccurs="unbounded" minOccurs="0">
2535       <xsd:element ref="xmi:Extension"/>
2536     </xsd:choice>
2537     <xsd:attribute ref="xmi:id"/>
2538     <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
2539   </xsd:complexType>
2540   <xsd:element name="SofaReference" type="uima.base:SofaReference"/>
2541   <xsd:complexType name="LocalSofaReference">
2542     <xsd:complexContent>
2543       <xsd:extension base="uima.base:SofaReference">
2544         <xsd:choice maxOccurs="unbounded" minOccurs="0">
2545           <xsd:element name="sofaObject" type="ecore:EObject"/>
2546         </xsd:choice>
2547         <xsd:attribute name="sofaFeature" type="xsd:string"/>
2548         <xsd:attribute name="sofaObject" type="xsd:string"/>
2549       </xsd:extension>
2550     </xsd:complexContent>
2551   </xsd:complexType>
2552   <xsd:element name="LocalSofaReference"
2553   type="uima.base:LocalSofaReference"/>
2554   <xsd:complexType name="RemoteSofaReference">
2555     <xsd:complexContent>
2556       <xsd:extension base="uima.base:SofaReference">
2557         <xsd:attribute name="sofaUri" type="xsd:string"/>
2558       </xsd:extension>
2559     </xsd:complexContent>
2560   </xsd:complexType>
2561   <xsd:element name="RemoteSofaReference"
2562   type="uima.base:RemoteSofaReference"/>
2563   <xsd:complexType name="TextAnnotation">
2564     <xsd:complexContent>
2565       <xsd:extension base="uima.base:Annotation">
2566         <xsd:attribute name="beginChar" type="xsd:int"/>

```

```

2567         <xsd:attribute name="endChar" type="xsd:int"/>
2568     </xsd:extension>
2569 </xsd:complexContent>
2570 </xsd:complexType>
2571 <xsd:element name="TextAnnotation" type="uima.base:TextAnnotation"/>
2572 <xsd:complexType name="TemporalAnnotation">
2573     <xsd:complexContent>
2574         <xsd:extension base="uima.base:Annotation">
2575             <xsd:attribute name="beginTime" type="xsd:float"/>
2576             <xsd:attribute name="endTime" type="xsd:float"/>
2577         </xsd:extension>
2578     </xsd:complexContent>
2579 </xsd:complexType>
2580 <xsd:element name="TemporalAnnotation"
2581 type="uima.base:TemporalAnnotation"/>
2582 <xsd:complexType name="AnnotationMetadata">
2583     <xsd:choice maxOccurs="unbounded" minOccurs="0">
2584         <xsd:element name="provenance" type="uima.base:Provenance"/>
2585         <xsd:element ref="xmi:Extension"/>
2586     </xsd:choice>
2587     <xsd:attribute ref="xmi:id"/>
2588     <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
2589     <xsd:attribute name="confidence" type="xsd:float"/>
2590     <xsd:attribute name="provenance" type="xsd:string"/>
2591 </xsd:complexType>
2592 <xsd:element name="AnnotationMetadata"
2593 type="uima.base:AnnotationMetadata"/>
2594 <xsd:complexType name="Provenance">
2595     <xsd:choice maxOccurs="unbounded" minOccurs="0">
2596         <xsd:element ref="xmi:Extension"/>
2597     </xsd:choice>
2598     <xsd:attribute ref="xmi:id"/>
2599     <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
2600 </xsd:complexType>
2601 <xsd:element name="Provenance" type="uima.base:Provenance"/>
2602 <xsd:complexType name="Entity">
2603     <xsd:choice maxOccurs="unbounded" minOccurs="0">
2604         <xsd:element name="occurrence" type="uima.base:Annotation"/>
2605         <xsd:element ref="xmi:Extension"/>
2606     </xsd:choice>
2607     <xsd:attribute ref="xmi:id"/>
2608     <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
2609     <xsd:attribute name="occurrence" type="xsd:string"/>
2610 </xsd:complexType>
2611 <xsd:element name="Entity" type="uima.base:Entity"/>
2612 <xsd:complexType name="SourceDocumentInformation">
2613     <xsd:choice maxOccurs="unbounded" minOccurs="0">
2614         <xsd:element ref="xmi:Extension"/>
2615     </xsd:choice>
2616     <xsd:attribute ref="xmi:id"/>
2617     <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
2618     <xsd:attribute name="uri" type="xsd:string"/>
2619     <xsd:attribute name="offsetInSource" type="xsd:int"/>
2620     <xsd:attribute name="documentSize" type="xsd:int"/>
2621 </xsd:complexType>
2622 <xsd:element name="SourceDocumentInformation"
2623 type="uima.base:SourceDocumentInformation"/>
2624 <xsd:complexType name="AnchoredView">

```

```

2625     <xsd:complexContent>
2626         <xsd:extension base="uima.base:View">
2627             <xsd:choice maxOccurs="unbounded" minOccurs="0">
2628                 <xsd:element name="sofa" type="uima.base:SofaReference"/>
2629             </xsd:choice>
2630             <xsd:attribute name="sofa" type="xsd:string"/>
2631         </xsd:extension>
2632     </xsd:complexContent>
2633 </xsd:complexType>
2634 <xsd:element name="AnchoredView" type="uima.base:AnchoredView"/>
2635 <xsd:complexType name="View">
2636     <xsd:choice maxOccurs="unbounded" minOccurs="0">
2637         <xsd:element name="IndexRepository" type="xmi:Any"/>
2638         <xsd:element name="member" type="ecore:EObject"/>
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="IndexRepository" type="xsd:string"/>
2645     <xsd:attribute name="member" type="xsd:string"/>
2646 </xsd:complexType>
2647 <xsd:element name="View" type="uima.base:View"/>
2648 </xsd:schema>
2649

```

2650 C.5 PE Metadata Ecore Model

2651 This Ecore model formally defines the UIMA Processing Element Metadata and Behavioral Metadata.

```

2652 <?xml version="1.0" encoding="UTF-8"?>
2653 <ecore:EPackage xmi:version="2.0"
2654     xmlns:xmi="http://www.omg.org/XMI"
2655     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2656     xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore" name="uima"
2657     nsURI="http://docs.oasis-open.org/uima/ns/uima.ecore" nsPrefix="uima">
2658     <eSubpackages name="peMetadata" nsURI="http://docs.oasis-
2659 open.org/uima/ns/peMetadata.ecore"
2660     nsPrefix="uima.peMetadata">
2661         <eClassifiers xsi:type="ecore:EClass" name="Identification">
2662             <eStructuralFeatures xsi:type="ecore:EAttribute" name="symbolicName"
2663 eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#/EString"/>
2664             <eStructuralFeatures xsi:type="ecore:EAttribute" name="name"
2665 eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#/EString"/>
2666             <eStructuralFeatures xsi:type="ecore:EAttribute" name="description"
2667 eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#/EString"/>
2668             <eStructuralFeatures xsi:type="ecore:EAttribute" name="vendor"
2669 eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#/EString"/>
2670             <eStructuralFeatures xsi:type="ecore:EAttribute" name="version"
2671 eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#/EString"/>
2672             <eStructuralFeatures xsi:type="ecore:EAttribute" name="url"
2673 eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#/EString"/>
2674         </eClassifiers>
2675         <eClassifiers xsi:type="ecore:EClass" name="ConfigurationParameter">
2676             <eStructuralFeatures xsi:type="ecore:EAttribute" name="name"
2677 eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#/EString"/>
2678             <eStructuralFeatures xsi:type="ecore:EAttribute" name="description"
2679 eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#/EString"/>

```

```

2680     <eStructuralFeatures xsi:type="ecore:EAttribute" name="type"
2681 eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
2682     <eStructuralFeatures xsi:type="ecore:EAttribute" name="multiValued"
2683 eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EBoolean"/>
2684     <eStructuralFeatures xsi:type="ecore:EAttribute" name="mandatory"
2685 eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EBoolean"/>
2686     <eStructuralFeatures xsi:type="ecore:EAttribute" name="defaultValue"
2687 upperBound="-1"
2688     eType="ecore:EDataType
2689 http://www.eclipse.org/emf/2002/Ecore#//EString"/>
2690     </eClassifiers>
2691     <eClassifiers xsi:type="ecore:EClass" name="TypeSystem">
2692     <eStructuralFeatures xsi:type="ecore:EAttribute" name="reference"
2693 upperBound="-1"
2694     eType="ecore:EDataType
2695 http://www.eclipse.org/emf/2002/Ecore#//EString"/>
2696     <eStructuralFeatures xsi:type="ecore:EReference" name="package"
2697 upperBound="-1"
2698     eType="ecore:EClass
2699 http://www.eclipse.org/emf/2002/Ecore#//EPackage" containment="true"/>
2700     </eClassifiers>
2701     <eClassifiers xsi:type="ecore:EClass" name="BehavioralMetadata">
2702     <eStructuralFeatures xsi:type="ecore:EAttribute"
2703 name="excludeReferenceClosure"
2704     eType="ecore:EDataType
2705 http://www.eclipse.org/emf/2002/Ecore#//EBooleanObject"/>
2706     <eStructuralFeatures xsi:type="ecore:EReference" name="analyzes"
2707 lowerBound="1"
2708     eType="#//peMetadata/BehaviorElement" containment="true"/>
2709     <eStructuralFeatures xsi:type="ecore:EReference" name="requiredInputs"
2710 lowerBound="1"
2711     eType="#//peMetadata/BehaviorElement" containment="true"/>
2712     <eStructuralFeatures xsi:type="ecore:EReference" name="optionalInputs"
2713 lowerBound="1"
2714     eType="#//peMetadata/BehaviorElement" containment="true"/>
2715     <eStructuralFeatures xsi:type="ecore:EReference" name="creates"
2716 lowerBound="1"
2717     eType="#//peMetadata/BehaviorElement" containment="true"/>
2718     <eStructuralFeatures xsi:type="ecore:EReference" name="modifies"
2719 lowerBound="1"
2720     eType="#//peMetadata/BehaviorElement" containment="true"/>
2721     <eStructuralFeatures xsi:type="ecore:EReference" name="deletes"
2722 lowerBound="1"
2723     eType="#//peMetadata/BehaviorElement" containment="true"/>
2724     <eStructuralFeatures xsi:type="ecore:EReference" name="precondition"
2725 lowerBound="1"
2726     eType="#//peMetadata/Condition" containment="true"/>
2727     <eStructuralFeatures xsi:type="ecore:EReference" name="postcondition"
2728 lowerBound="1"
2729     eType="#//peMetadata/Condition" containment="true"/>
2730     <eStructuralFeatures xsi:type="ecore:EReference"
2731 name="projectionCondition"
2732 lowerBound="1" eType="#//peMetadata/Condition" containment="true"/>
2733     <eStructuralFeatures xsi:type="ecore:EReference" name="requiredView"
2734 upperBound="-1"
2735     eType="#//peMetadata/ViewBehavioralMetadata" containment="true"/>
2736     <eStructuralFeatures xsi:type="ecore:EReference" name="optionalView"
2737 upperBound="-1"

```



```

2738         eType="#//peMetadata/ViewBehavioralMetadata" containment="true"/>
2739     <eStructuralFeatures xsi:type="ecore:EReference" name="createsView"
2740 upperBound="-1"
2741         eType="#//peMetadata/ViewBehavioralMetadata" containment="true"/>
2742 </eClassifiers>
2743 <eClassifiers xsi:type="ecore:EClass" name="ProcessingElementMetadata">
2744     <eStructuralFeatures xsi:type="ecore:EReference"
2745 name="configurationParameter"
2746         upperBound="-1" eType="#//peMetadata/ConfigurationParameter"
2747 containment="true"/>
2748     <eStructuralFeatures xsi:type="ecore:EReference" name="identification"
2749 lowerBound="1"
2750         eType="#//peMetadata/Identification" containment="true"/>
2751     <eStructuralFeatures xsi:type="ecore:EReference" name="typeSystem"
2752 lowerBound="1"
2753         eType="#//peMetadata/TypeSystem" containment="true"/>
2754     <eStructuralFeatures xsi:type="ecore:EReference"
2755 name="behavioralMetadata" lowerBound="1"
2756         eType="#//peMetadata/BehavioralMetadata" containment="true"/>
2757     <eStructuralFeatures xsi:type="ecore:EReference" name="extension"
2758 upperBound="-1"
2759         eType="#//peMetadata/Extension" containment="true"/>
2760 </eClassifiers>
2761 <eClassifiers xsi:type="ecore:EClass" name="Extension">
2762     <eStructuralFeatures xsi:type="ecore:EAttribute" name="extenderId"
2763 eType="ecore:EDatatype http://www.eclipse.org/emf/2002/Ecore#//EString"/>
2764     <eStructuralFeatures xsi:type="ecore:EReference" name="contents"
2765 lowerBound="1"
2766         eType="ecore:EClass
2767 http://www.eclipse.org/emf/2002/Ecore#//EObject" containment="true"/>
2768 </eClassifiers>
2769 <eClassifiers xsi:type="ecore:EClass" name="BehaviorElement">
2770     <eStructuralFeatures xsi:type="ecore:EReference" name="type"
2771 upperBound="-1"
2772         eType="#//peMetadata/Type" containment="true"/>
2773 </eClassifiers>
2774 <eClassifiers xsi:type="ecore:EClass" name="Type">
2775     <eStructuralFeatures xsi:type="ecore:EAttribute" name="name"
2776 eType="ecore:EDatatype http://www.eclipse.org/emf/2002/Ecore#//EString"/>
2777     <eStructuralFeatures xsi:type="ecore:EAttribute" name="feature"
2778 upperBound="-1"
2779         eType="ecore:EDatatype
2780 http://www.eclipse.org/emf/2002/Ecore#//EString"/>
2781 </eClassifiers>
2782 <eClassifiers xsi:type="ecore:EClass" name="Condition">
2783     <eStructuralFeatures xsi:type="ecore:EAttribute" name="language"
2784 eType="ecore:EDatatype http://www.eclipse.org/emf/2002/Ecore#//EString"/>
2785     <eStructuralFeatures xsi:type="ecore:EAttribute" name="expression"
2786 eType="ecore:EDatatype http://www.eclipse.org/emf/2002/Ecore#//EString"/>
2787 </eClassifiers>
2788 <eClassifiers xsi:type="ecore:EClass" name="ViewBehavioralMetadata"
2789 eSuperTypes="#//peMetadata/BehavioralMetadata"/>
2790 </eSubpackages>
2791 </ecore:EPackage>
2792

```

2793 C.6 PE Metadata XML Schema

2794 This XML schema was generated from the Ecore model in Appendix C.4 by the Eclipse Modeling
2795 Framework tools.

```
2796 <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2797 <xsd:schema xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore"
2798 xmlns:uima.peMetadata="http://docs.oasis-open.org/uima/ns/peMetadata.ecore"
2799 xmlns:xmi="http://www.omg.org/XMI"
2800 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
2801 targetNamespace="http://docs.oasis-open.org/uima/ns/peMetadata.ecore">
2802   <xsd:import namespace="http://www.eclipse.org/emf/2002/Ecore"
2803   schemaLocation="ecore.xsd"/>
2804   <xsd:import namespace="http://www.omg.org/XMI"
2805   schemaLocation="../../../../plugin/org.eclipse.emf.ecore/model/XMI.xsd"/>
2806   <xsd:complexType name="Identification">
2807     <xsd:choice maxOccurs="unbounded" minOccurs="0">
2808       <xsd:element ref="xmi:Extension"/>
2809     </xsd:choice>
2810     <xsd:attribute ref="xmi:id"/>
2811     <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
2812     <xsd:attribute name="symbolicName" type="xsd:string"/>
2813     <xsd:attribute name="name" type="xsd:string"/>
2814     <xsd:attribute name="description" type="xsd:string"/>
2815     <xsd:attribute name="vendor" type="xsd:string"/>
2816     <xsd:attribute name="version" type="xsd:string"/>
2817     <xsd:attribute name="url" type="xsd:string"/>
2818   </xsd:complexType>
2819   <xsd:element name="Identification" type="uima.peMetadata:Identification"/>
2820   <xsd:complexType name="ConfigurationParameter">
2821     <xsd:choice maxOccurs="unbounded" minOccurs="0">
2822       <xsd:element name="defaultValue" nillable="true" type="xsd:string"/>
2823       <xsd:element ref="xmi:Extension"/>
2824     </xsd:choice>
2825     <xsd:attribute ref="xmi:id"/>
2826     <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
2827     <xsd:attribute name="name" type="xsd:string"/>
2828     <xsd:attribute name="description" type="xsd:string"/>
2829     <xsd:attribute name="type" type="xsd:string"/>
2830     <xsd:attribute name="multiValued" type="xsd:boolean"/>
2831     <xsd:attribute name="mandatory" type="xsd:boolean"/>
2832   </xsd:complexType>
2833   <xsd:element name="ConfigurationParameter"
2834   type="uima.peMetadata:ConfigurationParameter"/>
2835   <xsd:complexType name="TypeSystem">
2836     <xsd:choice maxOccurs="unbounded" minOccurs="0">
2837       <xsd:element name="reference" nillable="true" type="xsd:string"/>
2838       <xsd:element name="package" type="ecore:EPackage"/>
2839       <xsd:element ref="xmi:Extension"/>
2840     </xsd:choice>
2841     <xsd:attribute ref="xmi:id"/>
2842     <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
2843   </xsd:complexType>
2844   <xsd:element name="TypeSystem" type="uima.peMetadata:TypeSystem"/>
2845   <xsd:complexType name="BehavioralMetadata">
2846     <xsd:choice maxOccurs="unbounded" minOccurs="0">
2847       <xsd:element name="analyzes" type="uima.peMetadata:BehaviorElement"/>
```

```

2848     <xsd:element name="requiredInputs"
2849 type="uima.peMetadata:BehaviorElement"/>
2850     <xsd:element name="optionalInputs"
2851 type="uima.peMetadata:BehaviorElement"/>
2852     <xsd:element name="creates" type="uima.peMetadata:BehaviorElement"/>
2853     <xsd:element name="modifies" type="uima.peMetadata:BehaviorElement"/>
2854     <xsd:element name="deletes" type="uima.peMetadata:BehaviorElement"/>
2855     <xsd:element name="precondition" type="uima.peMetadata:Condition"/>
2856     <xsd:element name="postcondition" type="uima.peMetadata:Condition"/>
2857     <xsd:element name="projectionCondition"
2858 type="uima.peMetadata:Condition"/>
2859     <xsd:element name="requiredView"
2860 type="uima.peMetadata:ViewBehavioralMetadata"/>
2861     <xsd:element name="optionalView"
2862 type="uima.peMetadata:ViewBehavioralMetadata"/>
2863     <xsd:element name="createsView"
2864 type="uima.peMetadata:ViewBehavioralMetadata"/>
2865     <xsd:element ref="xmi:Extension"/>
2866   </xsd:choice>
2867   <xsd:attribute ref="xmi:id"/>
2868   <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
2869   <xsd:attribute name="excludeReferenceClosure" type="xsd:boolean"/>
2870 </xsd:complexType>
2871 <xsd:element name="BehavioralMetadata"
2872 type="uima.peMetadata:BehavioralMetadata"/>
2873 <xsd:complexType name="ProcessingElementMetadata">
2874   <xsd:choice maxOccurs="unbounded" minOccurs="0">
2875     <xsd:element name="configurationParameter"
2876 type="uima.peMetadata:ConfigurationParameter"/>
2877     <xsd:element name="identification"
2878 type="uima.peMetadata:Identification"/>
2879     <xsd:element name="typeSystem" type="uima.peMetadata:TypeSystem"/>
2880     <xsd:element name="behavioralMetadata"
2881 type="uima.peMetadata:BehavioralMetadata"/>
2882     <xsd:element name="extension" type="uima.peMetadata:Extension"/>
2883     <xsd:element ref="xmi:Extension"/>
2884   </xsd:choice>
2885   <xsd:attribute ref="xmi:id"/>
2886   <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
2887 </xsd:complexType>
2888 <xsd:element name="ProcessingElementMetadata"
2889 type="uima.peMetadata:ProcessingElementMetadata"/>
2890 <xsd:complexType name="Extension">
2891   <xsd:choice maxOccurs="unbounded" minOccurs="0">
2892     <xsd:element name="contents" type="ecore:EObject"/>
2893     <xsd:element ref="xmi:Extension"/>
2894   </xsd:choice>
2895   <xsd:attribute ref="xmi:id"/>
2896   <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
2897   <xsd:attribute name="extenderId" type="xsd:string"/>
2898 </xsd:complexType>
2899 <xsd:element name="Extension" type="uima.peMetadata:Extension"/>
2900 <xsd:complexType name="BehaviorElement">
2901   <xsd:choice maxOccurs="unbounded" minOccurs="0">
2902     <xsd:element name="type" type="uima.peMetadata:Type"/>
2903     <xsd:element ref="xmi:Extension"/>
2904   </xsd:choice>
2905   <xsd:attribute ref="xmi:id"/>

```

```

2906     <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
2907 </xsd:complexType>
2908 <xsd:element name="BehaviorElement"
2909 type="uima.peMetadata:BehaviorElement"/>
2910 <xsd:complexType name="Type">
2911   <xsd:choice maxOccurs="unbounded" minOccurs="0">
2912     <xsd:element name="feature" nillable="true" type="xsd:string"/>
2913     <xsd:element ref="xmi:Extension"/>
2914   </xsd:choice>
2915   <xsd:attribute ref="xmi:id"/>
2916   <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
2917   <xsd:attribute name="name" type="xsd:string"/>
2918 </xsd:complexType>
2919 <xsd:element name="Type" type="uima.peMetadata:Type"/>
2920 <xsd:complexType name="Condition">
2921   <xsd:choice maxOccurs="unbounded" minOccurs="0">
2922     <xsd:element ref="xmi:Extension"/>
2923   </xsd:choice>
2924   <xsd:attribute ref="xmi:id"/>
2925   <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
2926   <xsd:attribute name="language" type="xsd:string"/>
2927   <xsd:attribute name="expression" type="xsd:string"/>
2928 </xsd:complexType>
2929 <xsd:element name="Condition" type="uima.peMetadata:Condition"/>
2930 <xsd:complexType name="ViewBehavioralMetadata">
2931   <xsd:complexContent>
2932     <xsd:extension base="uima.peMetadata:BehavioralMetadata"/>
2933   </xsd:complexContent>
2934 </xsd:complexType>
2935 <xsd:element name="ViewBehavioralMetadata"
2936 type="uima.peMetadata:ViewBehavioralMetadata"/>
2937 </xsd:schema>

```

2938 C.7 PE Service WSDL Definition

2939 This WSDL document formally defines a UIMA SOAP Service.

```

2940 <?xml version="1.0" encoding="UTF-8"?>
2941 <wsdl:definitions
2942   targetNamespace="http://docs.oasis-open.org/uima/ns/peService"
2943   xmlns:service="http://docs.oasis-open.org/uima/ns/peService"
2944   xmlns:pemd="http://docs.oasis-open.org/uima/ns/peMetadata.ecore"
2945   xmlns:pe="http://docs.oasis-open.org/uima/ns/pe.ecore"
2946   xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
2947   xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
2948   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
2949   xmlns:xmi="http://www.omg.org/XMI">
2950
2951   <wsdl:types>
2952     <!-- Import the PE Metadata Schema Definitions -->
2953     <xsd:import
2954       namespace="http://docs.oasis-open.org/uima/ns/peMetadata.ecore"
2955       schemaLocation="uima.peMetadataXMI.xsd"/>
2956
2957     <!-- Import the XMI schema. -->
2958     <xsd:import namespace="http://www.omg.org/XMI"
2959       schemaLocation="XMI.xsd"/>
2960
2961     <!-- Import other type definitions used as part of the service API. -->

```

```

2962     <xsd:import
2963         namespace="http://docs.oasis-open.org/uima/ns/pe.ecore"
2964         schemaLocation="uima.peServiceXMI.xsd"/>
2965 </wsdl:types>
2966
2967 <!-- Define the messages sent to and from the service. -->
2968
2969 <!-- Messages for all UIMA Processing Elements -->
2970 <wsdl:message name="getMetadataRequest">
2971 </wsdl:message>
2972
2973 <wsdl:message name="getMetadataResponse">
2974     <wsdl:part element="metadata"
2975         type="pemd:ProcessingElementMetadata" name="metadata"/>
2976 </wsdl:message>
2977
2978 <wsdl:message name="setConfigurationParametersRequest">
2979     <wsdl:part element="settings"
2980         type="pe:ConfigurationParameterSettings" name="settings"/>
2981 </wsdl:message>
2982
2983 <wsdl:message name="setConfigurationParametersResponse">
2984 </wsdl:message>
2985
2986 <wsdl:message name="uimaFault">
2987     <wsdl:part element="exception" type="pe:UimaException" name="exception"/>
2988 </wsdl:message>
2989
2990
2991 <!-- Messages for the Analyzer interface -->
2992
2993 <wsdl:message name="processCasRequest">
2994     <wsdl:part element="cas" type="xmi:XMI" name="cas"/>
2995     <wsdl:part element="sofas" type="pe:ObjectList" name="sofas"/>
2996 </wsdl:message>
2997
2998 <wsdl:message name="processCasResponse">
2999     <wsdl:part element="cas" type="xmi:XMI" name="cas"/>
3000 </wsdl:message>
3001
3002 <wsdl:message name="processCasBatchRequest">
3003     <wsdl:part element="casBatchInput" type="pe:CasBatchInput"
3004 name="casBatchInput"/>
3005 </wsdl:message>
3006
3007 <wsdl:message name="processCasBatchResponse">
3008     <wsdl:part element="casBatchResponse" type="pe:CasBatchResponse"
3009 name="casBatchResponse"/>
3010 </wsdl:message>
3011
3012
3013 <!-- Messages for the CasMultiplier interface -->
3014 <wsdl:message name="inputCasRequest">
3015     <wsdl:part element="cas" type="xmi:XMI" name="cas"/>
3016     <wsdl:part element="sofas" type="pe:ObjectList" name="sofas"/>
3017 </wsdl:message>
3018
3019 <wsdl:message name="inputCasResponse">

```

```

3020     </wsdl:message>
3021
3022     <wsdl:message name="getNextCasRequest">
3023     </wsdl:message>
3024
3025     <wsdl:message name="getNextCasResponse">
3026         <wsdl:part element="cas" type="xmi:XMI" name="cas"/>
3027     </wsdl:message>
3028
3029     <wsdl:message name="retrieveInputCasRequest">
3030     </wsdl:message>
3031
3032     <wsdl:message name="retrieveInputCasResponse">
3033         <wsdl:part element="cas" type="xmi:XMI" name="cas"/>
3034     </wsdl:message>
3035
3036     <wsdl:message name="getNextCasRequest">
3037         <wsdl:part element="maxCASesToReturn" type="xsd:integer"
3038 name="maxCASesToReturn"/>
3039         <wsdl:part element="timeToWait" type="xsd:integer" name="timeToWait"/>
3040     </wsdl:message>
3041
3042     <wsdl:message name="getNextCasResponse">
3043         <wsdl:part element="cas" type="xmi:XMI" name="cas"/>
3044     </wsdl:message>
3045
3046     <wsdl:message name="getNextCasBatchRequest">
3047         <wsdl:part element="maxCASesToReturn" type="xsd:integer"
3048 name="maxCASesToReturn"/>
3049         <wsdl:part element="timeToWait" type="xsd:integer" name="timeToWait"/>
3050     </wsdl:message>
3051
3052     <wsdl:message name="getNextCasBatchResponse">
3053         <wsdl:part element="reponse" type="pe:GetNextCasBatchResponse"
3054 name="response"/>
3055     </wsdl:message>
3056
3057     <!-- Messages for the FlowController interface -->
3058
3059     <wsdl:message name="addAvailableAnalyticsRequest">
3060         <wsdl:part element="analyticMetadataMap"
3061         type="pe:AnalyticMetadataMap" name="analyticMetadataMap"/>
3062     </wsdl:message>
3063
3064     <wsdl:message name="addAvailableAnalyticsResponse">
3065     </wsdl:message>
3066
3067     <wsdl:message name="removeAvailableAnalyticsRequest">
3068         <wsdl:part element="analyticKeys" type="pe:Keys"
3069         name="analyticKeys"/>
3070     </wsdl:message>
3071
3072     <wsdl:message name="removeAvailableAnalyticsResponse">
3073     </wsdl:message>
3074
3075     <wsdl:message name="setAggregateMetadataRequest">
3076         <wsdl:part element="metadata"
3077         type="pemd:ProcessingElementMetadata" name="metadata"/>

```

```

3078 </wsdl:message>
3079
3080 <wsdl:message name="setAggregateMetadataResponse">
3081 </wsdl:message>
3082
3083 <wsdl:message name="getNextDestinationsRequest">
3084 <wsdl:part element="cas" type="xmi:XMI" name="cas"/>
3085 </wsdl:message>
3086
3087 <wsdl:message name="getNextDestinationsResponse">
3088 <wsdl:part element="step" type="pe:Step" name="step"/>
3089 </wsdl:message>
3090
3091 <wsdl:message name="continueOnFailureRequest">
3092 <wsdl:part element="cas" type="xmi:XMI" name="cas"/>
3093 <wsdl:part element="failedAnalyticKey" type="xsd:string"
3094 name="failedAnalyticKey"/>
3095 <wsdl:part element="failure" type="pe:UimaException" name="failure"/>
3096 </wsdl:message>
3097
3098 <wsdl:message name="continueOnFailureResponse">
3099 <wsdl:part element="continue" type="xsd:boolean" name="continue"/>
3100 </wsdl:message>
3101
3102 <!-- Define a portType for each of the UIMA interfaces -->
3103 <wsdl:portType name="Analyzer">
3104
3105 <wsdl:operation name="getMetadata">
3106 <wsdl:input message="service:getMetadataRequest"
3107 name="getMetadataRequest"/>
3108 <wsdl:output message="service:getMetadataResponse"
3109 name="getMetadataResponse"/>
3110 <wsdl:fault message="service:uimaFault"
3111 name="uimaFault"/>
3112 </wsdl:operation>
3113
3114 <wsdl:operation name="setConfigurationParameters">
3115 <wsdl:input
3116 message="service:setConfigurationParametersRequest"
3117 name="setConfigurationParametersRequest"/>
3118 <wsdl:output
3119 message="service:setConfigurationParametersResponse"
3120 name="setConfigurationParametersResponse"/>
3121 <wsdl:fault message="service:uimaFault"
3122 name="uimaFault"/>
3123 </wsdl:operation>
3124
3125 <wsdl:operation name="processCas">
3126 <wsdl:input message="service:processCasRequest"
3127 name="processCasRequest"/>
3128 <wsdl:output message="service:processCasResponse"
3129 name="processCasResponse"/>
3130 <wsdl:fault message="service:uimaFault"
3131 name="uimaFault"/>
3132 </wsdl:operation>
3133
3134 <wsdl:operation name="processCasBatch">
3135 <wsdl:input message="service:processCasBatchRequest"

```

```

3136         name="processCasBatchRequest" />
3137     <wsdl:output message="service:processCasBatchResponse"
3138         name="processCasBatchResponse" />
3139     <wsdl:fault message="service:uimaFault"
3140         name="uimaFault" />
3141 </wsdl:operation>
3142 </wsdl:portType>
3143
3144 <wsdl:portType name="CasMultiplier">
3145
3146     <wsdl:operation name="getMetadata">
3147         <wsdl:input message="service:getMetadataRequest"
3148             name="getMetadataRequest" />
3149         <wsdl:output message="service:getMetadataResponse"
3150             name="getMetadataResponse" />
3151         <wsdl:fault message="service:uimaFault"
3152             name="uimaFault" />
3153     </wsdl:operation>
3154
3155     <wsdl:operation name="setConfigurationParameters">
3156         <wsdl:input
3157             message="service:setConfigurationParametersRequest"
3158             name="setConfigurationParametersRequest" />
3159         <wsdl:output
3160             message="service:setConfigurationParametersResponse"
3161             name="setConfigurationParametersResponse" />
3162         <wsdl:fault message="service:uimaFault"
3163             name="uimaFault" />
3164     </wsdl:operation>
3165
3166     <wsdl:operation name="inputCas">
3167         <wsdl:input message="service:inputCasRequest"
3168             name="inputCasRequest" />
3169         <wsdl:output message="service:inputCasResponse"
3170             name="inputCasResponse" />
3171         <wsdl:fault message="service:uimaFault"
3172             name="uimaFault" />
3173     </wsdl:operation>
3174
3175     <wsdl:operation name="getNextCas">
3176         <wsdl:input message="service:getNextCasRequest"
3177             name="getNextCasRequest" />
3178         <wsdl:output message="service:getNextCasResponse"
3179             name="getNextCasResponse" />
3180         <wsdl:fault message="service:uimaFault"
3181             name="uimaFault" />
3182     </wsdl:operation>
3183
3184     <wsdl:operation name="retrieveInputCas">
3185         <wsdl:input message="service:retrieveInputCasRequest"
3186             name="retrieveInputCasRequest" />
3187         <wsdl:output message="service:retrieveInputCasResponse"
3188             name="retrieveInputCasResponse" />
3189         <wsdl:fault message="service:uimaFault"
3190             name="uimaFault" />
3191     </wsdl:operation>
3192
3193     <wsdl:operation name="getNextCasBatch">

```



```

3194     <wsdl:input message="service:getNextCasBatchRequest"
3195         name="getNextCasBatchRequest" />
3196     <wsdl:output message="service:getNextCasBatchResponse"
3197         name="getNextCasBatchResponse" />
3198     <wsdl:fault message="service:uimaFault"
3199         name="uimaFault" />
3200 </wsdl:operation>
3201 </wsdl:portType>
3202
3203 <wsdl:portType name="FlowController">
3204
3205     <wsdl:operation name="getMetadata">
3206         <wsdl:input message="service:getMetadataRequest"
3207             name="getMetadataRequest" />
3208         <wsdl:output message="service:getMetadataResponse"
3209             name="getMetadataResponse" />
3210         <wsdl:fault message="service:uimaFault"
3211             name="uimaFault" />
3212     </wsdl:operation>
3213
3214     <wsdl:operation name="setConfigurationParameters">
3215         <wsdl:input
3216             message="service:setConfigurationParametersRequest"
3217             name="setConfigurationParametersRequest" />
3218         <wsdl:output
3219             message="service:setConfigurationParametersResponse"
3220             name="setConfigurationParametersResponse" />
3221         <wsdl:fault message="service:uimaFault"
3222             name="uimaFault" />
3223     </wsdl:operation>
3224
3225     <wsdl:operation name="addAvailableAnalytics">
3226         <wsdl:input message="service:addAvailableAnalyticsRequest"
3227             name="addAvailableAnalyticsRequest" />
3228         <wsdl:output message="service:addAvailableAnalyticsResponse"
3229             name="addAvailableAnalyticsResponse" />
3230         <wsdl:fault message="service:uimaFault"
3231             name="uimaFault" />
3232     </wsdl:operation>
3233
3234     <wsdl:operation name="removeAvailableAnalytics">
3235         <wsdl:input
3236             message="service:removeAvailableAnalyticsRequest"
3237             name="removeAvailableAnalyticsRequest" />
3238         <wsdl:output
3239             message="service:removeAvailableAnalyticsResponse"
3240             name="removeAvailableAnalyticsResponse" />
3241         <wsdl:fault message="service:uimaFault"
3242             name="uimaFault" />
3243     </wsdl:operation>
3244
3245     <wsdl:operation name="setAggregateMetadata">
3246         <wsdl:input message="service:setAggregateMetadataRequest"
3247             name="setAggregateMetadataRequest" />
3248         <wsdl:output message="service:setAggregateMetadataResponse"
3249             name="setAggregateMetadataResponse" />
3250         <wsdl:fault message="service:uimaFault"
3251             name="uimaFault" />

```

```

3252     </wsdl:operation>
3253
3254     <wsdl:operation name="getNextDestinations">
3255         <wsdl:input message="service:getNextDestinationsRequest"
3256             name="getNextDestinationsRequest" />
3257         <wsdl:output message="service:getNextDestinationsResponse"
3258             name="getNextDestinationsResponse" />
3259         <wsdl:fault message="service:uimaFault"
3260             name="uimaFault" />
3261     </wsdl:operation>
3262
3263     <wsdl:operation name="continueOnFailure">
3264         <wsdl:input message="service:continueOnFailureRequest"
3265             name="continueOnFailureRequest" />
3266         <wsdl:output message="service:continueOnFailureResponse"
3267             name="continueOnFailureResponse" />
3268         <wsdl:fault message="service:uimaFault"
3269             name="uimaFault" />
3270     </wsdl:operation>
3271
3272 </wsdl:portType>
3273
3274 <!-- Define a SOAP binding for each portType. -->
3275 <wsdl:binding name="AnalyzerSoapBinding" type="service:Analyzer">
3276
3277     <wsdlsoap:binding style="rpc"
3278         transport="http://schemas.xmlsoap.org/soap/http" />
3279
3280     <wsdl:operation name="getMetadata">
3281         <wsdlsoap:operation soapAction="" />
3282
3283         <wsdl:input name="getMetadataRequest">
3284             <wsdlsoap:body use="literal" />
3285         </wsdl:input>
3286
3287         <wsdl:output name="getMetadataResponse">
3288             <wsdlsoap:body use="literal" />
3289         </wsdl:output>
3290     </wsdl:operation>
3291
3292     <wsdl:operation name="setConfigurationParameters">
3293         <wsdlsoap:operation soapAction="" />
3294
3295         <wsdl:input name="setConfigurationParametersRequest">
3296             <wsdlsoap:body use="literal" />
3297         </wsdl:input>
3298
3299         <wsdl:output name="setConfigurationParametersResponse">
3300             <wsdlsoap:body use="literal" />
3301         </wsdl:output>
3302     </wsdl:operation>
3303
3304     <wsdl:operation name="processCas">
3305         <wsdlsoap:operation soapAction="" />
3306
3307         <wsdl:input name="processCasRequest">
3308             <wsdlsoap:body use="literal" />
3309     </wsdl:input>

```

```

3310
3311     <wsdl:output name="processCasResponse">
3312         <wsdlsoap:body use="literal"/>
3313     </wsdl:output>
3314 </wsdl:operation>
3315
3316 <wsdl:operation name="processCasBatch">
3317     <wsdlsoap:operation soapAction=""/>
3318
3319     <wsdl:input name="processCasBatchRequest">
3320         <wsdlsoap:body use="literal"/>
3321     </wsdl:input>
3322
3323     <wsdl:output name="processCasBatchResponse">
3324         <wsdlsoap:body use="literal"/>
3325     </wsdl:output>
3326 </wsdl:operation>
3327 </wsdl:binding>
3328
3329 <wsdl:binding name="CasMultiplierSoapBinding"
3330     type="service:CasMultiplier">
3331
3332     <wsdlsoap:binding style="rpc"
3333         transport="http://schemas.xmlsoap.org/soap/http"/>
3334
3335     <wsdl:operation name="getMetadata">
3336         <wsdlsoap:operation soapAction=""/>
3337
3338         <wsdl:input name="getMetadataRequest">
3339             <wsdlsoap:body use="literal"/>
3340         </wsdl:input>
3341
3342         <wsdl:output name="getMetadataResponse">
3343             <wsdlsoap:body use="literal"/>
3344         </wsdl:output>
3345
3346         <wsdl:fault name="uimaFault">
3347             <wsdlsoap:fault use="literal"/>
3348         </wsdl:fault>
3349     </wsdl:operation>
3350
3351     <wsdl:operation name="setConfigurationParameters">
3352         <wsdlsoap:operation soapAction=""/>
3353
3354         <wsdl:input name="setConfigurationParametersRequest">
3355             <wsdlsoap:body use="literal"/>
3356         </wsdl:input>
3357
3358         <wsdl:output name="setConfigurationParametersResponse">
3359             <wsdlsoap:body use="literal"/>
3360         </wsdl:output>
3361
3362         <wsdl:fault name="uimaFault">
3363             <wsdlsoap:fault use="literal"/>
3364         </wsdl:fault>
3365     </wsdl:operation>
3366
3367     <wsdl:operation name="inputCas">

```

```

3368     <wsdlsoap:operation soapAction="" />
3369
3370     <wsdl:input name="inputCasRequest">
3371       <wsdlsoap:body use="literal" />
3372     </wsdl:input>
3373
3374     <wsdl:output name="inputCasResponse">
3375       <wsdlsoap:body use="literal" />
3376     </wsdl:output>
3377
3378     <wsdl:fault name="uimaFault">
3379       <wsdlsoap:fault use="literal" />
3380     </wsdl:fault>
3381 </wsdl:operation>
3382
3383 <wsdl:operation name="getNextCas">
3384   <wsdlsoap:operation soapAction="" />
3385
3386   <wsdl:input name="getNextCasRequest">
3387     <wsdlsoap:body use="literal" />
3388   </wsdl:input>
3389
3390   <wsdl:output name="getNextCasResponse">
3391     <wsdlsoap:body use="literal" />
3392   </wsdl:output>
3393
3394   <wsdl:fault name="uimaFault">
3395     <wsdlsoap:fault use="literal" />
3396   </wsdl:fault>
3397 </wsdl:operation>
3398
3399 <wsdl:operation name="retrieveInputCas">
3400   <wsdlsoap:operation soapAction="" />
3401
3402   <wsdl:input name="retrieveInputCasRequest">
3403     <wsdlsoap:body use="literal" />
3404   </wsdl:input>
3405
3406   <wsdl:output name="retrieveInputCasResponse">
3407     <wsdlsoap:body use="literal" />
3408   </wsdl:output>
3409
3410   <wsdl:fault name="uimaFault">
3411     <wsdlsoap:fault use="literal" />
3412   </wsdl:fault>
3413 </wsdl:operation>
3414
3415 <wsdl:operation name="getNextCasBatch">
3416   <wsdlsoap:operation soapAction="" />
3417
3418   <wsdl:input name="getNextCasBatchRequest">
3419     <wsdlsoap:body use="literal" />
3420   </wsdl:input>
3421
3422   <wsdl:output name="getNextCasBatchResponse">
3423     <wsdlsoap:body use="literal" />
3424   </wsdl:output>
3425

```

```

3426     <wsdl:fault name="uimaFault">
3427         <wsdlsoap:fault use="literal"/>
3428     </wsdl:fault>
3429 </wsdl:operation>
3430 </wsdl:binding>
3431
3432 <wsdl:binding name="FlowControllerSoapBinding"
3433     type="service:FlowController">
3434
3435     <wsdlsoap:binding style="rpc"
3436         transport="http://schemas.xmlsoap.org/soap/http"/>
3437
3438     <wsdl:operation name="getMetadata">
3439         <wsdlsoap:operation soapAction=""/>
3440
3441         <wsdl:input name="getMetadataRequest">
3442             <wsdlsoap:body use="literal"/>
3443         </wsdl:input>
3444
3445         <wsdl:output name="getMetadataResponse">
3446             <wsdlsoap:body use="literal"/>
3447         </wsdl:output>
3448
3449         <wsdl:fault name="uimaFault">
3450             <wsdlsoap:fault use="literal"/>
3451         </wsdl:fault>
3452     </wsdl:operation>
3453
3454     <wsdl:operation name="setConfigurationParameters">
3455         <wsdlsoap:operation soapAction=""/>
3456
3457         <wsdl:input name="setConfigurationParametersRequest">
3458             <wsdlsoap:body use="literal"/>
3459         </wsdl:input>
3460
3461         <wsdl:output name="setConfigurationParametersResponse">
3462             <wsdlsoap:body use="literal"/>
3463         </wsdl:output>
3464
3465         <wsdl:fault name="uimaFault">
3466             <wsdlsoap:fault use="literal"/>
3467         </wsdl:fault>
3468     </wsdl:operation>
3469
3470     <wsdl:operation name="addAvailableAnalytics">
3471         <wsdlsoap:operation soapAction=""/>
3472
3473         <wsdl:input name="addAvailableAnalyticsRequest">
3474             <wsdlsoap:body use="literal"/>
3475         </wsdl:input>
3476
3477         <wsdl:output name="addAvailableAnalyticsResponse">
3478             <wsdlsoap:body use="literal"/>
3479         </wsdl:output>
3480
3481         <wsdl:fault name="uimaFault">
3482             <wsdlsoap:fault use="literal"/>
3483     </wsdl:fault>

```

```

3484     </wsdl:operation>
3485
3486     <wsdl:operation name="removeAvailableAnalytics">
3487         <wsdlsoap:operation soapAction=""/>
3488
3489         <wsdl:input name="removeAvailableAnalyticsRequest">
3490             <wsdlsoap:body use="literal"/>
3491         </wsdl:input>
3492
3493         <wsdl:output name="removeAvailableAnalyticsResponse">
3494             <wsdlsoap:body use="literal"/>
3495         </wsdl:output>
3496
3497         <wsdl:fault name="uimaFault">
3498             <wsdlsoap:fault use="literal"/>
3499         </wsdl:fault>
3500     </wsdl:operation>
3501
3502     <wsdl:operation name="setAggregateMetadata">
3503         <wsdlsoap:operation soapAction=""/>
3504
3505         <wsdl:input name="setAggregateMetadataRequest">
3506             <wsdlsoap:body use="literal"/>
3507         </wsdl:input>
3508
3509         <wsdl:output name="setAggregateMetadataResponse">
3510             <wsdlsoap:body use="literal"/>
3511         </wsdl:output>
3512
3513         <wsdl:fault name="uimaFault">
3514             <wsdlsoap:fault use="literal"/>
3515         </wsdl:fault>
3516     </wsdl:operation>
3517
3518     <wsdl:operation name="getNextDestinations">
3519         <wsdlsoap:operation soapAction=""/>
3520
3521         <wsdl:input name="getNextDestinationsRequest">
3522             <wsdlsoap:body use="literal"/>
3523         </wsdl:input>
3524
3525         <wsdl:output name="getNextDestinationsResponse">
3526             <wsdlsoap:body use="literal"/>
3527         </wsdl:output>
3528
3529         <wsdl:fault name="uimaFault">
3530             <wsdlsoap:fault use="literal"/>
3531         </wsdl:fault>
3532     </wsdl:operation>
3533
3534     <wsdl:operation name="continueOnFailure">
3535         <wsdlsoap:operation soapAction=""/>
3536
3537         <wsdl:input name="continueOnFailureRequest">
3538             <wsdlsoap:body use="literal"/>
3539         </wsdl:input>
3540
3541         <wsdl:output name="continueOnFailureResponse">

```

```

3542     <wsdlsoap:body use="literal"/>
3543 </wsdl:output>
3544
3545     <wsdl:fault name="uimaFault">
3546         <wsdlsoap:fault use="literal"/>
3547     </wsdl:fault>
3548 </wsdl:operation>
3549 </wsdl:binding>
3550
3551 <!-- Define an example service as including both portTypes. This is
3552      just an example, not part of the UIMA Specification -->
3553 <wsdl:service name="MyAnalyticService">
3554     <wsdl:port binding="service:AnalyzerSoapBinding"
3555         name="AnalyzerSoapPort">
3556         <wsdlsoap:address
3557
3558 location="http://localhost:8080/axis/services/MyAnalyticService/AnalyzerPort"
3559 />
3560     </wsdl:port>
3561     <wsdl:port binding="service:CasMultiplierSoapBinding"
3562         name="CasMultiplierSoapPort">
3563         <wsdlsoap:address
3564
3565 location="http://localhost:8080/axis/services/MyAnalyticService/CasMultiplier
3566 Port"/>
3567     </wsdl:port>
3568 </wsdl:service>
3569 </wsdl:definitions>

```

3570 **C.8 PE Service Data Types XML Schema (uima.peServiceXMI.xsd)**

3571 This XML schema is referenced from the WSDL definition in Appendix C.6

```

3572 <?xml version="1.0" encoding="UTF-8" standalone="no"?>
3573 <xsd:schema xmlns:uima.pe="http://docs.oasis-open.org/uima/ns/pe.ecore"
3574 xmlns:uima.peMetadata="http://docs.oasis-open.org/uima/ns/peMetadata.ecore"
3575 xmlns:xmi="http://www.omg.org/XMI"
3576 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
3577 targetNamespace="http://docs.oasis-open.org/uima/ns/pe.ecore">
3578     <xsd:import namespace="http://docs.oasis-open.org/uima/ns/peMetadata.ecore"
3579 schemaLocation="uima.peMetadataXMI.xsd"/>
3580     <xsd:import namespace="http://www.omg.org/XMI"
3581 schemaLocation="../../../plugin/org.eclipse.emf.ecore/model/XMI.xsd"/>
3582     <xsd:complexType name="ProcessingElement">
3583         <xsd:choice maxOccurs="unbounded" minOccurs="0">
3584             <xsd:element ref="xmi:Extension"/>
3585         </xsd:choice>
3586         <xsd:attribute ref="xmi:id"/>
3587         <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
3588     </xsd:complexType>
3589     <xsd:element name="ProcessingElement" type="uima.pe:ProcessingElement"/>
3590     <xsd:complexType name="Analyzer">
3591         <xsd:complexContent>
3592             <xsd:extension base="uima.pe:Analytic"/>
3593         </xsd:complexContent>
3594     </xsd:complexType>
3595     <xsd:element name="Analyzer" type="uima.pe:Analyzer"/>
3596     <xsd:complexType name="CasMultiplier">

```

```

3597     <xsd:complexContent>
3598         <xsd:extension base="uima.pe:Analytic"/>
3599     </xsd:complexContent>
3600 </xsd:complexType>
3601 <xsd:element name="CasMultiplier" type="uima.pe:CasMultiplier"/>
3602 <xsd:complexType name="FlowController">
3603     <xsd:complexContent>
3604         <xsd:extension base="uima.pe:ProcessingElement"/>
3605     </xsd:complexContent>
3606 </xsd:complexType>
3607 <xsd:element name="FlowController" type="uima.pe:FlowController"/>
3608 <xsd:complexType name="AnalyticMetadataMap">
3609     <xsd:choice maxOccurs="unbounded" minOccurs="0">
3610         <xsd:element name="AnalyticMetadataMapEntry"
3611 type="uima.pe:AnalyticMetadataMapEntry"/>
3612         <xsd:element ref="xmi:Extension"/>
3613     </xsd:choice>
3614     <xsd:attribute ref="xmi:id"/>
3615     <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
3616     <xsd:attribute name="AnalyticMetadataMapEntry" type="xsd:string"/>
3617 </xsd:complexType>
3618 <xsd:element name="AnalyticMetadataMap"
3619 type="uima.pe:AnalyticMetadataMap"/>
3620 <xsd:complexType name="AnalyticMetadataMapEntry">
3621     <xsd:choice maxOccurs="unbounded" minOccurs="0">
3622         <xsd:element name="ProcessingElementMetadata"
3623 type="uima.pe:Metadata:ProcessingElementMetadata"/>
3624         <xsd:element ref="xmi:Extension"/>
3625     </xsd:choice>
3626     <xsd:attribute ref="xmi:id"/>
3627     <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
3628     <xsd:attribute name="key" type="xsd:string"/>
3629     <xsd:attribute name="ProcessingElementMetadata" type="xsd:string"/>
3630 </xsd:complexType>
3631 <xsd:element name="AnalyticMetadataMapEntry"
3632 type="uima.pe:AnalyticMetadataMapEntry"/>
3633 <xsd:complexType name="Analytic">
3634     <xsd:complexContent>
3635         <xsd:extension base="uima.pe:ProcessingElement"/>
3636     </xsd:complexContent>
3637 </xsd:complexType>
3638 <xsd:element name="Analytic" type="uima.pe:Analytic"/>
3639 <xsd:complexType name="Step">
3640     <xsd:choice maxOccurs="unbounded" minOccurs="0">
3641         <xsd:element ref="xmi:Extension"/>
3642     </xsd:choice>
3643     <xsd:attribute ref="xmi:id"/>
3644     <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
3645 </xsd:complexType>
3646 <xsd:element name="Step" type="uima.pe:Step"/>
3647 <xsd:complexType name="SimpleStep">
3648     <xsd:complexContent>
3649         <xsd:extension base="uima.pe:Step">
3650             <xsd:attribute name="analyticKey" type="xsd:string"/>
3651         </xsd:extension>
3652     </xsd:complexContent>
3653 </xsd:complexType>
3654 <xsd:element name="SimpleStep" type="uima.pe:SimpleStep"/>

```



```

3655 <xsd:complexType name="MultiStep">
3656   <xsd:complexContent>
3657     <xsd:extension base="uima.pe:Step">
3658       <xsd:choice maxOccurs="unbounded" minOccurs="0">
3659         <xsd:element name="steps" type="uima.pe:Step"/>
3660       </xsd:choice>
3661       <xsd:attribute name="parallel" type="xsd:boolean"/>
3662     </xsd:extension>
3663   </xsd:complexContent>
3664 </xsd:complexType>
3665 <xsd:element name="MultiStep" type="uima.pe:MultiStep"/>
3666 <xsd:complexType name="FinalStep">
3667   <xsd:complexContent>
3668     <xsd:extension base="uima.pe:Step"/>
3669   </xsd:complexContent>
3670 </xsd:complexType>
3671 <xsd:element name="FinalStep" type="uima.pe:FinalStep"/>
3672 <xsd:complexType name="Keys">
3673   <xsd:choice maxOccurs="unbounded" minOccurs="0">
3674     <xsd:element name="key" nillable="true" type="xsd:string"/>
3675     <xsd:element ref="xmi:Extension"/>
3676   </xsd:choice>
3677   <xsd:attribute ref="xmi:id"/>
3678   <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
3679 </xsd:complexType>
3680 <xsd:element name="Keys" type="uima.pe:Keys"/>
3681 <xsd:complexType name="ObjectList">
3682   <xsd:choice maxOccurs="unbounded" minOccurs="0">
3683     <xsd:element name="objects" type="xmi:Any"/>
3684     <xsd:element ref="xmi:Extension"/>
3685   </xsd:choice>
3686   <xsd:attribute ref="xmi:id"/>
3687   <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
3688   <xsd:attribute name="objects" type="xsd:string"/>
3689 </xsd:complexType>
3690 <xsd:element name="ObjectList" type="uima.pe:ObjectList"/>
3691 <xsd:complexType name="UimaException">
3692   <xsd:choice maxOccurs="unbounded" minOccurs="0">
3693     <xsd:element ref="xmi:Extension"/>
3694   </xsd:choice>
3695   <xsd:attribute ref="xmi:id"/>
3696   <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
3697   <xsd:attribute name="message" type="xsd:string"/>
3698 </xsd:complexType>
3699 <xsd:element name="UimaException" type="uima.pe:UimaException"/>
3700 <xsd:complexType name="ConfigurationParameterSettings">
3701   <xsd:choice maxOccurs="unbounded" minOccurs="0">
3702     <xsd:element name="ConfigurationParameterSetting"
3703 type="uima.pe:ConfigurationParameterSetting"/>
3704     <xsd:element ref="xmi:Extension"/>
3705   </xsd:choice>
3706   <xsd:attribute ref="xmi:id"/>
3707   <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
3708 </xsd:complexType>
3709 <xsd:element name="ConfigurationParameterSettings"
3710 type="uima.pe:ConfigurationParameterSettings"/>
3711 <xsd:complexType name="ConfigurationParameterSetting">
3712   <xsd:choice maxOccurs="unbounded" minOccurs="0">

```

```

3713     <xsd:element name="values" nillable="true" type="xsd:string"/>
3714     <xsd:element ref="xmi:Extension"/>
3715 </xsd:choice>
3716 <xsd:attribute ref="xmi:id"/>
3717 <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
3718 <xsd:attribute name="parameterName" type="xsd:string"/>
3719 </xsd:complexType>
3720 <xsd:element name="ConfigurationParameterSetting"
3721 type="uima.pe:ConfigurationParameterSetting"/>
3722 <xsd:complexType name="CasBatchInput">
3723 <xsd:choice maxOccurs="unbounded" minOccurs="0">
3724 <xsd:element name="CasBatchInputElement"
3725 type="uima.pe:CasBatchInputElement"/>
3726 <xsd:element ref="xmi:Extension"/>
3727 </xsd:choice>
3728 <xsd:attribute ref="xmi:id"/>
3729 <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
3730 </xsd:complexType>
3731 <xsd:element name="CasBatchInput" type="uima.pe:CasBatchInput"/>
3732 <xsd:complexType name="CasBatchInputElement">
3733 <xsd:choice maxOccurs="unbounded" minOccurs="0">
3734 <xsd:element name="cas" type="xmi:Any"/>
3735 <xsd:element name="sofas" type="uima.pe:ObjectList"/>
3736 <xsd:element ref="xmi:Extension"/>
3737 </xsd:choice>
3738 <xsd:attribute ref="xmi:id"/>
3739 <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
3740 </xsd:complexType>
3741 <xsd:element name="CasBatchInputElement"
3742 type="uima.pe:CasBatchInputElement"/>
3743 <xsd:complexType name="CasBatchResponse">
3744 <xsd:choice maxOccurs="unbounded" minOccurs="0">
3745 <xsd:element name="CasBatchResponseElement"
3746 type="uima.pe:CasBatchResponseElement"/>
3747 <xsd:element ref="xmi:Extension"/>
3748 </xsd:choice>
3749 <xsd:attribute ref="xmi:id"/>
3750 <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
3751 </xsd:complexType>
3752 <xsd:element name="CasBatchResponse" type="uima.pe:CasBatchResponse"/>
3753 <xsd:complexType name="CasBatchResponseElement">
3754 <xsd:choice maxOccurs="unbounded" minOccurs="0">
3755 <xsd:element name="CAS" type="xmi:Any"/>
3756 <xsd:element name="UimaException" type="uima.pe:UimaException"/>
3757 <xsd:element ref="xmi:Extension"/>
3758 </xsd:choice>
3759 <xsd:attribute ref="xmi:id"/>
3760 <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
3761 </xsd:complexType>
3762 <xsd:element name="CasBatchResponseElement"
3763 type="uima.pe:CasBatchResponseElement"/>
3764 <xsd:complexType name="GetNextCasBatchResponse">
3765 <xsd:choice maxOccurs="unbounded" minOccurs="0">
3766 <xsd:element name="CAS" type="xmi:Any"/>
3767 <xsd:element ref="xmi:Extension"/>
3768 </xsd:choice>
3769 <xsd:attribute ref="xmi:id"/>
3770 <xsd:attributeGroup ref="xmi:ObjectAttribs"/>

```

```
3771     <xsd:attribute name="hasMoreCASes" type="xsd:boolean"/>
3772     <xsd:attribute name="estimatedRemainingCASes" type="xsd:int"/>
3773 </xsd:complexType>
3774 <xsd:element name="GetNextCasBatchResponse"
3775 type="uima.pe:GetNextCasBatchResponse"/>
3776 </xsd:schema>
```