



Unstructured Information Management Architecture (UIMA) Version 1.0

Working Draft 03

24 April 2008

Specification URIs:

This Version:

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

Previous Version:

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

Latest Version:

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

Latest Approved Version:

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

Technical Committee:

OASIS Unstructured Information Management Architecture (UIMA) TC

Chair(s):

David Ferrucci, IBM

Editor(s):

Adam Lally, IBM

Related work:

This specification replaces or supercedes:

- [specifications replaced by this standard]
- [specifications replaced by this standard]

This specification is related to:

- [related specifications]
- [related specifications]

Declared XML Namespace(s):

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

Abstract:

[Summary of the technical purpose of the document]

Status:

This document was last revised or approved by the [TC name | membership of OASIS] on the above date. The level of approval is also listed above. Check the “Latest Version” or “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/\[TC short name\] /](http://www.oasis-open.org/committees/[TC short name]/).

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/\[TC short name\] /ipr.php](http://www.oasis-open.org/committees/[TC short name] /ipr.php)).

The non-normative errata page for this specification is located at [http://www.oasis-open.org/committees/\[TC short name\] /](http://www.oasis-open.org/committees/[TC short name] /).

Notices

Copyright © OASIS® 2008. All Rights Reserved.

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

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

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

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

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

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

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

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

Table of Contents

1	Introduction	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	The Common Analysis Structure (CAS)	11
3.1.1	Basic Structure: Objects and Slots	12
3.1.2	Relationship to Type System.....	12
3.1.3	The XMI CAS Representation	14
3.1.4	Example (Not Normative)	14
3.1.5	Formal Specification	18
3.2	The Type System Model	18
3.2.1	Features of the Type System Model.....	19
3.2.2	Ecore as the UIMA Type System Representation.....	19
3.2.3	Discussion and Example (Not Normative).....	20
3.2.4	Formal Specification	23
3.3	Base Type System	23
3.3.1	Primitive Types	24
3.3.2	Annotation and Sofa Type System	24
3.3.3	View Type System	29
3.3.4	Source Document Information.....	31
3.3.5	Formal Specification	32
3.4	Abstract Interfaces	32
3.4.1	Processing Element.....	32
3.4.2	Analytic	33
3.4.3	Analyzer	33
3.4.4	CAS Multiplier	33
3.4.5	Flow Controller.....	34
3.4.6	Examples (Not Normative).....	36
3.4.7	Formal Specification	37
3.5	Behavioral Metadata	41
3.5.1	Goals.....	41
3.5.2	Elements of Behavioral Metadata.....	42
3.5.3	Example (Not Normative)	42
3.5.4	Using Views in Behavioral Metadata	43
3.5.5	Formal Semantics for Behavioral Metadata	43
3.5.6	Behavioral Metadata UML	44
3.5.7	Behavioral Metadata XML Representation.....	45
3.5.8	Formal Specification	51
3.6	Processing Element Metadata	54
3.6.1	Overview	54
3.6.2	Elements of PE Metadata	55

3.6.3 Example (Not Normative)	58
3.6.4 Formal Specification	59
3.7 Service WSDL Descriptions	59
3.7.1 Overview of the WSDL Definition	59
3.7.2 Delta Responses	63
3.7.3 SOAP Service Example (Not Normative)	63
3.7.4 Formal Specification	64
A. Acknowledgements.....	65
B. Formal Specification Artifacts	66
B.1 XMI XML Schema.....	66
B.2 Ecore XML Schema.....	69
B.3 Base Type System Ecore Model	74
B.4 PE Metadata and Behavioral Metadata Ecore Model	74
B.5 PE Metadata and Behavioral Metadata XML Schema	77
B.6 PE Service WSDL Definition	80
B.7 PE Service XML Schema (uima.peServiceXMI.xsd).....	93
C. Non-Normative Text.....	97
D. Revision History.....	98

1 Introduction

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

7
8 Unstructured information may be contrasted with the information in classic relational databases where the
9 intended interpretation for every field data is explicitly encoded in the database by column headings.
10 Consider information encoded in XML as another example. In an XML document some of the data is
11 wrapped by tags which provide explicit semantic information about how that data should be interpreted.
12 An XML document or a relational database may be considered semi-structured in practice, because the
13 content of some chunk of data, a blob of text in a text field labeled “description” for example, may be of
14 interest to an application but remain without any explicit tagging—that is, without any explicit semantics or
15 structure.

16
17 Unstructured information represents the largest, most current and fastest growing source of knowledge
18 available to businesses and governments worldwide. The web is just the tip of the iceberg. Consider the
19 for example the droves of corporate, scientific, social and technical documentation ranging from best
20 practices, research reports, medical abstracts, problem reports, customer communications, contracts,
21 emails and voice mails. Beyond these consider the growing number of broadcasts containing audio, video
22 and speech. In these mounds of natural language, speech and video artifacts often lay nuggets of
23 knowledge critical for analyzing and solving problems, detecting threats, realizing important trends and
24 relationships, creating new opportunities or preventing disasters.

25
26 For unstructured information to be processed by traditional applications that rely on specific structure, it
27 must be first analyzed to assign application-specific semantics to the unstructured content. Another way
28 to say this is that the unstructured information must become “structured” where the added structure
29 explicitly provides the semantics required by target applications to interpret the data.

30
31 An example of assigning semantics includes wrapping regions of text in a text document with appropriate
32 XML tags that might identify the names of organizations or products. Another example may extract
33 elements of a document and insert them in the appropriate fields of a relational database or use them to
34 create instances of concepts in a knowledgebase. Another example may analyze a voice stream and tag
35 it with the information explicitly identifying the speaker.

36
37 A simple analysis on documents may, for example, scan each token in each document of a collection to
38 identify names of organizations. It may insert a tag wrapping and identifying every found occurrence of an
39 organization name and output the XML that explicitly annotates each with the appropriate tag. An
40 application that manages a database of organizations may now use the structured information produced
41 by the document analysis to populate a relational database.

42
43 In general, we refer to the act of assigning semantics to a region of some unstructured content (e.g., a
44 document) as “analysis”. A software component or service that performs the analysis is an “analytic”.

45
46 Analytics are typically reused and combined together in different flows to perform application-specific
47 aggregate analyses.

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

53
54 The UIMA specification defines platform-independent data representations & interfaces for text & multi-
55 modal analytics. The principal objective of the UIMA specification is to support interoperability among
56 *analytics*. This objective is subdivided into the following four design goals:

- 57
58 1. **Data Representation.** Support the common representation of *artifacts* and *artifact metadata*
59 (analysis results) independently of *artifact modality* and *domain model*.
- 60
61 2. **Data Modeling and Interchange.** Support the platform-independent interchange of *analysis data*
62 in a form that facilitates a formal modeling approach and alignment with existing programming
63 systems and standards.
- 64
65 3. **Discovery, Reuse and Composition.** Support the discovery, reuse and composition of
66 independently-developed *analytics*.
- 67
68 4. **Service-Level Interoperability.** Support concrete interoperability of independently developed
69 *analytics* based on a common service description and associated SOAP bindings.

70
71 The text of this specification is normative with the exception of sections that explicitly state “Not
72 Normative” in their heading.

75 1.1 Terminology

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

79 1.2 Normative References

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

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

102 **1.3 Non-Normative References**

103 **[BPEL1]** http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel
104 **[EcoreEMOF1]** <http://dev.eclipse.org/newslists/news.eclipse.tools.emf/msg04197.html>
105
106 **[EMF1]** The Eclipse Modeling Framework (EMF) Overview.
107 <http://help.eclipse.org/help33/index.jsp?topic=/org.eclipse.emf.doc//references/overview/EMF.html>
108 **[EMF2]** Budinsky et al. Eclipse Modeling Framework. Addison-Wesley. 2004.
109 **[XMI2]** Grose et al. Mastering XMI. Java Programming with XMI, XML, and UML. John Wiley &
110 Sons, Inc. 2002

111 2 Basic Concepts and Terms

112 This specification defines and uses the following terms:

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

118

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

123

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

126

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

133

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

140

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

142

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

149

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

151

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

153

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

156

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

161 3 Elements of the UIMA Specification

162 In this section we define the seven elements of the UIMA standard. For each element, there is generally
163 a detailed description, UML model, and examples, followed by the Formal Specification for that element.
164 The Formal Specification sections list the precise requirements that UIMA implementations must satisfy in
165 order to comply with this standard.

166 The elements are listed in brief below:

- 169 1. **Common Analysis Structure (CAS).** Supports interoperability by providing a common data
170 structure shared among analytics. The CAS is a general object graph and is used to
171 represent the artifact and the artifact metadata. UIMA defines an XML representation of
172 analysis data using the XML Metadata Interchange (XMI) specification [XMI1][XMI2].
- 174 2. **Type System Model.** To support data modeling and interchange, a CAS must conform to a
175 user-defined schema called a Type System. Every object in a CAS must be associated with
176 a Type defined by a Type System. UIMA defines the Type System representation using
177 Ecore, which is the modeling language used in the Eclipse Modeling Framework [EMF1] and
178 is tightly aligned with the OMG's EMOF standard. The XML representation uses XMI.
- 180 3. **Base Type System.** Provides a Standard definition of commonly-used, domain-independent
181 types, in order to establish a basic level of interoperability among applications. For example
182 UIMA defines the type Annotation to represent objects that have references (e.g., offsets) into
183 the value of an attribute of another object. It is intended that annotations describe or
184 "annotate" the unstructured content in these values.
- 186 4. **Abstract Interfaces.** Defines the standard component types and operations that UIMA
187 developers can implement. This element is defined abstractly using a UML model.
- 189 5. **Behavioral Metadata.** Provides a formal declarative description of what a UIMA analytic
190 does. This includes: what types of CASeS it can process, what elements in a CAS it
191 analyzes, and what effects it may have on CAS contents as a result of its application.
- 193 6. **Processing Element Metadata.** Provides a standard declarative means for describing
194 identification, configuration and behavioral information about Processing Elements (analytics
195 and flow controllers). This section of the specification refers to the Behavioral Metadata
196 Specification to represent a processing element's behavioral information.
- 198 7. **WSDL Service Descriptions.** This specification element facilitates interoperability by
199 specifying a WSDL [WSDL1] description of the UIMA interfaces and a binding to a concrete
200 SOAP interface that compliant frameworks/services must implement.

201 3.1 The Common Analysis Structure (CAS)

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

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

208

209 The CAS is an Object Graph where Objects are instances of Classes and Classes are Types in a type
210 system. The Type System Model is described in detail in Section 3.2.

211

212 There are two fundamental types of objects in a CAS:

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

216

217 A data representation which stores the artifact metadata separately from the subject of analysis is
218 commonly referred to as *standoff annotation*. This approach is preferred to in-line annotation approaches
219 (such as SGML) which change the subject of analysis by inserting tags directly into the content.

220 Preserving the distinction between the original content and subsequent annotation provides better
221 support for interoperability of analytics. The definitions of the Sofa and Annotation types are introduced in
222 Section 3.3.2.

223

224 The CAS provides a domain neutral, object-based representation scheme that is aligned with UML and
225 XML standards. The CAS representation can easily be elaborated for specific domains of analysis by
226 defining domain-specific types; interoperability can be achieved across programming languages and
227 operating systems through the use of the CAS representation and its associated type system definition
228 (see 3.1.2).

229

230 **3.1.1 Basic Structure: Objects and Slots**

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

234

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

237

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

240

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

245

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

248

249 **3.1.2 Relationship to Type System**

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

251
 252 A type system defines a set of classes. A class may have multiple features. Features may either be
 253 attributes or references.
 254
 255 All features define their type. The type of an attribute is a primitive dataType. The type of a reference is a
 256 class. Features also have a cardinality (defined by a lower bound and a upper bound), which define how
 257 many values they may take. We sometimes refer to features with an upper bound greater than one as
 258 multi-valued features.
 259
 260 An object has one slot for each feature defined by its class.
 261
 262 Slots for attributes take primitive values; slots for references take objects as values. In general a slot may
 263 take multiple values; the number of allowed values is defined by the lower bound and upper bound of the
 264 feature.
 265
 266 The metamodel describing how a CAS relates to a type system is diagrammed in Figure 1.
 267
 268 Note that some UIMA components may manipulate a CAS without knowledge of its type system. A
 269 common example is a CAS Store, which might allow the storage and retrieval of any CAS regardless of
 270 what its type system might be.
 271

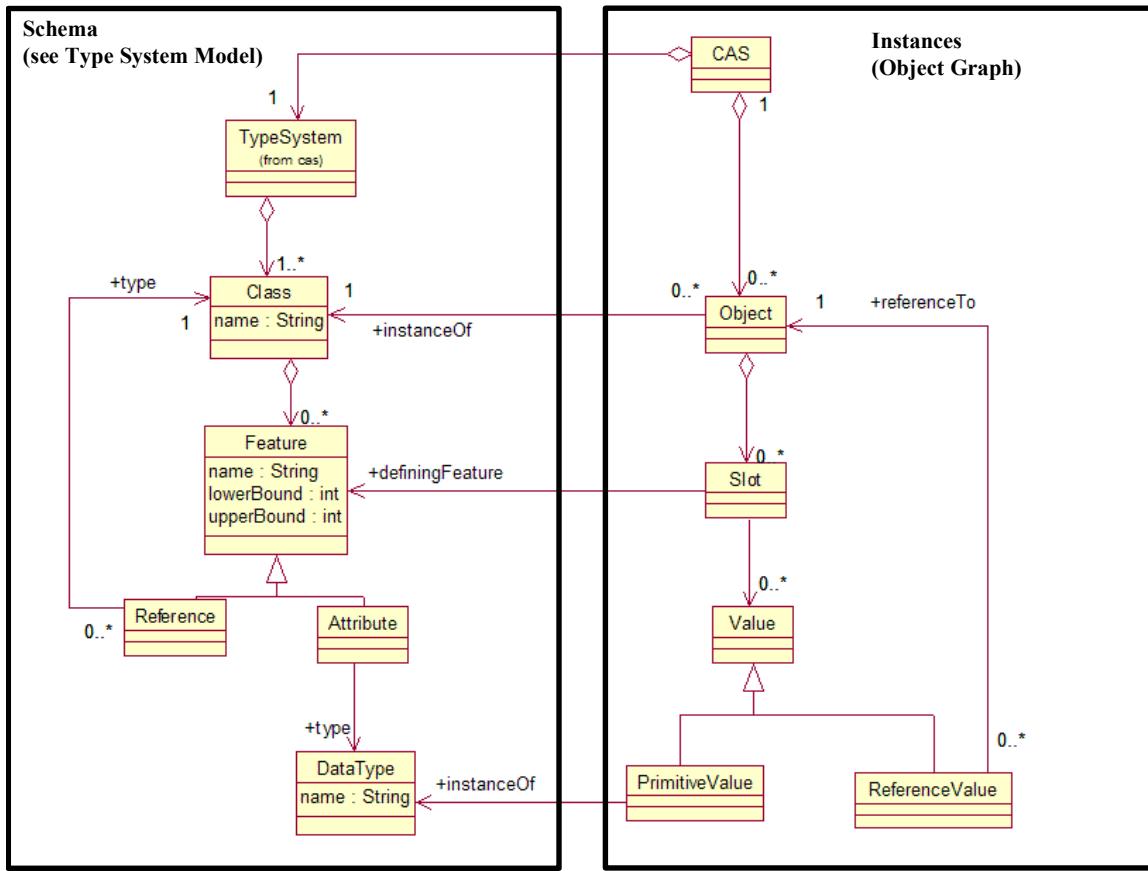


Figure 1: CAS Specification UML

272
 273
 274

275 **3.1.3 The XMI CAS Representation**

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

278

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

282

283 **3.1.4 Example (Not Normative)**

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

286 **3.1.4.1 XMI Tag**

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

290

```
291 <xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI">  
292   <!-- CAS Contents here -->  
293 </xmi:XMI>
```

294

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

297

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

300

301 **3.1.4.2 Objects**

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

304

305 For example consider the following XMI document:

```
306 <xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"  
307   xmlns:myorg="http://org/myorg.ecore">  
308   ...  
309   <myorg:Person xmi:id="1"/>  
310   ...  
311 </xmi:XMI>
```

312

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

317

318 Note that the order in which Objects are listed in the XMI is not important, and components that process
319 XMI are not required to maintain this order.
320
321 The xmi:id attribute can be used to refer to an object from elsewhere in the XMI document. It is not
322 required if the object is never referenced. If an xmi:id is provided, it must be unique among all xmi:ids on
323 all objects in this CAS.
324
325 All namespace prefixes (e.g., myorg) in this example must be bound to URIs using the
326 "xmlns..." attribute, as defined by the XML namespaces specification [XMLS1].
327

328 **3.1.4.3 Attributes (Primitive Features)**

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

332
333 For example, an *object* of *class Person*, with slots:

334
335 begin = 14
336 end = 25
337 name = "Fred Center"

338 could be mapped to the attribute serialization as follows:

339
340
341 <xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
342 xmlns:myorg="http://org/myorg.ecore">
343 ...
344 <myorg:Person xmi:id="1" begin="14" end="25" name="Fred Center"/>
345 ...
346 </xmi:XMI>

347 or alternatively to an element serialization as follows:

348
349
350 <xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
351 xmlns:myorg="http://org/myorg.ecore">
352 ...
353 <myorg:Person xmi:id="1">
354 <begin>14</begin>
355 <end>25</end>
356 <name>Fred Center</name>
357 </myorg:Person>
358 ...
359 </xmi:XMI>

361 The attribute serialization is preferred for compactness, but either representation is allowed and UIMA
362 framework components that process XMI are required to support both. Mixing the two styles is allowed;
363 some *features* can be represented as attributes and others as elements.

364 **3.1.4.4 References (Object-Valued Features)**

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

366

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

369

```
370 <xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"  
371   xmlns:myorg="http://org/myorg.ecore">  
372   ...  
373   <myorg:Person xmi:id="1" begin="14" end="25" name="Fred Center"/>  
374   <myorg:Organization xmi:id="2" myCEO="1"/>  
375   ...  
376 </xmi:XMI>
```

377

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

381

```
382 <myorg:Organization xmi:id="2">  
383   <myCEO href="#1"/>  
384 <myorg.Organization>
```

385

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

389 **3.1.4.5 Multi-valued Features**

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

392

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

394

395 or:

396

```

397   <myorg:Baz xmi:id="3">
398     <myIntArray>2</myIntArray>
399     <myIntArray>4</myIntArray>
400     <myIntArray>6</myIntArray>
401   </myorg:Baz>
402
403 Note that string arrays whose elements contain embedded spaces must use the latter mapping.1
404
405 Multi-valued references serialized in a similar way. For example a reference that refers to the elements
406 with xmi:ids "13" and "42" could be serialized as:
407
408   <myorg:Baz xmi:id="3" myRefFeature="13 42"/>
409
410 or:
411
412   <myorg:Baz xmi:id="3">
413     <myRefFeature href="#13"/>
414     <myRefFeature href="#42"/>
415   </myorg:Baz>
416
417 Note that the order in which the elements of a multi-valued feature are listed is meaningful, and
418 components that process XMI documents must maintain this order.
419

```

3.1.5 Linking an XMI Document to its Ecore Type System

The structure of a CAS is defined by a UIMA type system, which is represented by an Ecore model (see Section 3.2).

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

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

¹ It might be possible to use an escape sequence to encode a space, which would allow elements containing embedded spaces to be serialized as an attribute value. However, the XMI specification [XMI1] does not appear to specify such escape sequences.

431
432 xsi:schemaLocation="http://org/myorg.ecore file:/c:/typesystems/myorg.ecore"
433
434 would indicate that the definition for the org.myorg CAS types is contained in the file
435 c:/typesystems/myorg.ecore. You can specify a different mapping for each of your CAS namespaces. For
436 details see [EMF2].

437 **3.1.6 XMI Extensions**

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

441
442 <xmi:Extension extenderId="NAME">
443 <!-- arbitrary content can go inside the Extension element -->
444 </xmi:Extension>

445
446 The extenderId attribute allows a particular "extender" (e.g., a UIMA framework implementation) to record
447 metadata that's relevant only within that framework, without confusing other frameworks that my want to
448 process the same CAS.

449

450 **3.1.7 Formal Specification**

451 **3.1.7.1 Structure**

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

453

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

456 **3.1.7.2 Constraints**

457 **3.1.7.2.1 Linkage of CAS to Ecore Type System**

458 If the root element of the XML CAS contains an xsi:schemaLocation attribute, the CAS is said to be linked
459 to an Ecore Type System. The xsi:schemaLocation attribute defines a mapping from namespace URI to
460 physical URL as defined by the XML Schema specification [XMLS1]. Each of these physical URLs MUST
461 be a valid Ecore document as defined by the XML Schema for Ecore, presented in Appendix B.2.

462

463 A CAS that is linked to an Ecore Type System MUST be valid with respect to that Ecore Type System, as
464 defined in Section 3.2.4.2.

465 **3.2 The Type System Model**

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

468

469 A type system is a collection of inter-related **type** definitions. Each type defines the structure of any object
470 that is an instance of that type. For example, Person and Organization may be types defined as part of a
471 type system. Each type definition declares the attributes of the type and describes valid fillers for its

472 attributes. For example lastName, age, emergencyContact and employer may be attributes of the Person
473 type. The type system may further specify that the lastName must be filled with exactly one string value,
474 age exactly one integer value, emergencyContact exactly one instance of the same Person type and
475 employer zero or more instances of the Organization type.

476

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

480

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

487

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

494 **3.2.1 Features of the Type System Model**

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

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

508 **3.2.2 Ecore as the UIMA Type System Representation**

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

511

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

515

516 Ecore is the modeling language of the Eclipse Modeling Framework (EMF) [EMF1]. It affords the
517 equivalent modeling semantics provided by EMOF with some minor syntactic differences – see Section
518 3.2.3.2.

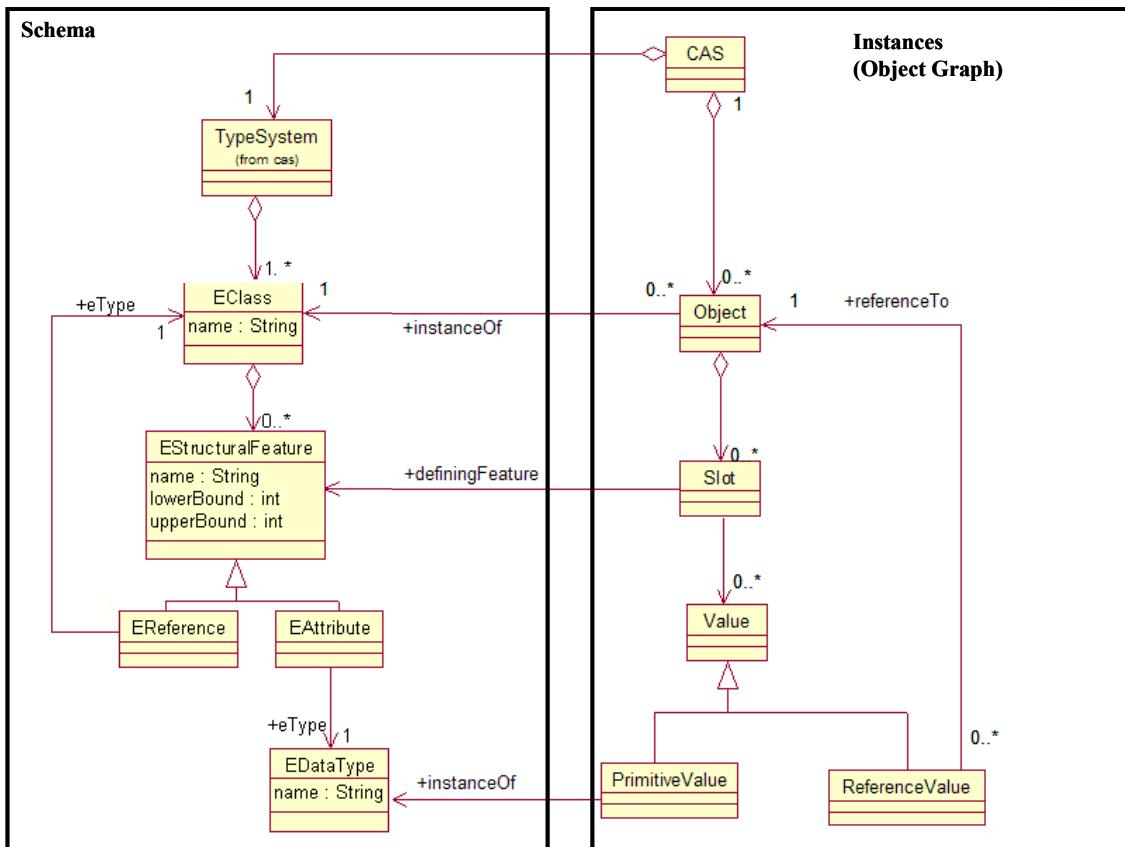
519

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

522

523 Figure 2 shows how Ecore is used to define the schema for a CAS.

524



525

Figure 2: Ecore defines schema for CAS

526

527 For an example of a UIMA Type System represented in Ecore, see the appendix 3.2.3.3 *Example*.

529 3.2.3 Discussion and Example (Not Normative)

530 3.2.3.1 An Introduction to Ecore

531 Ecore is well described by Budinsky et al. in the book *Eclipse Modeling Framework*. Some brief
532 introduction to Ecore can be found in a chapter of that book that is available online at
533 <http://www.awprofessional.com/content/images/0131425420/samplechapter/budinskych02.pdf> (see
534 section 2.3). As a convenience to 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.

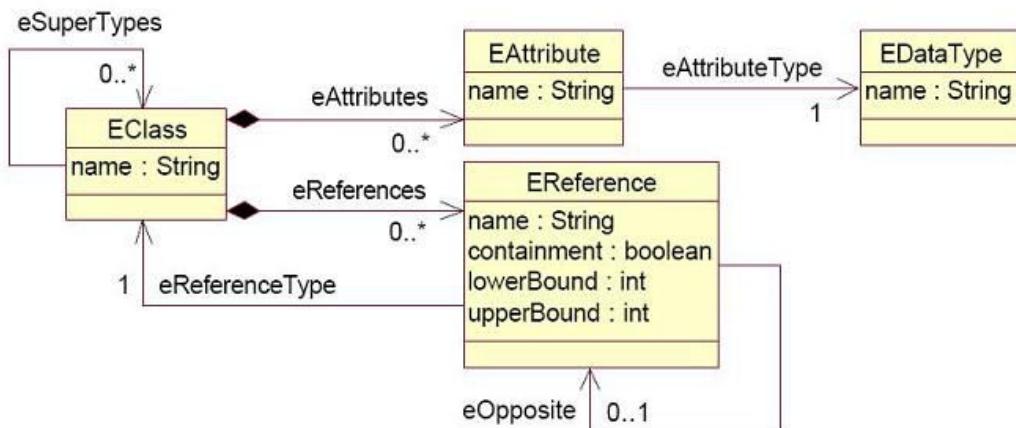


Figure 3: The Ecore Kernel

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.

535
536

537 3.2.3.2 Differences between Ecore and EMOF

538 The primary differences between Ecore and EMOF are:

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

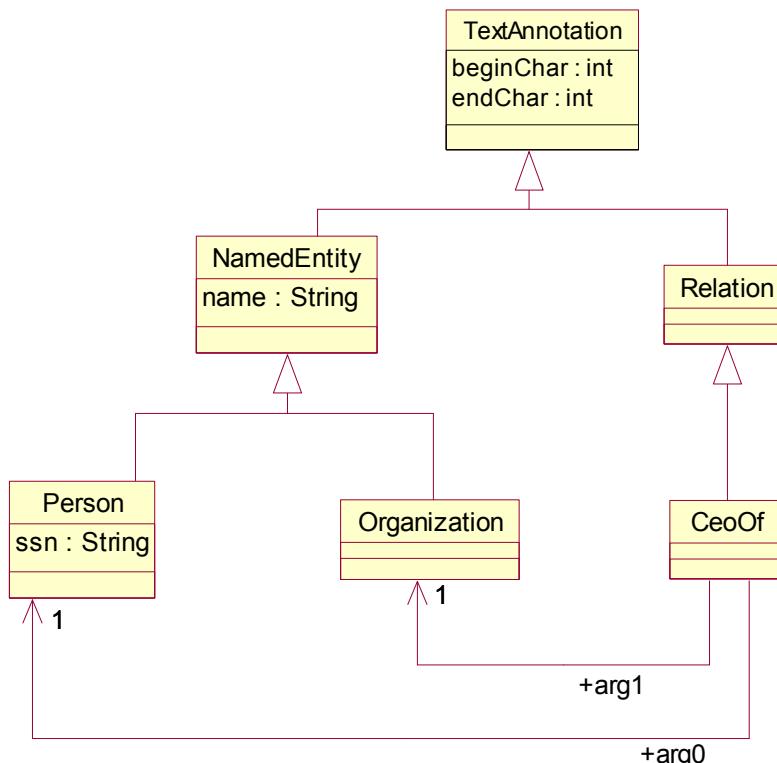
542

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

544 3.2.3.3 Example Ecore Model

545 Figure 4 shows a simple example of an object model in UML. This model describes two types of Named
 546 Entities: Person and Organization. They may participate in a CeoOf relation (i.e., a Person is the CEO of
 547 an Organization). The NamedEntity and Relation types are subtypes of TextAnnotation (a standard UIMA
 548 base type, see 3.3), so they will inherit beginChar and endChar features that specify their location within a
 549 text document.

550



551

552 **Figure 4: Example UML Model**

553

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

556

```

557 <?xml version="1.0" encoding="UTF-8"?>
558 <ecore:EPackage xmi:version="2.0"
559   xmlns:xmi="http://www.omg.org/XMI"
560   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
561   xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore"
562   xmlns:uima.cas="http://docs.oasis-open.org/uima/cas.ecore"
563   name="org" nsURI="http:///org.ecore" nsPrefix="org">
```

```

564      <eSubpackages name="example" nsURI="http://org/example.ecore"
565      nsPrefix="org.example">
566          <eClassifiers xsi:type="ecore:EClass" name="NamedEntity"
567          eSuperTypes="uima.cas:TextAnnotation">
568              <eStructuralFeatures xsi:type="ecore:EAttribute" name="name"
569              eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore//EString"/>
570          </eClassifiers>
571          <eClassifiers xsi:type="ecore:EClass" name="Relation"
572          eSuperTypes="uima.cas:TextAnnotation"/>
573              <eClassifiers xsi:type="ecore:EClass" name="Person"
574              eSuperTypes="#//example/NamedEntity">
575                  <eStructuralFeatures xsi:type="ecore:EAttribute" name="ssn"
576                  eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore//EString"/>
577              </eClassifiers>
578              <eClassifiers xsi:type="ecore:EClass" name="CeoOf"
579              eSuperTypes="#//example/Relation">
580                  <eStructuralFeatures xsi:type="ecore:EReference" name="arg0"
581                  lowerBound="1"
582                      eType="#//example/Person"/>
583                  <eStructuralFeatures xsi:type="ecore:EReference" name="arg1"
584                  lowerBound="1"
585                      eType="#//example/Organization"/>
586              </eClassifiers>
587          </eSubpackages>
588      </ecore:EPackage>

```

589 This XMI document is a valid representation of a UIMA Type System.

590 **3.2.4 Formal Specification**

591 **3.2.4.1 Structure**

592 *UIMA Type System XML* must be a valid Ecore/XMI document as defined by Ecore and the XMI Specification [XMI1].

594

595 This implies that UIMA Type System XML must be a valid instance of the XML Schema for Ecore, given
596 in Section B.2.

597 **3.2.4.2 Semantics**

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

601 **3.3 Base Type System**

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

604

605 The Base Type System includes the following:

- 606 • Primitive Types (defined by Ecore)
- 607 • Annotation and Sofa Types (Annotation representation and linkage to Sofas)
- 608 • Views (Specific collections of objects in a CAS)
- 609 • Source Document Information (Records information about the original source of unstructured
610 information in the CAS)

611
612 The XML namespace for types defined in the UIMA base model is <http://docs.oasis-open.org/uima/cas.ecore>. (With the exception of types defined as part of Ecore, listed in Section 3.3.1,
613 whose namespace is defined by Ecore.)
614

615 **3.3.1 Primitive Types**

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

- 618
- 619 • EString
 - 620 • EBoolean
 - 621 • EByte (8 bits)
 - 622 • EShort (16 bits)
 - 623 • EInt (32 bits)
 - 624 • ELong (64 bits)
 - 625 • EFloat (32 bits)
 - 626 • EDouble (64 bits)
- 627
628 Also Ecore defines the type EObject, which is defined as the superclass of all non-primitive types
629 (classes).

630 **3.3.2 Annotation and Sofa Type System**

631 **3.3.2.1 Overview**

632 A general and motivating UIMA use-case is one where analytics label or *annotate* regions of unstructured
633 content. A fundamental approach to representing annotations is referred to as “stand-off” annotation
634 model. In a “stand-off” annotation model, annotations are represented as objects of a domain model that
635 “point into” or reference elements of the unstructured content (e.g., document or video stream) rather than
636 as inserted tags that affect and/or are constrained by the original form of the content. A stand-off model
637 allows for multiple, potentially contradictory, interpretations of the content and different representations of
638 the same artifact to be created and manipulated independently.

639
640 In UIMA, a CAS stores the artifact (i.e., the unstructured content that is the subject of the analysis) and
641 the artifact metadata (i.e., structured data elements that describes the artifact). The metadata generated
642 by an analytic may include a set of annotations that label regions of the artifact with respect to some
643 domain model (e.g., persons, organizations, events, times, opinions, etc). These annotations are logically
644 and physical distinct from the subject of analysis, so UIMA adopts the “*stand-off*” model for annotations.

645
646 In UIMA the original content is not affected in the analysis process. Rather, an object graph is produced
647 that *stands off* from and annotates the content. Stand-off annotations in UIMA allow for multiple content
648 interpretations of graph complexity to be produced, co-exist, overlap and be retracted without affecting
649 the original content representation. The object model representing the stand-off annotations may be used
650 to produce different representations of the analysis results. A common form for capturing document
651 metadata for example is as in-line XML. An analytic in a UIM application, for example, can generate from
652 the UIMA representation an in-line XML document that conforms to some particular domain model or
653 markup language. Alternatively it can produce an XMI or RDF document.

654 **3.3.2.2 Annotation and Sofa Reference**

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

657

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

663

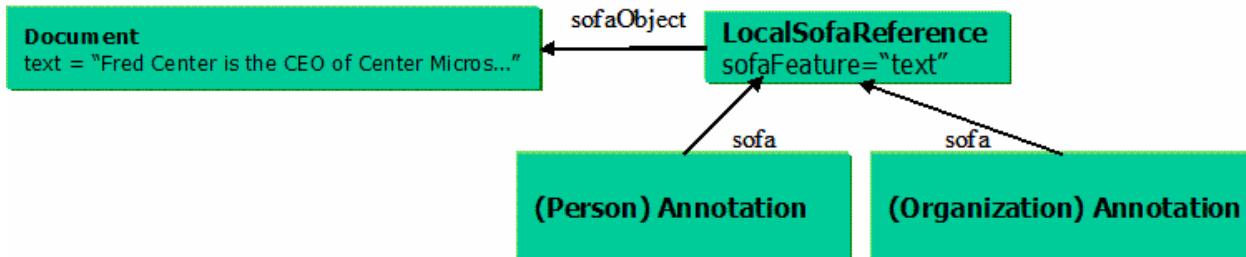
664 Figure 5 illustrates an example. The CAS contains an *object* of class Document with a *slot* text
665 containing the string value, "Fred Center is the CEO of Center Micros."

666

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

671

672



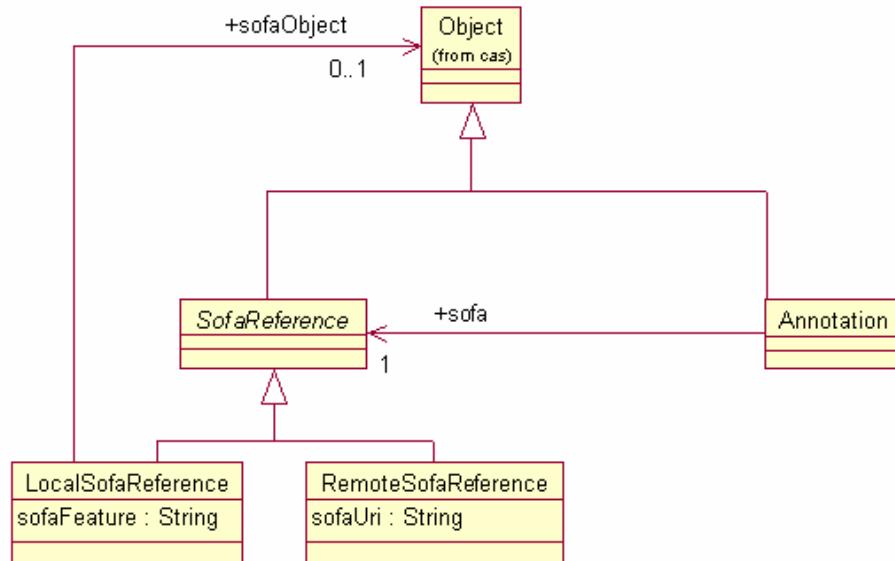
673

674 **Figure 5: Annotation and Subject of Analysis**

675

676 The UML model for the Annotation and SofaReference types is given in Figure 6.

677



678

679 **Figure 6: Annotation and Sofa Reference UML**

680

681

682 3.3.2.3 References to Regions of Sofas

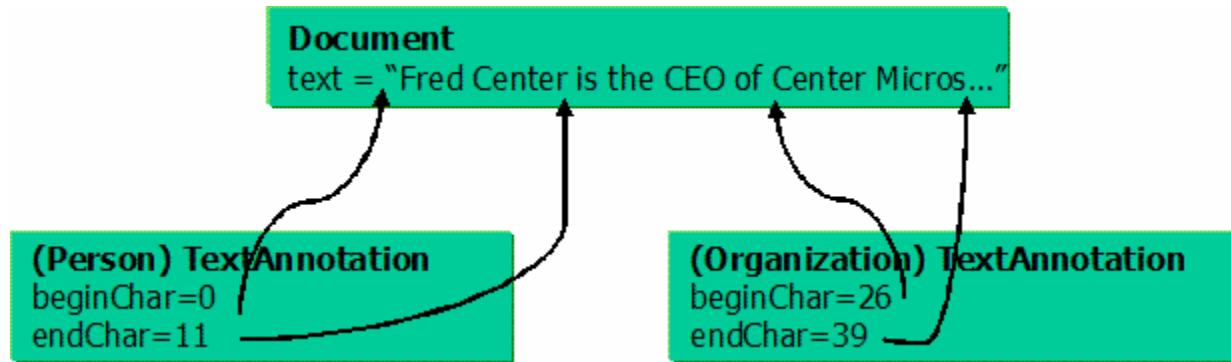
683 An annotation typically points to a region of the artifact data. One of UIMA's design goals is to be
684 independent of modality. For this reason UIMA does not constrain the data type that can function as a
685 subject of analysis and allows for different implementations of the linkage between an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

687

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

691

692 Figure 7 extends the previous example by showing how the TextAnnotation subtype of Annotation is
693 used to specify a range of character offsets to which the annotation applies.

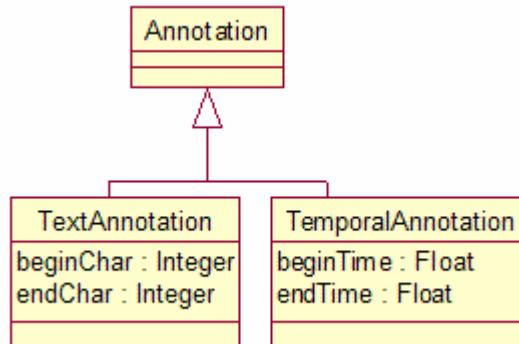


694
695 Figure 7: References from Annotations to Regions of the Sofa

696

697 Figure 8 shows the UML diagram for the TextAnnotation and TemporalAnnotation base types.

698



699

700 Figure 8: TextAnnotation and TemporalAnnotation UML

701

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

3.3.2.4 Options for Extending Annotation Type System (Not Normative)

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.

The first option is to subclass the Annotation types, as in Figure 9. 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.

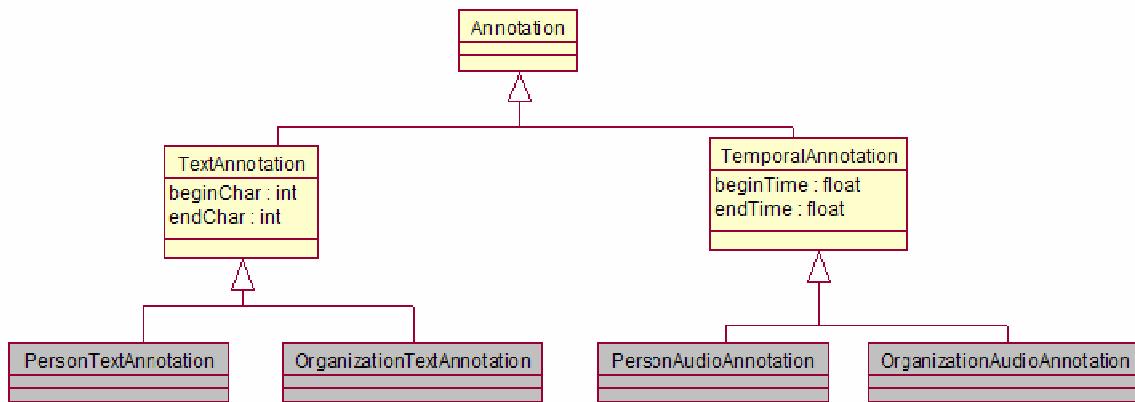


Figure 9: Extending the base type system through subclassing.

The second option, shown in Figure 10, is to create an Entity type that subsumes the relevant semantic classes, and associate the Annotation with the appropriate Entity type. In this model, an Annotation is viewed as an occurrence of an Entity reference in a particular modality. The advantage of this approach is that all annotations corresponding to a particular Entity type (e.g. Person or Organization), regardless of the modality they are expressed in, will have the same occurrence value and can thus be easily grouped together. It does, however, push the semantic information about the annotation into an associated type that needs to be investigated rather than being immediately available in the type of the Annotation object. In other words, it introduces a level of indirection for accessing the semantic information about the Annotation. However, an additional advantage of this approach is that it allows for multiple Annotations to be associated with a single Entity, so that for instance multiple distinct references to a person in a text can be linked to a single Entity object representing that person.

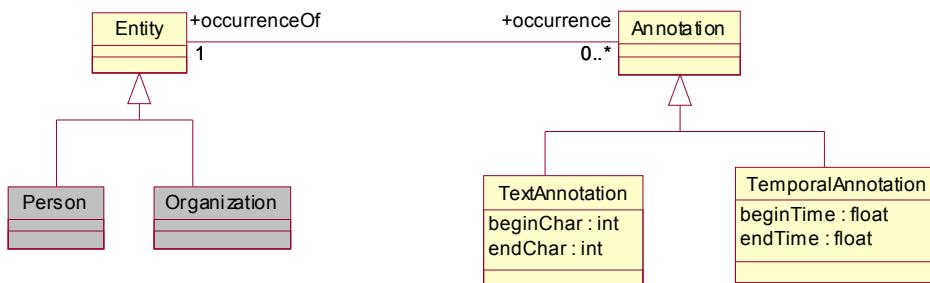
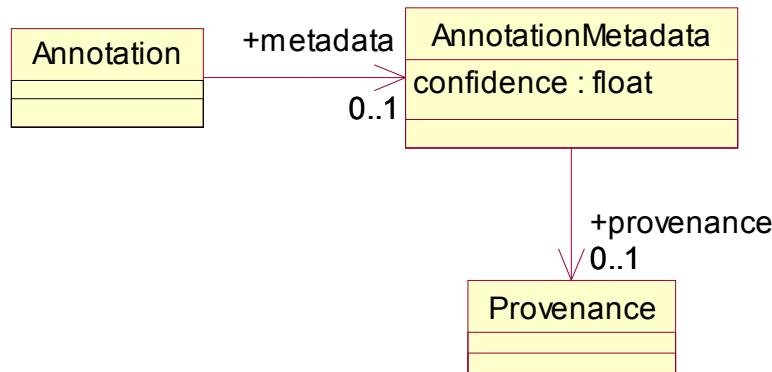


Figure 10: Associate Annotation with Entity type

735
736

737 3.3.2.5 Additional Annotation Metadata

738 In many applications, it will be important to capture metadata about each annotation. In the Base Type
739 System, we introduce an `AnnotationMetadata` class to capture this information. The standard, as
740 expressed in Figure 11, provides fields for *confidence*, a float indicating how confident the annotation
741 engine that produced the annotation was in that annotation, and *provenance*, a `Provenance` object
742 which stores information about the source of an annotation. Users may subclass `AnnotationMetadata`
743 and `Provenance` as needed to capture additional application-specific information about annotations.

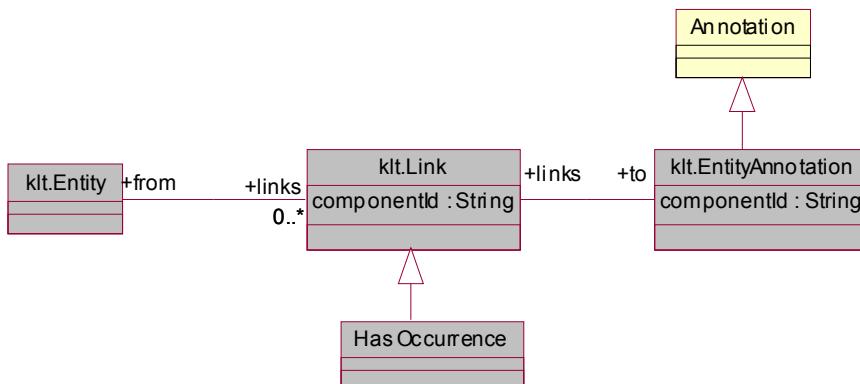


744
745

Figure 11: `AnnotationMetadata` types in the base type system.

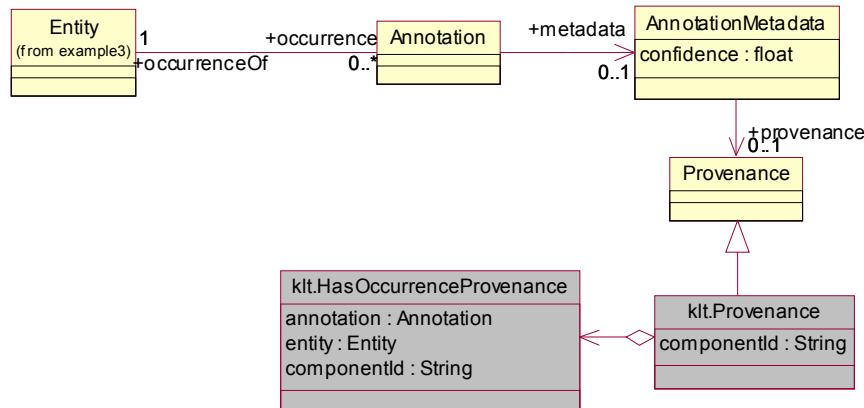
746 3.3.2.6 An Example of Annotation Model Extension (Not Normative)

747 The Base Type System is intended to specify only the top-level classes for the Annotation system used in
748 an application. Users will need to extend these classes in order to meet the particular needs of their
749 applications. An example of how an application might extend the base type system comes from
750 examining the redesign of IBM's Knowledge Level Types in terms of the standard. The current model in
751 KLT appears in Figure 12. It uses the `Annotation` class, but subclasses it with its own `EntityAnnotation`,
752 models coreference with a reified `HasOccurrence` link, and captures provenance through a *componentId*
753 attribute. Using the standard base type system, this type system could be refactored as in Figure 13.
754 This refactoring uses the standard definitions of `Annotation` and `Entity`, but captures provenance using the
755 an extended `Provenance` object.



756
757

Figure 12: IBM's Knowledge Level Types

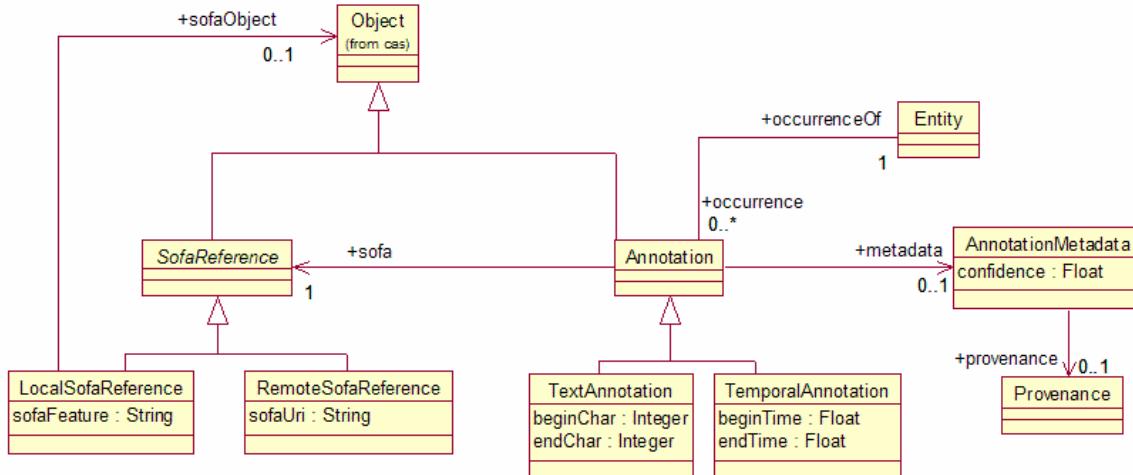


758
759
760
761

Figure 13: Refactoring of KLT using the standard base type system.

3.3.2.7 Complete Annotation Model

763 Figure 14 shows the complete UML definition for the Annotation Base Type System.



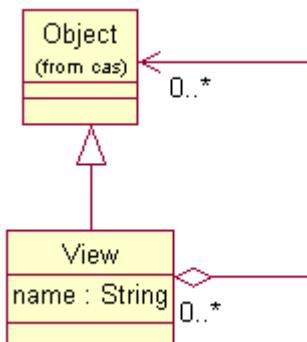
764
765 **Figure 14: Annotation Model Complete UML**
766

3.3.3 View Type System

768 A View, depicted in Figure 15, is a named collection of *objects* in a CAS. In general a view can represent
769 any subset of the *objects* in the CAS for any purpose. It is intended however that Views represent
770 different perspectives of the artifact represented by the CAS. Each View is intended to partition the
771 artifact metadata to capture a specific perspective.

772
773 For example, given a CAS representing a document, one View may capture the metadata describing an
774 English translation of the document while another may capture the metadata describing a French
775 translation of the document.

776 In another example, given a CAS representing a document, one view many contain an analysis produced
777 using company-confidential data another may produce an analysis using generally available data.
778



779
780 **Figure 15: View Type**
781
782 UIMA does not require the use of Views. However, our experiences developing Apache UIMA suggest
783 that it is a useful design pattern to organize the metadata in a complex CAS by partitioning it into Views.
784 Individual analytics may then declare that they require certain Views as input or produce certain Views as
785 output.

786
787 Any application-specific type system could define a *class* that represents a named collection of *objects*
788 and then refer to that *class* in an analytic's behavioral specification. However, since it is a common design
789 pattern we consider defining a standard View *class* to facilitate interoperability between components that
790 operate on such collections of *objects*.

791
792 The members of a view are those *objects* explicitly asserted to be contained in the View. Referring to the
793 UML in Figure 15, we mean that there is an explicit reference from the View to the member *object*.
794 Members of a view may have references to other *objects* that are not members of the same View. A
795 consequence of this is that we cannot in general "export" the members of a View to form a new self-
796 contained CAS, as there could be dangling references. We define the **reference closure of a view** to
797 mean the collection of objects that includes all of the members of the view but also contains all other
798 *objects* referenced either directly or indirectly from the members of the view.

799 **3.3.3.1 Anchored View**

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

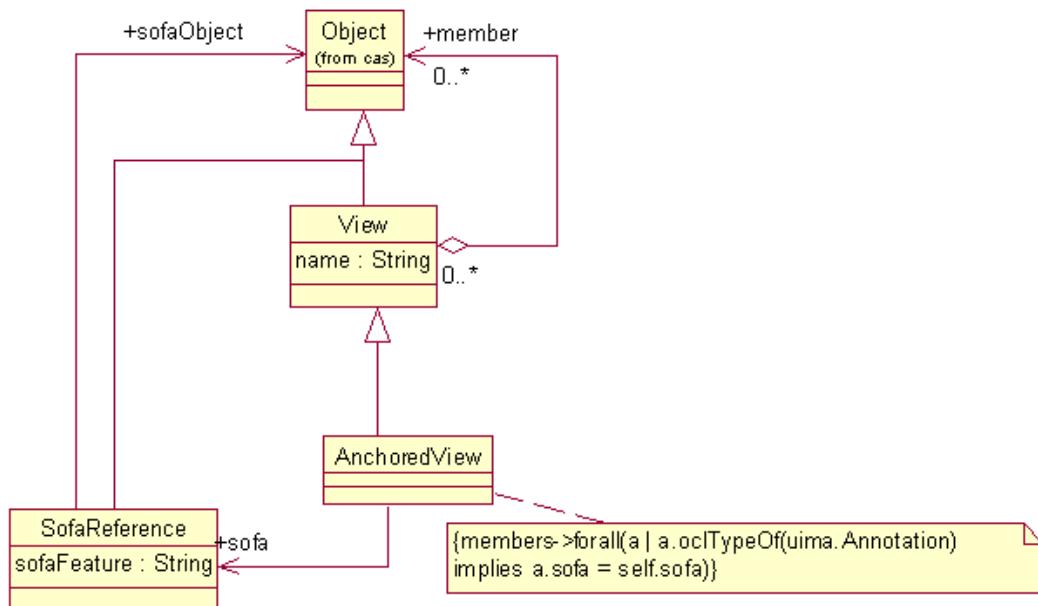
803
804 AnchoredView is as a subtype of View that has a named association with exactly one particular *object* via
805 the standard *feature sofa*.

806
807 An AnchoredView requires that all Annotation *objects* that are members of the AnchoredView have their
808 sofa *feature* refer to the same SofaReference that is referred to by the View's sofa *feature*.

809
810 Simply put, all annotations in an AnchoredView annotate the same subject of analysis.

811
812 Figure 16 shows a UML diagram for the AnchoredView type, including an OCL constraint
813 expression[OCL1] specifying the restriction on the sofa feature of its member annotations.

814



815

816

Figure 16: Anchored View Type

817

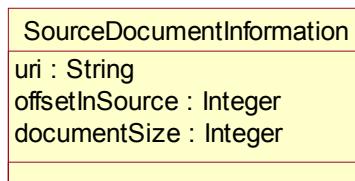
818 The concept of an AnchoredView addresses common use cases. For example, an analytic written to
819 analyze the detagged representation of a document will likely only be able to interpret Annotations that
820 label and therefore refer to regions in that detagged representation. Other Annotations,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

823

824 If a chain of analytics are intended to all analyze the same representation of the artifact, they can all
825 declare that AnchoredView as a precondition in their Behavioral Specification (see Section 3.5 Behavioral
826 Metadata). With AnchoredViews, all the analytics in the chain can simply assume that all regional
827 references of all Annotations that are members of the AnchoredView refer to the AnchoredView's sofa.
828 This saves them the trouble of filtering Annotations to ensure they all refer to a particular sofa.

829 **3.3.4 Source Document Information**

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



833

834

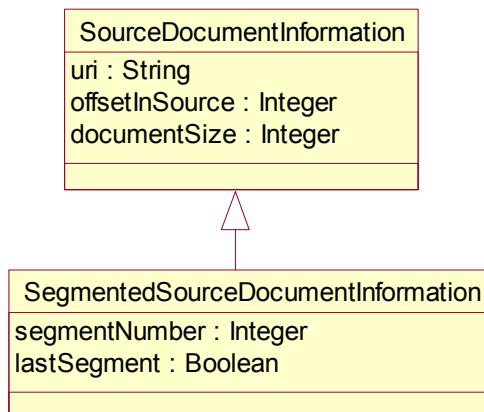
Figure 17: Source Document Information UML

835 Figure 17 contains the specification of a SourceDocumentInformation type included in the BaseType
836 System that can be stored in a CAS and used to capture this information. Here, the offsetInSource and
837 documentSize attributes must be byte offsets into the source, since that source may be in any modality.

838

839 **3.3.4.1 Example Extension of Source Document Information (Not Normative)**

840 If an application needs to process multiple segments of an artifact and later merger the results, then
841 additional offset information may also be needed on each segment. While not a standard part of the
842 specification, a representative extension to the `SourceDocumentInformation` type to capture such
843 information is shown in Figure 18. This `SegmentedSourceDocumentInformation` type adds features
844 to track information about the segment of the source document the CAS corresponds to. Specifically, it
845 adds an Integer `segmentNumber` to capture the segment number of this segment, and a Boolean
846 `lastSegment` that is true when this segment is the last segment derived from the source document.



847

848 **Figure 18: Segmented Source Document Information UML**

849

850 **3.3.5 Formal Specification**

851 The Base Type System is formally defined by the Ecore model in Appendix B.3. UIMA components or
852 applications are not required to use the Base Type System. However, the XML namespace
853 <http://docs.oasis-open.org/uima/cas.ecore> is reserved for use by the Base Type System Ecore model,
854 and user-defined Type Systems (such as those referenced in PE metadata as discussed in Section
855 3.6.2.3) MUST NOT define their own type definitions in this namespace.

856 **3.4 Abstract Interfaces**

857 The UIMA specification has defines two fundamental types of Processing Elements (PEs) that developers
858 may implement: *Analytics* and *Flow Controllers*. In this section we give an abstract definition of the
859 operations that these PE types support. Refer to Figure 19 for a UML model of the Analytic interfaces
860 and Figure 20 for a UML model of the FlowController interface. The abstract definitions in this section lay
861 the foundation for the concrete service specification defined in Section 3.7.

862 **3.4.1 Processing Element**

863 The base `ProcessingElement` interface defines the following operations, which are common to all
864 subtypes of `ProcessingElement`:

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

869

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

873 **3.4.2 Analytic**

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

879 **3.4.3 Analyzer**

880 The Analyzer interface defines two additional operations:

- 881 • processCas, which takes a single CAS plus a list of Sofas to analyze, and returns either an
882 updated CAS, or a set of updates to apply to the CAS.
- 883 • processCasBatch, which takes multiple CASes, each with a list of Sofas to analyze, and returns
884 a response that contains, for each of the input CASes: an updated CAS, a set of updates to apply
885 to the CAS, or an exception.

886

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

890

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

895 **3.4.4 CAS Multiplier**

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

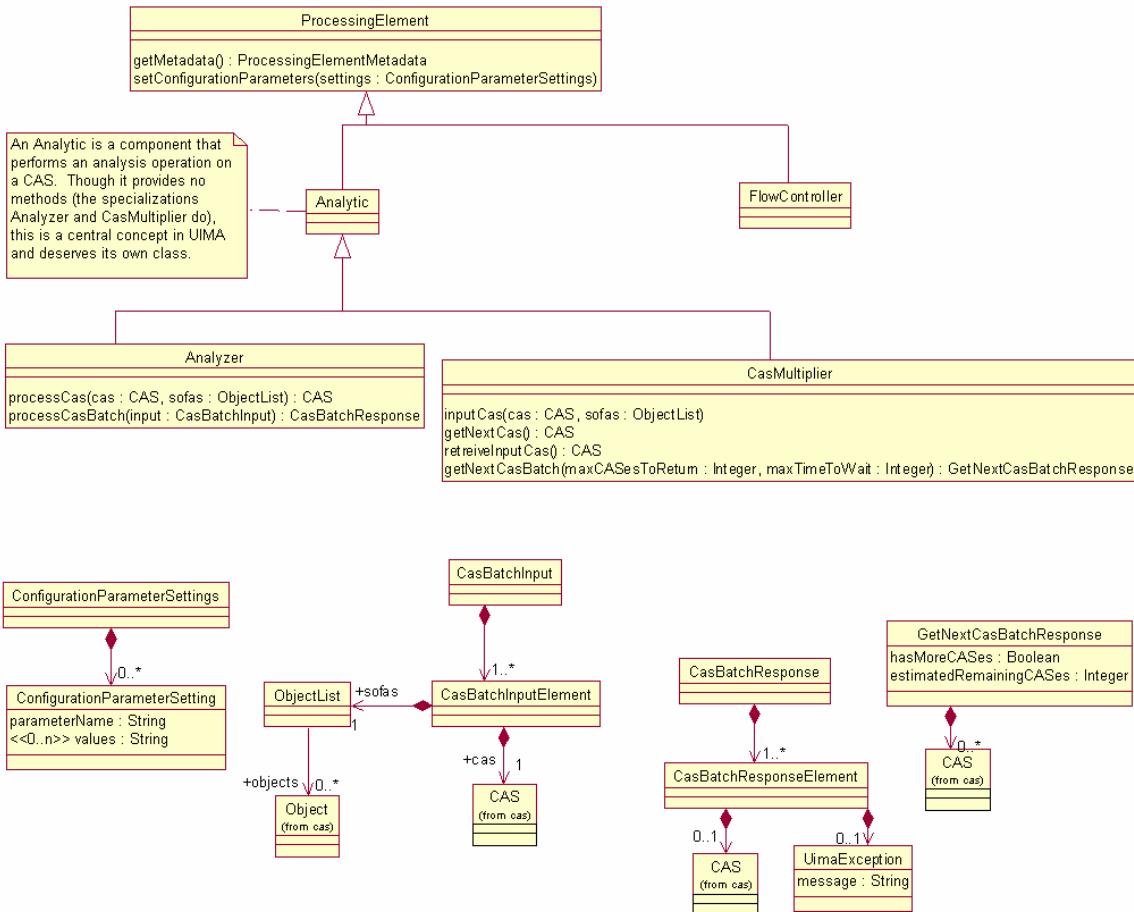
900

- 901 • inputCas, which takes a CAS plus a list of Sofas, but returns nothing.
- 902 • getNextCas, which takes no input and returns a CAS. This returns the next output CAS. An
903 empty response indicates no more output CASes.
- 904 • retrieveInputCas, which takes no arguments and returns the original input CAS, possibly
905 updated.
- 906 • getNextCasBatch, which takes a maximum number of CASes to return and a maximum
907 amount of time to wait (in milliseconds), and returns a response that contains: Zero or more
908 CASes (up to the maximum number specified), a Boolean indicating whether any more CASes
909 remain, and an estimate of the number of CASes remaining (if known).

910

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

915



916

917

Figure 19: Abstract Interfaces UML (Flow Controller Detail Omitted)

3.4.5 Flow Controller

919 The FlowController interface defines the operations:

- 920 • `addAvailableAnalytics`, which provides the Flow Controller with access to the Analytic Metadata
921 for all of the Analytics that the Flow Controller may route CAses to. This may be called multiple
922 times, if new analytics are added to the system after the original call is made.
- 923 • `removeAvailableAnalytics`, which instructs the Flow Controller to remove some Analytics from
924 consideration as possible destinations.
- 925 • `setAggregateMetadata`, which provides the Flow Controller with Processing Element Metadata
926 that identifies and describes the desired behavior of the entire flow of components that the
927 FlowController is. The most common use for this is to specify the desired outputs of the
928 aggregate, so that the Flow Controller can make decisions about which analytics need to be
929 invoked in order to produce those outputs.
- 930 • `getNextDestinations`, which takes a CAS and returns one or more destinations for this CAS.
- 931 • `continueOnFailure`, which can be called by the aggregate/application when a Step issued by the
932 FlowController failed. The FlowController returns true if it can continue, and can change the
933 subsequent flow in any way it chooses based on the knowledge that a failure occurred. The
934 FlowController returns false if it cannot continue.

935

936 The application or aggregate framework containing the FlowController is expected to call
937 `addAvailableAnalytics` and `setAggregateMetadata` before making calls to `getNextDestinations`. When

938 getNextDestinations is called, the FlowController implementation uses the available metadata along with
939 any data in the CAS to choose the next destinations from this set of analytics. The FlowController
940 responds with the a Step object, of which there are three subtypes:

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

948

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

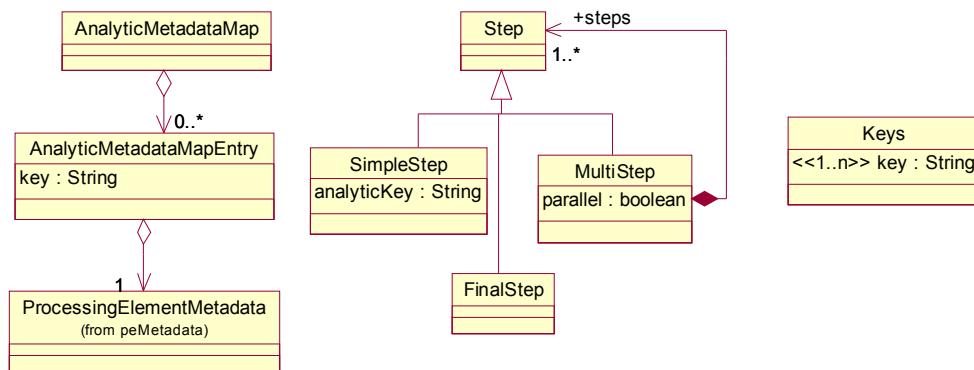
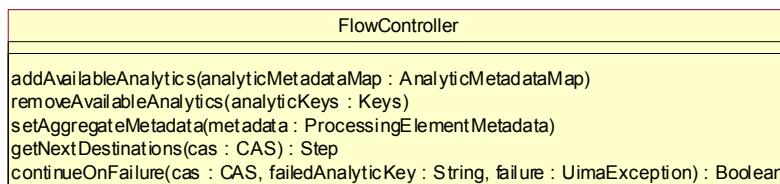
953

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

959

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

964



965

966

Figure 20: Flow Controller Abstract Interface UML

967

968 **3.4.6 Examples (Not Normative)**

969 **3.4.6.1 Analyzer Example**

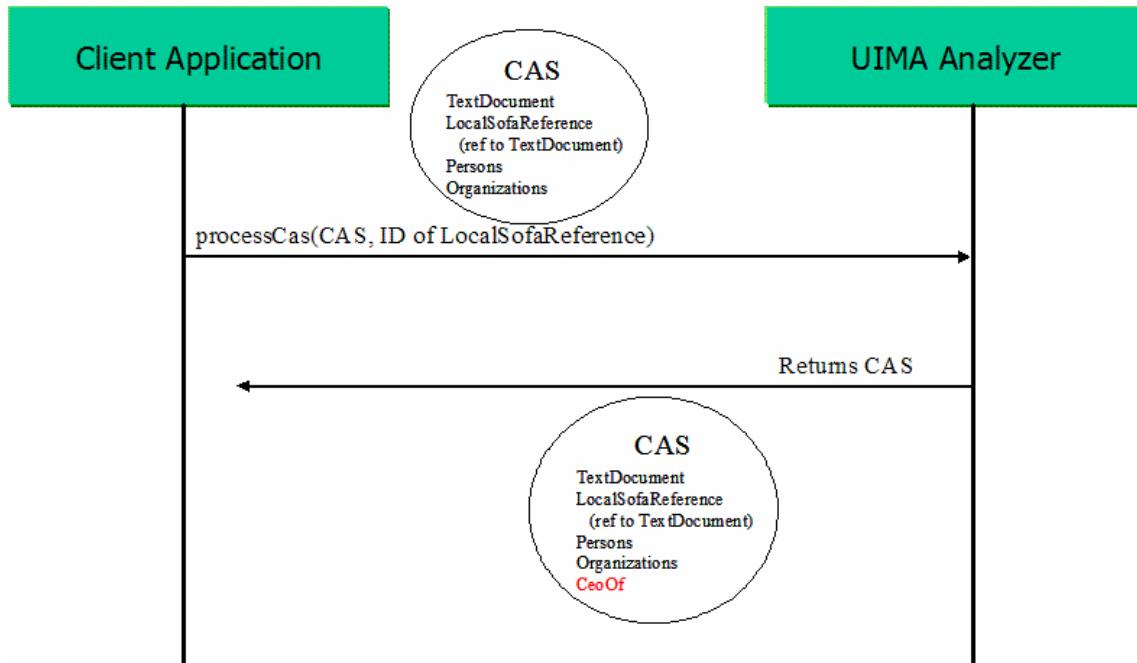
970 The sequence diagram in Figure 21 illustrates how a client interacts with a UIMA Analyzer service. In this
971 example the Analyzer is a "CEO Relation Detector," which given a text document with Person and
972 Organization annotations, can find occurrences of CeoOf relationships between them.

973

974 The example shows that the client calls the `processCas(cas, sofas)` operation. The first argument is
975 the CAS to be processed (in XMI format). It contains a TextDocument, a LocalSofaReference (see
976 Section 3.3.2.2) that points to a text field in that TextDocument, and Person and Organization annotations
977 that annotate regions in the TextDocument. The second argument is the xmi:id of the
978 LocalSofaReference object, indicating that this object should be considered the subject of analysis (Sofa)
979 for this operation.

980

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



983

984 **Figure 21: Analyzer Sequence Diagram**

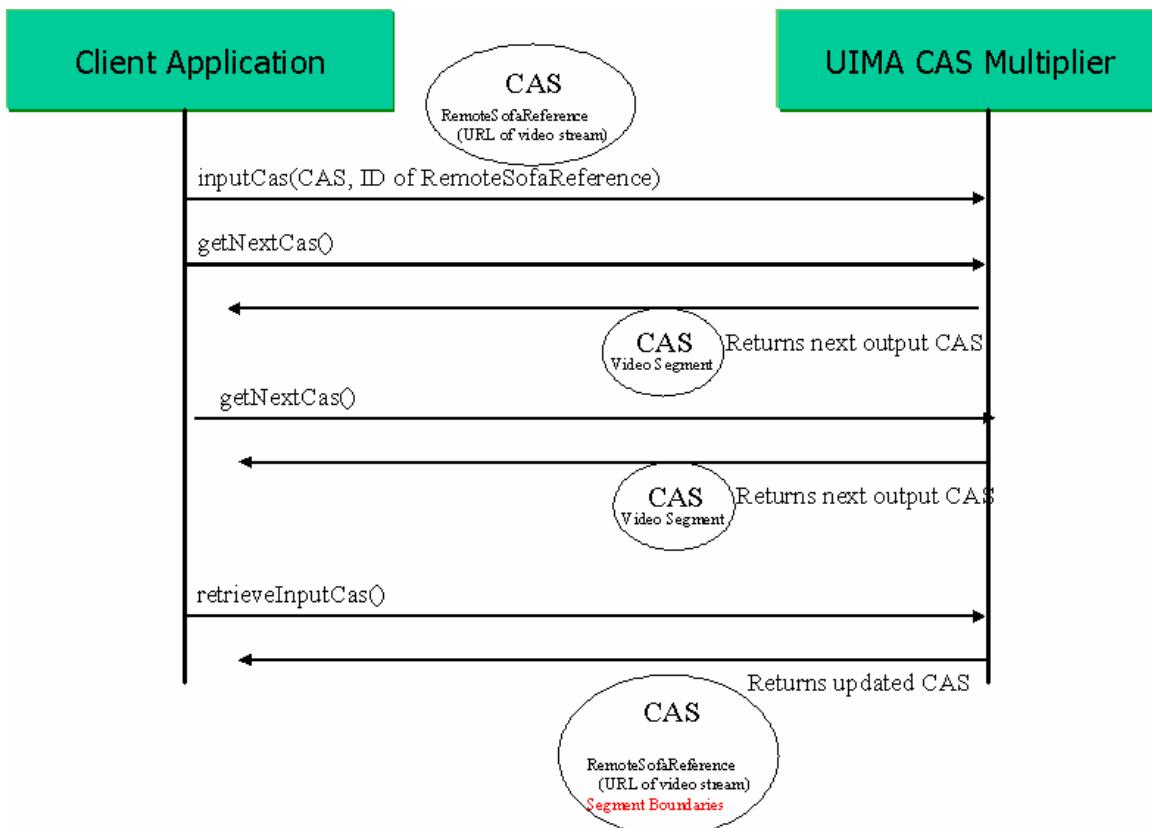
985 **3.4.6.2 CAS Multiplier Example**

986 The sequence diagram in Figure 22 illustrates how a client interacts with a UIMA CAS Multiplier service.
987 In this case the CAS Multiplier is a Video Segmente, which given a video stream divides it into individual
988 segments.

989

990 The client first calls the `inputCas(cas, sofas)` operation. The first argument is a CAS containing a
991 reference to the video stream to analyze. Typically a large artifact such as a video stream is represented
992 in the CAS as a reference (using the `RemoteSofaReference` base type introduced in section 3.3.2.2),
993 rather than included directly in the CAS as is typically done with a text document. The second argument

994 to `inputCas` is the `xmi:id` of the `RemoteSofaReference` object, so that the service knows that this is the
 995 subject of analysis for this operation.
 996
 997 The client then calls the `getNextCas` operation. This returns a CAS containing the data for the first
 998 segment (or possibly, a reference to it). The client repeatedly calls `getNextCas` to obtain each
 999 successive segment. Eventually, `getNextCas` returns null to indicate there are no more segments.
 1000
 1001 Finally, the client calls the `retrieveInputCas` operation. This returns the original CAS, with additional
 1002 information added. In this example, the Video Segmenter adds information to the original CAS indicating
 1003 at what time offsets each of the segment boundaries were detected.
 1004



1005
 1006
 1007

Figure 22: CAS Multiplier Sequence Diagram

1008 **3.4.7 Formal Specification**

1009 The following subsections specify requirements that a particular type of UIMA service must provide an
 1010 operation with certain inputs and outputs. For example, a UIMA PE service must implement a
 1011 `getMetaData` operation that returns standard UIMA PE Metadata. In all cases, the protocol for invoking
 1012 this operation is not defined by the standard. However, the format in which data is sent to and from the
 1013 service is MUST be the standard UIMA XML representation. Implementations MAY define additional
 1014 operations that use other formats.

1015 **3.4.7.1 ProcessingElement.getMetaData**

1016 A UIMA Processing Element (PE) Service MUST implement an operation named `getMetaData`. This
 1017 operation MUST take zero argument and MUST return PE Metadata XML as defined in Section 3.6.4. In

1018 the following sections, we use the term “this PE Service’s Metadata” to refer to the PE Metadata returned
1019 by this operation.

1020 **3.4.7.2 ProcessingElement.setConfigurationParameters**

1021 A UIMA Processing Element (PE) Service MUST implement an operation named
1022 `setConfigurationParameters`. This operation MUST accept one argument, an instance of the
1023 `ConfigurationParameterSettings` type defined by the XML Schema in Section B.7.

1025 The PE Service MUST return an error if the `ConfigurationParameterSettings` object passed to this
1026 method contains any of:

- 1027 1. a `parameterName` that does not match any of the parameter names declared in this PE Service’s
1028 Metadata.
- 1029 2. multiple values for a parameter that is not declared as `multiValued` in this PE Service’s Metadata.
- 1030 3. a value that is not a valid instance of the type of the parameter as declared in this PE Service’s
1031 Metadata. To be a valid instance of the UIMA configuration parameter type, the value must be a
1032 valid instance of the corresponding XML Schema datatype in Table 1: Mapping of UIMA
1033 Configuration Parameter Types to XML Schema Datatypes, as defined by the XML Schema
1034 specification [XMLS2].

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

1036 **Table 1: Mapping of UIMA Configuration Parameter Types to XML Schema Datatypes**

1037
1038 After a client calls `setConfigurationParameters`, those parameter settings MUST be applied to all
1039 subsequent requests from that client, until such time as a subsequent call to `setConfigurationParameters`
1040 specifies new values for the same parameter(s). If the PE service is shared by multiple clients, the PE
1041 service MUST provide a way to keep their configuration parameter settings separate.

1043 **3.4.7.3 Analyzer.processCas**

1044 A UIMA Analyzer Service MUST implement an operation named `processCas`. This operation MUST
1045 accept two arguments. The first argument is a CAS, represented in XMI as defined in Section 3.1.7. The
1046 second argument is a list of `xmi:ids` that identify `SofaReference` objects which the Analyzer is expected
1047 to analyze. This operation MUST return a valid XMI document which is either a valid CAS (as defined in
1048 Section 3.1.7) or a description of changes to be applied to the input CAS using the XMI differences
1049 language defined in [XMI1].

1050

1051 The output CAS of this operation represents an update of the input CAS. Formally, this means :

- 1052 1. All objects in the input CAS must appear in the output CAS, except where an explicit delete or
1053 modification was performed by the service (which is only allowed if such operations are declared
1054 in the Behavioral Metadata element of this service’s PE Metadata).

- 1055 2. For the processCas operation, an object that appears in both the input CAS and output CAS must
1056 have the same value for xmi:id.
1057 3. No newly created object in the output CAS may have the same xmi:id as any object in the input
1058 CAS.

1059

1060 The input CAS may contain a reference to its type system (see Section 3.1.5). If it does not, then the
1061 PE's type system (see Section 3.6.2.3) may provide definitions of the types. If the CAS contains an
1062 instance of a type that is not defined in either place, then the PE may decide to reject the CAS and return
1063 an error. Some PE's may be capable of handling undefined types, however, and these PE's need not
1064 return an error.

1065

1066 **3.4.7.4 Analyzer.processCasBatch**

1067 A UIMA Analyzer Service MUST implement an operation named `processCasBatch`. This operation
1068 MUST accept an argument which consists of one or more CASes, each with an associated list of xmi:ids
1069 that identify `SofaReference` objects in that CAS. This operation MUST return a response that consists of
1070 multiple elements, one for each input CAS, where each element is either valid XMI document which is
1071 either a valid CAS (as defined in Section 3.1.7), a description of changes to be applied to the input CAS
1072 using the XMI differences language defined in [XMI1], or an exception message.

1073

1074 The CASes that result from calling `processCasBatch` MUST be identical to the CASes that would result
1075 from several individual `processCas` operations each taking only one of the CASes as input.

1076 **3.4.7.5 CasMultiplier.inputCas**

1077 A UIMA CAS Multiplier service MUST implement an operation named `inputCas`. This operation MUST
1078 accept two arguments. The first argument is a CAS, represented in XMI as defined in Section 3.1.7. The
1079 second argument is a list of xmi:ids that identify `SofaReference` objects which the Analyzer is expected
1080 to analyze. This operation returns nothing.

1081

1082 The CAS that is passed to this operation becomes this CAS Multiplier's *active CAS*.

1083 **3.4.7.6 CasMultiplier.getNextCas**

1084 A UIMA CAS Multiplier service MUST implement an operation named `getNextCas`. This operation
1085 MUST take zero arguments. This operation MUST return a valid CAS as defined in Section 3.1.7, or a
1086 result indicating that there are no more CASes available.

1087

1088 If the client calls `getNextCas` when this CAS Multiplier has no active CAS, then this CAS Multiplier MUST
1089 return an error.

1090 **3.4.7.7 CasMultiplier.retrieveInputCas**

1091 A UIMA CAS Multiplier service MUST implement an operation named `retrieveInputCas`. This operation
1092 MUST take zero arguments and must return a valid XMI document which is either a valid CAS (as defined
1093 in Section 3.1.7) or a description of changes to be applied to the CAS Multiplier's active CAS using the
1094 XMI differences language defined in [XMI1].

1095

1096 If the client calls `retrieveInputCas` when this CAS Multiplier has no active CAS, then this CAS Multiplier
1097 MUST return an error.

1098

1099 After this method completes, this service no longer has an active CAS, until the client's next call to
1100 `inputCas`.

1101 **3.4.7.8 CasMultiplier.getNextCasBatch**

1102 A UIMA CAS Multiplier service MUST implement an operation named `getNextCasBatch`. This
1103 operation MUST take two arguments, both of which are integers. The first argument (named
1104 `maxCASEsToReturn`) specifies the maximum number of CASEs to be returned, and the second argument
1105 (named `maxTimeToWait`) indicates the maximum number of milliseconds to wait. This operation MUST
1106 return an object with three fields:

- 1107 1. Zero or more valid CASEs as defined in Section 3.1.7. The number of CASEs MUST NOT exceed
1108 the value of the `maxCASEsToReturn` argument.
- 1109 2. a Boolean indicating whether more CASE remain to be retrieved.
- 1110 3. An estimated number of remaining CASEs. The estimated number of remaining CASEs may be
1111 set to -1 to indicate an unknown number.

1112

1113 The call to `getNextCasBatch` SHOULD attempt to complete and return a response in no more than the
1114 amount of time specified (in milliseconds) by the `maxTimeToWait` argument.

1115

1116 If the client calls `getNextCasBatch` when this CAS Multiplier has no active CAS, then this CAS Multiplier
1117 MUST return an error.

1118

1119 CASEs returned from `getNextCasBatch` MUST be equivalent to the CASEs that would be returned from
1120 individual calls to `getNextCas`.

1121 **3.4.7.9 FlowController.addAvailableAnalytics**

1122 A UIMA Flow Controller service MUST implement an operation named `addAvailableAnalytics`. This
1123 operation MUST accept one argument, a Map from String keys to PE Metadata objects. Each of the
1124 String keys passed to this operation is added to the set of *available analytic keys* for this Flow Controller
1125 service.

1126 **3.4.7.10 FlowController.removeAvailableAnalytics**

1127 A UIMA Flow Controller service MUST implement an operation named `removeAvailableAnalytics`.
1128 This operation MUST accept one argument, which is a collection of one or more String keys. If any of the
1129 String keys passed to this operation are not a member of the set of *available analytic keys* for this Flow
1130 Controller service, an error MUST be returned. Each of the String keys passed to this operation is
1131 removed from the set of *available analytic keys* for this FlowController service.

1132 **3.4.7.11 FlowController.setAggregateMetadata**

1133 A UIMA Flow Controller service MUST implement an operation named `setAggregateMetadata`. This
1134 operation MUST take one argument, which is valid PE Metadata XML as defined in Section 3.6.4.

1135

1136 There are no formal requirements on what the Flow Controller does with this PE Metadata, but the
1137 intention is for the PE Metadata to specify the desired outputs of the workflow, so that the Flow Controller
1138 can make decisions about which analytics need to be invoked in order to produce those outputs.

1139 **3.4.7.12 FlowController.getNextDestinations**

1140 A UIMA Flow Controller service MUST implement an operation named `getNextDestinations`. This
1141 operation MUST accept one argument, which is an XML CAS as defined in Section 3.1.7 and MUST
1142 return an instance of the `Step` type defined by the XML Schema in Section B.7.

1143

1144 The different types of Step objects are defined in the UML diagram in Figure 20 and XML schema in
1145 Appendix B.7. Their intended meanings are documented in section 3.4.5.

1146

1147 Each analyticKey field of a Step object returned from the `getNextDestinations` operation MUST be a
1148 member of the set of *active analytic* keys of this Flow Controller service.

1149 **3.4.7.13 FlowController.continueOnFailure**

1150 A UIMA FlowController service MUST define an operation named `continueOnFailure`. This operation
1151 MUST accept three arguments as follows. The first argument is an XML CAS as defined in Section 3.1.7.
1152 The second argument is a String key. The third argument is an instance of the `UimaException` type
1153 defined in the XML schema in Section B.7.

1154

1155 If the String key is not a member of the set of *active analytic* keys of this Flow Controller, then an error
1156 must be returned.

1157

1158 This method is intended to be called by the client when there was a failure in executing a Step issued by
1159 the FlowController. The client is expected to pass the CAS that failed, the analytic key from the Step
1160 object that was being executed, and the exception that occurred.

1161

1162 Given that the above assumptions hold, the `continueOnFailure` operation SHOULD return true if a further
1163 call to `getNextDestinations` would succeed, and false if a further call to `getNextDestinations` would fail.

1164

1165 **3.5 Behavioral Metadata**

1166 The Behavioral Metadata of an analytic declaratively describes what the analytic does; for example, what
1167 types of CASs it can process, what elements in a CAS it analyzes, and what sorts of effects it may have
1168 on CAS contents as a result of its application.

1169 **3.5.1 Goals**

1170 1. **Discovery:** Enable both human developers and automated processes to search a repository and
1171 locate components that provide a particular function (i.e., works on certain input, produces certain
1172 output)

1173

1174 2. **Composition:** Support composition either by a human developer or an automated process.

1175 a. Analytics should be able to declare what they do in enough detail to assist manual
1176 and/or automated processes in considering their role in an application or in the
1177 composition of aggregate analytics.

1178 b. Through their Behavioral Metadata, Analytics should be able to declare enough detail
1179 as to enable an application or aggregate to detect “invalid” compositions/workflows
1180 (e.g., a workflow where it can be determined that one of the Analytic’s preconditions
1181 can never be satisfied by the preceding Analytic).

1182

1183 3. **Efficiency:** Facilitate efficient sharing of CAS content among cooperating analytics.
1184 If analytics declare which elements of the CAS (e.g., *views*) they need to receive and which elements they do not
1185 need to receive, the CAS can be filtered or split prior to sending it to target analytics, to achieve
1186 transport and parallelization efficiencies respectively.

1187
1188 Note that analytics are not required to declare behavioral metadata. If an analytic does not provide
1189 behavioral metadata, then an application using the analytic cannot assume anything about the operations
1190 that the analytic will perform on a CAS.

1192 3.5.2 Elements of Behavioral Metadata

1193 Behavioral Metadata breaks down into the following categories:

- 1194
- 1195 • **Analyzes:** Types of objects (Sofas) that the analytic intends to produce annotations over.
 - 1196 • **Required Inputs:** Types of objects that must be present in the CAS for the analytic to operate.
 - 1197 • **Optional Inputs:** Types of objects that the analytic would consult if they were present in the CAS.
 - 1198 • **Creates:** Types of objects that the analytic may create.
 - 1199 • **Modifies:** Types of objects that the analytic may modify.
 - 1200 • **Deletes:** Types of objects that the analytic may delete.

1201
1202 For each of these elements, if an analytic declares the element at all, it must completely declare its
1203 behavior with respect to that element. For example, if an analytic declares a *creates* expression
1204 containing only type X, then it must not create instances of any types other than X. This is a requirement
1205 for the composition and efficiency goals that we describe next.

1207 3.5.3 Example (Not Normative)

1208 Consider a “CeoOf Relation Detector” analytic that receives as input a text document in which Persons
1209 and Organizations have been annotated, and looks for a relationship that a Person is the CEO Of an
1210 Organization. This analytic would declare its Behavioral Metadata as shown in Figure 23.

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

<pre><analyzes> <type name="org.example:TextDocument"/> </analyzes></pre>	Type of Sofa that the Analytic will process
<pre><requiredInputs> <type name="org.example:Person"/> <type name="org.example:Organization"/> </requiredInputs></pre>	Inputs – may be required or optional
<pre><creates> <type name="org.example:CeoOf"/> </creates></pre>	Effects – objects that the analytic creates, modifies, or deletes

```
</behavioralMetadata>
```

1211
1212 Figure 23: Behavioral Metadata Example

1213 This satisfies the three design goals of Behavioral Metadata:

- 1214 • **Discovery:**
 - 1215 – A component repository can be searched to locate an analytic that produces CeoOf
1216 annotations.
- 1217 • **Composition:**
 - 1218 – Person and Organization annotations are required inputs, so a user knows to combine a
1219 Person annotator and a Relation annotator with the CeoOf annotator to produce a valid
1220 composition.
- 1221 • **Efficiency:**
 - 1222 – If the CAS contains objects in the CAS that are not declared in the analyzes, required
1223 inputs, or optional inputs (e.g., Place annotations), then these do not need to be sent to
1224 the analytic.

1225 **3.5.4 Using Views in Behavioral Metadata**

1226 An issue with the above example is the lack of any relationship of the Sofa to the Annotations. It is not
1227 explicit that the Person, Organization, and CeoOf annotations refer to the TextDocument Sofa. Worse,
1228 things become completely unclear for analytics that works with multiple Sofas. To address this problem,
1229 Behavioral Metadata may be expressed in terms of Views. For example:

```
1230
1231 <behavioralMetadata xmlns:org.example="http://docs.oasis-
1232 open.org/uima/org/example.ecore">
1233   <requiredView sofaType="org.example:Document">
1234     <requiredInputs>
1235       <type name="org.example:Person"/>
1236       <type name="org.example:Organization"/>
1237     </requiredInputs>
1238     <creates>
1239       <type name="org.example:CeoOf"/>
1240     </creates>
1241   </requiredView>
1242 </behavioralMetadata>
```

1243 **3.5.5 Formal Semantics for Behavioral Metadata**

1244 All Behavioral Metadata elements may be mapped to THREE kinds of expressions in a formal language:
1245 a **Precondition**, a **Postcondition**, and a **Projection Condition**.

1246 A *Precondition* is a predicate that qualifies CASs that the analytic considers valid input. More precisely
1247 the analytic's behavior would be considered unspecified for any CAS that did not satisfy the pre-condition.
1248 The pre-condition may be used by a framework or application to filter or skip CASs routed to an analytic
1249 whose pre-condition is not satisfied by the CASs. A human assembler or automated composition process
1250 can interpret the pre-conditions to determine if the analytic is suitable for playing a role in some aggregate
1251 composition.

1252 A *Postcondition* is a predicate that is declared to be true of any CAS after having been processed by the
1253 analytic, assuming that the CAS satisfied the precondition when it was input to the analytic.

1254 For example, if the pre-condition requires that valid input CASs contain People, Places and
1255 Organizations, but the Postconditions of the previously run Analytic asserts that the CAS will not contain
1256 all of these objects, then the composition is clearly invalid.

1263 A *Projection Condition* is a predicate that is evaluated over a CAS and which evaluates to a subset of the
1264 objects in the CAS. This is the set of objects that the Analytic declares that it will consider to perform its
1265 function.

1266

1267 The following is a high-level description of the mapping from Behavioral Metadata Elements to
1268 preconditions, postconditions, and projection conditions. For a precise definition of the mapping, see
1269 Section 3.5.8.3.

1270

1271 An `analyzes` or `requiredInputs` predicate translates into a precondition that all input CASes contain the
1272 objects that satisfy the predicates.

1273

1274 A `deletes` predicate translates into a postcondition that for each object O in the input CAS, if O does not
1275 satisfy the `deletes` predicate, then O is present in the output CAS.

1276

1277 A `modifies` predicate translates into a postcondition that for each object O in the input CAS, if O does not
1278 satisfy the `modifies` predicate, and if O is present in the output CAS (i.e. it was not deleted), then O has
1279 the same values for all of its slots.

1280

1281 For views, we add the additional constraint that objects are members of that View (and therefore
1282 annotations refer to the View's sofa). For example:

1283 `<requiredView sofaType="org.example:TextDocument">`

1284 `<requiredInputs>`

1285 `<type> org.example:Token</type>`

1286 `</requiredInputs>`

1287 `</requiredView>`

1288

1289 Translates into a precondition that the input CAS must contain an anchored view V where V is linked to a
1290 Sofa of type TextDocument and V.members contains at least one object of type Token.

1291

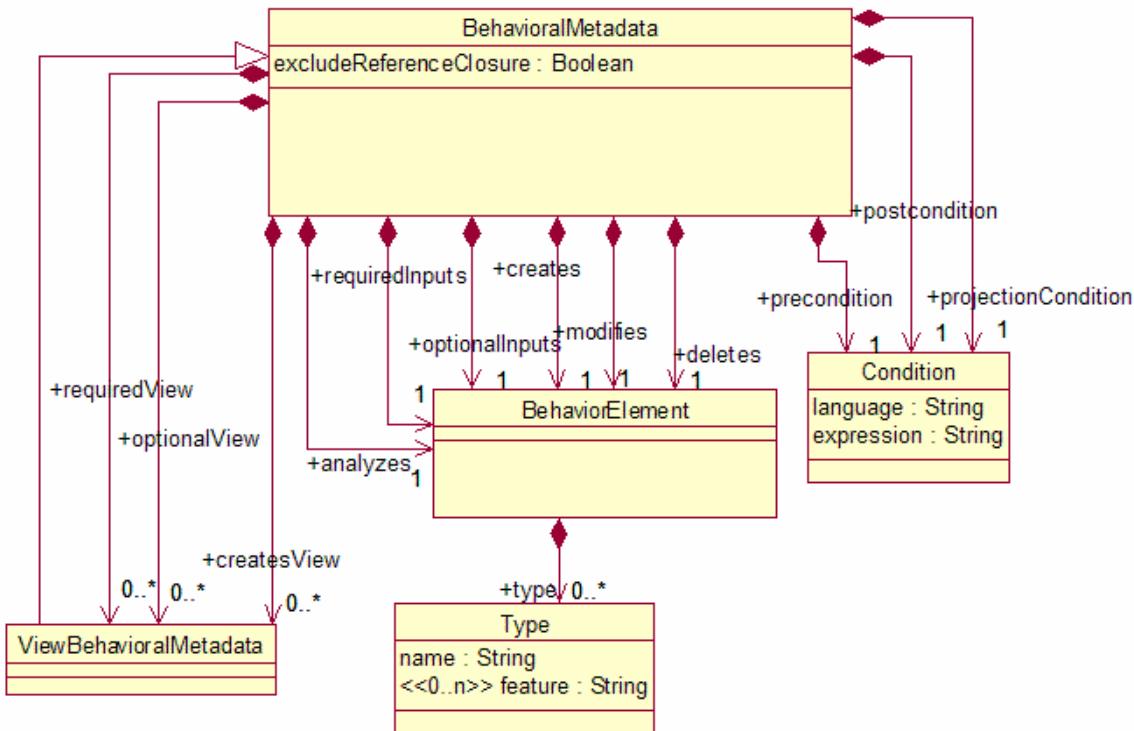
1292 Finally, the projection condition is formed from a disjunction of the “`analyzes`,” “`required inputs`,” and
1293 “`optional inputs`” predicates, so that any object which satisfies any of these predicates will satisfy the
1294 projection condition.

1295

1296 UIMA does not mandate a particular expression language for representing these conditions.
1297 Implementations are free to use any language they wish. However, to ensure a standard interpretation of
1298 the standard UIMA Behavior Elements, the UIMA specification defines how the Behavior Elements map
1299 to preconditions, postconditions, and projection conditions in the Object Constraint Language [OCL1], an
1300 OMG standard. See Section 3.5.8.3 for details.

1301 **3.5.6 Behavioral Metadata UML**

1302 The following UML diagram defines the UIMA Behavioral Metadata representation:



1303

1304

Figure 24: Behavioral Metadata UML**3.5.7 Behavioral Metadata XML Representation**

1306 For each of the Behavioral Metadata Elements (analyzes, required inputs, optional inputs, creates, 1307 modifies, and deletes), there will be a corresponding XML element. For each element a list of type 1308 names is declared.

1309

1310 To address some common situations where an analytic operates on a *view* (a collection of objects all 1311 referring to the same subject of analysis), we also provide a simple way for behavioral metadata to refer 1312 to views.

3.5.7.1 Type Naming Conventions

1314 In the XML behavioral metadata, type names are represented in the same way as in Ecore and XMI.

1315

1316 In UML (and Ecore), a *Package* is a collection of classes and/or other packages. All classes must be 1317 contained in a package.

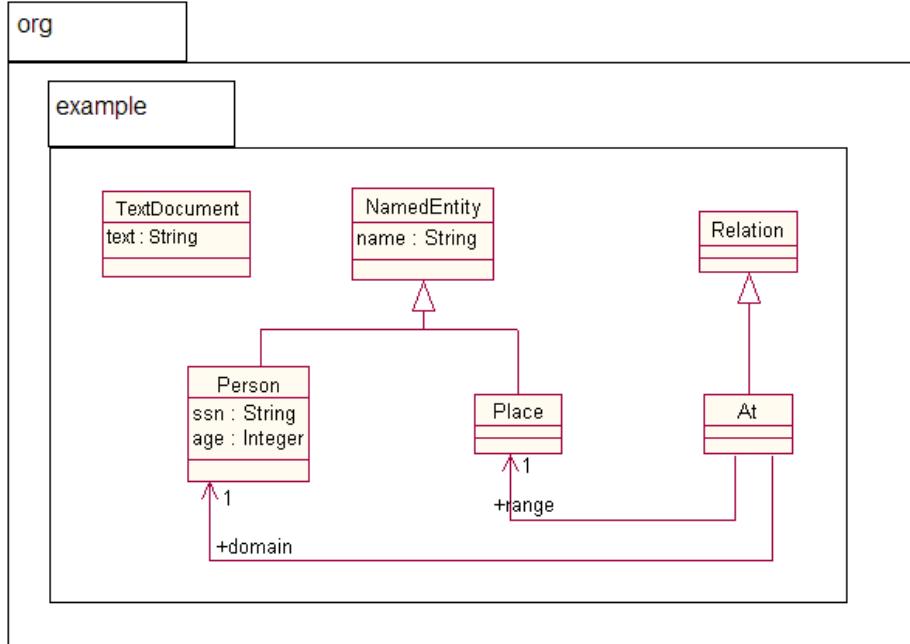
1318

1319 Figure 1 is a UML diagram of an example type system. It depicts a Package “org” containing a Package 1320 “example” containing several classes.

1321

1322

1323



1324

1325

Figure 25: Example Type System UML Model

1326

1327 In the Ecore model, each package is assigned (by the developer) three identifiers: a *name*, a *namespace URI*, and a *namespace prefix*. The *name* is a simple string that must be unique within the containing
 1328 package (top-level package names must be globally unique). The namespace URI and namespace prefix
 1329 are standard concepts in the XML namespaces spec [2] are used to refer to that package in XML,
 1330 including the behavioral metadata as well as the XMI CAS. An example is given below.

1332

1333 Figure 26 shows the relevant parts of the Ecore definition for this type system. Some details have been
 1334 omitted (marked with an ellipsis) to show only the parts where packages and namespaces are concerned,
 1335 and only a subset of the classes in the diagram are shown.

1336

```

<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="Place"
      eSuperTypes="#//example/NamedEntity"/>

```

1337

1338

Figure 26: Partial Ecore Representation of Example Type System

1339

1340 In this example, the namespace URI for the nested “example” project is `http://docs.oasis-`
1341 `open.org/uima/org/example.ecore`², and the corresponding prefix is `org.example`. It is
1342 important to note that the URI and prefix are arbitrarily determined by the type system developer and
1343 there is no required mapping from the package names “org” and “example” to the URI and prefix. In the
1344 above example, the namespace prefix have been set to “foo” and it would be completely valid. (However,
1345 for UIMA we could recommend or require the use of particular naming conventions.)
1346

1347 Now, to refer to a type name within the behavioral metadata XML, we use the namespace URI and prefix
1348 in the normal XML namespaces way, for example:
1349

```
1350 <behavioralMetadata xmlns:org.example="http://docs.oasis-
1351 open.org/uima/org/example.ecore">
1352     ...
1353     <type name="org.example:Place"/>
1354     ...
1355 </behavioralMetadata>
```

1356

1357 The “xmlns” attribute declares that the prefix “org.example” is bound to the URI `http://docs.oasis-`
1358 `open.org/uima/org/example.ecore`. Then, each time we want to refer to a type in that package, we use
1359 the prefix “org.example.”
1360

1361 Technically, the XML document does not have to use the same namespace prefix as what is in the Ecore
1362 model. It is only a guideline. The namespace URI is what matters. For example, the above XML is
1363 completely equivalent to the following
1364

```
1365 <behavioralMetadata xmlns:foo="http://docs.oasis-
1366 open.org/uima/org/example.ecore">
1367     ...
1368     <type name="foo:Place"/>
1369     ...
1370 </behavioralMetadata>
```

1371

1372 This is because the namespace URI is a globally unique identifier for the package, but the namespace
1373 prefix need only be unique within the current XML document. For more information on XML namespace
1374 syntax, see [XML1].
1375

² The use of the “http” scheme is a common XML namespace convention and does not imply that any actual http communication is occurring.

1376 The above discussion centered on the representation of type names in XML. There is a different
1377 representation needed within OCL expressions. Since OCL is not primarily XML-based, it does not use
1378 the XML namespace URIs or prefixes to refer to packages. Instead, OCL expressions refer directly to the
1379 simple package names separated by double colons, as in “org::example::Person”. For more information
1380 see [OCL1].

1381 **3.5.7.2 XML Syntax for Behavioral Metadata Elements**

1382 The following example is the behavioral metadata for an analytic that analyzes a Sofa of type
1383 TextDocument, requires objects of type Person, and will inspect objects of type Place if they are present.
1384 It may create objects of type At.

1385

```
1386 <behavioralMetadata xmlns:org.example="http://docs.oasis-
1387 open.org/uima/org/example.ecore" excludeReferenceClosure="true">
1388   <analyzes>
1389     <type name="org.example:TextDocument"/>
1390   </analyzes>
1391   <requiredInputs>
1392     <type name="org.example:Person"/>
1393     <type name="org.example:Place"/>
1394   </requiredInputs>
1395   <creates>
1396     <type name="org.example:At"/>
1397   </creates>
1398 </behavioralMetadata>
```

1399

1400 Note that the inheritance hierarchy declared in the type system is respected. So for example a CAS
1401 containing objects of type GovernmentOfficial and Country would be valid input to this analytic, assuming
1402 that the type system declared these to be subtypes of org.example:Person and org.example:Place,
1403 respectively.

1404

1405 The “excludeReferenceClosure” attribute on the Behavioral Metadata element, when set to true, indicates
1406 that objects that are referenced from optional/required inputs of this analytic will not be guaranteed to be
1407 included in the CAS passed to the analytic. This attribute defaults to false.

1408

1409 For example, assume in this example the Person object had an employer feature of type Company. With
1410 excludeReferenceClosure set to true, the caller of this analytic is not required to include Company objects
1411 in the CAS that is delivered to this analytic. If Company objects are filtered then the employer feature
1412 would become null. If excludeReferenceClosure were not set, then Company objects would be
1413 guaranteed to be included in the CAS.

1414 **3.5.7.3 Views**

1415 As described in section 3.5.4, we allow the behavioral metadata to refer to a View, where a View may
1416 collect all annotations referring to a particular Sofa.

1417

```
1418 <behavioralMetadata xmlns:org.example="http://docs.oasis-
1419 open.org/uima/org/example.ecore">
1420   <requiredView sofaType="org.example:TextDocument">
1421     <requiredInputs>
```

```

1422      <type name="org.example:Token"/>
1423  </requiredInputs>
1424  <creates>
1425      <type name="org.example:Person"/>
1426  </creates>
1427 </requiredView>
1428 <optionalView sofaType="org.example:RawAudio">
1429  <requiredInputs>
1430      <type name="org.example:SpeakerBoundary"/>
1431  </requiredInputs>
1432  <creates>
1433      <type name="org.example:AudioPerson"/>
1434  </creates>
1435 </optionalView>
1436 </behavioralMetadata>
1437
1438 This example requires a TextDocument Sofa and optionally accepts a RawAudio Sofa. It has different
1439 input and output types for the different Sofas.
1440
1441 As with an optional input, an “optional view” is one that the analytic would consider if it were present in the
1442 CAS. Views that do not satisfy the required view or optional view expressions might not be delivered to
1443 the analytic.
1444
1445 The meaning of an optionalView having a requiredInput is that a view not containing the required input
1446 types is not considered to satisfy the optionalView expression and might not be delivered to the analytic.
1447
1448 An analytic can also declare that it creates a View along with an associated Sofa and annotations. For
1449 example, this Analytic transcribes audio to text, and also outputs Person annotations over that text:
1450
1451 <behavioralMetadata xmlns:org.example="http://docs.oasis-
1452 open.org/uima/org/example.ecore">
1453  <requiredView sofaType="org.example:RawAudio">
1454    <requiredInputs>
1455      <type name="org.example:SpeakerBoundary"/>
1456    </requiredInputs>
1457  </requiredView>
1458  <createsView sofaType="org.example:TextDocument">
1459    <creates>
1460      <type name="org.example:Person"/>
1461    </creates>
1462  </createsView>
1463 </behavioralMetadata>

```

1464 **3.5.7.4 Specifying Which Features Are Modified**

1465 For the “modifies” predicate we allow an additional piece of information: the names of the features that
1466 may be modified. This is primarily to support discovery. For example:

1467

```
1468 <behavioralMetadata xmlns:org.example="http://docs.oasis-
1469 open.org/uima/org/example.ecore">
1470   <requiredInputs>
1471     <type name="org.example:Person"/>
1472   </requiredInputs>
1473   <modifies>
1474     <type name="org.example:Person">
1475       <feature name="age"/>
1476       <feature name="ssn"/>
1477     </type>
1478   </modifies>
1479 </behavioralMetadata>
1480
```

1481 **3.5.7.5 Specifying Preconditions, Postconditions, and Projection Conditions**

1482 Although we expect it to be rare, analytic developers may declare preconditions, postconditions, and
1483 projection conditions directly. The syntax for this is straightforward:

1484

```
1485 <behavioralMetadata>
1486   <precondition language="OCL"
1487     expression="exists(s | s.oclKindOf(org::example::Sofa) and
1488 s.mimeMajor = 'audio')"/>
1489   <postcondition language="OCL"
1490     expr="exists(p | p.oclKindOf(org::example::Sofa) and s.mimeMajor =
1491 'text')"/>
1492   <projectionCondition language="OCL"
1493     expr=" select(p | p.oclKindOf(org::example::NamedEntity)) "/>
1494 </behavioralMetadata>
```

1495

1496 UIMA does not define what language must be used for expression these conditions. OCL is just one
1497 example.

1498

1499 Preconditions and postconditions are expressions that evaluate to a Boolean value. Projection conditions
1500 are expressions that evaluate to a collection of objects.

1501

1502 Behavioral Metadata can include these conditions as well as the other elements (analyzes,
1503 requiredInputs, etc.). In that case, the overall precondition and postcondition of the analytic are a
1504 combination of the user-specified conditions and the conditions derived from the other behavioral
1505 metadata elements as described in the next section. (For precondition and postcondition it is a
1506 conjunction; for projection condition it is a union.)

1507

1508 **3.5.8 Formal Specification**

1509 **3.5.8.1 Structure**

1510 *UIMA Behavioral Metadata XML* is a part of *UIMA Processing Element Metadata XML*. Its structure is
1511 defined by the definitions of the BehavioralMetadata class in the Ecore model in B.3.

1512

1513 This implies that UIMA Behavioral Metadata XML must be a valid instance of the BehavioralMetadata
1514 element definition in the XML schema given in Section B.5.

1515 **3.5.8.2 Constraints**

1516 Field values must satisfy the following constraints

1517 **3.5.8.2.1 Type**

- 1518 • name must be a valid QName (Qualified Name) as defined by the Namesapces for XML specification
1519 [XML2]. The namespace of this QName must match the namespace URI of an EPackage defined in an
1520 Ecore model referenced by the PE's *TypeSystemReference*. The local part of the QName must match
1521 the name of an EClass within that EPackage.
- 1522 • Values for the `feature` attribute must not be specified unless the Type is contained in a `modifies`
1523 element.
- 1524 • Each value of feature must be a valid UnprefixedName as specified in [XML2], and must match the
1525 name of an EStructuralFeature in the EClass corresponding to the value of the name field as described
1526 in the previous bullet.

1527 **3.5.8.2.2 Condition**

- 1528 • language must be one of:

- 1529 ○ The exact string OCL. If the value of the language field is OCL, then the value of the
1530 expression field must be a valid OCL expression as defined by [OCL1].
- 1531 ○ A user-defined language, which must be a String containing the '.' Character (for example
1532 "org.example.MyLanguage"). Strings not containing the '.' are reserved by the UIMA
1533 standard and may be defined at a later date.

1534 **3.5.8.3 Semantics**

1535 To give a formal meaning to the *analyzes*, *required inputs*, *optional inputs*, *creates*, *modifies*, and *deletes*
1536 expressions, UIMA defines how these map into formal preconditions, postconditions, and projection
1537 conditions in the Object Constraint Language [OCL1], an OMG standard.

1538

1539 The UIMA specification defines this mapping in order to ensure a standard interpretation of UIMA
1540 Behavior Metadata Elements. There is no requirement on any implementation to evaluate or enforce
1541 these expressions. Implementations are free to use other languages for expressing and/or processing
1542 preconditions, postconditions, and projection conditions.

1543 **3.5.8.3.1 Mapping to OCL Precondition**

1544 An OCL precondition is formed from the `analyzes`, `requiredInputs`, and `requiredView`
1545 BehavioralMetadata elements as follows.

1546

1547 In these OCL expressions the keyword `input` refers to the collection of objects in the CAS when it is input
1548 to the analytic.

1549

- 1550 For each type T in an `analyzes` or `requiredInputs` element, produce the OCL expression:
- 1551

```
input->exists(p | p.oclKindOf(T))
```
- 1552
- 1553 For each `requiredView` element that contains `analyzes` or `requiredInputs` elements with types T_1, T_2, \dots, T_n , produce the OCL expression:
- 1555

```
input->exists(v | ViewExpression and v.members->exists(p | p.oclKindOf(T2))
```
- 1556

```
and ... and v.members->exists(p | p.oclKindOf(Tn)))
```
- 1557 (There may be zero `analyzes` or `requiredInputs` elements, in which case there will be no `v.members` clauses in the OCL expression.)
- 1559
- 1560 In the above we define `ViewExpression` as follows:
- 1561 If the `requiredView` element has no value for its `sofaType` slot, then `ViewExpression` is:
- 1562

```
v.oclKindOf(uima::cas::View)
```
- 1563 If the `requiredView` has a `sofaType` slot with value then `ViewExpression` is defined as:
- 1564

```
v.oclKindOf(uima::cas::AnchoredView) and v.sofa.sofaObject.oclKindOf(S)
```
- 1565
- 1566 The final precondition expression for the analytic is the conjunction of all the expressions generated from
1567 the productions defined in this section, as well as any explicitly declared precondition as defined in
1568 Section 3.5.7.5.
- 1569

3.5.8.3.2 Mapping to OCL Postcondition
- 1570 In these OCL expressions the keyword `input` refers to the collection of objects in the CAS when it was
1571 input to the analytic, and the keyword `result` refers to the collection of objects in the CAS at the end of
1572 the analytic's processing. Also note that the suffix `@pre` applied to any attribute references the value of
1573 that attribute at the start of the analytic's operation.
- 1574
- 1575 For types T_1, T_2, \dots, T_n specified in `creates` elements, produce the OCL expression:
- 1576

```
result->forAll(p | input->includes(p) or p.oclKindOf(T1) or p.oclKindOf(T2) or
```
- 1577

```
... or p.oclKindOf(Tn))
```
- 1578
- 1579 For types T_1, T_2, \dots, T_n specified in `deletes` elements, produce the OCL expression:
- 1580

```
input->forAll(p | result->includes(p) or p.oclKindOf(T1) or p.oclKindOf(T2) or
```
- 1581

```
... or p.oclKindOf(Tn))
```
- 1582
- 1583 For each `modifies` element specifying type T with features $F = \{F_1, F_2, \dots, F_n\}$, for each feature g defined
1584 on type T where $g \notin F$, produce the OCL expression:
- 1585

```
result->forAll(p | (input->includes(p) and p.oclKindOf(T)) implies p.g =
```
- 1586

```
p.g@pre)
```
- 1587
- 1588 For each `createsView`, `requiredView` or `optionalView` containing `creates` elements with types
1589 T_1, T_2, \dots, T_n , produce the OCL expression:
- 1590

```
result->forAll(v | (ViewExpression) implies v.members->forAll(p |
```
- 1591

```
v.members->includes(p) or p.oclKindOf(T1) or p.oclKindOf(T2) or ... or
```
- 1592

```
p.oclKindOf(Tn)))
```

1593 where ViewExpression is as defined in Section 3.5.8.3.1.
1594
1595 For each requiredView or optionalView containing deletes elements with types T1,T2,...,Tn, produce
1596 the OCL expression:

```
1597 result->forAll(v | (ViewExpression) implies v.members@pre->forAll(p |
1598 v.members->includes(p) or p.oclkIndOf(T1) or p.oclkIndOf(T2) or ... or
1599 p.oclkIndOf(Tn))
```

1600 where ViewExpression is as defined in Section 3.5.8.3.1.
1601

1602 Within each requiredView or optionalView, for each modifies element specifying type T with features
1603 F={F1, F2, ...Fn}, for each feature g defined on type T where gnotinF, produce the OCL expression:

```
1604 result->forAll(v | (ViewExpression) implies v.members->forAll(p |
1605 (v.members@pre->includes(p) and p.oclkIndOf(T)) implies p.g = p.g@pre))
```

1606 where ViewExpression is as defined in Section 3.5.8.3.1.
1607

1608 The final postcondition expression for the analytic is the conjunction of all the expressions generated from
1609 the productions defined in this section, as well as any explicitly declared postcondition as defined in
1610 Section 3.5.7.5.

1611 3.5.8.3.3 Mapping to OCL Projection Condition

1612 In these OCL expressions the keyword input refers to the collection of objects in the entire CAS when it
1613 is about to be delivered to the analytic. The OCL expression evaluates to a collection of objects that the
1614 analytic declares it will consider while performing its operation. When an application or framework calls
1615 this analytic, it MUST deliver to the analytic all objects in this collection.
1616

1617 If the excludeReferenceClosure attribute of the BehavioralMetadata is set to false (or omitted), then the
1618 application or framework MUST also deliver all objects that are referenced (directly or indirectly) from any
1619 object in the collection resulting from evaluation of the projection condition.
1620

1621 For types T1, T2, ... Tn specified in analyzes, requiredInputs, or optionalInputs elements, produce
1622 the OCL expression:

```
1623 input->select(p | p.oclkIndOf(T1) or p.oclkIndOf(T2) or ... or
1624 p.oclkIndOf(Tn))
```

1625
1626 For each requiredView or optionalView produce the OCL expression:

```
1627 input->select(v | ViewExpression)
```

1628 where ViewExpression is as defined in Section 3.5.8.3.1.
1629

1630 If the requiredView or optionalView contains types T1, T2,...Tn specified in analyzes,
1631 requiredInputs, or optionalInputs elements, produce the OCL expression:

```
1632 input->select(v | ViewExpression)->collect(v.members()->select(p |
1633 p.oclkIndOf(T1) or p.oclkIndOf(T2) or ... or p.oclkIndOf(Tn)))
```

1634

1635 The final projection condition expression for the analytic is the result of the OCL `union` operator applied
1636 consecutively to all of the expressions generated from the productions defined in this section, as well as
1637 any explicitly declared projection condition as defined in Section 3.5.7.5.

1638

1639 **3.6 Processing Element Metadata**

1640 All UIMA Processing Elements (PEs) must publish ***processing element metadata***, which describes the
1641 analytic to support discovery and composition. This section of the spec defines the structure of this
1642 metadata and provides an XML schema in which PEs must publish this metadata.

1643 **3.6.1 Overview**

1644 The PE Metadata is subdivided into the following parts:

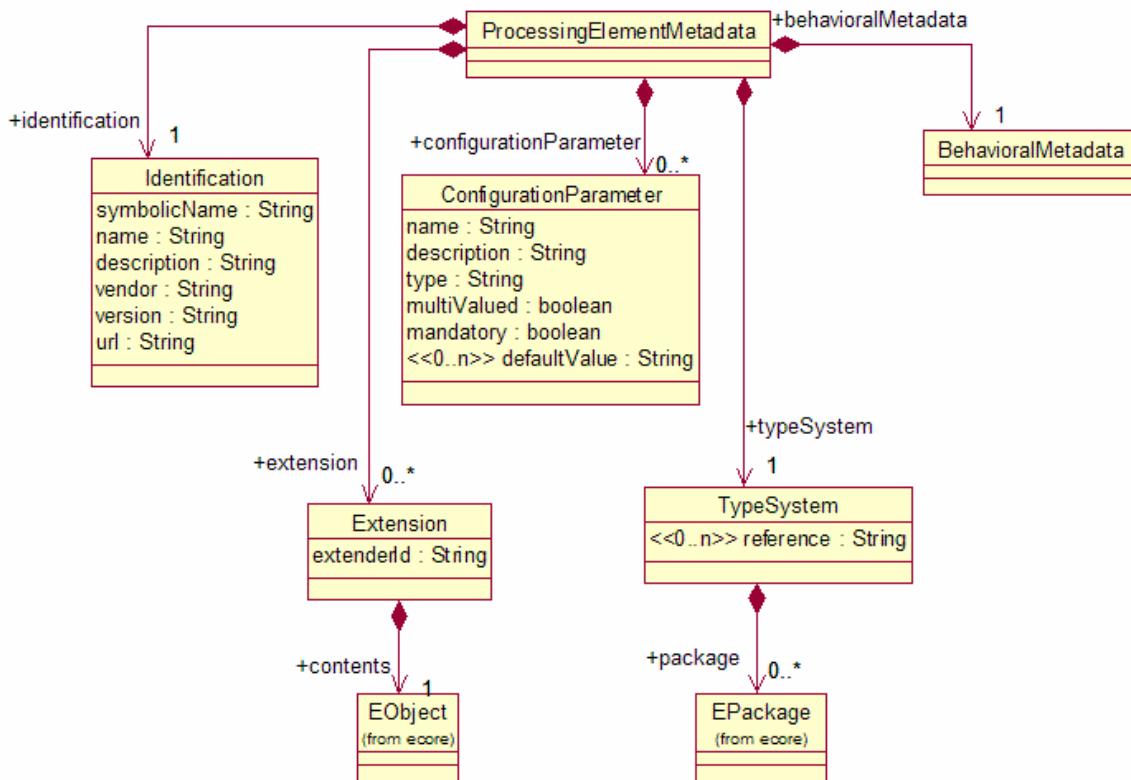
1645

- 1646 1. **Identification Information.** Identifies the PE. It includes for example a symbolic/unique name, a
1647 descriptive name, vendor and version information.
- 1648 2. **Configuration Parameters.** Declares the names of parameters used by the PE to affect its
1649 behavior, as well as the parameters' default values.
- 1650 3. **Behavioral Specification.** Describes the PEs input requirements and the operations that the PE
1651 may perform.
- 1652 4. **Type System.** Defines types used by the PE and referenced from the behavioral specification.
- 1653 5. **Extensions.** Allows the PE metadata to contain additional elements, , the contents of which are
1654 not defined by the UIMA specification. This can be used by framework implementations to
1655 extend the PE metadata with additional information that may be meaningful only to that
1656 framework.

1657

1658 Figure 27 is a UML model for the PE metadata. We describe each subpart of the PE metadata in detail in
1659 the following sections.

1660



1661
1662
1663

Figure 27: Processing Element Metadata UML Model

1664 **3.6.2 Elements of PE Metadata**

1665 **3.6.2.1 Identification Information**

1666 The Identification Information section of the descriptor defines a small set of properties that developers
1667 should fill in with information that describes their PE. The main objectives of this information are to:

- 1668 1. Provide human-readable information about the analytic to assist developers in understanding
1669 what the purpose of each PE is.
1670 2. Facilitate the development of repositories of PEs.

1671

1672 The following properties are included:

- 1673 1. Symbolic Name: A unique name (such as a Java-style dotted name) for this PE.
- 1674 2. Name: A human-readable name for the PE. Not necessarily unique.
- 1675 3. Description: A textual description of the PE.
- 1676 4. Version: A version number. This is necessary for PE repositories that need to distinguish
1677 different versions of the same component. The syntax of a version number is as defined in
1678 [OSGi1]: up to four dot-separated components where the first three must be numeric but the
1679 fourth may be alphanumeric. For example 1.2.3.4 and 1.2.3.abc are valid version numbers but
1680 1.2.abc is not.
- 1681 5. Vendor: The provider of the component.
- 1682 6. URL: website providing information about the component and possibly allowing download of the
1683 component

1684

3.6.2.2 Configuration Parameters

Many kinds of PEs may be configured to operate in different ways³. UIMA provides a standard way for PEs to declare configuration parameters so that application developers are aware of the options that are available to them.

1689

1690 UIMA provides a standard interface for setting the values of parameters; see Section 3.4 Abstract
1691 Interfaces.

1692

1693 For each configuration parameter we should allow the PE developer to specify:

1694

- 1695 1. The name of the parameter
- 1696 2. A description for the parameter
- 1697 3. The type of value that the parameter may take
- 1698 4. Whether the parameter accepts multiple values or only one
- 1699 5. Whether the parameter is mandatory
- 1700 6. A default value or values for the parameter

1701

1702 One common use of configuration parameters is to refer to external resource data, such as files
1703 containing patterns or statistical models. Frameworks such as Apache UIMA may wish to provide
1704 additional support for such parameters, such as resolution of relative URLs (using classpath/datapath)
1705 and/or caching of shared data. It is therefore important for the UIMA configuration parameter schema to
1706 be expressive enough to distinguish parameters that represent resource locations from parameters that
1707 are just arbitrary strings.

1708

1709 The type of a parameter must be one of the following:

- 1710 • String
- 1711 • Integer (32-bit)
- 1712 • Float (32-bit)
- 1713 • Boolean
- 1714 • ResourceURL

1715

1716 The ResourceURL satisfies the requirement to explicitly identify parameters that represent resource
1717 locations.

1718

1719 Note that parameters may take multiple values so it is not necessary to have explicit parameter types
1720 such as StringArray, IntegerArray, etc.

³ Different configuration parameter settings may affect the behavior of an analytic. UIMA does not provide any mechanism to keep the behavioral specification in sync with the different configurations. It may be suggested as a best practices that configuration settings should not affect behavioral specifications.

1721 **3.6.2.3 Type System**

1722 There are two ways that PE metadata may provide type system information: It can either include it or refer
1723 to it. This specification is only concerned with the format of that reference or inclusion. For the actual
1724 definition of the type system, we have adopted the Ecore/XMI representation. See Section 3.2 The Type
1725 System for details.

1726

1727 If reference is chosen as the way to provide the type system information, then the `reference` field of the
1728 `TypeSystem` object must be set to a valid URI (or multiple URIs). URIs are used as references by many
1729 web-based standards (e.g., RDF), and they are also used within Ecore. Thus we use a URI to refer to the
1730 type system. To achieve interoperability across frameworks, each URI should be a URL which resolves
1731 to a location where Ecore/XMI type system data is located.

1732

1733 If embedding is chosen as the way to provide the type system information, then the `package` reference of
1734 the `TypeSystem` object must be set to one or more `EPackages`, where an `EPackage` contains
1735 subpackages and/or classes as defined by Ecore.

1736

1737 The role of this type system is to provide definitions of the types referenced in the PE's behavioral
1738 specification. It is important to note that this is not a restriction on the CASes that may be input to the PE
1739 (if that is desired, it can be expressed using a precondition in the behavioral specification). If the input
1740 CAS contains instances of types that are not defined by the PE's type system, then the CAS itself may
1741 indicate a URI where definitions of these types may be found (see 3.1.5 Linking an XMI Document to its
1742 Ecore Type System). Also, some PE's may be capable of processing CASes without being aware of the
1743 type system at all.⁴

1744

1745 Some analytics may be capable of operating on any types. These analytics need not refer to any specific
1746 type system and in their behavioral metadata may declare that they analyze or inspect instances of the
1747 most general type (`EObject` in Ecore).

1748

1749 **3.6.2.4 Behavioral Metadata**

1750 The Behavioral Metadata is discussed in detail in 3.5.

1751 **3.6.2.5 Extensions**

1752 Extension objects allow a framework implementation to extend the PE metadata descriptor with additional
1753 elements, which other frameworks may not necessarily respect. For example Apache UIMA defines an
1754 element `fslIndexCollection` that defines the CAS indexes that the component uses. Other frameworks
1755 could ignore that.

1756

⁴ Some PE's may not be able to process undefined types, and may return an error if given a CAS that contains an instance of an undefined type. It might be useful to have a place in the behavioral metadata for a PE to declare whether it can accept undefined types.

1757 This extensibility is enabled by the Extension class in Figure 27. The Extension class defines two
1758 features, extenderId and contents.
1759
1760 The extenderId *feature* identifies the framework implementation that added the extension, which allows
1761 framework implementations to ignore extensions that they were not meant to process.
1762
1763 The contents *feature* can contain any EObject. (EObject is the superclass of all classes in Ecore.) To add
1764 an extension, a framework must provide an Ecore model that defines the structure of the extension.

1765 3.6.3 Example (Not Normative)

1766 The following XML fragment is an example of Processing Element Metadata for a “CeoOf Relation
1767 Detector” analytic.

```
1768 <pemd:ProcessingElementMetadata xmi:version="2.0"
1769   xmlns:xmi="http://www.omg.org/XMI" xmlns:pemd="http://docs.oasis-
1770   open.org/uima/pemetadata.ecore">
1771   <identification
1772     symbolicName="org.oasis-open.uima.example.CeoRelationAnnotator"
1773     name="Ceo Relation Annotator"
1774     description="Detects CeoOf relationships between Persons and
1775     Organizations in a text document."
1776     vendor="OASIS"
1777     version="1.0.0"/>
1778
1779   <configurationParameter
1780     name="PatternFile"
1781     description="Location of external file containing patterns that
1782     indicate a CeoOf relation in text."
1783     type="ResourceURL">
1784     <defaultValue>myResources/ceoPatterns.dat</defaultValue>
1785   </configurationParameter>
1786
1787   <typeSystem
1788     reference="http://docs.oasis-
1789     open.org/uima/types/exampleTypeSystem.ecore"/>
1790
1791   <behavioralMetadata>
1792     <analyzes>
1793       <type name="org.example:Document"/>
1794     </analyzes>
1795     <requiredInputs>
1796       <type name="org.example:Person"/>
1797       <type name="org.example:Organization"/>
1798     </requiredInputs>
1799     <creates>
1800       <type name="org.example:CeoOf"/>
1801     </creates>
1802   </behavioralMetadata>
1803
1804   <extension extenderId="org.apache.uima">
1805     ...
1806   </extension>
1807 </pemd:ProcessingElementMetadata>
```

1808 **3.6.4 Formal Specification**

1809 **3.6.4.1 Structure**

1810 *UIMA Processing Element Metadata XML* must be a valid XMI document that is an instance of the UIMA
1811 Processing Element Metadata Ecore model given in Section B.3.

1812

1813 This implies that UIMA Processing Element Metadata XML must be a valid instance of the UIMA
1814 Processing Element Metadata XML schema given in Section B.5.

1815 **3.6.4.2 Constraints**

1816 Field values must satisfy the following constraints

1817

1818 **Identification Information:**

- symbolicName must be a valid symbolic-name as defined by the OSGi specification [OSGi1].
- version must be a valid version as defined by the OSGi specification [OSGi1].
- url must be a valid URL as defined by [\[URL1\]](#).

1822

1823 **Configuration Parameter**

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

1826

1827 **Type System Reference**

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

1830

1831 **Extensions**

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

1833

1834 **3.7 Service WSDL Descriptions**

1835 This specification element facilitates interoperability by specifying a WSDL [\[WSDL1\]](#) description of the
1836 UIMA interfaces and a binding to a concrete SOAP interface that compliant frameworks and services
1837 MUST implement.

1838

1839 This SOAP interface implements the Abstract Interfaces defined in Section 3.4 Abstract Interfaces. The
1840 use of SOAP facilitates standard use of web services as a CAS transport.

1841

1842 In this section we describe the WSDL service definition at a high level. The formal WSDL document is
1843 given in Section B.6.

1844

1845 **3.7.1 Overview of the WSDL Definition**

1846 Before discussing the elements of the UIMA WSDL definition, as a convenience to the reader we first
1847 provide an overview of WSDL excerpted from the WSDL Specification.

1848

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.

1849
1850

1851 **3.7.1.1 Types**

1852 Type Definitions for the UIMA WSDL service are defined using XML schema. These draw from other
1853 elements of the specification. For example the `ProcessingElementMetadata` type, which is returned
1854 from the `getMetadata` operation, is defined by the PE Metadata specification element.

1855

1856 **3.7.1.2 Messages**

1857 Messages are used to define the structure of the request and response of the various operations
1858 supported by the service. Operations are described in the next section.

1859

1860 Messages refer to the XML schema defined under the `<wsdl:types>` element. So wherever a message
1861 includes a CAS (for example the `processCasRequest` and `processCasResponse`, we indicate that the
1862 type of the data is `xmi:XMI` (a type defined by `XMI.xsd`), and where the message consists of PE metadata
1863 (the `getMetadataResponse`), we indicate that the type of the data is `uima:ProcessingElementMetadata` (a
1864 type defined by `UimaDescriptorSchema.xsd`).

1865
1866 The messages defined by the UIMA WSDL service definition are:
1867 For ALL PEs:
1868 • getMetadataRequest – takes no arguments
1869 • getMetadataResponse – returns ProcessingElementMetadata
1870 • setConfigurationParametersRequest – takes one argument: ConfigurationParameterSettings
1871 • setConfigurationParameterResponse – returns nothing
1872
1873 For Analyzers:
1874 • processCasRequest – takes two arguments – a CAS and a list of Sofas (object IDs) to process
1875 • processCasResponse – returns a CAS
1876 • processCasBatchRequest – takes one argument, an Object that includes multiple CASEs, each with an
1877 associated list of Sofas (object IDs) to process
1878 • processCasResponse – returns a list of elements, each of which is a CAS or an exception message
1879
1880 For CAS Multipliers:
1881 • inputCasRequest – takes two arguments – a CAS and a list of Sofas (object IDs) to process
1882 • inputCasResponse – returns nothing
1883 • getNextCasRequest – takes no arguments
1884 • getNextCasResponse – returns a CAS
1885 • retrieveInputCasRequest – takes no arguments
1886 • retrieveInputCasResponse – returns a CAS
1887 • getNextCasBatchRequest – takes two arguments, an integer that specifies the maximum number of
1888 CASEs to return and an integer which specifies the maximum number of milliseconds to wait
1889 • getNextCasBatchResponse – returns an object with three fields: a list of zero or more CASEs, a
1890 Boolean indicating whether any CASEs remain to be retrieved, and an integer indicating the estimated
1891 number of remaining CASEs (-1 if not known).
1892
1893 For Flow Controllers:
1894 • addAvailableAnalyticsRequest – takes one argument, a Map from String keys to PE Metadata objects.
1895 • addAvailableAnalyticsResponse – returns nothing
1896 • removeAvailableAnalyticsRequest – takes one argument, a collection of one or more String keys
1897 • removeAvailableAnalyticsResponse – returns nothing
1898 • setAggregateMetadataRequest – takes one argument – a ProcessingElementMetadata
1899 • setAggregateMetadataResponse – returns nothing
1900 • getNextDestinationsRequest – takes one argument, a CAS
1901 • getNextDestionsResponse – returns a Step object
1902 • continueOnFailureRequest – takes three arguments, a CAS, a String key, and a UimaException
1903 • continueOnFailureResponse – returns a Boolean
1904
1905 **3.7.1.3 Port Types and Operations**
1906 A *port type* is a collection of *operations*, where each operation is an action that can be performed by the
1907 service. We define a separate port type for each of the three interfaces defined in Section 3.4 Abstract
1908 Interfaces.
1909
1910 The port types and their operations defined by the UIMA WSDL definition are as follows. Each operation
1911 refers to its input and output message, defined in the previous section. Operations also have fault
1912 messages, returned in the case of an error.

1913
1914 • **Analyzer Port Type**
1915 • getMetadata
1916 • setConfigurationParameters
1917 • processCas
1918 • processCasBatch
1919
1920 • **CasMultiplier Port Type**
1921 • getMetadata
1922 • setConfigurationParameters
1923 • inputCas
1924 • getNextCas
1925 • retrieveInputCas
1926 • getNextCasBatch
1927
1928 **FlowController Port Type**
1929 • getMetadata
1930 • setConfigurationPsrameters
1931 • addAvailableAnalytics
1932 • removeAvailableAnalytics
1933 • setAggregateMetadata
1934 • getNextDestinations
1935 • continueOnFailure
1936

3.7.1.4 SOAP Bindings

1937 For each port type, we define a binding to the SOAP protocol. There are a few configuration choices to
1938 be made:
1939
1940 In <wsdlsoap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>:
1941 • The style attribute defines that our operation is an RPC, meaning that our XML messages contain
1942 parameters and return values. The alternative is "document" style, which is used for services that
1943 logically send and receive XML documents without a parameter structure. This has an effect on
1944 how the body of the SOAP message is constructed.
1945 • The transport operation defines that this binding uses the HTTP protocol (the SOAP spec allows
1946 other protocols, such as FTP or SMTP, but HTTP is by far the most common)
1947 For each parameter (message part) in each abstract operation, we have a <wsdlsoap:body use="literal"/>
1948 element:
1949 • The use of the <wsdlsoap:body> tag indicates that this parameter is sent in the body of the SOAP
1950 message. Alternatively we could use <wsdlsoap:header> to choose to send parameters in the
1951 SOAP header. This is an arbitrary choice, but a good rule of thumb is that the data being
1952 processed by the service should be sent in the body, and "control information" (i.e., *how* the
1953 message should be processed) can be sent in the header.
1954 • The use="literal" attribute states that the content of the message must *exactly* conform to the
1955 XML Schema defined earlier in the WSDL definitions. The other option is "encoded", which treats
1956 the XML Schema as an abstract type definition and applies SOAP encoding rules to determine
1957 the exact XML syntax of the messages. The "encoded" style makes more sense if you are
1958 starting from an abstract object model and you want to let the SOAP rules determine your XML
1959 syntax. In our case, we already know what XML syntax we want (e.g., XMI), so the "literal" style
1960 is more appropriate.
1961

3.7.2 Delta Responses

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

1967

3.7.3 SOAP Service Example (Not Normative)

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

1971

1972 The processCas request message is shown here:

```
1973 <soapenv:Envelope...>
1974   <soapenv:Body>
1975     <processCas xmlns="">
1976       <cas xmi:version="2.0" ... >
1977         <org.example:Document xmi:id="1"
1978           text="Fred Center is the CEO of Center Micros."/>
1979         <cas:LocalSofaReference xmi:id="2" sofaObject="1" sofaFeature="text"/>
1980         <org.example:Person xmi:id="3" sofa="2" begin="0" end="11"/>
1981         <org.example:Organization xmi:id="4" sofa="2" begin="26" end="39"/>
1982       </cas>
1983       <sofas>1</sofas>
1984     </processCas>
1985   </soapenv:Body>
1986 </soapenv:Envelope>
```

1987 This message is simply an XMI CAS wrapped in an appropriate SOAP envelope, indicating which
1988 operation is being invoked (processCas).

1989

1990 The processCas response message returned from the service is shown here:

1991

```
1992 <soapenv:Envelope...>
1993   <soapenv:Body>
1994     <processCas xmlns="">
1995       <cas xmi:version="2.0" ... >
1996         <org.example:Document xmi:id="1"
1997           text="Fred Center is the CEO of Center Micros."/>
1998         <cas:SofaReference xmi:id="2" sofaObject="1" sofaFeature="text"/>
1999         <org.example:Person xmi:id="3" sofa="2" begin="0" end="11"/>
2000         <org.example:Organization xmi:id="4" sofa="2" begin="26" end="39"/>
2001         <org.example:CeoOf xmi:id="5" sofa="2" begin="0" end="31" arg0="3"
2002           arg1="4"/>
2003         </cas>
2004       </processCas>
2005     </soapenv:Body>
2006   </soapenv:Envelope>
```

2007 Again this is just an XMI CAS wrapped in a SOAP envelope. Note that the “CeoOf” object has been
2008 added to the CAS.

2009

2010 Alternatively, the service could have responded with a “delta” using the XMI differences language. Here
2011 is an example:

```
2012 <soapenv:Envelope...>
2013   <soapenv:Body>
```

```
2014 <processCas xmlns="">
2015   <cas xmi:version="2.0" ... >
2016     <xmi:Difference>
2017       <target href="input.xmi"/>
2018       <xmi:Add addition="5">
2019     </xmi:Difference>
2020       <org.example:CeoOf xmi:id="5" sofa="2" begin="0" end="31" arg0="3"
2021 arg1="4"/>
2022     </cas>
2023   </processCas>
2024 </soapenv:Body>
2025 </soapenv:Envelope>
```

2026

2027 Note that the `target` element is defined in the XMI specification to hold an href to the original XMI file to
2028 which these differences will get applied. In UIMA we don't really have a URI for that - it is just the input to
2029 the Process CAS Request. The example conventionally input.xmi for this URI.

2030

2031 **3.7.4 Formal Specification**

2032 A *UIMA SOAP Service* must conform to the WSDL document given in Section B.6 and must implement at
2033 least one of the portTypes and corresponding SOAP bindings defined in that WSDL document, as defined
2034 in [WSDL1] and [SOAP1].

2035

2036 A *UIMA Analyzer SOAP Service* must implement the Analyzer portType and the AnalyzerSoapBinding.

2037

2038 A *UIMA CAS Multiplier SOAP Service* must implement the CasMultiplier portType and the
2039 CasMultiplierSoapBinding.

2040

2041

A. Acknowledgements

2042

The following individuals have participated in the creation of this specification and are gratefully
2043 acknowledged:

2044

Participants:

2045

[Participant Name, Affiliation | Individual Member]

2046

[Participant Name, Affiliation | Individual Member]

2047

2048 B. Formal Specification Artifacts

2049 This section includes artifacts such as Ecore models and XML Schemata, which formally define elements
2050 of the UIMA specification.

2051 B.1 XMI XML Schema

2052 This XML schema is defined by the XMI specification [XMI1] and repeated here for completeness:

```
2053
2054 <?xml version="1.0" encoding="UTF-8"?>
2055 <xsd:schema xmlns:xmi="http://www.omg.org/XMI"
2056   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
2057   targetNamespace="http://www.omg.org/XMI">
2058   <xsd:attribute name="id" type="xsd:ID"/>
2059   <xsd:attributeGroup name="IdentityAttribs">
2060     <xsd:attribute form="qualified" name="label" type="xsd:string"
2061       use="optional"/>
2062     <xsd:attribute form="qualified" name="uuid" type="xsd:string"
2063       use="optional"/>
2064   </xsd:attributeGroup>
2065   <xsd:attributeGroup name="LinkAttribs">
2066     <xsd:attribute name="href" type="xsd:string" use="optional"/>
2067     <xsd:attribute form="qualified" name="idref" type="xsd:IDREF"
2068       use="optional"/>
2069   </xsd:attributeGroup>
2070   <xsd:attributeGroup name="ObjectAttribs">
2071     <xsd:attributeGroup ref="xmi:IdentityAttribs"/>
2072     <xsd:attributeGroup ref="xmi:LinkAttribs"/>
2073     <xsd:attribute fixed="2.0" form="qualified" name="version"
2074       type="xsd:string" use="optional"/>
2075     <xsd:attribute form="qualified" name="type" type="xsd:QName"
2076       use="optional"/>
2077   </xsd:attributeGroup>
2078   <xsd:complexType name="XMI">
2079     <xsd:choice maxOccurs="unbounded" minOccurs="0">
2080       <xsd:any processContents="strict"/>
2081     </xsd:choice>
2082     <xsd:attributeGroup ref="xmi:IdentityAttribs"/>
2083     <xsd:attributeGroup ref="xmi:LinkAttribs"/>
2084     <xsd:attribute form="qualified" name="type" type="xsd:QName"
2085       use="optional"/>
2086     <xsd:attribute fixed="2.0" form="qualified" name="version"
2087       type="xsd:string" use="required"/>
```

```

2088    </xsd:complexType>
2089    <xsd:element name="XMI" type="xmi:XMI"/>
2090    <xsd:complexType name="PackageReference">
2091        <xsd:choice maxOccurs="unbounded" minOccurs="0">
2092            <xsd:element name="name" type="xsd:string"/>
2093            <xsd:element name="version" type="xsd:string"/>
2094        </xsd:choice>
2095        <xsd:attributeGroup ref="xmi:ObjectAttrs"/>
2096            <xsd:attribute name="name" type="xsd:string" use="optional"/>
2097    </xsd:complexType>
2098    <xsd:element name="PackageReference"
2099        type="xmi:PackageReference"/>
2100    <xsd:complexType name="Model">
2101        <xsd:complexContent>
2102            <xsd:extension base="xmi:PackageReference"/>
2103        </xsd:complexContent>
2104    </xsd:complexType>
2105    <xsd:element name="Model" type="xmi:Model"/>
2106    <xsd:complexType name="Import">
2107        <xsd:complexContent>
2108            <xsd:extension base="xmi:PackageReference"/>
2109        </xsd:complexContent>
2110    </xsd:complexType>
2111    <xsd:element name="Import" type="xmi:Import"/>
2112    <xsd:complexType name="MetaModel">
2113        <xsd:complexContent>
2114            <xsd:extension base="xmi:PackageReference"/>
2115        </xsd:complexContent>
2116    </xsd:complexType>
2117    <xsd:element name="MetaModel" type="xmi:MetaModel"/>
2118    <xsd:complexType name="Documentation">
2119        <xsd:choice maxOccurs="unbounded" minOccurs="0">
2120            <xsd:element name="contact" type="xsd:string"/>
2121            <xsd:element name="exporter" type="xsd:string"/>
2122            <xsd:element name="exporterVersion" type="xsd:string"/>
2123            <xsd:element name="longDescription" type="xsd:string"/>
2124            <xsd:element name="shortDescription" type="xsd:string"/>
2125            <xsd:element name="notice" type="xsd:string"/>
2126            <xsd:element name="owner" type="xsd:string"/>
2127        </xsd:choice>
2128        <xsd:attributeGroup ref="xmi:ObjectAttrs"/>
2129            <xsd:attribute name="contact" type="xsd:string" use="optional"/>
2130            <xsd:attribute name="exporter" type="xsd:string"

```

```

2131      use="optional"/>
2132      <xsd:attribute name="exporterVersion" type="xsd:string"
2133          use="optional"/>
2134      <xsd:attribute name="longDescription" type="xsd:string"
2135          use="optional"/>
2136      <xsd:attribute name="shortDescription" type="xsd:string"
2137          use="optional"/>
2138      <xsd:attribute name="notice" type="xsd:string" use="optional"/>
2139      <xsd:attribute name="owner" type="xsd:string" use="optional"/>
2140  </xsd:complexType>
2141  <xsd:element name="Documentation" type="xmi:Documentation"/>
2142  <xsd:complexType name="Extension">
2143      <xsd:choice maxOccurs="unbounded" minOccurs="0">
2144          <xsd:any processContents="lax"/>
2145      </xsd:choice>
2146      <xsd:attributeGroup ref="xmi:ObjectAttrs"/>
2147      <xsd:attribute name="extender" type="xsd:string"
2148          use="optional"/>
2149      <xsd:attribute name="extenderID" type="xsd:string"
2150          use="optional"/>
2151  </xsd:complexType>
2152  <xsd:element name="Extension" type="xmi:Extension"/>
2153  <xsd:complexType name="Difference">
2154      <xsd:choice maxOccurs="unbounded" minOccurs="0">
2155          <xsd:element name="target">
2156              <xsd:complexType>
2157                  <xsd:choice maxOccurs="unbounded" minOccurs="0">
2158                      <xsd:any processContents="skip"/>
2159                  </xsd:choice>
2160                  <xsd:anyAttribute processContents="skip"/>
2161              </xsd:complexType>
2162          </xsd:element>
2163          <xsd:element name="difference" type="xmi:Difference"/>
2164          <xsd:element name="container" type="xmi:Difference"/>
2165      </xsd:choice>
2166      <xsd:attributeGroup ref="xmi:ObjectAttrs"/>
2167      <xsd:attribute name="target" type="xsd:IDREFS" use="optional"/>
2168      <xsd:attribute name="container" type="xsd:IDREFS"
2169          use="optional"/>
2170  </xsd:complexType>
2171  <xsd:element name="Difference" type="xmi:Difference"/>
2172  <xsd:complexType name="Add">
2173      <xsd:complexContent>
```

```

2174      <xsd:extension base="xmi:Difference">
2175          <xsd:attribute name="position" type="xsd:string"
2176              use="optional"/>
2177          <xsd:attribute name="addition" type="xsd:IDREFS"
2178              use="optional"/>
2179      </xsd:extension>
2180  </xsd:complexContent>
2181 </xsd:complexType>
2182 <xsd:element name="Add" type="xmi:Add"/>
2183 <xsd:complexType name="Replace">
2184     <xsd:complexContent>
2185         <xsd:extension base="xmi:Difference">
2186             <xsd:attribute name="position" type="xsd:string"
2187                 use="optional"/>
2188             <xsd:attribute name="replacement" type="xsd:IDREFS"
2189                 use="optional"/>
2190         </xsd:extension>
2191     </xsd:complexContent>
2192 </xsd:complexType>
2193 <xsd:element name="Replace" type="xmi:Replace"/>
2194 <xsd:complexType name="Delete">
2195     <xsd:complexContent>
2196         <xsd:extension base="xmi:Difference"/>
2197     </xsd:complexContent>
2198 </xsd:complexType>
2199 <xsd:element name="Delete" type="xmi:Delete"/>
2200 <xsd:complexType name="Any">
2201     <xsd:choice maxOccurs="unbounded" minOccurs="0">
2202         <xsd:any processContents="skip"/>
2203     </xsd:choice>
2204     <xsd:anyAttribute processContents="skip"/>
2205 </xsd:complexType>
2206 </xsd:schema>

```

2207 B.2 Ecore XML Schema

2208 This XML schema is defined by Ecore [Need Ref] and repeated here for completeness:

```

2209 <?xml version="1.0" encoding="UTF-8"?>
2210 <xsd:schema xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore"
2211   xmlns:xmi="http://www.omg.org/XMI"
2212   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
2213   targetNamespace="http://www.eclipse.org/emf/2002/Ecore">
2214     <xsd:import namespace="http://www.omg.org/XMI" schemaLocation="XMI.xsd"/>
2215     <xsd:complexType name="EAttribute">
2216       <xsd:complexContent>

```

```

2217      <xsd:extension base="ecore:EStructuralFeature">
2218          <xsd:attribute name="id" type="xsd:boolean"/>
2219      </xsd:extension>
2220  </xsd:complexContent>
2221 </xsd:complexType>
2222 <xsd:element name="EAttribute" type="ecore:EAttribute"/>
2223 <xsd:complexType name="EAnnotation">
2224     <xsd:complexContent>
2225         <xsd:extension base="ecore:EModelElement">
2226             <xsd:choice maxOccurs="unbounded" minOccurs="0">
2227                 <xsd:element name="details" type="ecore:EStringToStringMapEntry"/>
2228                 <xsd:element name="contents" type="ecore:EObject"/>
2229                 <xsd:element name="references" type="ecore:EObject"/>
2230             </xsd:choice>
2231             <xsd:attribute name="source" type="xsd:string"/>
2232             <xsd:attribute name="references" type="xsd:string"/>
2233         </xsd:extension>
2234     </xsd:complexContent>
2235 </xsd:complexType>
2236 <xsd:element name="EAnnotation" type="ecore:EAnnotation"/>
2237 <xsd:complexType name="EClass">
2238     <xsd:complexContent>
2239         <xsd:extension base="ecore:EClassifier">
2240             <xsd:choice maxOccurs="unbounded" minOccurs="0">
2241                 <xsd:element name="eSuperTypes" type="ecore:EClass"/>
2242                 <xsd:element name="eOperations" type="ecore:EOperation"/>
2243                 <xsd:element name="eStructuralFeatures"
2244 type="ecore:EStructuralFeature"/>
2245             </xsd:choice>
2246             <xsd:attribute name="abstract" type="xsd:boolean"/>
2247             <xsd:attribute name="interface" type="xsd:boolean"/>
2248             <xsd:attribute name="eSuperTypes" type="xsd:string"/>
2249         </xsd:extension>
2250     </xsd:complexContent>
2251 </xsd:complexType>
2252 <xsd:element name="EClass" type="ecore:EClass"/>
2253 <xsd:complexType abstract="true" name="EClassifier">
2254     <xsd:complexContent>
2255         <xsd:extension base="ecore:ENamedElement">
2256             <xsd:attribute name="instanceClassName" type="xsd:string"/>
2257         </xsd:extension>
2258     </xsd:complexContent>
2259 </xsd:complexType>
```

```

2260 <xsd:element name="EClassifier" type="ecore:EClassifier"/>
2261 <xsd:complexType name="EDataType">
2262   <xsd:complexContent>
2263     <xsd:extension base="ecore:EClassifier">
2264       <xsd:attribute name="serializable" type="xsd:boolean"/>
2265     </xsd:extension>
2266   </xsd:complexContent>
2267 </xsd:complexType>
2268 <xsd:element name="EDataType" type="ecore:EDataType"/>
2269 <xsd:complexType name="EEEnum">
2270   <xsd:complexContent>
2271     <xsd:extension base="ecore:EDataType">
2272       <xsd:choice maxOccurs="unbounded" minOccurs="0">
2273         <xsd:element name="eLiterals" type="ecore:EEEnumLiteral"/>
2274       </xsd:choice>
2275     </xsd:extension>
2276   </xsd:complexContent>
2277 </xsd:complexType>
2278 <xsd:element name="EEEnum" type="ecore:EEEnum"/>
2279 <xsd:complexType name="EEEnumLiteral">
2280   <xsd:complexContent>
2281     <xsd:extension base="ecore:ENamedElement">
2282       <xsd:attribute name="value" type="xsd:int"/>
2283       <xsd:attribute name="literal" type="xsd:string"/>
2284     </xsd:extension>
2285   </xsd:complexContent>
2286 </xsd:complexType>
2287 <xsd:element name="EEEnumLiteral" type="ecore:EEEnumLiteral"/>
2288 <xsd:complexType name="EFactory">
2289   <xsd:complexContent>
2290     <xsd:extension base="ecore:EModelElement"/>
2291   </xsd:complexContent>
2292 </xsd:complexType>
2293 <xsd:element name="EFactory" type="ecore:EFactory"/>
2294 <xsd:complexType abstract="true" name="EModelElement">
2295   <xsd:complexContent>
2296     <xsd:extension base="ecore:EObject">
2297       <xsd:choice maxOccurs="unbounded" minOccurs="0">
2298         <xsd:element name="eAnnotations" type="ecore:EAnnotation"/>
2299       </xsd:choice>
2300     </xsd:extension>
2301   </xsd:complexContent>
2302 </xsd:complexType>

```

```

2303 <xsd:element name="EModelElement" type="ecore:EModelElement"/>
2304 <xsd:complexType abstract="true" name="ENamedElement">
2305   <xsd:complexContent>
2306     <xsd:extension base="ecore:EModelElement">
2307       <xsd:attribute name="name" type="xsd:string"/>
2308     </xsd:extension>
2309   </xsd:complexContent>
2310 </xsd:complexType>
2311 <xsd:element name="ENamedElement" type="ecore:ENamedElement"/>
2312 <xsd:complexType name="EObject">
2313   <xsd:choice maxOccurs="unbounded" minOccurs="0">
2314     <xsd:element ref="xmi:Extension"/>
2315   </xsd:choice>
2316   <xsd:attribute ref="xmi:id"/>
2317   <xsd:attributeGroup ref="xmi:ObjectAttrs"/>
2318 </xsd:complexType>
2319 <xsd:element name="EObject" type="ecore:EObject"/>
2320 <xsd:complexType name="EOperation">
2321   <xsd:complexContent>
2322     <xsd:extension base="ecore:ETypedElement">
2323       <xsd:choice maxOccurs="unbounded" minOccurs="0">
2324         <xsd:element name="eParameters" type="ecore:EParameter"/>
2325         <xsd:element name="eExceptions" type="ecore:EClassifier"/>
2326       </xsd:choice>
2327         <xsd:attribute name="eExceptions" type="xsd:string"/>
2328       </xsd:extension>
2329   </xsd:complexContent>
2330 </xsd:complexType>
2331 <xsd:element name="EOperation" type="ecore:EOperation"/>
2332 <xsd:complexType name="EPackage">
2333   <xsd:complexContent>
2334     <xsd:extension base="ecore:ENamedElement">
2335       <xsd:choice maxOccurs="unbounded" minOccurs="0">
2336         <xsd:element name="eClassifiers" type="ecore:EClassifier"/>
2337         <xsd:element name="eSubpackages" type="ecore:EPackage"/>
2338       </xsd:choice>
2339         <xsd:attribute name="nsURI" type="xsd:string"/>
2340         <xsd:attribute name="nsPrefix" type="xsd:string"/>
2341       </xsd:extension>
2342   </xsd:complexContent>
2343 </xsd:complexType>
2344 <xsd:element name="EPackage" type="ecore:EPackage"/>
2345 <xsd:complexType name="EParameter">

```

```

2346      <xsd:complexContent>
2347          <xsd:extension base="ecore:ETypedElement"/>
2348      </xsd:complexContent>
2349  </xsd:complexType>
2350  <xsd:element name="EParameter" type="ecore:EParameter"/>
2351  <xsd:complexType name="EReference">
2352      <xsd:complexContent>
2353          <xsd:extension base="ecore:EStructuralFeature">
2354              <xsd:choice maxOccurs="unbounded" minOccurs="0">
2355                  <xsd:element name="eOpposite" type="ecore:EReference"/>
2356              </xsd:choice>
2357              <xsd:attribute name="containment" type="xsd:boolean"/>
2358              <xsd:attribute name="resolveProxies" type="xsd:boolean"/>
2359              <xsd:attribute name="eOpposite" type="xsd:string"/>
2360          </xsd:extension>
2361      </xsd:complexContent>
2362  </xsd:complexType>
2363  <xsd:element name="EReference" type="ecore:EReference"/>
2364  <xsd:complexType abstract="true" name="EStructuralFeature">
2365      <xsd:complexContent>
2366          <xsd:extension base="ecore:ETypedElement">
2367              <xsd:attribute name="changeable" type="xsd:boolean"/>
2368              <xsd:attribute name="volatile" type="xsd:boolean"/>
2369              <xsd:attribute name="transient" type="xsd:boolean"/>
2370              <xsd:attribute name="defaultValueLiteral" type="xsd:string"/>
2371              <xsd:attribute name="unsettable" type="xsd:boolean"/>
2372              <xsd:attribute name="derived" type="xsd:boolean"/>
2373          </xsd:extension>
2374      </xsd:complexContent>
2375  </xsd:complexType>
2376  <xsd:element name="EStructuralFeature" type="ecore:EStructuralFeature"/>
2377  <xsd:complexType abstract="true" name="ETypedElement">
2378      <xsd:complexContent>
2379          <xsd:extension base="ecore:ENamedElement">
2380              <xsd:choice maxOccurs="unbounded" minOccurs="0">
2381                  <xsd:element name="eType" type="ecore:EClassifier"/>
2382              </xsd:choice>
2383              <xsd:attribute name="ordered" type="xsd:boolean"/>
2384              <xsd:attribute name="unique" type="xsd:boolean"/>
2385              <xsd:attribute name="lowerBound" type="xsd:int"/>
2386              <xsd:attribute name="upperBound" type="xsd:int"/>
2387              <xsd:attribute name="eType" type="xsd:string"/>
2388          </xsd:extension>

```

```

2389      </xsd:complexContent>
2390  </xsd:complexType>
2391  <xsd:element name="ETypedElement" type=".ecore:ETypedElement"/>
2392  <xsd:complexType name="EStringToStringMapEntry">
2393      <xsd:choice maxOccurs="unbounded" minOccurs="0">
2394          <xsd:element ref="xmi:Extension"/>
2395      </xsd:choice>
2396      <xsd:attribute ref="xmi:id"/>
2397      <xsd:attributeGroup ref="xmi:ObjectAttrs"/>
2398          <xsd:attribute name="key" type="xsd:string"/>
2399          <xsd:attribute name="value" type="xsd:string"/>
2400  </xsd:complexType>
2401  <xsd:element name="EStringToStringMapEntry"
2402 type=".ecore:EStringToStringMapEntry"/>
2403 </xsd:schema>
2404

```

2405 **B.3 Base Type System Ecore Model**

2406 **TODO**

2407 **B.4 PE Metadata and Behavioral Metadata Ecore Model**

```

2408 TODO:Out of date. Also ffix capitalization: is it peMetadata or pemetadata?
2409 <?xml version="1.0" encoding="UTF-8"?>
2410 <ecore:EPackage xmi:version="2.0"
2411     xmlns:xmi="http://www.omg.org/XMI"
2412     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2413     xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore" name="uima"
2414     nsURI="http://uima.ecore" nsPrefix="uima">
2415     <eSubpackages name="peMetadata" nsURI="http://docs.oasis-
2416 open.org/uima/pemetadata.ecore"
2417         nsPrefix="uima.peMetadata">
2418         <eClassifiers xsi:type="ecore:EClass" name="Identification">
2419             <eStructuralFeatures xsi:type="ecore:EAttribute" name="symbolicName"
2420             eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#/EString"/>
2421                 <eStructuralFeatures xsi:type="ecore:EAttribute" name="name"
2422             eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#/EString"/>
2423                 <eStructuralFeatures xsi:type="ecore:EAttribute" name="description"
2424             eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#/EString"/>
2425                 <eStructuralFeatures xsi:type="ecore:EAttribute" name="vendor"
2426             eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#/EString"/>
2427                 <eStructuralFeatures xsi:type="ecore:EAttribute" name="version"
2428             eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#/EString"/>
2429                 <eStructuralFeatures xsi:type="ecore:EAttribute" name="url"
2430             eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#/EString"/>
2431         </eClassifiers>

```

```

2432      <eClassifiers xsi:type="ecore:EClass" name="ConfigurationParameter">
2433          <eStructuralFeatures xsi:type="ecore:EAttribute" name="name"
2434          eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
2435              <eStructuralFeatures xsi:type="ecore:EAttribute" name="description"
2436              eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
2437                  <eStructuralFeatures xsi:type="ecore:EAttribute" name="type"
2438                  eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
2439                      <eStructuralFeatures xsi:type="ecore:EAttribute" name="multiValued"
2440                      eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EBoolean"/>
2441                          <eStructuralFeatures xsi:type="ecore:EAttribute" name="mandatory"
2442                          eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EBoolean"/>
2443                              <eStructuralFeatures xsi:type="ecore:EAttribute" name="defaultValue"
2444                              upperBound="-1"
2445
2446          eType="ecore:EDataType
2446          http://www.eclipse.org/emf/2002/Ecore#//EString"/>
2447      </eClassifiers>
2448      <eClassifiers xsi:type="ecore:EClass" name="TypeSystemReference">
2449          <eStructuralFeatures xsi:type="ecore:EAttribute" name="uri"
2450          eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
2451      </eClassifiers>
2452      <eClassifiers xsi:type="ecore:EClass" name="BehavioralMetadata">
2453          <eStructuralFeatures xsi:type="ecore:EReference" name="analyzes"
2454          lowerBound="1"
2455
2456          eType="#//peMetadata/BehaviorElement" containment="true"/>
2457
2458          <eStructuralFeatures xsi:type="ecore:EReference" name="requiredInputs"
2459          lowerBound="1"
2460
2461          eType="#//peMetadata/BehaviorElement" containment="true"/>
2462
2463          <eStructuralFeatures xsi:type="ecore:EReference" name="optionalInputs"
2464          lowerBound="1"
2465
2466          eType="#//peMetadata/BehaviorElement" containment="true"/>
2467
2468          <eStructuralFeatures xsi:type="ecore:EReference" name="creates"
2469          lowerBound="1"
2470
2471          eType="#//peMetadata/BehaviorElement" containment="true"/>
2472
2473          <eStructuralFeatures xsi:type="ecore:EReference" name="modifies"
2474          lowerBound="1"
2475
2476          eType="#//peMetadata/BehaviorElement" containment="true"/>
2477
2478          <eStructuralFeatures xsi:type="ecore:EReference"

```

```

2479         lowerBound="1" eType="#//peMetadata/Condition" containment="true"/>
2480             <eStructuralFeatures xsi:type="ecore:EReference" name="requiredView"
2481             upperBound="-1"
2482                 eType="#//peMetadata/ViewBehavioralMetadata" containment="true"/>
2483             <eStructuralFeatures xsi:type="ecore:EReference" name="optionalView"
2484             upperBound="-1"
2485                 eType="#//peMetadata/ViewBehavioralMetadata" containment="true"/>
2486             </eClassifiers>
2487             <eClassifiers xsi:type="ecore:EClass" name="ProcessingElementMetadata">
2488                 <eStructuralFeatures xsi:type="ecore:EReference"
2489                 name="configurationParameter"
2490                     upperBound="-1" eType="#//peMetadata/ConfigurationParameter"
2491                     containment="true"/>
2492                     <eStructuralFeatures xsi:type="ecore:EReference" name="identification"
2493                     lowerBound="1"
2494                         eType="#//peMetadata/Identification" containment="true"/>
2495                         <eStructuralFeatures xsi:type="ecore:EReference"
2496                         name="typeSystemReference"
2497                             lowerBound="1" eType="#//peMetadata/TypeSystemReference"
2498                             containment="true"/>
2499                             <eStructuralFeatures xsi:type="ecore:EReference"
2500                             name="behavioralMetadata" lowerBound="1"
2501                                 eType="#//peMetadata/BehavioralMetadata" containment="true"/>
2502                                 <eStructuralFeatures xsi:type="ecore:EReference" name="extension"
2503                                 upperBound="-1"
2504                                     eType="#//peMetadata/Extension" containment="true"/>
2505                                     </eClassifiers>
2506                                     <eClassifiers xsi:type="ecore:EClass" name="Extension">
2507                                         <eStructuralFeatures xsi:type="ecore:EAttribute" name="extenderId"
2508                                         eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#/EString"/>
2509                                         <eStructuralFeatures xsi:type="ecore:EReference" name="contents"
2510                                         lowerBound="1"
2511                                             eType="ecore:EClass
2512                                             http://www.eclipse.org/emf/2002/Ecore#/EObject"/>
2513                                             </eClassifiers>
2514                                             <eClassifiers xsi:type="ecore:EClass" name="BehaviorElement">
2515                                                 <eStructuralFeatures xsi:type="ecore:EReference" name="type"
2516                                                 upperBound="-1"
2517                                                     eType="#//peMetadata/Type" containment="true"/>
2518                                                     </eClassifiers>
2519                                                     <eClassifiers xsi:type="ecore:EClass" name="Type">
2520                                                         <eStructuralFeatures xsi:type="ecore:EAttribute" name="name"
2521                                                         eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#/EString"/>
2522                                                         <eStructuralFeatures xsi:type="ecore:EAttribute" name="feature"
2523                                                         upperBound="-1"
2524                                                 eType="ecore:EDataType
2525                                                 http://www.eclipse.org/emf/2002/Ecore#/EString"/>

```

```

2526      </eClassifiers>
2527      <eClassifiers xsi:type="ecore:EClass" name="Condition">
2528          <eStructuralFeatures xsi:type="ecore:EAttribute" name="language"
2529          eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
2530          <eStructuralFeatures xsi:type="ecore:EAttribute" name="expression"
2531          eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
2532          <eStructuralFeatures xsi:type="ecore:EAttribute" name="feature"
2533          upperBound="-1"
2534          eType="ecore:EDataType
2535          http://www.eclipse.org/emf/2002/Ecore#//EString"/>
2536      </eClassifiers>
2537      <eClassifiers xsi:type="ecore:EClass" name="ViewBehavioralMetadata"
2538      eSuperTypes="#//peMetadata/BehavioralMetadata"/>
2539      </eSubpackages>
2540  </ecore:EPackage>
2541

```

2542 B.5 PE Metadata and Behavioral Metadata XML Schema

2543 **TODO: Out of Date**

2544 This XML schema was generated from the Ecore model in Appendix B.4 by the Eclipse Modeling
2545 Framework tools.

```

2546  <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2547  <xsd:schema xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore"
2548  xmlns:uima.peMetadata="http://docs.oasis-open.org/uima/pemetadata.ecore"
2549  xmlns:xmi="http://www.omg.org/XMI"
2550  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
2551  targetNamespace="http://docs.oasis-open.org/uima/pemetadata.ecore">
2552      <xsd:import namespace="http://www.eclipse.org/emf/2002/Ecore"
2553      schemaLocation="ecore.xsd"/>
2554      <xsd:import namespace="http://www.omg.org/XMI" schemaLocation="XMI.xsd"/>
2555      <xsd:complexType name="Identification">
2556          <xsd:choice maxOccurs="unbounded" minOccurs="0">
2557              <xsd:element ref="xmi:Extension"/>
2558          </xsd:choice>
2559          <xsd:attribute ref="xmi:id"/>
2560          <xsd:attributeGroup ref="xmi:ObjectAttrs"/>
2561          <xsd:attribute name="symbolicName" type="xsd:string"/>
2562          <xsd:attribute name="name" type="xsd:string"/>
2563          <xsd:attribute name="description" type="xsd:string"/>
2564          <xsd:attribute name="vendor" type="xsd:string"/>
2565          <xsd:attribute name="version" type="xsd:string"/>
2566          <xsd:attribute name="url" type="xsd:string"/>
2567      </xsd:complexType>
2568      <xsd:element name="Identification" type="uima.peMetadata:Identification"/>
2569      <xsd:complexType name="ConfigurationParameter">
2570          <xsd:choice maxOccurs="unbounded" minOccurs="0">

```

```

2571      <xsd:element name="defaultValue" nillable="true" type="xsd:string"/>
2572      <xsd:element ref="xmi:Extension"/>
2573  </xsd:choice>
2574  <xsd:attribute ref="xmi:id"/>
2575  <xsd:attributeGroup ref="xmi:ObjectAttrs"/>
2576  <xsd:attribute name="name" type="xsd:string"/>
2577  <xsd:attribute name="description" type="xsd:string"/>
2578  <xsd:attribute name="type" type="xsd:string"/>
2579  <xsd:attribute name="multiValued" type="xsd:boolean"/>
2580  <xsd:attribute name="mandatory" type="xsd:boolean"/>
2581 </xsd:complexType>
2582 <xsd:element name="ConfigurationParameter"
2583 type="uima.peMetadata:ConfigurationParameter"/>
2584 <xsd:complexType name="TypeSystemReference">
2585   <xsd:choice maxOccurs="unbounded" minOccurs="0">
2586     <xsd:element ref="xmi:Extension"/>
2587   </xsd:choice>
2588   <xsd:attribute ref="xmi:id"/>
2589   <xsd:attributeGroup ref="xmi:ObjectAttrs"/>
2590   <xsd:attribute name="uri" type="xsd:string"/>
2591 </xsd:complexType>
2592 <xsd:element name="TypeSystemReference"
2593 type="uima.peMetadata>TypeSystemReference"/>
2594 <xsd:complexType name="BehavioralMetadata">
2595   <xsd:choice maxOccurs="unbounded" minOccurs="0">
2596     <xsd:element name="analyzes" type="uima.peMetadata:BehaviorElement"/>
2597     <xsd:element name="requiredInputs"
2598 type="uima.peMetadata:BehaviorElement"/>
2599     <xsd:element name="optionalInputs"
2600 type="uima.peMetadata:BehaviorElement"/>
2601     <xsd:element name="creates" type="uima.peMetadata:BehaviorElement"/>
2602     <xsd:element name="modifies" type="uima.peMetadata:BehaviorElement"/>
2603     <xsd:element name="deletes" type="uima.peMetadata:BehaviorElement"/>
2604     <xsd:element name="precondition" type="uima.peMetadata:Condition"/>
2605     <xsd:element name="postcondition" type="uima.peMetadata:Condition"/>
2606     <xsd:element name="projectionCondition"
2607 type="uima.peMetadata:Condition"/>
2608     <xsd:element name="requiredView"
2609 type="uima.peMetadata:ViewBehavioralMetadata"/>
2610     <xsd:element name="optionalView"
2611 type="uima.peMetadata:ViewBehavioralMetadata"/>
2612     <xsd:element ref="xmi:Extension"/>
2613   </xsd:choice>
2614   <xsd:attribute ref="xmi:id"/>
2615   <xsd:attributeGroup ref="xmi:ObjectAttrs"/>

```

```

2616    </xsd:complexType>
2617    <xsd:element name="BehavioralMetadata"
2618 type="uima.peMetadata:BehavioralMetadata"/>
2619    <xsd:complexType name="ProcessingElementMetadata">
2620        <xsd:choice maxOccurs="unbounded" minOccurs="0">
2621            <xsd:element name="configurationParameter"
2622 type="uima.peMetadata:ConfigurationParameter"/>
2623            <xsd:element name="identification"
2624 type="uima.peMetadata:Identification"/>
2625            <xsd:element name="typeSystemReference"
2626 type="uima.peMetadata>TypeSystemReference"/>
2627            <xsd:element name="behavioralMetadata"
2628 type="uima.peMetadata:BehavioralMetadata"/>
2629                <xsd:element name="extension" type="uima.peMetadata:Extension"/>
2630                <xsd:element ref="xmi:Extension"/>
2631            </xsd:choice>
2632            <xsd:attribute ref="xmi:id"/>
2633            <xsd:attributeGroup ref="xmi:ObjectAttrs"/>
2634        </xsd:complexType>
2635        <xsd:element name="ProcessingElementMetadata"
2636 type="uima.peMetadata:ProcessingElementMetadata"/>
2637        <xsd:complexType name="Extension">
2638            <xsd:choice maxOccurs="unbounded" minOccurs="0">
2639                <xsd:element name="contents" type="ecore:EObject"/>
2640                <xsd:element ref="xmi:Extension"/>
2641            </xsd:choice>
2642            <xsd:attribute ref="xmi:id"/>
2643            <xsd:attributeGroup ref="xmi:ObjectAttrs"/>
2644            <xsd:attribute name="extenderId" type="xsd:string"/>
2645            <xsd:attribute name="contents" type="xsd:string"/>
2646        </xsd:complexType>
2647        <xsd:element name="Extension" type="uima.peMetadata:Extension"/>
2648        <xsd:complexType name="BehaviorElement">
2649            <xsd:choice maxOccurs="unbounded" minOccurs="0">
2650                <xsd:element name="type" type="uima.peMetadata>Type"/>
2651                <xsd:element ref="xmi:Extension"/>
2652            </xsd:choice>
2653            <xsd:attribute ref="xmi:id"/>
2654            <xsd:attributeGroup ref="xmi:ObjectAttrs"/>
2655        </xsd:complexType>
2656        <xsd:element name="BehaviorElement"
2657 type="uima.peMetadata:BehaviorElement"/>
2658        <xsd:complexType name="Type">
2659            <xsd:choice maxOccurs="unbounded" minOccurs="0">
2660                <xsd:element name="feature" nillable="true" type="xsd:string"/>

```

```

2661      <xsd:element ref="xmi:Extension"/>
2662  </xsd:choice>
2663  <xsd:attribute ref="xmi:id"/>
2664  <xsd:attributeGroup ref="xmi:ObjectAttrs"/>
2665    <xsd:attribute name="name" type="xsd:string"/>
2666 </xsd:complexType>
2667 <xsd:element name="Type" type="uima.peMetadata>Type"/>
2668 <xsd:complexType name="Condition">
2669  <xsd:choice maxOccurs="unbounded" minOccurs="0">
2670    <xsd:element name="feature" nullable="true" type="xsd:string"/>
2671    <xsd:element ref="xmi:Extension"/>
2672 </xsd:choice>
2673  <xsd:attribute ref="xmi:id"/>
2674  <xsd:attributeGroup ref="xmi:ObjectAttrs"/>
2675    <xsd:attribute name="language" type="xsd:string"/>
2676    <xsd:attribute name="expression" type="xsd:string"/>
2677 </xsd:complexType>
2678 <xsd:element name="Condition" type="uima.peMetadata:Condition"/>
2679 <xsd:complexType name="ViewBehavioralMetadata">
2680  <xsd:complexContent>
2681    <xsd:extension base="uima.peMetadata:BehavioralMetadata"/>
2682  </xsd:complexContent>
2683 </xsd:complexType>
2684 <xsd:element name="ViewBehavioralMetadata"
2685 type="uima.peMetadata:ViewBehavioralMetadata"/>
2686 </xsd:schema>

```

2687 B.6 PE Service WSDL Definition

2688 **TODO: This is out of date**

```

2689 <?xml version="1.0" encoding="UTF-8"?>
2690 <wsdl:definitions
2691   targetNamespace="http://docs.oasis-open.org/uima/peService"
2692   xmlns:service="http://docs.oasis-open.org/uima/peService"
2693   xmlns:pemd="http://docs.oasis-open.org/uima/peMetadata.ecore"
2694   xmlns:pe="http://docs.oasis-open.org/uima/pe.ecore"
2695   xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
2696   xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
2697   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
2698   xmlns:xmi="http://www.omg.org/XMI">
2699
2700 <wsdl:types>
2701  <!-- Import the PE Metadata Schema Definitions -->
2702  <xsd:import

```

```

2703     namespace="http://docs.oasis-open.org/uima/peMetadata.ecore"
2704     schemaLocation="uima.peMetadataXMI.xsd"/>
2705
2706     <!-- Import the XMI schema. -->
2707     <xsd:import namespace="http://www.omg.org/XMI"
2708         schemaLocation="XMI.xsd"/>
2709
2710     <!-- Import other type definitions used as part of the service API. -->
2711     <xsd:import
2712         namespace="http://docs.oasis-open.org/uima/pe.ecore"
2713         schemaLocation="uima.peServiceXMI.xsd"/>
2714 </wsdl:types>
2715
2716     <!-- Define the messages sent to and from the service. -->
2717
2718     <!-- Messages for all UIMA Processing Elements -->
2719     <wsdl:message name="getMetadataRequest">
2720     </wsdl:message>
2721
2722     <wsdl:message name="getMetadataResponse">
2723         <wsdl:part element="metadata"
2724             type="pemd:ProcessingElementMetadata" name="metadata"/>
2725     </wsdl:message>
2726
2727     <wsdl:message name="setConfigurationParametersRequest">
2728         <wsdl:part element="settings"
2729             type="pemd:ConfigurationParameterSettings" name="settings"/>
2730     </wsdl:message>
2731
2732     <wsdl:message name="setConfigurationParametersResponse">
2733     </wsdl:message>
2734
2735     <wsdl:message name="uimaFault">
2736         <wsdl:part element="exception" type="pe:UimaException" name="exception"/>
2737     </wsdl:message>
2738
2739
2740     <!-- Messages for the Analyzer interface -->
2741     <!-- Note that processCasRequest and processCasResponse allow
2742         multiple CASes to be sent in one batch, for performance
2743         reasons. -->
2744
2745     <wsdl:message name="processCasRequest">

```

```

2746     <wsdl:part element="casList" type="pe:CasList" name="casList"/>
2747     <wsdl:part element="sofas" type="pe:ObjectList" name="sofas"/>
2748 </wsdl:message>
2749
2750     <wsdl:message name="processCasResponse">
2751         <wsdl:part element="casList" type="pe:CasList" name="casList"/>
2752     </wsdl:message>
2753
2754     <!-- Messages for the CasMultiplier interface -->
2755     <!-- Note that inputCasRequest and getNextResponse allow
2756         multiple CASEs to be sent in one batch, for performance
2757         reasons.  -->
2758     <wsdl:message name="inputCasRequest">
2759         <wsdl:part element="casList" type="pe:CasList" name="casList"/>
2760         <wsdl:part element="sofas" type="pe:ObjectList" name="sofas"/>
2761     </wsdl:message>
2762
2763     <wsdl:message name="inputCasResponse">
2764     </wsdl:message>
2765
2766     <wsdl:message name="getNextRequest">
2767         <wsdl:part element="maxCASEsToReturn" type="xsd:integer"
2768 name="maxCASEsToReturn"/>
2769         <wsdl:part element="timeToWait" type="xsd:integer" name="timeToWait"/>
2770     </wsdl:message>
2771
2772     <wsdl:message name="getNextResponse">
2773         <wsdl:part element="reponse" type="pe:GetNextResponse" name="response"/>
2774     </wsdl:message>
2775
2776     <wsdl:message name="retrieveInputCasRequest">
2777     </wsdl:message>
2778
2779     <wsdl:message name="retrieveInputCasResponse">
2780         <wsdl:part element="casList" type="pe:CasList" name="casList"/>
2781     </wsdl:message>
2782
2783     <!-- Messages for the FlowController interface -->
2784
2785     <wsdl:message name="addAvailableAnalyticsRequest">
2786         <wsdl:part element="analyticMetadataMap"
2787             type="pe:AnalyticMetadataMap" name="analyticMetadataMap"/>
2788     </wsdl:message>

```

```

2789
2790     <wsdl:message name="addAvailableAnalyticsResponse">
2791     </wsdl:message>
2792
2793     <wsdl:message name="removeAvailableAnalyticsRequest">
2794         <wsdl:part element="analyticKeys" type="pe:Keys"
2795             name="analyticKeys"/>
2796     </wsdl:message>
2797
2798     <wsdl:message name="removeAvailableAnalyticsResponse">
2799     </wsdl:message>
2800
2801     <wsdl:message name="setAggregateMetadataRequest">
2802         <wsdl:part element="metadata"
2803             type="pemd:ProcessingElementMetadata" name="metadata"/>
2804     </wsdl:message>
2805
2806     <wsdl:message name="setAggregateMetadataResponse">
2807     </wsdl:message>
2808
2809     <wsdl:message name="getNextDestinationsRequest">
2810         <wsdl:part element="cas" type="xmi:XMI" name="cas"/>
2811     </wsdl:message>
2812
2813     <wsdl:message name="getNextDestinationsResponse">
2814         <wsdl:part element="step" type="pe:Step" name="step"/>
2815     </wsdl:message>
2816
2817     <wsdl:message name="continueOnFailureRequest">
2818         <wsdl:part element="cas" type="xmi:XMI" name="cas"/>
2819         <wsdl:part element="failedAnalyticKey" type="xsd:string"
2820 name="failedAnalyticKey"/>
2821         <wsdl:part element="failure" type="pe:UimaException" name="failure"/>
2822     </wsdl:message>
2823
2824     <wsdl:message name="continueOnFailureResponse">
2825         <wsdl:part element="continue" type="xsd:boolean" name="continue"/>
2826     </wsdl:message>
2827
2828     <!-- Define a portType for each of the UIMA interfaces -->
2829     <wsdl:portType name="Analyzer">
2830
2831         <wsdl:operation name="getMetadata">

```

```

2832     <wsdl:input message="service:getMetadataRequest"
2833         name="getMetadataRequest"/>
2834     <wsdl:output message="service:getMetadataResponse"
2835         name="getMetadataResponse"/>
2836     <wsdl:fault message="service:uimaFault"
2837         name="uimaFault"/>
2838   </wsdl:operation>
2839
2840   <wsdl:operation name="setConfigurationParameters">
2841     <wsdl:input
2842         message="service:setConfigurationParametersRequest"
2843         name="setConfigurationParametersRequest"/>
2844     <wsdl:output
2845         message="service:setConfigurationParametersResponse"
2846         name="setConfigurationParametersResponse"/>
2847     <wsdl:fault message="service:uimaFault"
2848         name="uimaFault"/>
2849   </wsdl:operation>
2850
2851   <wsdl:operation name="processCas">
2852     <wsdl:input message="service:processCasRequest"
2853         name="processCasRequest"/>
2854     <wsdl:output message="service:processCasResponse"
2855         name="processCasResponse"/>
2856     <wsdl:fault message="service:uimaFault"
2857         name="uimaFault"/>
2858   </wsdl:operation>
2859
2860 </wsdl:portType>
2861
2862 <wsdl:portType name="CasMultiplier">
2863
2864   <wsdl:operation name="getMetadata">
2865     <wsdl:input message="service:getMetadataRequest"
2866         name="getMetadataRequest"/>
2867     <wsdl:output message="service:getMetadataResponse"
2868         name="getMetadataResponse"/>
2869     <wsdl:fault message="service:uimaFault"
2870         name="uimaFault"/>
2871   </wsdl:operation>
2872
2873   <wsdl:operation name="setConfigurationParameters">
2874     <wsdl:input

```

```

2875     message="service:setConfigurationParametersRequest"
2876     name="setConfigurationParametersRequest"/>
2877   <wsdl:output
2878     message="service:setConfigurationParametersResponse"
2879     name="setConfigurationParametersResponse"/>
2880   <wsdl:fault message="service:uimaFault"
2881     name="uimaFault"/>
2882 </wsdl:operation>
2883
2884 <wsdl:operation name="inputCas">
2885   <wsdl:input message="service:inputCasRequest"
2886     name="inputCasRequest"/>
2887   <wsdl:output message="service:inputCasResponse"
2888     name="inputCasResponse"/>
2889   <wsdl:fault message="service:uimaFault"
2890     name="uimaFault"/>
2891 </wsdl:operation>
2892
2893 <wsdl:operation name="getNext">
2894   <wsdl:input message="service:getNextRequest"
2895     name="getNextRequest"/>
2896   <wsdl:output message="service:getNextResponse"
2897     name="getNextResponse"/>
2898   <wsdl:fault message="service:uimaFault"
2899     name="uimaFault"/>
2900 </wsdl:operation>
2901
2902 <wsdl:operation name="retrieveInputCas">
2903   <wsdl:input message="service:retrieveInputCasRequest"
2904     name="retrieveInputCasRequest"/>
2905   <wsdl:output message="service:retrieveInputCasResponse"
2906     name="retrieveInputCasResponse"/>
2907   <wsdl:fault message="service:uimaFault"
2908     name="uimaFault"/>
2909 </wsdl:operation>
2910 </wsdl:portType>
2911
2912 <wsdl:portType name="FlowController">
2913
2914   <wsdl:operation name="getMetadata">
2915     <wsdl:input message="service:getMetadataRequest"
2916       name="getMetadataRequest"/>
2917     <wsdl:output message="service:getMetadataResponse"

```

```

2918     name="getMetadataResponse"/>
2919     <wsdl:fault message="service:uimaFault"
2920         name="uimaFault"/>
2921 </wsdl:operation>
2922
2923 <wsdl:operation name="setConfigurationParameters">
2924     <wsdl:input
2925         message="service:setConfigurationParametersRequest"
2926         name="setConfigurationParametersRequest"/>
2927     <wsdl:output
2928         message="service:setConfigurationParametersResponse"
2929         name="setConfigurationParametersResponse"/>
2930     <wsdl:fault message="service:uimaFault"
2931         name="uimaFault"/>
2932 </wsdl:operation>
2933
2934 <wsdl:operation name="addAvailableAnalytics">
2935     <wsdl:input message="service:addAvailableAnalyticsRequest"
2936         name="addAvailableAnalyticsRequest"/>
2937     <wsdl:output message="service:addAvailableAnalyticsResponse"
2938         name="addAvailableAnalyticsResponse"/>
2939     <wsdl:fault message="service:uimaFault"
2940         name="uimaFault"/>
2941 </wsdl:operation>
2942
2943 <wsdl:operation name="removeAvailableAnalytics">
2944     <wsdl:input
2945         message="service:removeAvailableAnalyticsRequest"
2946         name="removeAvailableAnalyticsRequest"/>
2947     <wsdl:output
2948         message="service:removeAvailableAnalyticsResponse"
2949         name="removeAvailableAnalyticsResponse"/>
2950     <wsdl:fault message="service:uimaFault"
2951         name="uimaFault"/>
2952 </wsdl:operation>
2953
2954 <wsdl:operation name="setAggregateMetadata">
2955     <wsdl:input message="service:setAggregateMetadataRequest"
2956         name="setAggregateMetadataRequest"/>
2957     <wsdl:output message="service:setAggregateMetadataResponse"
2958         name="setAggregateMetadataResponse"/>
2959     <wsdl:fault message="service:uimaFault"
2960         name="uimaFault"/>

```

```

2961      </wsdl:operation>
2962
2963      <wsdl:operation name="getNextDestinations">
2964          <wsdl:input message="service:getNextDestinationsRequest"
2965              name="getNextDestinationsRequest"/>
2966          <wsdl:output message="service:getNextDestinationsResponse"
2967              name="getNextDestinationsResponse"/>
2968          <wsdl:fault message="service:uimaFault"
2969              name="uimaFault"/>
2970      </wsdl:operation>
2971
2972      <wsdl:operation name="continueOnFailure">
2973          <wsdl:input message="service:continueOnFailureRequest"
2974              name="continueOnFailureRequest"/>
2975          <wsdl:output message="service:continueOnFailureResponse"
2976              name="continueOnFailureResponse"/>
2977          <wsdl:fault message="service:uimaFault"
2978              name="uimaFault"/>
2979      </wsdl:operation>
2980
2981  </wsdl:portType>
2982
2983  <!-- Define a SOAP binding for each portType. -->
2984  <wsdl:binding name="AnalyzerSoapBinding" type="service:Analyzer">
2985
2986      <wsdlsoap:binding style="rpc"
2987          transport="http://schemas.xmlsoap.org/soap/http"/>
2988
2989      <wsdl:operation name="getMetadata">
2990          <wsdlsoap:operation soapAction="" />
2991
2992          <wsdl:input name="getMetadataRequest">
2993              <wsdlsoap:body use="literal"/>
2994          </wsdl:input>
2995
2996          <wsdl:output name="getMetadataResponse">
2997              <wsdlsoap:body use="literal"/>
2998          </wsdl:output>
2999      </wsdl:operation>
3000
3001      <wsdl:operation name="setConfigurationParameters">
3002          <wsdlsoap:operation soapAction="" />
3003
```

```

3004      <wsdl:input name="setConfigurationParametersRequest">
3005          <wsdlsoap:body use="literal"/>
3006      </wsdl:input>
3007
3008      <wsdl:output name="setConfigurationParametersResponse">
3009          <wsdlsoap:body use="literal"/>
3010      </wsdl:output>
3011  </wsdl:operation>
3012
3013  <wsdl:operation name="processCas">
3014      <wsdlsoap:operation soapAction="" />
3015
3016      <wsdl:input name="processCasRequest">
3017          <wsdlsoap:body use="literal"/>
3018      </wsdl:input>
3019
3020      <wsdl:output name="processCasResponse">
3021          <wsdlsoap:body use="literal"/>
3022      </wsdl:output>
3023  </wsdl:operation>
3024 </wsdl:binding>
3025
3026 <wsdl:binding name="CasMultiplierSoapBinding"
3027   type="service:CasMultiplier">
3028
3029     <wsdlsoap:binding style="rpc"
3030       transport="http://schemas.xmlsoap.org/soap/http"/>
3031
3032     <wsdl:operation name="getMetadata">
3033         <wsdlsoap:operation soapAction="" />
3034
3035         <wsdl:input name="getMetadataRequest">
3036             <wsdlsoap:body use="literal"/>
3037         </wsdl:input>
3038
3039         <wsdl:output name="getMetadataResponse">
3040             <wsdlsoap:body use="literal"/>
3041         </wsdl:output>
3042
3043         <wsdl:fault name="uimaFault">
3044             <wsdlsoap:fault use="literal"/>
3045         </wsdl:fault>
3046     </wsdl:operation>

```

```

3047
3048 <wsdl:operation name="setConfigurationParameters">
3049   <wsdlsoap:operation soapAction="" />
3050
3051   <wsdl:input name="setConfigurationParametersRequest">
3052     <wsdlsoap:body use="literal" />
3053   </wsdl:input>
3054
3055   <wsdl:output name="setConfigurationParametersResponse">
3056     <wsdlsoap:body use="literal" />
3057   </wsdl:output>
3058
3059   <wsdl:fault name="uimaFault">
3060     <wsdlsoap:fault use="literal" />
3061   </wsdl:fault>
3062 </wsdl:operation>
3063
3064 <wsdl:operation name="inputCas">
3065   <wsdlsoap:operation soapAction="" />
3066
3067   <wsdl:input name="inputCasRequest">
3068     <wsdlsoap:body use="literal" />
3069   </wsdl:input>
3070
3071   <wsdl:output name="inputCasResponse">
3072     <wsdlsoap:body use="literal" />
3073   </wsdl:output>
3074
3075   <wsdl:fault name="uimaFault">
3076     <wsdlsoap:fault use="literal" />
3077   </wsdl:fault>
3078 </wsdl:operation>
3079
3080 <wsdl:operation name="getNext">
3081   <wsdlsoap:operation soapAction="" />
3082
3083   <wsdl:input name="getNextRequest">
3084     <wsdlsoap:body use="literal" />
3085   </wsdl:input>
3086
3087   <wsdl:output name="getNextResponse">
3088     <wsdlsoap:body use="literal" />
3089   </wsdl:output>

```

```

3090
3091     <wsdl:fault name="uimaFault">
3092         <wsdlsoap:fault use="literal"/>
3093     </wsdl:fault>
3094 </wsdl:operation>
3095
3096 <wsdl:operation name="retrieveInputCas">
3097     <wsdlsoap:operation soapAction="" />
3098
3099     <wsdl:input name="retrieveInputCasRequest">
3100         <wsdlsoap:body use="literal"/>
3101     </wsdl:input>
3102
3103     <wsdl:output name="retrieveInputCasResponse">
3104         <wsdlsoap:body use="literal"/>
3105     </wsdl:output>
3106
3107     <wsdl:fault name="uimaFault">
3108         <wsdlsoap:fault use="literal"/>
3109     </wsdl:fault>
3110 </wsdl:operation>
3111 </wsdl:binding>
3112
3113 <wsdl:binding name="FlowControllerSoapBinding"
3114     type="service:FlowController">
3115
3116     <wsdlsoap:binding style="rpc"
3117         transport="http://schemas.xmlsoap.org/soap/http"/>
3118
3119     <wsdl:operation name="getMetadata">
3120         <wsdlsoap:operation soapAction="" />
3121
3122         <wsdl:input name="getMetadataRequest">
3123             <wsdlsoap:body use="literal"/>
3124         </wsdl:input>
3125
3126         <wsdl:output name="getMetadataResponse">
3127             <wsdlsoap:body use="literal"/>
3128         </wsdl:output>
3129
3130         <wsdl:fault name="uimaFault">
3131             <wsdlsoap:fault use="literal"/>
3132         </wsdl:fault>

```

```

3133     </wsdl:operation>
3134
3135     <wsdl:operation name="setConfigurationParameters">
3136         <wsdlsoap:operation soapAction="" />
3137
3138         <wsdl:input name="setConfigurationParametersRequest">
3139             <wsdlsoap:body use="literal" />
3140         </wsdl:input>
3141
3142         <wsdl:output name="setConfigurationParametersResponse">
3143             <wsdlsoap:body use="literal" />
3144         </wsdl:output>
3145
3146         <wsdl:fault name="uimaFault">
3147             <wsdlsoap:fault use="literal" />
3148         </wsdl:fault>
3149     </wsdl:operation>
3150
3151     <wsdl:operation name="addAvailableAnalytics">
3152         <wsdlsoap:operation soapAction="" />
3153
3154         <wsdl:input name="addAvailableAnalyticsRequest">
3155             <wsdlsoap:body use="literal" />
3156         </wsdl:input>
3157
3158         <wsdl:output name="addAvailableAnalyticsResponse">
3159             <wsdlsoap:body use="literal" />
3160         </wsdl:output>
3161
3162         <wsdl:fault name="uimaFault">
3163             <wsdlsoap:fault use="literal" />
3164         </wsdl:fault>
3165     </wsdl:operation>
3166
3167     <wsdl:operation name="removeAvailableAnalytics">
3168         <wsdlsoap:operation soapAction="" />
3169
3170         <wsdl:input name="removeAvailableAnalyticsRequest">
3171             <wsdlsoap:body use="literal" />
3172         </wsdl:input>
3173
3174         <wsdl:output name="removeAvailableAnalyticsResponse">
3175             <wsdlsoap:body use="literal" />

```

```

3176      </wsdl:output>
3177
3178      <wsdl:fault name="uimaFault">
3179          <wsdlsoap:fault use="literal"/>
3180      </wsdl:fault>
3181  </wsdl:operation>
3182
3183  <wsdl:operation name="setAggregateMetadata">
3184      <wsdlsoap:operation soapAction="" />
3185
3186      <wsdl:input name="setAggregateMetadataRequest">
3187          <wsdlsoap:body use="literal"/>
3188      </wsdl:input>
3189
3190      <wsdl:output name="setAggregateMetadataResponse">
3191          <wsdlsoap:body use="literal"/>
3192      </wsdl:output>
3193
3194      <wsdl:fault name="uimaFault">
3195          <wsdlsoap:fault use="literal"/>
3196      </wsdl:fault>
3197  </wsdl:operation>
3198
3199  <wsdl:operation name="getNextDestinations">
3200      <wsdlsoap:operation soapAction="" />
3201
3202      <wsdl:input name="getNextDestinationsRequest">
3203          <wsdlsoap:body use="literal"/>
3204      </wsdl:input>
3205
3206      <wsdl:output name="getNextDestinationsResponse">
3207          <wsdlsoap:body use="literal"/>
3208      </wsdl:output>
3209
3210      <wsdl:fault name="uimaFault">
3211          <wsdlsoap:fault use="literal"/>
3212      </wsdl:fault>
3213  </wsdl:operation>
3214
3215  <wsdl:operation name="continueOnFailure">
3216      <wsdlsoap:operation soapAction="" />
3217
3218      <wsdl:input name="continueOnFailureRequest">

```

```

3219      <wsdlsoap:body use="literal"/>
3220    </wsdl:input>
3221
3222    <wsdl:output name="continueOnFailureResponse">
3223      <wsdlsoap:body use="literal"/>
3224    </wsdl:output>
3225
3226    <wsdl:fault name="uimaFault">
3227      <wsdlsoap:fault use="literal"/>
3228    </wsdl:fault>
3229  </wsdl:operation>
3230 </wsdl:binding>
3231
3232  <!-- Define an example service as including both portTypes. This is
3233      just an example, not part of the UIMA Specification -->
3234 <wsdl:service name="MyAnalyticService">
3235   <wsdl:port binding="service:AnalyzerSoapBinding"
3236     name="AnalyzerSoapPort">
3237     <wsdlsoap:address
3238
3239   location="http://localhost:8080/axis/services/MyAnalyticService/AnalyzerPort"
3240 />
3241   </wsdl:port>
3242   <wsdl:port binding="service:CasMultiplierSoapBinding"
3243     name="CasMultiplierSoapPort">
3244     <wsdlsoap:address
3245
3246   location="http://localhost:8080/axis/services/MyAnalyticService/CasMultiplier
3247 Port"/>
3248   </wsdl:port>
3249 </wsdl:service>
3250 </wsdl:definitions>

```

3251 **B.7 PE Service XML Schema (uima.peServiceXMI.xsd)**

3252 **TODo: Out of date**

3253 This XML schema is referenced from the WSDL definition in Appendix B.6

```

3254 <?xml version="1.0" encoding="UTF-8"?>
3255 <xsd:schema xmlns:pe="http://docs.oasis-open.org/uima/pe.ecore"
3256   xmlns:pemd="http://docs.oasis-open.org/uima/peMetadata.ecore"
3257   xmlns:xmi="http://www.omg.org/XMI"
3258   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
3259   targetNamespace="http://docs.oasis-open.org/uima/pe.ecore">
3260   <xsd:import
3261     namespace="http://docs.oasis-open.org/uima/peMetadata.ecore"

```

```

3262     schemaLocation="uima.peMetadataXMI.xsd"/>
3263 <xsd:import namespace="http://www.omg.org/XMI"
3264   schemaLocation="XMI.xsd"/>
3265 <xsd:complexType name="InputBindings">
3266   <xsd:choice maxOccurs="unbounded" minOccurs="0">
3267     <xsd:element name="InputBinding" type="pe:InputBinding"/>
3268     <xsd:element ref="xmi:Extension"/>
3269   </xsd:choice>
3270   <xsd:attribute ref="xmi:id"/>
3271   <xsd:attributeGroup ref="xmi:ObjectAttrs"/>
3272   <xsd:attribute name="InputBinding" type="xsd:string"/>
3273 </xsd:complexType>
3274 <xsd:element name="InputBindings" type="pe:InputBindings"/>
3275 <xsd:complexType name="InputBinding">
3276   <xsd:choice maxOccurs="unbounded" minOccurs="0">
3277     <xsd:element name="objects" nillable="true"
3278       type="xsd:string"/>
3279     <xsd:element ref="xmi:Extension"/>
3280   </xsd:choice>
3281   <xsd:attribute ref="xmi:id"/>
3282   <xsd:attributeGroup ref="xmi:ObjectAttrs"/>
3283   <xsd:attribute name="handle" type="xsd:string"/>
3284 </xsd:complexType>
3285 <xsd:element name="InputBinding" type="pe:InputBinding"/>
3286 <xsd:complexType name="AnalyticMetadataMap">
3287   <xsd:choice maxOccurs="unbounded" minOccurs="0">
3288     <xsd:element name="AnalyticMetadataMapEntry"
3289       type="pe:AnalyticMetadataMapEntry"/>
3290     <xsd:element ref="xmi:Extension"/>
3291   </xsd:choice>
3292   <xsd:attribute ref="xmi:id"/>
3293   <xsd:attributeGroup ref="xmi:ObjectAttrs"/>
3294   <xsd:attribute name="AnalyticMetadataMapEntry"
3295     type="xsd:string"/>
3296 </xsd:complexType>
3297 <xsd:element name="AnalyticMetadataMap"
3298   type="pe:AnalyticMetadataMap"/>
3299 <xsd:complexType name="AnalyticMetadataMapEntry">
3300   <xsd:choice maxOccurs="unbounded" minOccurs="0">
3301     <xsd:element name="ProcessingElementMetadata"
3302       type="pemd:ProcessingElementMetadata"/>
3303     <xsd:element ref="xmi:Extension"/>
3304   </xsd:choice>

```

```

3305    <xsd:attribute ref="xmi:id"/>
3306    <xsd:attributeGroup ref="xmi:ObjectAttrs"/>
3307    <xsd:attribute name="key" type="xsd:string"/>
3308    <xsd:attribute name="ProcessingElementMetadata"
3309        type="xsd:string"/>
3310  </xsd:complexType>
3311  <xsd:element name="AnalyticMetadataMapEntry"
3312      type="pe:AnalyticMetadataMapEntry"/>
3313  <xsd:complexType name="Step">
3314    <xsd:choice maxOccurs="unbounded" minOccurs="0">
3315      <xsd:element ref="xmi:Extension"/>
3316    </xsd:choice>
3317    <xsd:attribute ref="xmi:id"/>
3318    <xsd:attributeGroup ref="xmi:ObjectAttrs"/>
3319  </xsd:complexType>
3320  <xsd:element name="Step" type="pe:Step"/>
3321  <xsd:complexType name="SimpleStep">
3322    <xsd:complexContent>
3323      <xsd:extension base="pe:Step">
3324        <xsd:attribute name="analyticKey" type="xsd:string"/>
3325      </xsd:extension>
3326    </xsd:complexContent>
3327  </xsd:complexType>
3328  <xsd:element name="SimpleStep" type="pe:SimpleStep"/>
3329  <xsd:complexType name="MultiStep">
3330    <xsd:complexContent>
3331      <xsd:extension base="pe:Step">
3332        <xsd:choice maxOccurs="unbounded" minOccurs="0">
3333          <xsd:element name="steps" type="pe:Step"/>
3334        </xsd:choice>
3335          <xsd:attribute name="parallel" type="xsd:boolean"/>
3336        </xsd:extension>
3337      </xsd:complexContent>
3338  </xsd:complexType>
3339  <xsd:element name="MultiStep" type="pe:MultiStep"/>
3340  <xsd:complexType name="FinalStep">
3341    <xsd:complexContent>
3342      <xsd:extension base="pe:Step"/>
3343    </xsd:complexContent>
3344  </xsd:complexType>
3345  <xsd:element name="FinalStep" type="pe:FinalStep"/>
3346  <xsd:complexType name="Keys">
3347    <xsd:choice maxOccurs="unbounded" minOccurs="0">

```

```
3348      <xsd:element name="key" nillable="true" type="xsd:string"/>
3349      <xsd:element ref="xmi:Extension"/>
3350    </xsd:choice>
3351    <xsd:attribute ref="xmi:id"/>
3352    <xsd:attributeGroup ref="xmi:ObjectAttrs"/>
3353  </xsd:complexType>
3354  <xsd:element name="Keys" type="pe:Keys"/>
3355</xsd:schema>
3356
3357
```

3359

D. Revision History

3360

[optional; should not be included in OASIS Standards]

3361

Revision	Date	Editor	Changes Made
1	11 March 2008	Adam Lally	First spec revision in OASIS template
2	10 April 2008	Adam Lally	Integrated Section 3.3 text from Karin. Rewrote Abstract Interface Compliance points to require standard XMLdata representation. Expanded Section 3.5.8 Behavioral Metadata Formal Specification, to include mapping to OCL. Other cleanup to sections 3.5 and 3.7.
3	24 April 2008	Adam Lally	Integrated Section 1 text from Dave, Section 3.1 and 3.2 text from Eric, additional Section 3.3 updates from Karen, and section 3.6 text from Thomas. Also fixed some UML diagrams in these sections.
			Added processCasBatch and getNextCasBatch operations to Abstract Interfaces so they would be in sync with the WSDL spec.
			Added 3.2.4.2 to reference XMI, UML, and MOF for definition of an object being a valid instance of a class.
			Fixed OCL in 3.5.8.3.1.

3362

3363