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] .doc
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] .doc
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] .doc
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] .doc
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] /.

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.

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

Notices

Table of Contents

1 Introduction

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

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

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

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

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

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

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

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

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

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

1. **Data Representation.** Support the common representation of *artifacts* and *artifact metadata* (analysis results) independently of *artifact modality* and *domain model*.

2. **Data Modeling and Interchange.** Support the platform-independent interchange of *analysis data* in a form that facilitates a formal modeling approach and alignment with existing programming systems and standards.

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

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

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

1.1 Terminology

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

1.2 Normative References

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

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

[OCL1] Object Management Group. Object Constraing Language Version 2.0. http://www.omg.org/technology/documents/formal/ocl.htm

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

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

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

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

[XML1] W3C. Extensible Markup Language (XML) 1.0 (Fourth Edition). http://www.w3.org/TR/REC-xml

[XML2] W3C. Namespaces in XML 1.0 (Second Edition). http://www.w3.org/TR/REC-xml-names/

[XMLS1] XML Schema Part 1: Structures Second Edition. http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/structures.html

[XMLS2] XML Schema Part 2: Datatypes Second Edition. http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/datatypes.html.

1.3 Non-Normative References

[BPEL1] http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel

[EcoreEMOF1] http://dev.eclipse.org/newslists/news.eclipse.tools.emf/msg04197.html

[EMF1] The Eclipse Modeling Framework (EMF) Overview. http://help.eclipse.org/help33/index.jsp?topic=/org.eclipse.emf.doc//references/overview/EMF.html

[EMF2] Budinsky et al. Eclipse Modeling Framework. Addison-Wesley. 2004.

[XMI2] Grose et al. Mastering XMI. Java Programming with XMI, XML, and UML. John Wiley & Sons, Inc. 2002

2 Basic Concepts and Terms

This specification defines and uses the following terms:

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

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

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

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

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

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

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

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

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

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

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

3 Elements of the UIMA Specification

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

The elements are listed in brief below:

1. **Common Analysis Structure (CAS).** Supports interoperability by providing a common data structure shared among analytics. The CAS is a general object graph and is used to represent the artifact and the artifact metadata. UIMA defines an XML representation of analysis data using the XML Metadata Interchange (XMI) specification [XMI1][XMI2].

2. **Type System Model**. To support data modeling and interchange, a CAS must conform to a user-defined schema called a Type System. Every object in a CAS must be associated with a Type defined by a Type System. UIMA defines the Type System representation using Ecore, which is the modeling language used in the Eclipse Modeling Framework [EMF1] and is tightly aligned with the OMG's EMOF standard. The XML representation uses XMI.

3. **Base Type System**. Provides a Standard definition of commonly-used, domain-independent types, in order to establish a basic level of interoperability among applications. For example UIMA defines the type Annotation to represent objects that have references (e.g., offsets) into the value of an attribute of another object. It is intended that annotations describe or "annotate" the unstructured content in these values.

4. **Abstract Interfaces.** Defines the standard component types and operations that UIMA developers can implement. This element is defined abstractly using a UML model.

5. **Behavioral Metadata.** Provides a formal declarative description of what a UIMA analytic does. This includes: what types of CASes it can process, what elements in a CAS it analyzes, and what effects it may have on CAS contents as a result of its application.

6. **Processing Element Metadata**. Provides a standard declarative means for describing identification, configuration and behavioral information about Processing Elements (analytics and flow controllers). This section of the specification refers to the Behavioral Metadata Specification to represent a processing element's behavioral information.

7. **WSDL Service Descriptions.** This specification element facilitates interoperability by specifying a WSDL [WSDL1] description of the UIMA interfaces and a binding to a concrete SOAP interface that compliant frameworks/services must implement.

3.1 The Common Analysis Structure (CAS)

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

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

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

There are two fundamental types of objects in a CAS:

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

A data representation which stores the artifact metadata separately from the subject of analysis is commonly referred to as *standoff annotation*. This approach is preferred to in-line annotation approaches (such as SGML) which change the subject of analysis by inserting tags directly into the content. Preserving the distinction between the original content and subsequent annotation provides better support for interoperability of analytics. The definitions of the Sofa and Annotation types are introduced in Section 3.3.2.

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

3.1.1 Basic Structure: Objects and Slots

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

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

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

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

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

3.1.2 Relationship to Type System

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

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

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

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

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

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

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

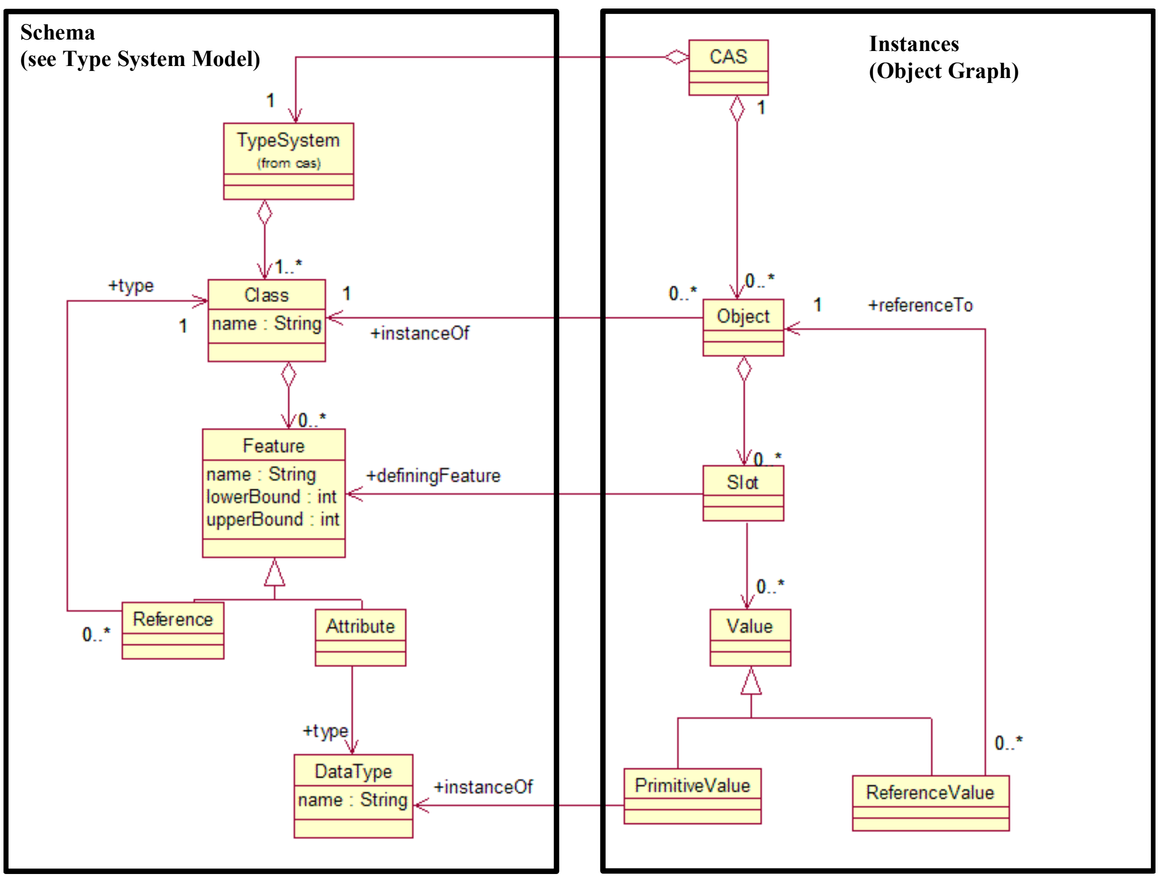

Figure 1: CAS Specification UML

3.1.3 The XMI CAS Representation

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

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

3.1.4 Example (Not Normative)

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

3.1.4.1 XMI Tag

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

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

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

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

3.1.4.2 Objects

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

For example consider the following XMI document:

```
<xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
  xmlns:myorg="http:///org/myorg.ecore">
  ...
  <myorg:Person xmi:id="1"/>
  ...
</xmi:XMI>
```

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

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

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

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

3.1.4.3 Attributes (Primitive Features)

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

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

begin = 14
end = 25
name = "Fred Center"

could be mapped to the attribute serialization as follows:

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

or alternatively to an element serialization as follows:

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

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

3.1.4.4 References (Object-Valued Features)

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

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

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

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

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

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

3.1.4.5 Multi-valued Features

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

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

or:

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

Note that string arrays whose elements contain embedded spaces must use the latter mapping.[1]

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

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

or:

```
<myorg:Baz xmi:id="3">
  <myRefFeature href="#13"/>
  <myRefFeature href="#42"/>
</myorg:Baz>
```

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

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.

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

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

3.1.6 XMI Extensions

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

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

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

3.1.7 Formal Specification

3.1.7.1 Structure

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

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

3.1.7.2 Constraints

3.1.7.2.1 Linkage of CAS to Ecore Type System

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

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

3.2 The Type System Model

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

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

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

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

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

3.2.1 Features of the Type System Model

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

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

3.2.2 Ecore as the UIMA Type System Representation

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

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

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

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

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

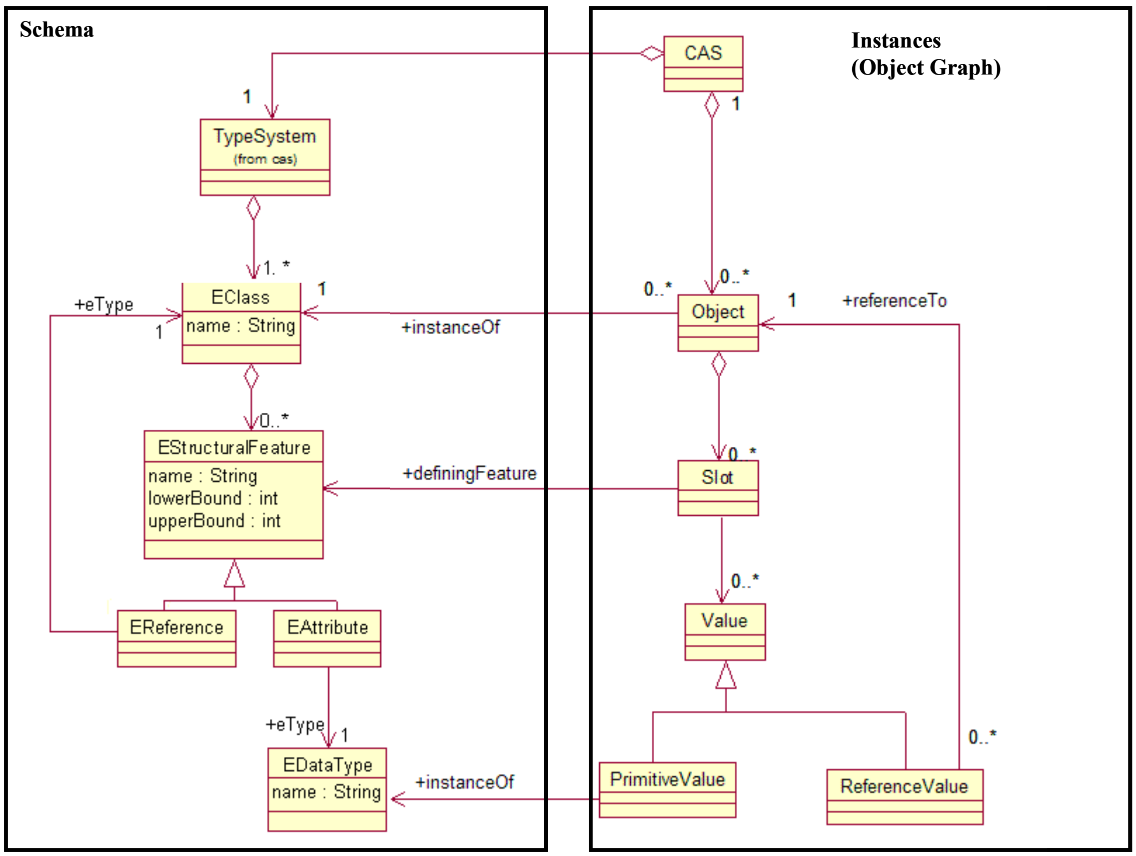

Figure 2: Ecore defines schema for CAS

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

3.2.3 Discussion and Example (Not Normative)

3.2.3.1 An Introduction to Ecore

Ecore is well described by Budisnky et al. in the book *Eclipse Modeling Framework*. Some brief introduction to Ecore can be found in a chapter of that book that is available online at http://www.awprofessional.com/content/images/0131425420/samplechapter/budinskych02.pdf (see 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.

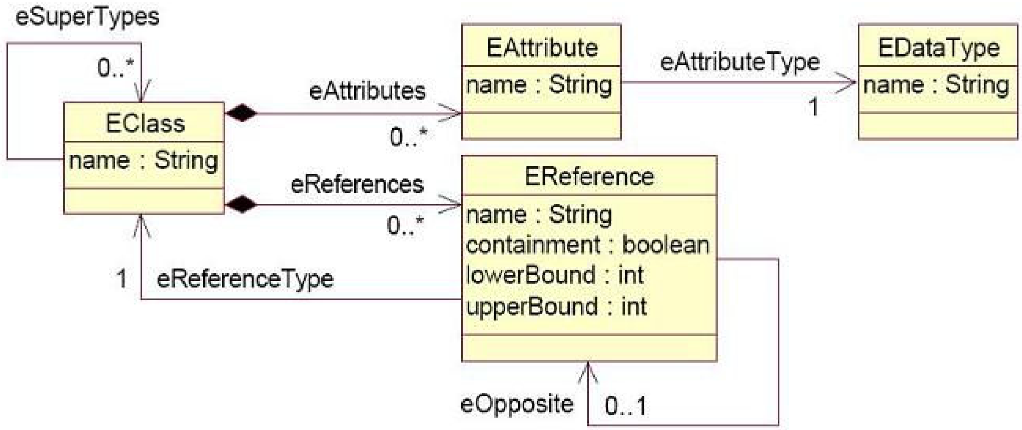

Figure 3: The Ecore Kernel

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.

3.2.3.2 Differences between Ecore and EMOF

The primary differences between Ecore and EMOF are:

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

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

3.2.3.3 Example Ecore Model

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

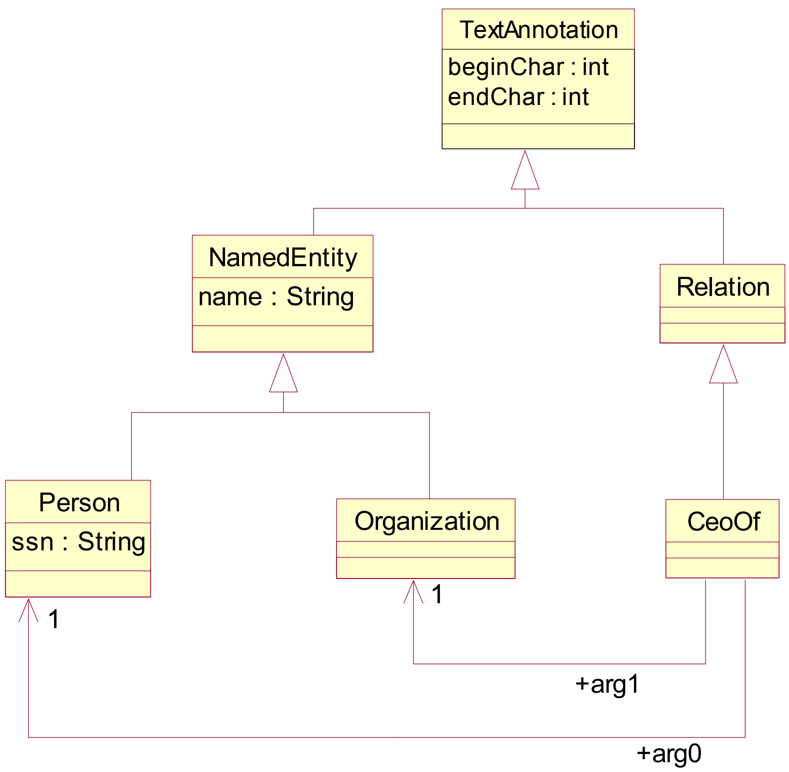

Figure 4: Example UML Model

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

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

```
  <eSubpackages name="example" nsURI="http:///org/example.ecore"
nsPrefix="org.example">
    <eClassifiers xsi:type="ecore:EClass" name="NamedEntity"
eSuperTypes="uima.cas:TextAnnotation">
      <eStructuralFeatures xsi:type="ecore:EAttribute" name="name"
eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    </eClassifiers>
    <eClassifiers xsi:type="ecore:EClass" name="Relation"
eSuperTypes="uima.cas:TextAnnotation"/>
    <eClassifiers xsi:type="ecore:EClass" name="Person"
eSuperTypes="#//example/NamedEntity">
      <eStructuralFeatures xsi:type="ecore:EAttribute" name="ssn"
eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    </eClassifiers>
    <eClassifiers xsi:type="ecore:EClass" name="CeoOf"
eSuperTypes="#//example/Relation">
      <eStructuralFeatures xsi:type="ecore:EReference" name="arg0"
lowerBound="1"
          eType="#//example/Person"/>
      <eStructuralFeatures xsi:type="ecore:EReference" name="arg1"
lowerBound="1"
          eType="#//example/Organization"/>
    </eClassifiers>
  </eSubpackages>
</ecore:EPackage>
```

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

3.2.4 Formal Specification

3.2.4.1 Structure

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

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

3.2.4.2 Semantics

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

3.3 Base Type System

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

The Base Type System includes the following:

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

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, whose namespace is defined by Ecore.)

3.3.1 Primitive Types

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

- EString
- EBoolean
- EByte (8 bits)
- EShort (16 bits)
- EInt (32 bits)
- ELong (64 bits)
- EFloat (32 bits)
- EDouble (64 bits)

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

3.3.2 Annotation and Sofa Type System

3.3.2.1 Overview

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

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

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

3.3.2.2 Annotation and Sofa Reference

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

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

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

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

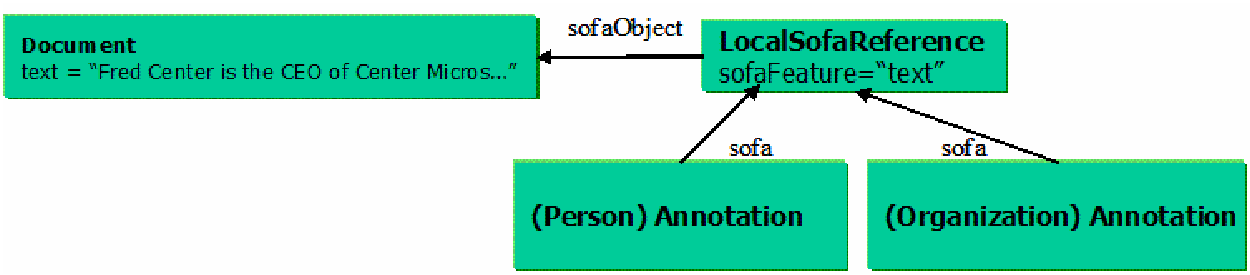

Figure 5: Annotation and Subject of Analysis

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

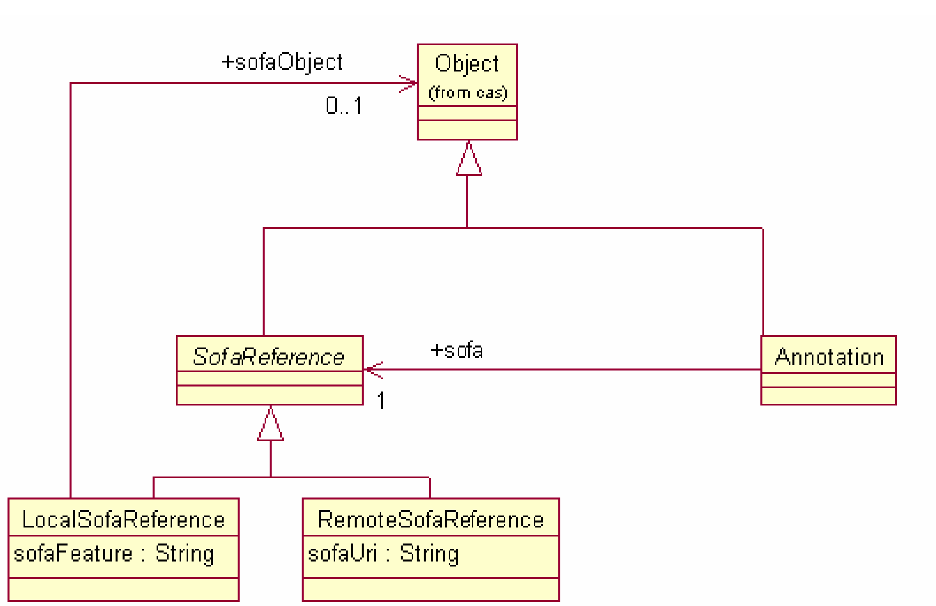

Figure 6: Annotation and Sofa Reference UML

3.3.2.3 References to Regions of Sofas

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

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

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

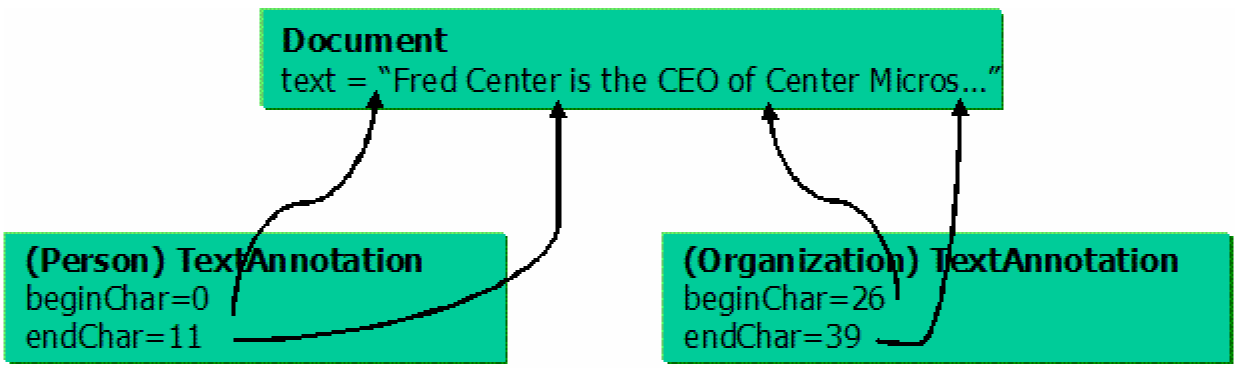

Figure 7: References from Annotations to Regions of the Sofa

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

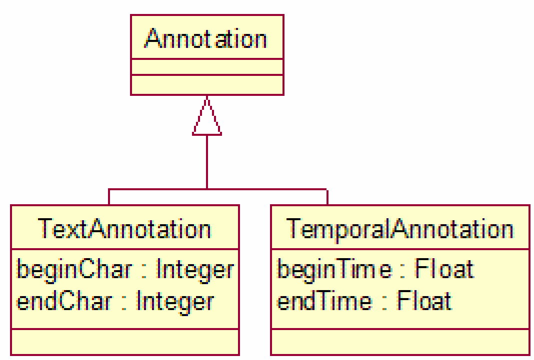

Figure 8: TextAnnotation and TemporalAnnotation UML

In TextAnnotation, beginChar and endChar refer to Unicode character offsets in the corresponding Sofa string. For TemporalAnnotation, beginTime and endTime are offsets measured in seconds from the start of the Sofa. Note that applications that require a different interpretation of these fields must accept the 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.

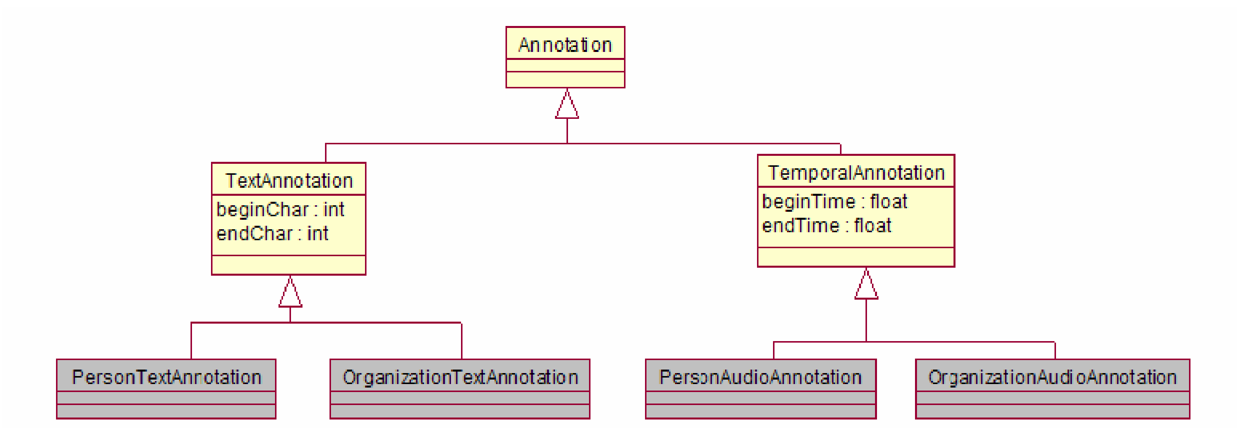

Figure 9: Extending the base type system through subclassing.

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.

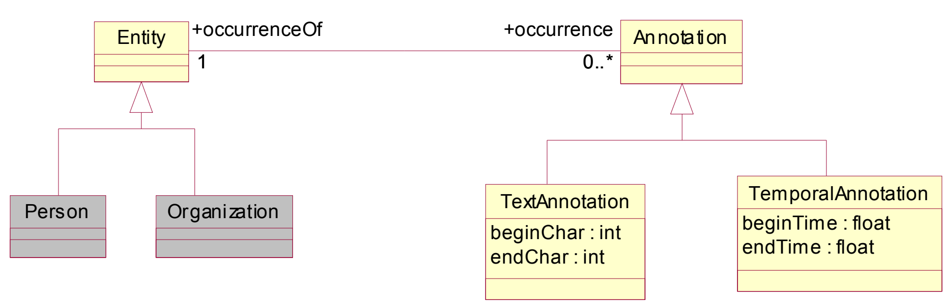

Figure 10: Associate Annotation with Entity type

3.3.2.5 Additional Annotation Metadata

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

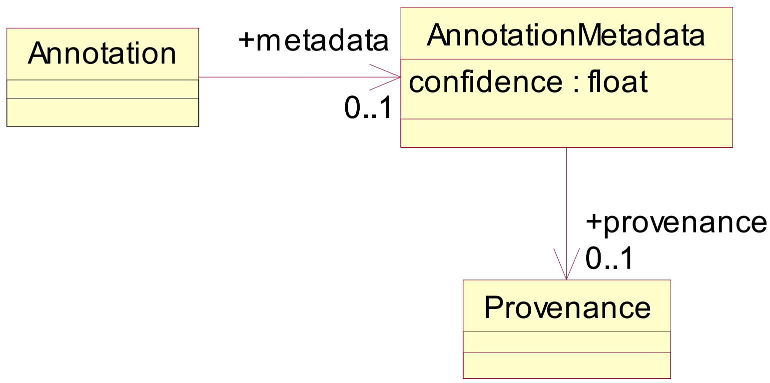

Figure 11: AnnotationMetadata types in the base type system.

3.3.2.6 An Example of Annotation Model Extension (Not Normative)

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

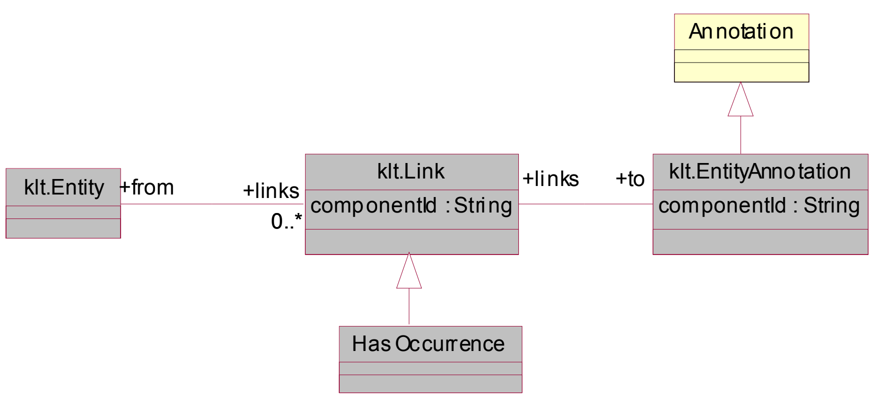

Figure 12: IBM's Knowledge Level Types

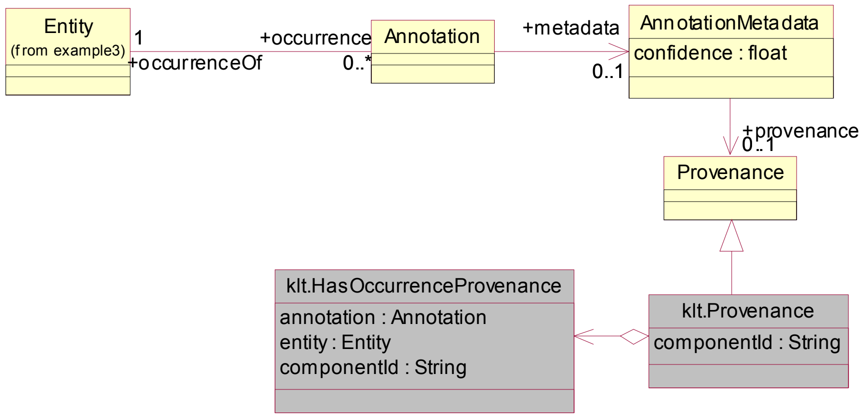

Figure 13: Refactoring of KLT using the standard base type system.

3.3.2.7 Complete Annotation Model

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

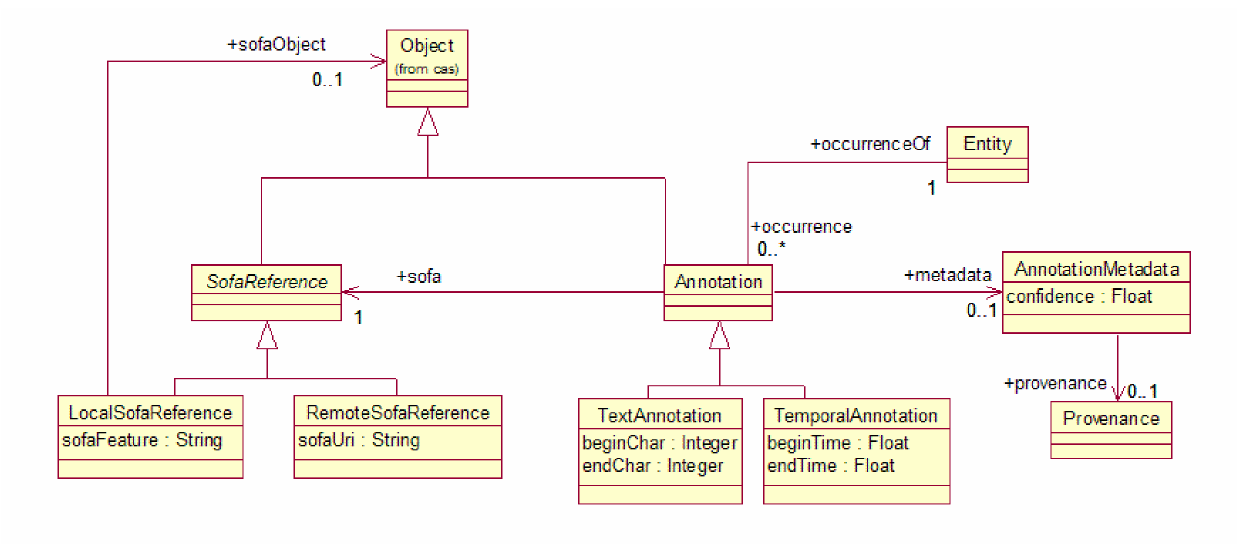

Figure 14: Annotation Model Complete UML

3.3.3 View Type System

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

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

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

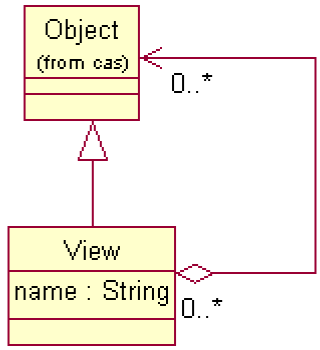

Figure 15: View Type

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

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

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

3.3.3.1 Anchored View

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

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

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

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

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

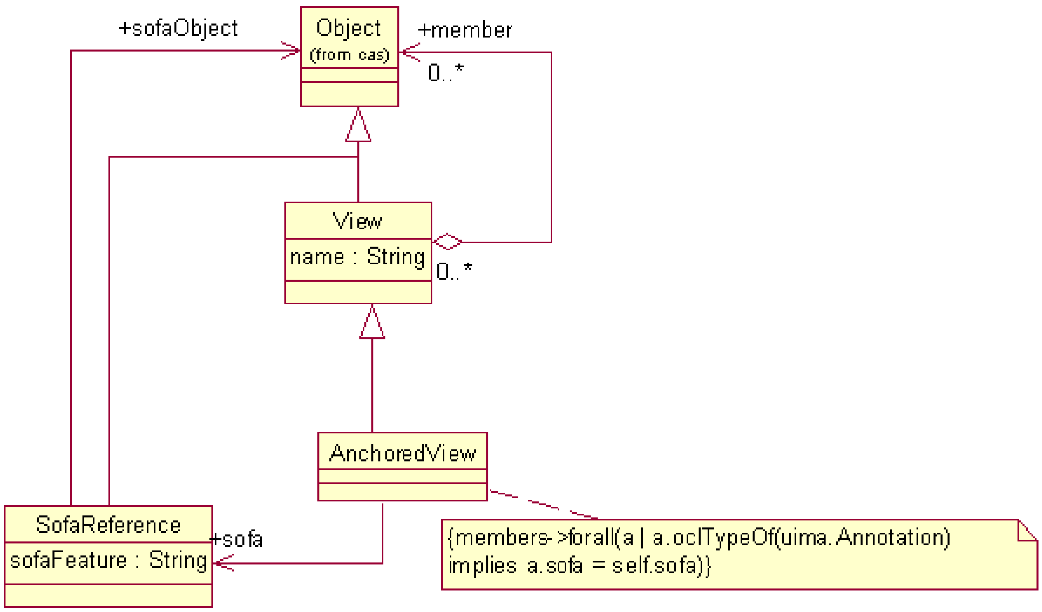

Figure 16: Anchored View Type

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

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

3.3.4 Source Document Information

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

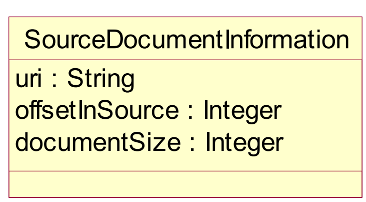

Figure 17: Source Document Information UML

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

3.3.4.1 Example Extension of Source Document Information (Not Normative)

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

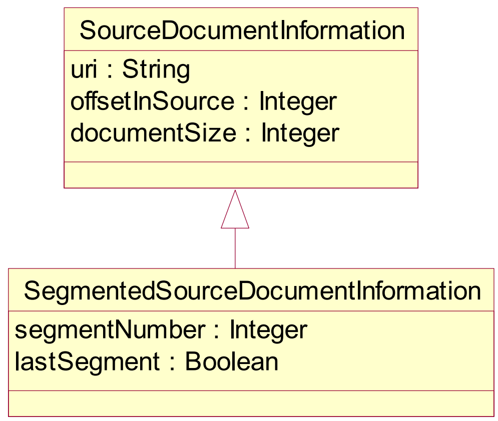

Figure 18: Segmented Source Document Information UML

3.3.5 Formal Specification

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

3.4 Abstract Interfaces

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

3.4.1 Processing Element

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

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

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

3.4.2 Analytic

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

3.4.3 Analyzer

The Analyzer interface defines two additional operations:

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

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

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

3.4.4 CAS Multiplier

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

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

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

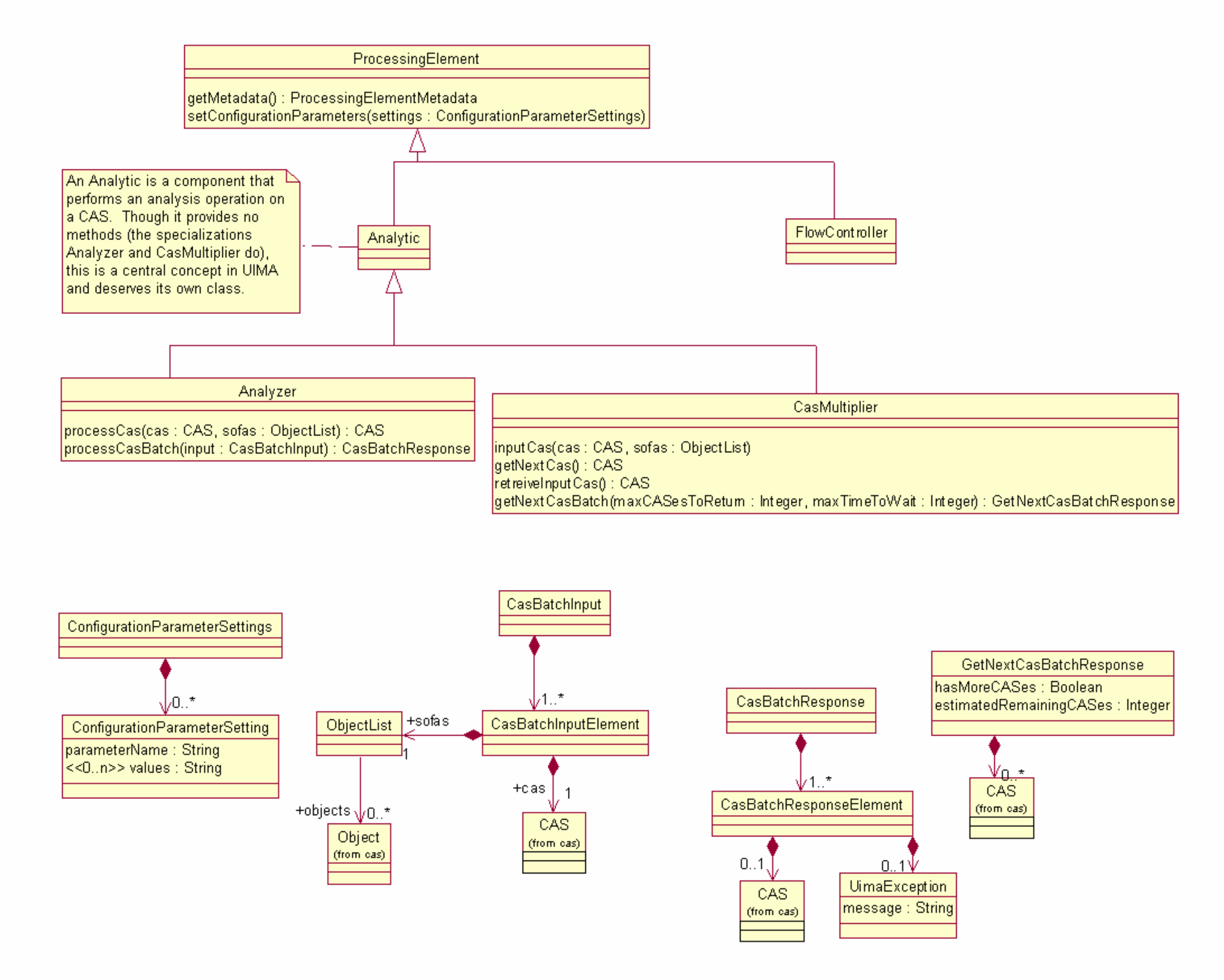

Figure 19: Abstract Interfaces UML (Flow Controller Detail Omitted)

3.4.5 Flow Controller

The FlowController interface defines the operations:

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

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

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

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

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

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

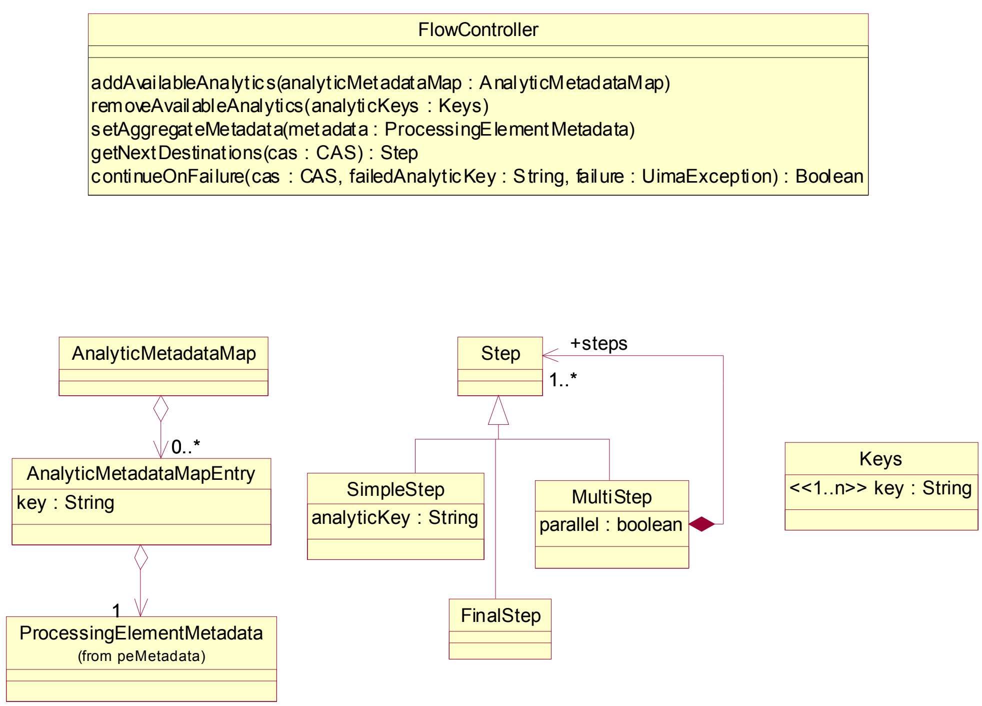

Figure 20: Flow Controller Abstract Interface UML

3.4.6 Examples (Not Normative)

3.4.6.1 Analyzer Example

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

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

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

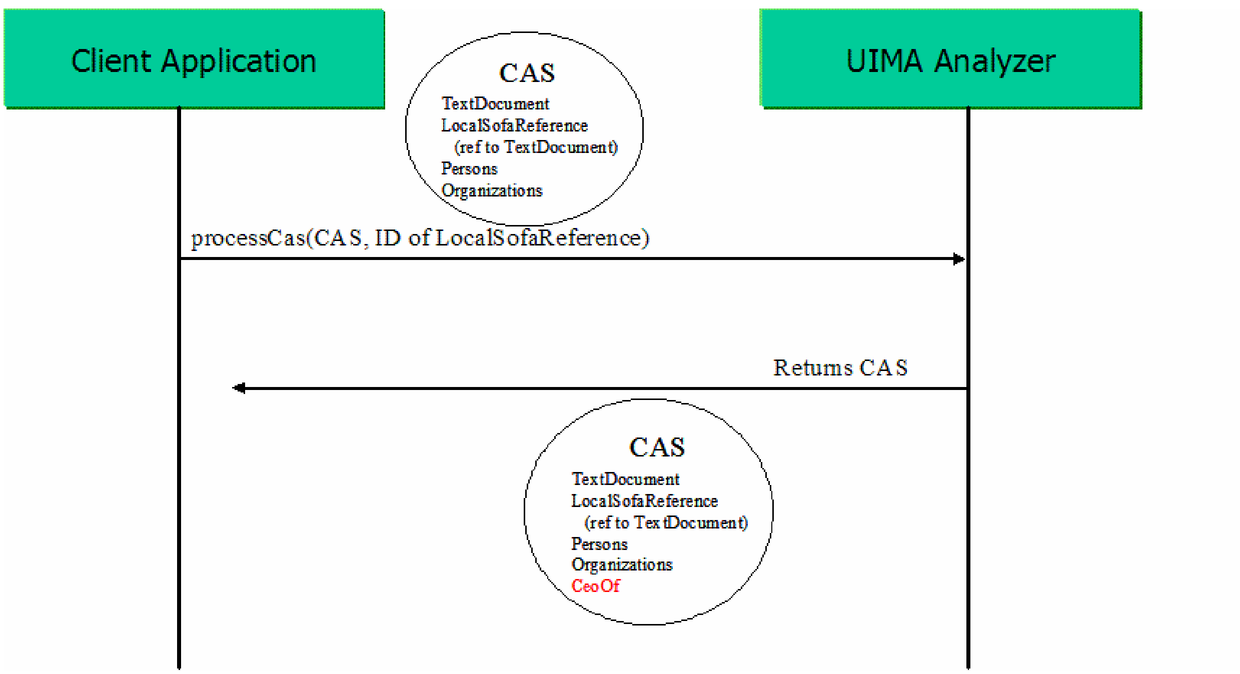

Figure 21: Analyzer Sequence Diagram

3.4.6.2 CAS Multiplier Example

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

The client first calls the `inputCas(cas, sofas)` operation. The first argument is a CAS containing a reference to the video stream to analyze. Typically a large artifact such as a video stream is represented in the CAS as a reference (using the `RemoteSofaReference` base type introduced in section 3.3.2.2), rather than included directly in the CAS as is typically done with a text document. The second argument to `inputCas` is the xmi:id of the `RemoteSofaReference` object, so that the service knows that this is the subject of analysis for this operation.

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

Finally, the client calls the `retrieveInputCas` operation. This returns the original CAS, with additional information added. In this example, the Video Segmenter adds information to the original CAS indicating at what time offsets each of the segment boundaries were detected.

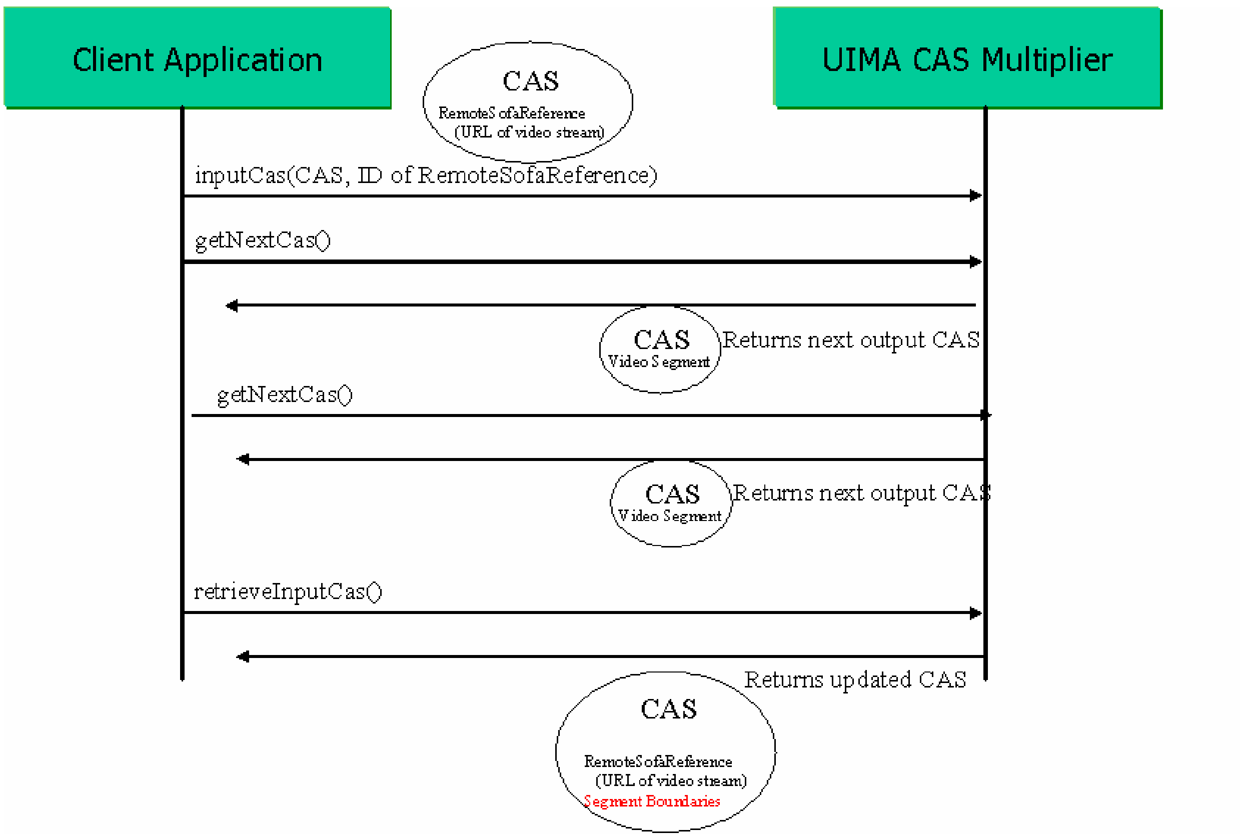

Figure 22: CAS Multiplier Sequence Diagram

3.4.7 Formal Specification

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

3.4.7.1 ProcessingElement.getMetaData

A UIMA Processing Element (PE) Service MUST implement an operation named `getMetaData`. This operation MUST take zero argument and MUST return PE Metadata XML as defined in Section 3.6.4. In the following sections, we use the term “this PE Service’s Metadata” to refer to the PE Metadata returned by this operation.

3.4.7.2 ProcessingElement.setConfigurationParameters

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

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

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

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

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

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

3.4.7.3 Analyzer.processCas

A UIMA Analyzer Service MUST implement an operation named `processCas`. This operation MUST accept two arguments. The first argument is a CAS, represented in XMI as defined in Section 3.1.7. The second argument is a list of xmi:ids that identify `SofaReference` objects which the Analyzer is expected to analyze. This operation MUST return a valid XMI document which is either a valid CAS (as defined in Section3.1.7) or a description of changes to be applied to the input CAS using the XMI differences language defined in [XMI1].

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

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

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

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

3.4.7.4 Analyzer.processCasBatch

A UIMA Analyzer Service MUST implement an operation named `processCasBatch`. This operation MUST accept an argument which consists of one or more CASes, each with an associated list of xmi:ids that identify `SofaReference` objects in that CAS. This operation MUST return a response that consists of multiple elements, one for each input CAS, where each element is either valid XMI document which is either a valid CAS (as defined in Section3.1.7), a description of changes to be applied to the input CAS using the XMI differences language defined in [XMI1], or an exception message.

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

3.4.7.5 CasMultiplier.inputCas

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

The CAS that is passed to this operation becomes this CAS Multipliers *active CAS*.

3.4.7.6 CasMultiplier.getNextCas

A UIMA CAS Multiplier service MUST implement an operation named `getNextCas`. This operation MUST take zero arguments. This operation MUST return a valid CAS as defined in Section3.1.7, or a result indicating that there are no more CASes available.

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

3.4.7.7 CasMultiplier.retrieveInputCas

A UIMA CAS Multiplier service MUST implement an operation named `retrieveInputCas`. This operation MUST take zero arguments and must return a valid XMI document which is either a valid CAS (as defined in Section3.1.7) or a description of changes to be applied to the CAS Multiplier's active CAS using the XMI differences language defined in [XMI1].

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

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

3.4.7.8 CasMultiplier.getNextCasBatch

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

1. Zero or more valid CASes as defined in Section3.1.7. The number of CASes MUST NOT exceed the value of the `maxCASesToReturn` argument.
2. a Boolean indicating whether more CAS remain to be retrieved.
3. An estimated number of remaining CASes. The estimated number of remaining CASes may be set to -1 to indicate an unknown number.

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

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

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

3.4.7.9 FlowController.addAvailableAnalytics

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

3.4.7.10 FlowController.removeAvailableAnalytics

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

3.4.7.11 FlowController.setAggregateMetadata

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

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

3.4.7.12 FlowController.getNextDestinations

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

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

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

3.4.7.13 FlowController.continueOnFailure

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

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

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

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

3.5 Behavioral Metadata

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

3.5.1 Goals

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

2. ***Composition:*** Support composition either by a human developer or an automated process.
    a. Analytics should be able to declare what they do in enough detail to assist manual and/or automated processes in considering their role in an application or in the composition of aggregate analytics.
    b. Through their Behavioral Metadata, Analytics should be able to declare enough detail as to enable an application or aggregate to detect “invalid” compositions/workflows (e.g., a workflow where it can be determined that one of the Analytic’s preconditions can never be satisfied by the preceding Analytic).

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

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

3.5.2 Elements of Behavioral Metadata

Behavioral Metadata breaks down into the following categories:

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

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

3.5.3 Example (Not Normative)

Consider a “CeoOf Relation Detector” analytic that receives as input a text document in which Persons and Organizations have been annotated, and looks for a relationship that a Person is the CEO Of an 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">

    <analyzes>
      <type name="org.example:TextDocument"/>
    </analyzes>

    <requiredInputs>
      <type name="org.example:Person"/>
      <type name="org.example:Organization"/>
    </requiredInputs>

    <creates>
      <type name="org.example:CeoOf"/>
    </creates>

</behavioralMetadata>
```

Type of Sofa that the Analytic will process

Inputs – may be required or optional

Effects – objects that the analytic creates, modifies, or deletes

Figure 23: Behavioral Metadata Example

This satisfies the three design goals of Behavioral Metadata:

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

3.5.4 Using Views in Behavioral Metadata

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

```
<behavioralMetadata xmlns:org.example="http://docs.oasis-
open.org/uima/org/example.ecore">
  <requiredView sofaType="org.example:Document">
    <requiredInputs>
      <type name="org.example:Person"/>
      <type name="org.example:Organization"/>
    </requiredInputs>
    <creates>
      <type name="org.example:CeoOf"/>
    </creates>
  </requiredView>
 </behavioralMetadata>
```

3.5.5 Formal Semantics for Behavioral Metadata

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

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

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

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

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

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

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

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

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

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

```
<requiredView sofaType="org.example:TextDocument">
  <requiredInputs>
    <type> org.example:Token</type>
  </requiredInputs>
</requiredView>
```

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

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

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

3.5.6 Behavioral Metadata UML

The following UML diagram defines the UIMA Behavioral Metadata representation:

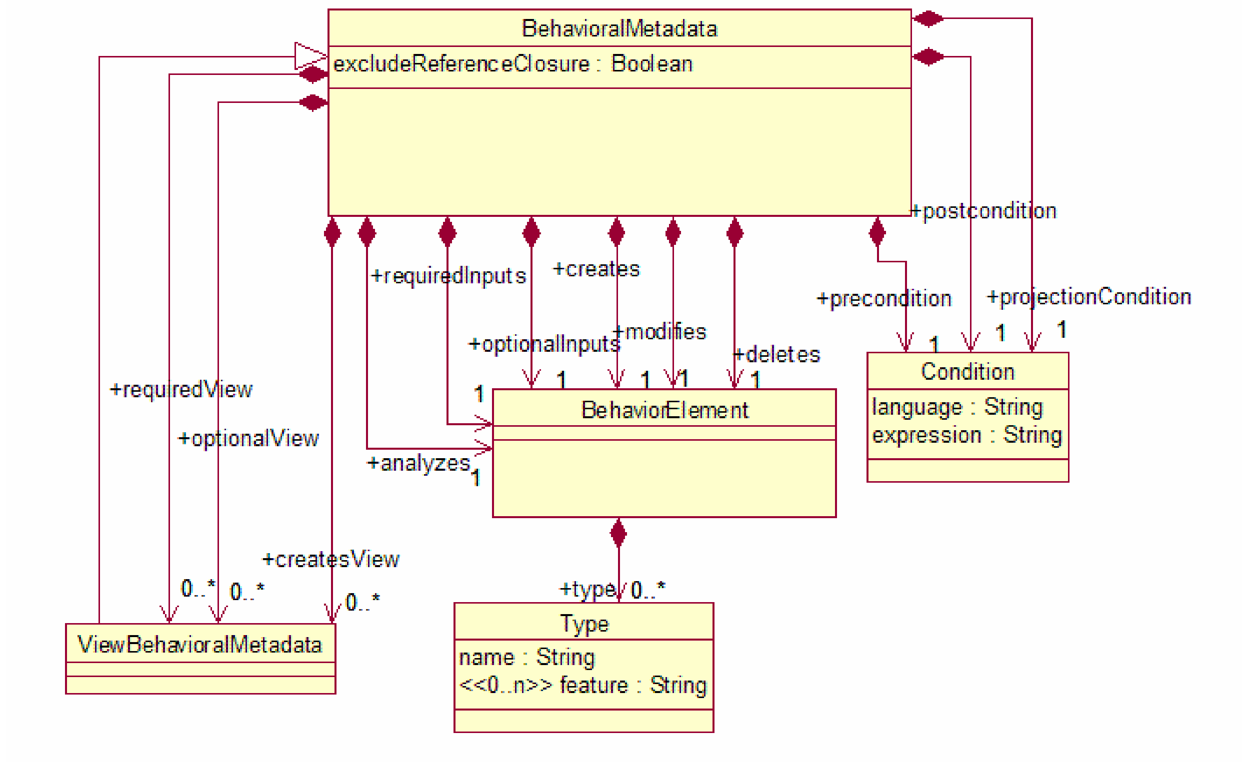

Figure 24: Behavioral Metadata UML

3.5.7 Behavioral Metadata XML Representation

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

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

3.5.7.1 Type Naming Conventions

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

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

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

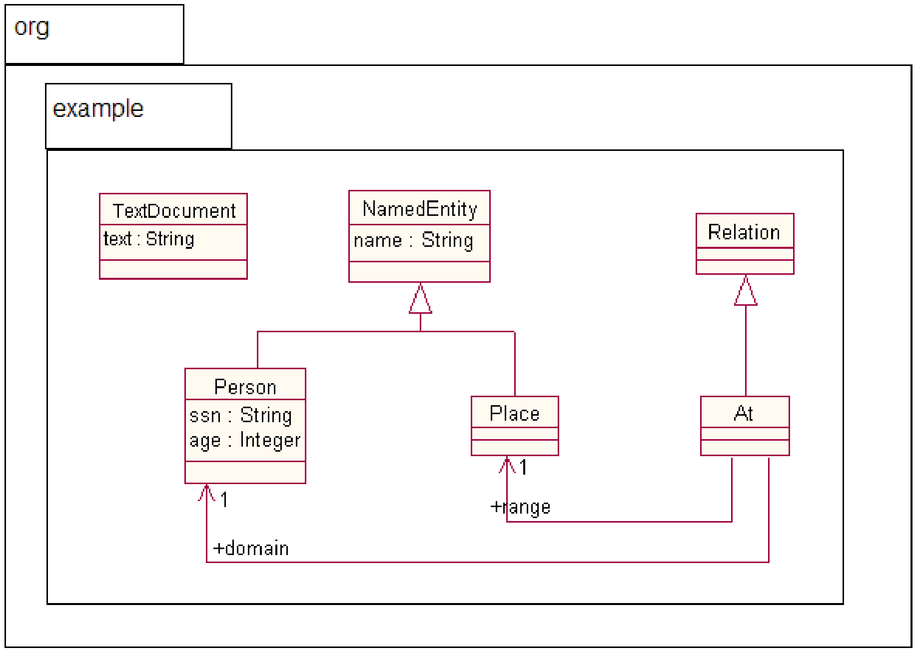

Figure 25: Example Type System UML Model

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

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

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

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

Figure 26: Partial Ecore Representation of Example Type System

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

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

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

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

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

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

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

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

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

3.5.7.2 XML Syntax for Behavioral Metadata Elements

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

```
  <behavioralMetadata xmlns:org.example="http://docs.oasis-
open.org/uima/org/example.ecore" excludeReferenceClosure="true">
    <analyzes>
      <type name="org.example:TextDocument"/>
    </analyzes>
    <requiredInputs>
      <type name="org.example:Person"/>
      <type name="org.example:Place"/>
    </requiredInputs>
    <creates>
      <type name="org.example:At"/>
    </creates>
  </behavioralMetadata>
```

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

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

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

3.5.7.3 Views

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

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

```
      <type name="org.example:Token"/>
    </requiredInputs>
    <creates>
      <type name="org.example:Person"/>
    </creates>
  </requiredView>
  <optionalView sofaType="org.example:RawAudio">
    <requiredInputs>
      <type name="org.example:SpeakerBoundary"/>
    </requiredInputs>
    <creates>
      <type name="org.example:AudioPerson"/>
    </creates>
  </optionalView>
</behavioralMetadata>
```

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

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

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

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

```
<behavioralMetadata xmlns:org.example="http://docs.oasis-
open.org/uima/org/example.ecore">
  <requiredView sofaType="org.example:RawAudio">
    <requiredInputs>
      <type name="org.example:SpeakerBoundary"/>
    </requiredInputs>
  </requiredView>
  <createsView sofaType="org.example:TextDocument">
    <creates>
      <type name="org.example:Person"/>
    </creates>
  </createsView>
</behavioralMetadata>
```

3.5.7.4 Specifying Which Features Are Modified

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

```
<behavioralMetadata xmlns:org.example="http://docs.oasis-
open.org/uima/org/example.ecore">
  <requiredInputs>
    <type name="org.example:Person"/>
  </requiredInputs>
  <modifies>
    <type name="org.example:Person">
      <feature name="age"/>
      <feature name="ssn"/>
    </type>
  </modifies>
</behavioralMetadata>
```

3.5.7.5 Specifying Preconditions, Postconditions, and Projection Conditions

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

```
  <behavioralMetadata>
    <precondition language="OCL"
       expression="exists(s | s.oclKindOf(org::example::Sofa) and
s.mimeTypeMajor = 'audio')"/>
    <postcondition language="OCL"
       expr="exists(p | p.oclKindOf(org::example::Sofa) and s.mimeTypeMajor =
'text')"/>
    <projectionCondition language="OCL"
      expr=" select(p | p.oclKindOf(org::example::NamedEntity))"/>
  </behavioralMetadata>
```

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

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

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

3.5.8 Formal Specification

3.5.8.1 Structure

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

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

3.5.8.2 Constraints

Field values must satisfy the following constraints

3.5.8.2.1 Type

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

3.5.8.2.2 Condition

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

3.5.8.3 Semantics

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

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

3.5.8.3.1 Mapping to OCL Precondition

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

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

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

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

For each `requiredView` element that contains `analyzes` or `requiredInputs` elements with types T1, T2, …, Tn, produce the OCL expression:

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

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

In the above we define ViewExpression as follows:

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

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

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

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

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

3.5.8.3.2 Mapping to OCL Postcondition

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

For types T1, T2, … Tn specified in `creates` elements, produce the OCL expression:

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

For types T1, T2, … Tn specified in `deletes` elements, produce the OCL expression:

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

For each `modifies` element specifying type T with features F={F1, F2, …Fn}, for each feature g defined on type T where g∉F, produce the OCL expression:

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

For each `createsView`, `requiredView` or `optionalView` containing `creates` elements with types T1,T2,…,Tn, produce the OCL expression:

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

where ViewExpression is as defined in Section 3.5.8.3.1.

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

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

where ViewExpression is as defined in Section 3.5.8.3.1.

Within each `requiredView` or `optionalView`, for each `modifies` element specifying type T with features F={F1, F2, …Fn}, for each feature g defined on type T where $g \notin F$, produce the OCL expression:

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

where ViewExpression is as defined in Section 3.5.8.3.1.

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

3.5.8.3.3 Mapping to OCL Projection Condition

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

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

For types T1, T2, … Tn specified in `analyzes`, `requiredInputs`, or `optionalInputs` elements, produce the OCL expression:

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

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

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

where ViewExpression is as defined in Section 3.5.8.3.1.

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

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

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

3.6 Processing Element Metadata

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

3.6.1 Overview

The PE Metadata is subdivided into the following parts:

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

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

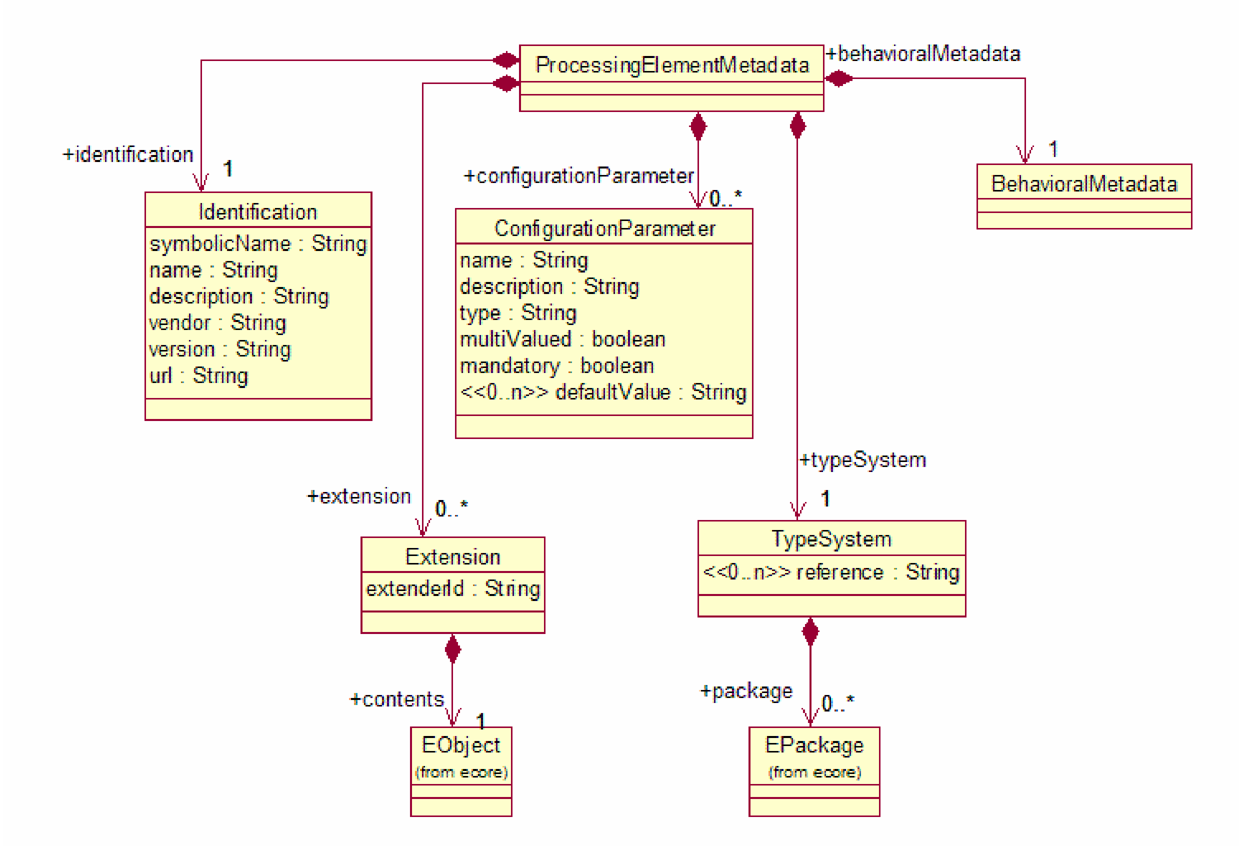

Figure 27: Processing Element Metadata UML Model

3.6.2 Elements of PE Metadata

3.6.2.1 Identification Information

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

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

The following properties are included:

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

3.6.2.2 Configuration Parameters

Many kinds of PEs may be configured to operate in different ways[3]. 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.

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

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

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

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

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

- String
- Integer (32-bit)
- Float (32-bit)
- Boolean
- ResourceURL

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

Note that parameters may take multiple values so it is not necessary to have explicit parameter types 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.

3.6.2.3 Type System

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

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

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

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

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

3.6.2.4 Behavioral Metadata

The Behavioral Metadata is discussed in detail in 3.5.

3.6.2.5 Extensions

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

---

[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.

This extensibility is enabled by the Extension class in Figure 27. The Extension class defines two *features*, extenderId and contents.

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

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

3.6.3 Example (Not Normative)

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

```
<pemd:ProcessingElementMetadata xmi:version="2.0"
xmlns:xmi="http://www.omg.org/XMI" xmlns:pemd="http://docs.oasis-
open.org/uima/pemetadata.ecore">
  <identification
        symbolicName="org.oasis-open.uima.example.CeoRelationAnnotator"
        name="Ceo Relation Annotator"
        description="Detects CeoOf relationships between Persons and
Organizations in a text document."
        vendor="OASIS"
        version="1.0.0"/>

  <configurationParameter
        name="PatternFile"
        description="Location of external file containing patterns that
indicate a CeoOf relation in text."
        type="ResourceURL">
            <defaultValue>myResources/ceoPatterns.dat</defaultValue>
  </configurationParameter>

  <typeSystem
        reference="http://docs.oasis-
open.org/uima/types/exampleTypeSystem.ecore"/>

  <behavioralMetadata>
    <analyzes>
      <type name="org.example:Document"/>
    </analyzes>
    <requiredInputs>
      <type name="org.example:Person"/>
      <type name="org.example:Organization"/>
    </requiredInputs>
    <creates>
      <type name="org.example:CeoOf"/>
    </creates>
  </behavioralMetadata>

  <extension extenderId="org.apache.uima">
        ...
  </extension>
</pemd:ProcessingElementMetadata>
```

3.6.4 Formal Specification

3.6.4.1 Structure

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

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

3.6.4.2 Constraints

Field values must satisfy the following constraints

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].

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}

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.

Extensions

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

3.7 Service WSDL Descriptions

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

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

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

3.7.1 Overview of the WSDL Definition

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

> **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.

3.7.1.1 Types

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

3.7.1.2 Messages

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

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

The messages defined by the UIMA WSDL service definition are:

For ALL PEs:

- getMetadataRequest – takes no arguments
- getMetadataResponse – returns ProcessingElementMetadata
- setConfigurationParametersRequest – takes one argument: ConfigurationParameterSettings
- setConfigurationParameterResponse – returns nothing

For Analyzers:

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

For CAS Multipliers:

- inputCasRequest – takes two arguments – a CAS and a list of Sofas (object IDs) to process
- inputCasResponse – returns nothing
- getNextCasRequest – takes no arguments
- getNextCasResponse – returns a CAS
- retrieveInputCasRequest – takes no arguments
- retrieveInputCasResponse – returns a CAS
- getNextCasBatchRequest – takes two arguments, an integer that specifies the maximum number of CASes to return and an integer which specifies the maximum number of milliseconds to wait
- getNextCasBatchResponse – returns an object with three fields: a list of zero or more CASes, a Boolean indicating whether any CASes remain to be retrieved, and an integer indicating the estimated number of remaining CASes (-1 if not known).

For Flow Controllers:

- addAvailableAnalyticsRequest – takes one argument, a Map from String keys to PE Metadata objects.
- addAvailableAnalyticsResponse – returns nothing
- removeAvailableAnalyticsRequest – takes one argument, a collection of one or more String keys
- removeAvailableAnalyticsResponse – returns nothing
- setAggregateMetadataRequest – takes one argument – a ProcessingElementMetadata
- setAggregateMetadataResponse – returns nothing
- getNextDestinationsRequest – takes one argument, a CAS
- getNextDestiontionsResponse – returns a Step object
- continueOnFailureRequest – takes three arguments, a CAS, a String key, and a UimaException
- continueOnFailureResponse – returns a Boolean

3.7.1.3 Port Types and Operations

A *port type* is a collection of *operations*, where each operation is an action that can be performed by the service. We define a separate port type for each of the three interfaces defined in Section 3.4 Abstract Interfaces.

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

- **Analyzer Port Type**
  - getMetadata
  - setConfigurationParameters
  - processCas
  - processCasBatch

- **CasMultiplier Port Type**
  - getMetadata
  - setConfigurationParameters
  - inputCas
  - getNextCas
  - retrieveInputCas
  - getNextCasBatch

FlowController Port Type

- getMetadata
- setConfigurationPsrameters
- addAvailableAnalytics
- removeAvailableAnalytics
- setAggregateMetadata
- getNextDestinations
- continueOnFailure

3.7.1.4 SOAP Bindings

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

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

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

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

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

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.

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.

The processCas request message is shown here:

```
<soapenv:Envelope...>
  <soapenv:Body>
    <processCas xmlns="">
      <cas xmi:version="2.0" ... >
        <org.example:Document xmi:id="1"
          text="Fred Center is the CEO of Center Micros."/>
        <cas:LocalSofaReference xmi:id="2" sofaObject="1" sofaFeature="text"/>
        <org.example:Person xmi:id="3" sofa="2" begin="0" end="11"/>
        <org.example:Organization xmi:id="4" sofa="2" begin="26" end="39"/>
       </cas>
      <sofas>1</sofas>
    </processCas>
  </soapenv:Body>
</soapenv:Envelope>
```

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

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

```
<soapenv:Envelope...>
  <soapenv:Body>
    <processCas xmlns="">
      <cas xmi:version="2.0" ... >
        <org.example:Document xmi:id="1"
          text="Fred Center is the CEO of Center Micros."/>
        <cas:SofaReference xmi:id="2" sofaObject="1" sofaFeature="text"/>
        <org.example:Person xmi:id="3" sofa="2" begin="0" end="11"/>
        <org.example:Organization xmi:id="4" sofa="2" begin="26" end="39"/>
        <org.example:CeoOf xmi:id="5" sofa="2" begin="0" end="31" arg0="3"
arg1="4"/>
      </cas>
    </processCas>
  </soapenv:Body>
</soapenv:Envelope>
```

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

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

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

```
    <processCas xmlns="">
      <cas xmi:version="2.0" ... >
        <xmi:Difference>
          <target href="input.xmi"/>
          <xmi:Add addition="5">
        </xmi:Difference>
          <org.example:CeoOf xmi:id="5" sofa="2" begin="0" end="31" arg0="3"
arg1="4"/>
       </cas>
    </processCas>
  </soapenv:Body>
</soapenv:Envelope>
```

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

3.7.4 Formal Specification

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

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

A *UIMA CAS Multiplier SOAP Serv*ice must implement the CasMultiplier portType and the CasMultiplierSoapBinding.

A. Acknowledgements

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

Participants:

[Participant Name, Affiliation | Individual Member]
[Participant Name, Affiliation | Individual Member]

B. Formal Specification Artifacts

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

B.1 XMI XML Schema

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

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xmi="http://www.omg.org/XMI"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.omg.org/XMI">
  <xsd:attribute name="id" type="xsd:ID"/>
  <xsd:attributeGroup name="IdentityAttribs">
    <xsd:attribute form="qualified" name="label" type="xsd:string"
      use="optional"/>
    <xsd:attribute form="qualified" name="uuid" type="xsd:string"
      use="optional"/>
  </xsd:attributeGroup>
  <xsd:attributeGroup name="LinkAttribs">
    <xsd:attribute name="href" type="xsd:string" use="optional"/>
    <xsd:attribute form="qualified" name="idref" type="xsd:IDREF"
      use="optional"/>
  </xsd:attributeGroup>
  <xsd:attributeGroup name="ObjectAttribs">
    <xsd:attributeGroup ref="xmi:IdentityAttribs"/>
    <xsd:attributeGroup ref="xmi:LinkAttribs"/>
    <xsd:attribute fixed="2.0" form="qualified" name="version"
      type="xsd:string" use="optional"/>
    <xsd:attribute form="qualified" name="type" type="xsd:QName"
      use="optional"/>
  </xsd:attributeGroup>
  <xsd:complexType name="XMI">
    <xsd:choice maxOccurs="unbounded" minOccurs="0">
      <xsd:any processContents="strict"/>
    </xsd:choice>
    <xsd:attributeGroup ref="xmi:IdentityAttribs"/>
    <xsd:attributeGroup ref="xmi:LinkAttribs"/>
    <xsd:attribute form="qualified" name="type" type="xsd:QName"
      use="optional"/>
    <xsd:attribute fixed="2.0" form="qualified" name="version"
      type="xsd:string" use="required"/>
```

```
</xsd:complexType>
<xsd:element name="XMI" type="xmi:XMI"/>
<xsd:complexType name="PackageReference">
  <xsd:choice maxOccurs="unbounded" minOccurs="0">
    <xsd:element name="name" type="xsd:string"/>
    <xsd:element name="version" type="xsd:string"/>
  </xsd:choice>
  <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
  <xsd:attribute name="name" type="xsd:string" use="optional"/>
</xsd:complexType>
<xsd:element name="PackageReference"
  type="xmi:PackageReference"/>
<xsd:complexType name="Model">
  <xsd:complexContent>
    <xsd:extension base="xmi:PackageReference"/>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="Model" type="xmi:Model"/>
<xsd:complexType name="Import">
  <xsd:complexContent>
    <xsd:extension base="xmi:PackageReference"/>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="Import" type="xmi:Import"/>
<xsd:complexType name="MetaModel">
  <xsd:complexContent>
    <xsd:extension base="xmi:PackageReference"/>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="MetaModel" type="xmi:MetaModel"/>
<xsd:complexType name="Documentation">
  <xsd:choice maxOccurs="unbounded" minOccurs="0">
    <xsd:element name="contact" type="xsd:string"/>
    <xsd:element name="exporter" type="xsd:string"/>
    <xsd:element name="exporterVersion" type="xsd:string"/>
    <xsd:element name="longDescription" type="xsd:string"/>
    <xsd:element name="shortDescription" type="xsd:string"/>
    <xsd:element name="notice" type="xsd:string"/>
    <xsd:element name="owner" type="xsd:string"/>
  </xsd:choice>
  <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
  <xsd:attribute name="contact" type="xsd:string" use="optional"/>
  <xsd:attribute name="exporter" type="xsd:string"
```

```
      use="optional"/>
    <xsd:attribute name="exporterVersion" type="xsd:string"
      use="optional"/>
    <xsd:attribute name="longDescription" type="xsd:string"
      use="optional"/>
    <xsd:attribute name="shortDescription" type="xsd:string"
      use="optional"/>
    <xsd:attribute name="notice" type="xsd:string" use="optional"/>
    <xsd:attribute name="owner" type="xsd:string" use="optional"/>
  </xsd:complexType>
  <xsd:element name="Documentation" type="xmi:Documentation"/>
  <xsd:complexType name="Extension">
    <xsd:choice maxOccurs="unbounded" minOccurs="0">
      <xsd:any processContents="lax"/>
    </xsd:choice>
    <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
    <xsd:attribute name="extender" type="xsd:string"
      use="optional"/>
    <xsd:attribute name="extenderID" type="xsd:string"
      use="optional"/>
  </xsd:complexType>
  <xsd:element name="Extension" type="xmi:Extension"/>
  <xsd:complexType name="Difference">
    <xsd:choice maxOccurs="unbounded" minOccurs="0">
      <xsd:element name="target">
        <xsd:complexType>
          <xsd:choice maxOccurs="unbounded" minOccurs="0">
            <xsd:any processContents="skip"/>
          </xsd:choice>
          <xsd:anyAttribute processContents="skip"/>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="difference" type="xmi:Difference"/>
      <xsd:element name="container" type="xmi:Difference"/>
    </xsd:choice>
    <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
    <xsd:attribute name="target" type="xsd:IDREFS" use="optional"/>
    <xsd:attribute name="container" type="xsd:IDREFS"
      use="optional"/>
  </xsd:complexType>
  <xsd:element name="Difference" type="xmi:Difference"/>
  <xsd:complexType name="Add">
    <xsd:complexContent>
```

```
        <xsd:extension base="xmi:Difference">
          <xsd:attribute name="position" type="xsd:string"
            use="optional"/>
          <xsd:attribute name="addition" type="xsd:IDREFS"
            use="optional"/>
        </xsd:extension>
      </xsd:complexContent>
    </xsd:complexType>
    <xsd:element name="Add" type="xmi:Add"/>
    <xsd:complexType name="Replace">
      <xsd:complexContent>
        <xsd:extension base="xmi:Difference">
          <xsd:attribute name="position" type="xsd:string"
            use="optional"/>
          <xsd:attribute name="replacement" type="xsd:IDREFS"
            use="optional"/>
        </xsd:extension>
      </xsd:complexContent>
    </xsd:complexType>
    <xsd:element name="Replace" type="xmi:Replace"/>
    <xsd:complexType name="Delete">
      <xsd:complexContent>
        <xsd:extension base="xmi:Difference"/>
      </xsd:complexContent>
    </xsd:complexType>
    <xsd:element name="Delete" type="xmi:Delete"/>
    <xsd:complexType name="Any">
      <xsd:choice maxOccurs="unbounded" minOccurs="0">
        <xsd:any processContents="skip"/>
      </xsd:choice>
      <xsd:anyAttribute processContents="skip"/>
    </xsd:complexType>
  </xsd:schema>
```

B.2 Ecore XML Schema

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

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

```
      <xsd:extension base="ecore:EStructuralFeature">
        <xsd:attribute name="iD" type="xsd:boolean"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:element name="EAttribute" type="ecore:EAttribute"/>
  <xsd:complexType name="EAnnotation">
    <xsd:complexContent>
      <xsd:extension base="ecore:EModelElement">
        <xsd:choice maxOccurs="unbounded" minOccurs="0">
          <xsd:element name="details" type="ecore:EStringToStringMapEntry"/>
          <xsd:element name="contents" type="ecore:EObject"/>
          <xsd:element name="references" type="ecore:EObject"/>
        </xsd:choice>
        <xsd:attribute name="source" type="xsd:string"/>
        <xsd:attribute name="references" type="xsd:string"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:element name="EAnnotation" type="ecore:EAnnotation"/>
  <xsd:complexType name="EClass">
    <xsd:complexContent>
      <xsd:extension base="ecore:EClassifier">
        <xsd:choice maxOccurs="unbounded" minOccurs="0">
          <xsd:element name="eSuperTypes" type="ecore:EClass"/>
          <xsd:element name="eOperations" type="ecore:EOperation"/>
          <xsd:element name="eStructuralFeatures"
type="ecore:EStructuralFeature"/>
        </xsd:choice>
        <xsd:attribute name="abstract" type="xsd:boolean"/>
        <xsd:attribute name="interface" type="xsd:boolean"/>
        <xsd:attribute name="eSuperTypes" type="xsd:string"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:element name="EClass" type="ecore:EClass"/>
  <xsd:complexType abstract="true" name="EClassifier">
    <xsd:complexContent>
      <xsd:extension base="ecore:ENamedElement">
        <xsd:attribute name="instanceClassName" type="xsd:string"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
```

```
    <xsd:element name="EClassifier" type="ecore:EClassifier"/>
    <xsd:complexType name="EDataType">
      <xsd:complexContent>
        <xsd:extension base="ecore:EClassifier">
          <xsd:attribute name="serializable" type="xsd:boolean"/>
        </xsd:extension>
      </xsd:complexContent>
    </xsd:complexType>
    <xsd:element name="EDataType" type="ecore:EDataType"/>
    <xsd:complexType name="EEnum">
      <xsd:complexContent>
        <xsd:extension base="ecore:EDataType">
          <xsd:choice maxOccurs="unbounded" minOccurs="0">
            <xsd:element name="eLiterals" type="ecore:EEnumLiteral"/>
          </xsd:choice>
        </xsd:extension>
      </xsd:complexContent>
    </xsd:complexType>
    <xsd:element name="EEnum" type="ecore:EEnum"/>
    <xsd:complexType name="EEnumLiteral">
      <xsd:complexContent>
        <xsd:extension base="ecore:ENamedElement">
          <xsd:attribute name="value" type="xsd:int"/>
          <xsd:attribute name="literal" type="xsd:string"/>
        </xsd:extension>
      </xsd:complexContent>
    </xsd:complexType>
    <xsd:element name="EEnumLiteral" type="ecore:EEnumLiteral"/>
    <xsd:complexType name="EFactory">
      <xsd:complexContent>
        <xsd:extension base="ecore:EModelElement"/>
      </xsd:complexContent>
    </xsd:complexType>
    <xsd:element name="EFactory" type="ecore:EFactory"/>
    <xsd:complexType abstract="true" name="EModelElement">
      <xsd:complexContent>
        <xsd:extension base="ecore:EObject">
          <xsd:choice maxOccurs="unbounded" minOccurs="0">
            <xsd:element name="eAnnotations" type="ecore:EAnnotation"/>
          </xsd:choice>
        </xsd:extension>
      </xsd:complexContent>
    </xsd:complexType>
```

```
<xsd:element name="EModelElement" type="ecore:EModelElement"/>
<xsd:complexType abstract="true" name="ENamedElement">
  <xsd:complexContent>
    <xsd:extension base="ecore:EModelElement">
      <xsd:attribute name="name" type="xsd:string"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="ENamedElement" type="ecore:ENamedElement"/>
<xsd:complexType name="EObject">
  <xsd:choice maxOccurs="unbounded" minOccurs="0">
    <xsd:element ref="xmi:Extension"/>
  </xsd:choice>
  <xsd:attribute ref="xmi:id"/>
  <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
</xsd:complexType>
<xsd:element name="EObject" type="ecore:EObject"/>
<xsd:complexType name="EOperation">
  <xsd:complexContent>
    <xsd:extension base="ecore:ETypedElement">
      <xsd:choice maxOccurs="unbounded" minOccurs="0">
        <xsd:element name="eParameters" type="ecore:EParameter"/>
        <xsd:element name="eExceptions" type="ecore:EClassifier"/>
      </xsd:choice>
      <xsd:attribute name="eExceptions" type="xsd:string"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="EOperation" type="ecore:EOperation"/>
<xsd:complexType name="EPackage">
  <xsd:complexContent>
    <xsd:extension base="ecore:ENamedElement">
      <xsd:choice maxOccurs="unbounded" minOccurs="0">
        <xsd:element name="eClassifiers" type="ecore:EClassifier"/>
        <xsd:element name="eSubpackages" type="ecore:EPackage"/>
      </xsd:choice>
      <xsd:attribute name="nsURI" type="xsd:string"/>
      <xsd:attribute name="nsPrefix" type="xsd:string"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="EPackage" type="ecore:EPackage"/>
<xsd:complexType name="EParameter">
```

```
      <xsd:complexContent>
        <xsd:extension base="ecore:ETypedElement"/>
      </xsd:complexContent>
    </xsd:complexType>
    <xsd:element name="EParameter" type="ecore:EParameter"/>
    <xsd:complexType name="EReference">
      <xsd:complexContent>
        <xsd:extension base="ecore:EStructuralFeature">
          <xsd:choice maxOccurs="unbounded" minOccurs="0">
            <xsd:element name="eOpposite" type="ecore:EReference"/>
          </xsd:choice>
          <xsd:attribute name="containment" type="xsd:boolean"/>
          <xsd:attribute name="resolveProxies" type="xsd:boolean"/>
          <xsd:attribute name="eOpposite" type="xsd:string"/>
        </xsd:extension>
      </xsd:complexContent>
    </xsd:complexType>
    <xsd:element name="EReference" type="ecore:EReference"/>
    <xsd:complexType abstract="true" name="EStructuralFeature">
      <xsd:complexContent>
        <xsd:extension base="ecore:ETypedElement">
          <xsd:attribute name="changeable" type="xsd:boolean"/>
          <xsd:attribute name="volatile" type="xsd:boolean"/>
          <xsd:attribute name="transient" type="xsd:boolean"/>
          <xsd:attribute name="defaultValueLiteral" type="xsd:string"/>
          <xsd:attribute name="unsettable" type="xsd:boolean"/>
          <xsd:attribute name="derived" type="xsd:boolean"/>
        </xsd:extension>
      </xsd:complexContent>
    </xsd:complexType>
    <xsd:element name="EStructuralFeature" type="ecore:EStructuralFeature"/>
    <xsd:complexType abstract="true" name="ETypedElement">
      <xsd:complexContent>
        <xsd:extension base="ecore:ENamedElement">
          <xsd:choice maxOccurs="unbounded" minOccurs="0">
            <xsd:element name="eType" type="ecore:EClassifier"/>
          </xsd:choice>
          <xsd:attribute name="ordered" type="xsd:boolean"/>
          <xsd:attribute name="unique" type="xsd:boolean"/>
          <xsd:attribute name="lowerBound" type="xsd:int"/>
          <xsd:attribute name="upperBound" type="xsd:int"/>
          <xsd:attribute name="eType" type="xsd:string"/>
        </xsd:extension>
```

```
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:element name="ETypedElement" type="ecore:ETypedElement"/>
  <xsd:complexType name="EStringToStringMapEntry">
    <xsd:choice maxOccurs="unbounded" minOccurs="0">
      <xsd:element ref="xmi:Extension"/>
    </xsd:choice>
    <xsd:attribute ref="xmi:id"/>
    <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
    <xsd:attribute name="key" type="xsd:string"/>
    <xsd:attribute name="value" type="xsd:string"/>
  </xsd:complexType>
  <xsd:element name="EStringToStringMapEntry"
type="ecore:EStringToStringMapEntry"/>
</xsd:schema>
```

B.3 Base Type System Ecore Model

TODO

B.4 PE Metadata and Behavioral Metadata Ecore Model

TODO:Out of date. Also ffix capitalization: is it peMetadata or pemetdata?

```
<?xml version="1.0" encoding="UTF-8"?>
<ecore:EPackage xmi:version="2.0"
    xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore" name="uima"
    nsURI="http:///uima.ecore" nsPrefix="uima">
  <eSubpackages name="peMetadata" nsURI="http://docs.oasis-
open.org/uima/pemetadata.ecore"
      nsPrefix="uima.peMetadata">
    <eClassifiers xsi:type="ecore:EClass" name="Identification">
      <eStructuralFeatures xsi:type="ecore:EAttribute" name="symbolicName"
eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
      <eStructuralFeatures xsi:type="ecore:EAttribute" name="name"
eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
      <eStructuralFeatures xsi:type="ecore:EAttribute" name="description"
eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
      <eStructuralFeatures xsi:type="ecore:EAttribute" name="vendor"
eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
      <eStructuralFeatures xsi:type="ecore:EAttribute" name="version"
eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
      <eStructuralFeatures xsi:type="ecore:EAttribute" name="url"
eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    </eClassifiers>
```

```
    <eClassifiers xsi:type="ecore:EClass" name="ConfigurationParameter">
      <eStructuralFeatures xsi:type="ecore:EAttribute" name="name"
eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
      <eStructuralFeatures xsi:type="ecore:EAttribute" name="description"
eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
      <eStructuralFeatures xsi:type="ecore:EAttribute" name="type"
eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
      <eStructuralFeatures xsi:type="ecore:EAttribute" name="multiValued"
eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EBoolean"/>
      <eStructuralFeatures xsi:type="ecore:EAttribute" name="mandatory"
eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EBoolean"/>
      <eStructuralFeatures xsi:type="ecore:EAttribute" name="defaultValue"
upperBound="-1"
          eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    </eClassifiers>
    <eClassifiers xsi:type="ecore:EClass" name="TypeSystemReference">
      <eStructuralFeatures xsi:type="ecore:EAttribute" name="uri"
eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    </eClassifiers>
    <eClassifiers xsi:type="ecore:EClass" name="BehavioralMetadata">
      <eStructuralFeatures xsi:type="ecore:EReference" name="analyzes"
lowerBound="1"
          eType="#//peMetadata/BehaviorElement" containment="true"/>
      <eStructuralFeatures xsi:type="ecore:EReference" name="requiredInputs"
lowerBound="1"
          eType="#//peMetadata/BehaviorElement" containment="true"/>
      <eStructuralFeatures xsi:type="ecore:EReference" name="optionalInputs"
lowerBound="1"
          eType="#//peMetadata/BehaviorElement" containment="true"/>
      <eStructuralFeatures xsi:type="ecore:EReference" name="creates"
lowerBound="1"
          eType="#//peMetadata/BehaviorElement" containment="true"/>
      <eStructuralFeatures xsi:type="ecore:EReference" name="modifies"
lowerBound="1"
          eType="#//peMetadata/BehaviorElement" containment="true"/>
      <eStructuralFeatures xsi:type="ecore:EReference" name="deletes"
lowerBound="1"
          eType="#//peMetadata/BehaviorElement" containment="true"/>
      <eStructuralFeatures xsi:type="ecore:EReference" name="precondition"
lowerBound="1"
          eType="#//peMetadata/Condition" containment="true"/>
      <eStructuralFeatures xsi:type="ecore:EReference" name="postcondition"
lowerBound="1"
          eType="#//peMetadata/Condition" containment="true"/>
      <eStructuralFeatures xsi:type="ecore:EReference"
name="projectionCondition"
```

```
            lowerBound="1" eType="#//peMetadata/Condition" containment="true"/>
        <eStructuralFeatures xsi:type="ecore:EReference" name="requiredView"
upperBound="-1"
            eType="#//peMetadata/ViewBehavioralMetadata" containment="true"/>
        <eStructuralFeatures xsi:type="ecore:EReference" name="optionalView"
upperBound="-1"
            eType="#//peMetadata/ViewBehavioralMetadata" containment="true"/>
    </eClassifiers>
    <eClassifiers xsi:type="ecore:EClass" name="ProcessingElementMetadata">
        <eStructuralFeatures xsi:type="ecore:EReference"
name="configurationParameter"
            upperBound="-1" eType="#//peMetadata/ConfigurationParameter"
containment="true"/>
        <eStructuralFeatures xsi:type="ecore:EReference" name="identification"
lowerBound="1"
            eType="#//peMetadata/Identification" containment="true"/>
        <eStructuralFeatures xsi:type="ecore:EReference"
name="typeSystemReference"
            lowerBound="1" eType="#//peMetadata/TypeSystemReference"
containment="true"/>
        <eStructuralFeatures xsi:type="ecore:EReference"
name="behavioralMetadata" lowerBound="1"
            eType="#//peMetadata/BehavioralMetadata" containment="true"/>
        <eStructuralFeatures xsi:type="ecore:EReference" name="extension"
upperBound="-1"
            eType="#//peMetadata/Extension" containment="true"/>
    </eClassifiers>
    <eClassifiers xsi:type="ecore:EClass" name="Extension">
        <eStructuralFeatures xsi:type="ecore:EAttribute" name="extenderId"
eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
        <eStructuralFeatures xsi:type="ecore:EReference" name="contents"
lowerBound="1"
            eType="ecore:EClass
http://www.eclipse.org/emf/2002/Ecore#//EObject"/>
    </eClassifiers>
    <eClassifiers xsi:type="ecore:EClass" name="BehaviorElement">
        <eStructuralFeatures xsi:type="ecore:EReference" name="type"
upperBound="-1"
            eType="#//peMetadata/Type" containment="true"/>
    </eClassifiers>
    <eClassifiers xsi:type="ecore:EClass" name="Type">
        <eStructuralFeatures xsi:type="ecore:EAttribute" name="name"
eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
        <eStructuralFeatures xsi:type="ecore:EAttribute" name="feature"
upperBound="-1"
            eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EString"/>
```

```
    </eClassifiers>
    <eClassifiers xsi:type="ecore:EClass" name="Condition">
      <eStructuralFeatures xsi:type="ecore:EAttribute" name="language"
eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
      <eStructuralFeatures xsi:type="ecore:EAttribute" name="expression"
eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
      <eStructuralFeatures xsi:type="ecore:EAttribute" name="feature"
upperBound="-1"
          eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    </eClassifiers>
    <eClassifiers xsi:type="ecore:EClass" name="ViewBehavioralMetadata"
eSuperTypes="#//peMetadata/BehavioralMetadata"/>
  </eSubpackages>
</ecore:EPackage>
```

B.5 PE Metadata and Behavioral Metadata XML Schema

TODO: Out of Date

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

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<xsd:schema xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore"
xmlns:uima.peMetadata="http://docs.oasis-open.org/uima/pemetadata.ecore"
xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://docs.oasis-open.org/uima/pemetadata.ecore">
  <xsd:import namespace="http://www.eclipse.org/emf/2002/Ecore"
schemaLocation="ecore.xsd"/>
  <xsd:import namespace="http://www.omg.org/XMI" schemaLocation="XMI.xsd"/>
  <xsd:complexType name="Identification">
    <xsd:choice maxOccurs="unbounded" minOccurs="0">
      <xsd:element ref="xmi:Extension"/>
    </xsd:choice>
    <xsd:attribute ref="xmi:id"/>
    <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
    <xsd:attribute name="symbolicName" type="xsd:string"/>
    <xsd:attribute name="name" type="xsd:string"/>
    <xsd:attribute name="description" type="xsd:string"/>
    <xsd:attribute name="vendor" type="xsd:string"/>
    <xsd:attribute name="version" type="xsd:string"/>
    <xsd:attribute name="url" type="xsd:string"/>
  </xsd:complexType>
  <xsd:element name="Identification" type="uima.peMetadata:Identification"/>
  <xsd:complexType name="ConfigurationParameter">
    <xsd:choice maxOccurs="unbounded" minOccurs="0">
```

```
        <xsd:element name="defaultValue" nillable="true" type="xsd:string"/>
        <xsd:element ref="xmi:Extension"/>
      </xsd:choice>
      <xsd:attribute ref="xmi:id"/>
      <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
      <xsd:attribute name="name" type="xsd:string"/>
      <xsd:attribute name="description" type="xsd:string"/>
      <xsd:attribute name="type" type="xsd:string"/>
      <xsd:attribute name="multiValued" type="xsd:boolean"/>
      <xsd:attribute name="mandatory" type="xsd:boolean"/>
    </xsd:complexType>
    <xsd:element name="ConfigurationParameter"
  type="uima.peMetadata:ConfigurationParameter"/>
    <xsd:complexType name="TypeSystemReference">
      <xsd:choice maxOccurs="unbounded" minOccurs="0">
        <xsd:element ref="xmi:Extension"/>
      </xsd:choice>
      <xsd:attribute ref="xmi:id"/>
      <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
      <xsd:attribute name="uri" type="xsd:string"/>
    </xsd:complexType>
    <xsd:element name="TypeSystemReference"
  type="uima.peMetadata:TypeSystemReference"/>
    <xsd:complexType name="BehavioralMetadata">
      <xsd:choice maxOccurs="unbounded" minOccurs="0">
        <xsd:element name="analyzes" type="uima.peMetadata:BehaviorElement"/>
        <xsd:element name="requiredInputs"
  type="uima.peMetadata:BehaviorElement"/>
        <xsd:element name="optionalInputs"
  type="uima.peMetadata:BehaviorElement"/>
        <xsd:element name="creates" type="uima.peMetadata:BehaviorElement"/>
        <xsd:element name="modifies" type="uima.peMetadata:BehaviorElement"/>
        <xsd:element name="deletes" type="uima.peMetadata:BehaviorElement"/>
        <xsd:element name="precondition" type="uima.peMetadata:Condition"/>
        <xsd:element name="postcondition" type="uima.peMetadata:Condition"/>
        <xsd:element name="projectionCondition"
  type="uima.peMetadata:Condition"/>
        <xsd:element name="requiredView"
  type="uima.peMetadata:ViewBehavioralMetadata"/>
        <xsd:element name="optionalView"
  type="uima.peMetadata:ViewBehavioralMetadata"/>
        <xsd:element ref="xmi:Extension"/>
      </xsd:choice>
      <xsd:attribute ref="xmi:id"/>
      <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
```

```
  </xsd:complexType>
  <xsd:element name="BehavioralMetadata"
type="uima.peMetadata:BehavioralMetadata"/>
  <xsd:complexType name="ProcessingElementMetadata">
    <xsd:choice maxOccurs="unbounded" minOccurs="0">
      <xsd:element name="configurationParameter"
type="uima.peMetadata:ConfigurationParameter"/>
      <xsd:element name="identification"
type="uima.peMetadata:Identification"/>
      <xsd:element name="typeSystemReference"
type="uima.peMetadata:TypeSystemReference"/>
      <xsd:element name="behavioralMetadata"
type="uima.peMetadata:BehavioralMetadata"/>
      <xsd:element name="extension" type="uima.peMetadata:Extension"/>
      <xsd:element ref="xmi:Extension"/>
    </xsd:choice>
    <xsd:attribute ref="xmi:id"/>
    <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
  </xsd:complexType>
  <xsd:element name="ProcessingElementMetadata"
type="uima.peMetadata:ProcessingElementMetadata"/>
  <xsd:complexType name="Extension">
    <xsd:choice maxOccurs="unbounded" minOccurs="0">
      <xsd:element name="contents" type="ecore:EObject"/>
      <xsd:element ref="xmi:Extension"/>
    </xsd:choice>
    <xsd:attribute ref="xmi:id"/>
    <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
    <xsd:attribute name="extenderId" type="xsd:string"/>
    <xsd:attribute name="contents" type="xsd:string"/>
  </xsd:complexType>
  <xsd:element name="Extension" type="uima.peMetadata:Extension"/>
  <xsd:complexType name="BehaviorElement">
    <xsd:choice maxOccurs="unbounded" minOccurs="0">
      <xsd:element name="type" type="uima.peMetadata:Type"/>
      <xsd:element ref="xmi:Extension"/>
    </xsd:choice>
    <xsd:attribute ref="xmi:id"/>
    <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
  </xsd:complexType>
  <xsd:element name="BehaviorElement"
type="uima.peMetadata:BehaviorElement"/>
  <xsd:complexType name="Type">
    <xsd:choice maxOccurs="unbounded" minOccurs="0">
      <xsd:element name="feature" nillable="true" type="xsd:string"/>
```

```
        <xsd:element ref="xmi:Extension"/>
      </xsd:choice>
      <xsd:attribute ref="xmi:id"/>
      <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
      <xsd:attribute name="name" type="xsd:string"/>
    </xsd:complexType>
    <xsd:element name="Type" type="uima.peMetadata:Type"/>
    <xsd:complexType name="Condition">
      <xsd:choice maxOccurs="unbounded" minOccurs="0">
        <xsd:element name="feature" nillable="true" type="xsd:string"/>
        <xsd:element ref="xmi:Extension"/>
      </xsd:choice>
      <xsd:attribute ref="xmi:id"/>
      <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
      <xsd:attribute name="language" type="xsd:string"/>
      <xsd:attribute name="expression" type="xsd:string"/>
    </xsd:complexType>
    <xsd:element name="Condition" type="uima.peMetadata:Condition"/>
    <xsd:complexType name="ViewBehavioralMetadata">
      <xsd:complexContent>
        <xsd:extension base="uima.peMetadata:BehavioralMetadata"/>
      </xsd:complexContent>
    </xsd:complexType>
    <xsd:element name="ViewBehavioralMetadata"
type="uima.peMetadata:ViewBehavioralMetadata"/>
</xsd:schema>
```

B.6 PE Service WSDL Definition

TODO: This is out of date

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions
  targetNamespace="http://docs.oasis-open.org/uima/peService"
  xmlns:service="http://docs.oasis-open.org/uima/peService"
  xmlns:pemd="http://docs.oasis-open.org/uima/peMetadata.ecore"
  xmlns:pe="http://docs.oasis-open.org/uima/pe.ecore"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xmi="http://www.omg.org/XMI">

  <wsdl:types>
    <!-- Import the PE Metadata Schema Definitions -->
    <xsd:import
```

```
        namespace="http://docs.oasis-open.org/uima/peMetadata.ecore"
        schemaLocation="uima.peMetadataXMI.xsd"/>

      <!-- Import the XMI schema. -->
      <xsd:import namespace="http://www.omg.org/XMI"
        schemaLocation="XMI.xsd"/>

      <!-- Import other type definitions used as part of the service API. -->
      <xsd:import
        namespace="http://docs.oasis-open.org/uima/pe.ecore"
        schemaLocation="uima.peServiceXMI.xsd"/>
    </wsdl:types>

    <!-- Define the messages sent to and from the service. -->

    <!-- Messages for all UIMA Processing Elements -->
    <wsdl:message name="getMetadataRequest">
    </wsdl:message>

    <wsdl:message name="getMetadataResponse">
      <wsdl:part element="metadata"
        type="pemd:ProcessingElementMetadata" name="metadata"/>
    </wsdl:message>

    <wsdl:message name="setConfigurationParametersRequest">
      <wsdl:part element="settings"
        type="pemd:ConfigurationParameterSettings" name="settings"/>
    </wsdl:message>

    <wsdl:message name="setConfigurationParametersResponse">
    </wsdl:message>

    <wsdl:message name="uimaFault">
      <wsdl:part element="exception" type="pe:UimaException" name="exception"/>
    </wsdl:message>


    <!-- Messages for the Analyzer interface -->
    <!--  Note that processCasRequest and processCasResponse allow
          multiple CASes to be sent in one batch, for performance
          reasons.  -->

    <wsdl:message name="processCasRequest">
```

```
    <wsdl:part element="casList" type="pe:CasList" name="casList"/>
    <wsdl:part element="sofas" type="pe:ObjectList" name="sofas"/>
  </wsdl:message>

  <wsdl:message name="processCasResponse">
    <wsdl:part element="casList" type="pe:CasList" name="casList"/>
  </wsdl:message>

  <!-- Messages for the CasMultiplier interface -->
  <!--  Note that inputCasRequest and getNextResponse allow
        multiple CASes to be sent in one batch, for performance
        reasons.  -->
  <wsdl:message name="inputCasRequest">
    <wsdl:part element="casList" type="pe:CasList" name="casList"/>
    <wsdl:part element="sofas" type="pe:ObjectList" name="sofas"/>
  </wsdl:message>

  <wsdl:message name="inputCasResponse">
  </wsdl:message>

  <wsdl:message name="getNextRequest">
    <wsdl:part element="maxCASesToReturn" type="xsd:integer"
name="maxCASesToReturn"/>
    <wsdl:part element="timeToWait" type="xsd:integer" name="timeToWait"/>
  </wsdl:message>

  <wsdl:message name="getNextResponse">
    <wsdl:part element="reponse" type="pe:GetNextResponse" name="response"/>
  </wsdl:message>

  <wsdl:message name="retrieveInputCasRequest">
  </wsdl:message>

  <wsdl:message name="retrieveInputCasResponse">
    <wsdl:part element="casList" type="pe:CasList" name="casList"/>
  </wsdl:message>

  <!-- Messages for the FlowController interface -->

  <wsdl:message name="addAvailableAnalyticsRequest">
    <wsdl:part element="analyticMetadataMap"
      type="pe:AnalyticMetadataMap" name="analyticMetadataMap"/>
  </wsdl:message>
```

```
  <wsdl:message name="addAvailableAnalyticsResponse">
  </wsdl:message>

  <wsdl:message name="removeAvailableAnalyticsRequest">
    <wsdl:part element="analyticKeys" type="pe:Keys"
      name="analyticKeys"/>
  </wsdl:message>

  <wsdl:message name="removeAvailableAnalyticsResponse">
  </wsdl:message>

  <wsdl:message name="setAggregateMetadataRequest">
    <wsdl:part element="metadata"
      type="pemd:ProcessingElementMetadata" name="metadata"/>
  </wsdl:message>

  <wsdl:message name="setAggregateMetadataResponse">
  </wsdl:message>

  <wsdl:message name="getNextDestinationsRequest">
    <wsdl:part element="cas" type="xmi:XMI" name="cas"/>
  </wsdl:message>

  <wsdl:message name="getNextDestinationsResponse">
    <wsdl:part element="step" type="pe:Step" name="step"/>
  </wsdl:message>

  <wsdl:message name="continueOnFailureRequest">
    <wsdl:part element="cas" type="xmi:XMI" name="cas"/>
    <wsdl:part element="failedAnalyticKey" type="xsd:string"
name="failedAnalyticKey"/>
    <wsdl:part element="failure" type="pe:UimaException" name="failure"/>
  </wsdl:message>

  <wsdl:message name="continueOnFailureResponse">
    <wsdl:part element="continue" type="xsd:boolean" name="continue"/>
  </wsdl:message>

  <!-- Define a portType for each of the UIMA interfaces -->
  <wsdl:portType name="Analyzer">

    <wsdl:operation name="getMetadata">
```

```
      <wsdl:input message="service:getMetadataRequest"
        name="getMetadataRequest"/>
      <wsdl:output message="service:getMetadataResponse"
        name="getMetadataResponse"/>
      <wsdl:fault message="service:uimaFault"
        name="uimaFault"/>
    </wsdl:operation>

    <wsdl:operation name="setConfigurationParameters">
      <wsdl:input
        message="service:setConfigurationParametersRequest"
        name="setConfigurationParametersRequest"/>
      <wsdl:output
        message="service:setConfigurationParametersResponse"
        name="setConfigurationParametersResponse"/>
      <wsdl:fault message="service:uimaFault"
        name="uimaFault"/>
    </wsdl:operation>

    <wsdl:operation name="processCas">
      <wsdl:input message="service:processCasRequest"
        name="processCasRequest"/>
      <wsdl:output message="service:processCasResponse"
        name="processCasResponse"/>
      <wsdl:fault message="service:uimaFault"
        name="uimaFault"/>
    </wsdl:operation>

  </wsdl:portType>

  <wsdl:portType name="CasMultiplier">

    <wsdl:operation name="getMetadata">
      <wsdl:input message="service:getMetadataRequest"
        name="getMetadataRequest"/>
      <wsdl:output message="service:getMetadataResponse"
        name="getMetadataResponse"/>
      <wsdl:fault message="service:uimaFault"
        name="uimaFault"/>
    </wsdl:operation>

    <wsdl:operation name="setConfigurationParameters">
      <wsdl:input
```

```
        message="service:setConfigurationParametersRequest"
        name="setConfigurationParametersRequest"/>
      <wsdl:output
        message="service:setConfigurationParametersResponse"
        name="setConfigurationParametersResponse"/>
      <wsdl:fault message="service:uimaFault"
        name="uimaFault"/>
    </wsdl:operation>

    <wsdl:operation name="inputCas">
      <wsdl:input message="service:inputCasRequest"
        name="inputCasRequest"/>
      <wsdl:output message="service:inputCasResponse"
        name="inputCasResponse"/>
      <wsdl:fault message="service:uimaFault"
        name="uimaFault"/>
    </wsdl:operation>

    <wsdl:operation name="getNext">
      <wsdl:input message="service:getNextRequest"
        name="getNextRequest"/>
      <wsdl:output message="service:getNextResponse"
        name="getNextResponse"/>
      <wsdl:fault message="service:uimaFault"
        name="uimaFault"/>
    </wsdl:operation>

    <wsdl:operation name="retrieveInputCas">
      <wsdl:input message="service:retrieveInputCasRequest"
        name="retrieveInputCasRequest"/>
      <wsdl:output message="service:retrieveInputCasResponse"
        name="retrieveInputCasResponse"/>
      <wsdl:fault message="service:uimaFault"
        name="uimaFault"/>
    </wsdl:operation>
  </wsdl:portType>

  <wsdl:portType name="FlowController">

    <wsdl:operation name="getMetadata">
      <wsdl:input message="service:getMetadataRequest"
        name="getMetadataRequest"/>
      <wsdl:output message="service:getMetadataResponse"
```

```
          name="getMetadataResponse"/>
        <wsdl:fault message="service:uimaFault"
          name="uimaFault"/>
      </wsdl:operation>

      <wsdl:operation name="setConfigurationParameters">
        <wsdl:input
          message="service:setConfigurationParametersRequest"
          name="setConfigurationParametersRequest"/>
        <wsdl:output
          message="service:setConfigurationParametersResponse"
          name="setConfigurationParametersResponse"/>
        <wsdl:fault message="service:uimaFault"
          name="uimaFault"/>
      </wsdl:operation>

      <wsdl:operation name="addAvailableAnalytics">
        <wsdl:input message="service:addAvailableAnalyticsRequest"
          name="addAvailableAnalyticsRequest"/>
        <wsdl:output message="service:addAvailableAnalyticsResponse"
          name="addAvailableAnalyticsResponse"/>
        <wsdl:fault message="service:uimaFault"
          name="uimaFault"/>
      </wsdl:operation>

      <wsdl:operation name="removeAvailableAnalytics">
        <wsdl:input
          message="service:removeAvailableAnalyticsRequest"
          name="removeAvailableAnalyticsRequest"/>
        <wsdl:output
          message="service:removeAvailableAnalyticsResponse"
          name="removeAvailableAnalyticsResponse"/>
        <wsdl:fault message="service:uimaFault"
          name="uimaFault"/>
      </wsdl:operation>

      <wsdl:operation name="setAggregateMetadata">
        <wsdl:input message="service:setAggregateMetadataRequest"
          name="setAggregateMetadataRequest"/>
        <wsdl:output message="service:setAggregateMetadataResponse"
          name="setAggregateMetadataResponse"/>
        <wsdl:fault message="service:uimaFault"
          name="uimaFault"/>
```

```
    </wsdl:operation>

    <wsdl:operation name="getNextDestinations">
      <wsdl:input message="service:getNextDestinationsRequest"
        name="getNextDestinationsRequest"/>
      <wsdl:output message="service:getNextDestinationsResponse"
        name="getNextDestinationsResponse"/>
      <wsdl:fault message="service:uimaFault"
        name="uimaFault"/>
    </wsdl:operation>

    <wsdl:operation name="continueOnFailure">
      <wsdl:input message="service:continueOnFailureRequest"
        name="continueOnFailureRequest"/>
      <wsdl:output message="service:continueOnFailureResponse"
        name="continueOnFailureResponse"/>
      <wsdl:fault message="service:uimaFault"
        name="uimaFault"/>
    </wsdl:operation>

  </wsdl:portType>

  <!-- Define a SOAP binding for each portType. -->
  <wsdl:binding name="AnalyzerSoapBinding" type="service:Analyzer">

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

    <wsdl:operation name="getMetadata">
      <wsdlsoap:operation soapAction=""/>

      <wsdl:input name="getMetadataRequest">
        <wsdlsoap:body use="literal"/>
      </wsdl:input>

      <wsdl:output name="getMetadataResponse">
        <wsdlsoap:body use="literal"/>
      </wsdl:output>
    </wsdl:operation>

    <wsdl:operation name="setConfigurationParameters">
      <wsdlsoap:operation soapAction=""/>

```

```
        <wsdl:input name="setConfigurationParametersRequest">
          <wsdlsoap:body use="literal"/>
        </wsdl:input>

        <wsdl:output name="setConfigurationParametersResponse">
          <wsdlsoap:body use="literal"/>
        </wsdl:output>
      </wsdl:operation>

      <wsdl:operation name="processCas">
        <wsdlsoap:operation soapAction=""/>

        <wsdl:input name="processCasRequest">
          <wsdlsoap:body use="literal"/>
        </wsdl:input>

        <wsdl:output name="processCasResponse">
          <wsdlsoap:body use="literal"/>
        </wsdl:output>
      </wsdl:operation>
    </wsdl:binding>

    <wsdl:binding name="CasMultiplierSoapBinding"
      type="service:CasMultiplier">

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

      <wsdl:operation name="getMetadata">
        <wsdlsoap:operation soapAction=""/>

        <wsdl:input name="getMetadataRequest">
          <wsdlsoap:body use="literal"/>
        </wsdl:input>

        <wsdl:output name="getMetadataResponse">
          <wsdlsoap:body use="literal"/>
        </wsdl:output>

        <wsdl:fault name="uimaFault">
          <wsdlsoap:fault use="literal"/>
        </wsdl:fault>
      </wsdl:operation>
```

```
    <wsdl:operation name="setConfigurationParameters">
      <wsdlsoap:operation soapAction=""/>

      <wsdl:input name="setConfigurationParametersRequest">
        <wsdlsoap:body use="literal"/>
      </wsdl:input>

      <wsdl:output name="setConfigurationParametersResponse">
        <wsdlsoap:body use="literal"/>
      </wsdl:output>

      <wsdl:fault name="uimaFault">
        <wsdlsoap:fault use="literal"/>
      </wsdl:fault>
    </wsdl:operation>

    <wsdl:operation name="inputCas">
      <wsdlsoap:operation soapAction=""/>

      <wsdl:input name="inputCasRequest">
        <wsdlsoap:body use="literal"/>
      </wsdl:input>

      <wsdl:output name="inputCasResponse">
        <wsdlsoap:body use="literal"/>
      </wsdl:output>

      <wsdl:fault name="uimaFault">
        <wsdlsoap:fault use="literal"/>
      </wsdl:fault>
    </wsdl:operation>

    <wsdl:operation name="getNext">
      <wsdlsoap:operation soapAction=""/>

      <wsdl:input name="getNextRequest">
        <wsdlsoap:body use="literal"/>
      </wsdl:input>

      <wsdl:output name="getNextResponse">
        <wsdlsoap:body use="literal"/>
      </wsdl:output>
```

```
      <wsdl:fault name="uimaFault">
        <wsdlsoap:fault use="literal"/>
      </wsdl:fault>
    </wsdl:operation>

    <wsdl:operation name="retrieveInputCas">
      <wsdlsoap:operation soapAction=""/>

      <wsdl:input name="retrieveInputCasRequest">
        <wsdlsoap:body use="literal"/>
      </wsdl:input>

      <wsdl:output name="retrieveInputCasResponse">
        <wsdlsoap:body use="literal"/>
      </wsdl:output>

      <wsdl:fault name="uimaFault">
        <wsdlsoap:fault use="literal"/>
      </wsdl:fault>
    </wsdl:operation>
  </wsdl:binding>

  <wsdl:binding name="FlowControllerSoapBinding"
    type="service:FlowController">

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

    <wsdl:operation name="getMetadata">
      <wsdlsoap:operation soapAction=""/>

      <wsdl:input name="getMetadataRequest">
        <wsdlsoap:body use="literal"/>
      </wsdl:input>

      <wsdl:output name="getMetadataResponse">
        <wsdlsoap:body use="literal"/>
      </wsdl:output>

      <wsdl:fault name="uimaFault">
        <wsdlsoap:fault use="literal"/>
      </wsdl:fault>
```

```
    </wsdl:operation>

    <wsdl:operation name="setConfigurationParameters">
      <wsdlsoap:operation soapAction=""/>

      <wsdl:input name="setConfigurationParametersRequest">
        <wsdlsoap:body use="literal"/>
      </wsdl:input>

      <wsdl:output name="setConfigurationParametersResponse">
        <wsdlsoap:body use="literal"/>
      </wsdl:output>

      <wsdl:fault name="uimaFault">
        <wsdlsoap:fault use="literal"/>
      </wsdl:fault>
    </wsdl:operation>

    <wsdl:operation name="addAvailableAnalytics">
      <wsdlsoap:operation soapAction=""/>

      <wsdl:input name="addAvailableAnalyticsRequest">
        <wsdlsoap:body use="literal"/>
      </wsdl:input>

      <wsdl:output name="addAvailableAnalyticsResponse">
        <wsdlsoap:body use="literal"/>
      </wsdl:output>

      <wsdl:fault name="uimaFault">
        <wsdlsoap:fault use="literal"/>
      </wsdl:fault>
    </wsdl:operation>

    <wsdl:operation name="removeAvailableAnalytics">
      <wsdlsoap:operation soapAction=""/>

      <wsdl:input name="removeAvailableAnalyticsRequest">
        <wsdlsoap:body use="literal"/>
      </wsdl:input>

      <wsdl:output name="removeAvailableAnalyticsResponse">
        <wsdlsoap:body use="literal"/>
```

```
        </wsdl:output>

        <wsdl:fault name="uimaFault">
          <wsdlsoap:fault use="literal"/>
        </wsdl:fault>
      </wsdl:operation>

      <wsdl:operation name="setAggregateMetadata">
        <wsdlsoap:operation soapAction=""/>

        <wsdl:input name="setAggregateMetadataRequest">
          <wsdlsoap:body use="literal"/>
        </wsdl:input>

        <wsdl:output name="setAggregateMetadataResponse">
          <wsdlsoap:body use="literal"/>
        </wsdl:output>

        <wsdl:fault name="uimaFault">
          <wsdlsoap:fault use="literal"/>
        </wsdl:fault>
      </wsdl:operation>

      <wsdl:operation name="getNextDestinations">
        <wsdlsoap:operation soapAction=""/>

        <wsdl:input name="getNextDestinationsRequest">
          <wsdlsoap:body use="literal"/>
        </wsdl:input>

        <wsdl:output name="getNextDestinationsResponse">
          <wsdlsoap:body use="literal"/>
        </wsdl:output>

        <wsdl:fault name="uimaFault">
          <wsdlsoap:fault use="literal"/>
        </wsdl:fault>
      </wsdl:operation>

      <wsdl:operation name="continueOnFailure">
        <wsdlsoap:operation soapAction=""/>

        <wsdl:input name="continueOnFailureRequest">
```

```
        <wsdlsoap:body use="literal"/>
      </wsdl:input>

      <wsdl:output name="continueOnFailureResponse">
        <wsdlsoap:body use="literal"/>
      </wsdl:output>

      <wsdl:fault name="uimaFault">
        <wsdlsoap:fault use="literal"/>
      </wsdl:fault>
    </wsdl:operation>
  </wsdl:binding>

  <!-- Define an example service as including both portTypes.  This is
        just an example, not part of the UIMA Specification -->
  <wsdl:service name="MyAnalyticService">
    <wsdl:port binding="service:AnalyzerSoapBinding"
      name="AnalyzerSoapPort">
      <wsdlsoap:address

location="http://localhost:8080/axis/services/MyAnalyticService/AnalyzerPort"
/>
    </wsdl:port>
    <wsdl:port binding="service:CasMultiplierSoapBinding"
      name="CasMultiplierSoapPort">
      <wsdlsoap:address

location="http://localhost:8080/axis/services/MyAnalyticService/CasMultiplier
Port"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

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

TODO: Out of date

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

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

```
    schemaLocation="uima.peMetadataXMI.xsd"/>
  <xsd:import namespace="http://www.omg.org/XMI"
    schemaLocation="XMI.xsd"/>
  <xsd:complexType name="InputBindings">
    <xsd:choice maxOccurs="unbounded" minOccurs="0">
      <xsd:element name="InputBinding" type="pe:InputBinding"/>
      <xsd:element ref="xmi:Extension"/>
    </xsd:choice>
    <xsd:attribute ref="xmi:id"/>
    <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
    <xsd:attribute name="InputBinding" type="xsd:string"/>
  </xsd:complexType>
  <xsd:element name="InputBindings" type="pe:InputBindings"/>
  <xsd:complexType name="InputBinding">
    <xsd:choice maxOccurs="unbounded" minOccurs="0">
      <xsd:element name="objects" nillable="true"
        type="xsd:string"/>
      <xsd:element ref="xmi:Extension"/>
    </xsd:choice>
    <xsd:attribute ref="xmi:id"/>
    <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
    <xsd:attribute name="handle" type="xsd:string"/>
  </xsd:complexType>
  <xsd:element name="InputBinding" type="pe:InputBinding"/>
  <xsd:complexType name="AnalyticMetadataMap">
    <xsd:choice maxOccurs="unbounded" minOccurs="0">
      <xsd:element name="AnalyticMetadataMapEntry"
        type="pe:AnalyticMetadataMapEntry"/>
      <xsd:element ref="xmi:Extension"/>
    </xsd:choice>
    <xsd:attribute ref="xmi:id"/>
    <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
    <xsd:attribute name="AnalyticMetadataMapEntry"
      type="xsd:string"/>
  </xsd:complexType>
  <xsd:element name="AnalyticMetadataMap"
    type="pe:AnalyticMetadataMap"/>
  <xsd:complexType name="AnalyticMetadataMapEntry">
    <xsd:choice maxOccurs="unbounded" minOccurs="0">
      <xsd:element name="ProcessingElementMetadata"
        type="pemd:ProcessingElementMetadata"/>
      <xsd:element ref="xmi:Extension"/>
    </xsd:choice>
```

```
    <xsd:attribute ref="xmi:id"/>
    <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
    <xsd:attribute name="key" type="xsd:string"/>
    <xsd:attribute name="ProcessingElementMetadata"
      type="xsd:string"/>
  </xsd:complexType>
  <xsd:element name="AnalyticMetadataMapEntry"
    type="pe:AnalyticMetadataMapEntry"/>
  <xsd:complexType name="Step">
    <xsd:choice maxOccurs="unbounded" minOccurs="0">
      <xsd:element ref="xmi:Extension"/>
    </xsd:choice>
    <xsd:attribute ref="xmi:id"/>
    <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
  </xsd:complexType>
  <xsd:element name="Step" type="pe:Step"/>
  <xsd:complexType name="SimpleStep">
    <xsd:complexContent>
      <xsd:extension base="pe:Step">
        <xsd:attribute name="analyticKey" type="xsd:string"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:element name="SimpleStep" type="pe:SimpleStep"/>
  <xsd:complexType name="MultiStep">
    <xsd:complexContent>
      <xsd:extension base="pe:Step">
        <xsd:choice maxOccurs="unbounded" minOccurs="0">
          <xsd:element name="steps" type="pe:Step"/>
        </xsd:choice>
        <xsd:attribute name="parallel" type="xsd:boolean"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:element name="MultiStep" type="pe:MultiStep"/>
  <xsd:complexType name="FinalStep">
    <xsd:complexContent>
      <xsd:extension base="pe:Step"/>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:element name="FinalStep" type="pe:FinalStep"/>
  <xsd:complexType name="Keys">
    <xsd:choice maxOccurs="unbounded" minOccurs="0">
```

```
      <xsd:element name="key" nillable="true" type="xsd:string"/>
      <xsd:element ref="xmi:Extension"/>
    </xsd:choice>
    <xsd:attribute ref="xmi:id"/>
    <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
  </xsd:complexType>
  <xsd:element name="Keys" type="pe:Keys"/>
</xsd:schema>
```

C. Non-Normative Text

D. Revision History

[optional; should not be included in OASIS Standards]

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 Complaince 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.