RC24122 (W0611-188) November 28, 2006
Computer Science

IBM Research Report

Towards an Interoperability Standard for Text and Multi-Modal Analytics

David Ferrucci, Adam Lally, Daniel Gruhl[1], Edward Epstein, Marshall Schor, J. William Murdock, Andy Frenkiel, Eric W. Brown, Thomas Hampp[2], Yurdaer Doganata, Christopher Welty, Lisa Amini, Galina Kofman, Lev Kozakov, Yosi Mass[3]

IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

[1]IBM Research Division
Almaden Research Center
650 Harry Road
San Jose, CA 95120-6099

[2]IBM Software Group
Schönaicher Str. 220
Boeblingen 71032
Germany

[3]IBM Research Division
Haifa Research Laboratory
Mt. Carmel 31905
Haifa, Israel

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Table of Contents

1 Introduction ... 3
1.1 Background ... 3
1.2 Toward a Standard Specification ... 4
1.3 Report Overview ... 6
2 Basic Concepts and Terms ... 6
2.1 Unstructured Information and Analysis ... 6
2.2 Architecture and Frameworks ... 8
2.3 Levels of Interoperability ... 10
3 Design Goals ... 11
4 Use Cases ... 12
4.1 Text analysis of email for SOX compliance ... 13
4.2 Enterprise Semantic Search ... 14
4.3 Cross language summarization of multimodal news feeds ... 15
4.4 Populating a knowledge-base of metabolic pathways ... 15
4.5 Video Segmentation for Just-in-Time Learning ... 16
5 Proposed Elements of a Standard Specification ... 17
5.1 The CAS Specification ... 19
5.2 The Type-System Language Specification ... 29
5.3 Type-System Base Model ... 33
5.4 Behavioral Metadata Specification ... 42
5.5 Processing Element Metadata ... 52
5.6 Abstract Interfaces ... 60
5.7 Service WSDL Descriptions ... 64
5.8 Aggregate Analytic Descriptor ... 72
6 Interoperability Case Studies ... 77
6.1 GATE Case Study ... 78
6.2 OpenNLP Case Study ... 79
6.3 WebFountain Semantic Super Computer Case Study ... 80
6.4 System S Case Study ... 81
7 Discussion Topics ... 82
7.1 Mapping Between Type-Systems ... 82
7.2 Supporting Multiple Sessions ... 83
7.3 End of Collection Processing ... 84
7.4 Provenance ... 85
7.5 Privacy and Security ... 87
7.6 Configuration Parameters Affecting Behavioral Metadata ... 87
7.7 Efficient Stream Processing ... 88
8 References ... 90
9 Appendices/Attachments ... 92
9.1 SOAP Bindings and Example SOAP Messages ... 92
9.2 Referenced XML Schemata ... 98

1 Introduction

This report motivates and proposes elements of an architecture specification for creating and composing text and multi-modal analytics for processing unstructured information, based on the UIMA project originated at IBM Research.

1.1 Background

1.1.1 Unstructured information

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 the explicit semantics (“structure”) needed by computer programs to interpret the information as intended by the human author or required by the application.

Unstructured information may be contrasted with the information in relational databases where the intended interpretation for every data field 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 field labeled “description” for example, may be of interest to an application but remain without any explicit tagging that would reveal its intended semantics.

1.1.2 Assigning Semantics – Structuring the Unstructured

For unstructured information to be processed by traditional applications, it must first be 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 provides the explicit semantics required by target applications to correctly interpret the data.

An example of assigning semantics includes identifying and wrapping regions of text in a document with appropriate XML tags. These tags may identify the regions as mentions of persons, organizations, times, events or products, for example. Another example may extract elements of a document and insert them in the appropriate fields of a relational database or use them to create instances of concepts in a knowledgebase. Another example may analyze a voice stream and tag it with the information explicitly identifying the speaker or where the speaker changes.

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

1.1.3 Unstructured Information Management (UIM) Applications

UIM applications are about generating structured information from unstructured content.

A growing number of applications see increasing value in exploiting unstructured information. This growth is largely driven by the wealth of unstructured information found on the external web, in corporate intranets, document repositories, call-centers, and in customer and employee business communications.

UIM applications tend to be highly decomposable; that is, they may be broken-down into finer grained parts, each performing a specialized function in an overall analysis workflow. Each of these functions, or *analytics,* may be reused in different flows to perform different aggregate analyses. Even in our simple example above, a first, very common, analytic in the overall process is to tokenize the document (identify each individual word). This tokenization function may be reused as a first step in many different analysis tasks for many different applications.

Because UIM applications are naturally decomposable into analytic functions that focus on different parts of a solution and are logically reusable across different solutions, there is an advantage to defining an architecture that supports standards for interoperability between analytic functions. Such an architecture would facilitate the sharing and composition of analytic functions within and between applications.

1.1.4 UIMA – The Unstructured Information Management Architecture

UIMA refers to a software architecture for defining and composing interoperable text and multi-modal analytics. UIMA builds on the work of prior IBM researchers and projects dedicated to advancing the state of the art in frameworks for text and multimodal analytics including TAF, TALENT [TAL1] and WebFountain [WF1,WF2]. It has been inspired and influenced by other projects outside of IBM including TIPSTER [TIP1], Mallet [MAL1], GATE [GATE1,GATE2], OpenNLP [ONLP1], Atlas [Laprun1] and Catalyst [CAT1].

UIMA formally began at IBM in 2001 as a project dedicated to enabling interoperability of analytics across research projects underway in different geographies across the globe by scores of researchers and to facilitate their integration and deployment into a variety of IBM Information Management products. The technical goals were to provide a common software framework that would facilitate the creation, discovery, integration and deployment of component text and multi-modal analytics.

The immediate problem motivating the project was that numerous efforts throughout IBM Research and IBM Software Group were focused on creating independent unstructured information analytics based on different technologies, interfaces, implementations and data representations. These independently developed analytics included for example: tokenization, language identification, document structure parsing, grammatical parsing, named-entity detection, chemical name and relationship detection, summarization, document classification, topic detection and tracking, speaker identification, speech transcription, natural language translation, image analysis, video object detection and tracking etc.

Complete solutions or end-user applications often required a combination of numerous highly specialized capabilities. The technical hurdles involved in reusing, combining and deploying these independently developed analytics across research projects and product platforms were often prohibitive. The results were inefficiencies and/or sub-optimal analytic performance.

1.2 Toward a Standard Specification

In late 2004 IBM released the *UIMA Software Developers Kit (SDK)* on IBM alphaWorks (http://www.ibm.com/alphaworks/tech/uima). The SDK is freely available and provides the tools

and run-time necessary for creating, composing and deploying component analytics. These analytics may be implemented by the developer to analyze and assign semantics to multi-modal data including, for example, combinations of text, audio or video. The SDK includes a semantic search engine based on the XMLFragment query language [XMLFrags1] for users to experiment with a search facility designed to exploit the semantic elements produced by a workflow of analytics.

Since 2004 industrial, academic and government research & development projects inside and outside of IBM have applied the UIMA SDK as a foundation for building and/or enhancing applications that process unstructured information. Feedback from users has helped inform the ideas presented in this report. At the time of publication of this report, there have been over 8000 downloads of the UIMA SDK by over 1000 distinct entities.

In early 2006 IBM contributed an implementation of the UIMA Framework to the open-source community through source forge [UIMASrc1]. The source includes everything in the SDK except the XMLFragment search engine. In 4Q 2006, the open-source framework was accepted as an Apache incubation project, where IBM and non-IBM *committers* will participate in its collaborative development. [ApacheUIMA1]. Throughout this report we use the term *Apache UIMA* to refer to this implementation.

The Apache UIMA implementation requires application and programming commitments at the Java API level to provide a rich set of functions and to facilitate high-degrees of component-level interoperability. Any given framework implementation, however, may not satisfy the requirements of all UIM applications. Some, for example, may require very lightweight browser-based analytics or a very different programming model, while others may require heavyweight, carefully managed, highly scalable solutions and yet others may depend on legacy infrastructure or middleware.

This diverse variety of potential implementations suggests that there should be a more general specification for interoperability that may allow for different framework implementations and different levels of compliance facilitating interoperability for a broader range of application and programming requirements.

To help define a broader, platform independent standard that can guide the open-source collaborative development of Apache UIMA and other related frameworks, applications and tools while maintaining broad interoperability, IBM has convened a Technical Committee to develop a standard specification under the auspices of OASIS (www.oasis-open.org) a Standards Development Organization (SDO). The intent is that such a standard would allow different frameworks to emerge, while also allowing applications built on different platforms and programming models to have a standard means to share analysis data and analytic services. Such a standard would lower the barrier for getting analytics to interoperate, allowing a broader community to discover, reuse and compose independently-developed text and multi-modal analytics in UIM applications.

In this report we propose elements of an architecture specification for interoperable text and multi-modal analytics, based on our work with UIMA, that we believe can provide the foundation for such standard. We identify and discuss requirements and open-issues, which we believe a complete specification should address.

For the purposes of this report we refer to the proposed specification as the ***UIMA Specification.***

1.3 Report Overview

The body of this report is organized as follows:

Basic Concepts and Terms: This section introduces the reader to a set of terms used throughout the report to refer to concepts related to unstructured information analysis, architecture, frameworks and interoperability.

Design Goals: This section proposes a set of high-level design goals for the UIMA Specification.

Use Cases: This section provides a few brief use-cases describing a sampling of different types of applications that have requirements for interoperable text and multi-modal analytics.

Proposed Elements of a Specification: This section is the body of the report. We propose eight principal elements of the UIMA specification and discuss each in turn.

Interoperability Case Studies: We have worked with several different frameworks and a wide array of analytics for unstructured information. Many elements of this proposal for a standard architecture based on UIMA, emerged from and/or were influenced by our experiences adapting or bridging Apache UIMA to interoperate with other frameworks. In this section we provide a brief overview of these experiences as case-studies highlighting interoperability issues that motivate various aspects of the architecture specification

Discussion Topics: This section discusses a few topics that repeatedly came up as high-level requirements, but which have not been addressed in Apache UIMA nor have we deeply considered as candidate elements of the UIMA specification. We believe these topics should be explored to determine if they are requirements that should be addressed by the UIMA specification.

Appendices: XML Schema definitions for the concepts defined in this specification are included as an appendix.

2 Basic Concepts and Terms

In this section we provide definitions for basic terms we use throughout the body of this paper.

2.1 Unstructured Information and Analysis

2.1.1 Artifact

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.

2.1.2 Artifact Modality

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

2.1.3 Artifact Metadata

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.

2.1.4 Domain Model

Domain model generally refers to a conceptualization of a system, often cast in a formal modeling language. In this report 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.

2.1.5 Analysis Data

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

2.1.6 Analysis Operations

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.

2.1.7 Behavioral Metadata

Behavioral Metadata (or Analytic Metadata) is data that declaratively characterizes the behavior of an Analysis Operation, describing, for example, the types of artifact metadata the operation produces given an artifact to analyze. A specific example of behavioral metadata might declare that given a video stream, the analysis operation produces annotations labeling instances of vehicles that occur in frames of the video.

2.1.8 Analysis Components

Analysis Components are software objects that perform Analysis Operations. They may be composed by some workflow of other *analysis components* or *analysis services*.

2.1.9 Analysis Services

Analysis Services are network services (e.g., SOAP Services) that perform an analysis operation. They may be composed of a flow of other *analysis components* or *analysis services*.

2.1.10 Analytic

An **Analytic** is an Analysis Service or an Analysis Component or some combination or two or more of the same.

2.1.11 Flow Controller

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

2.1.12 Processing Element (PE)

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.

2.1.13 Aggregate Analytic

An **Aggregate Analytic** is an analytic that is composed of some workflow of constituent analytics. An Aggregate Analytic exposes the same interface as any analytic but is implemented by a set of constituent analytics plus a flow controller.

2.1.14 Analysis Workflow

An **Analysis Workflow** is a graph of analytics intended to process a stream of artifacts in a particular order determined by Flow Controller.

2.1.15 Processing Element Metadata (PE Metadata)

Processing Element 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.

2.2 Architecture and Frameworks

In this section we propose definitions for terms fundamental to any discussion around architectures, frameworks and standards for unstructured information that may be ambiguous or often vaguely defined. The intent is to provide clarity in our intent for the proceeding discussion rather than to propose ideal definitions.

2.2.1 Platform

Platform refers to a specific programming language, operating-system, application-server or database or some combination thereof that provides a foundation for implementing a higher-level software system. A platform-specific model describes how to implement a platform-independent model on a particular platform.

2.2.2 Logical Architecture or Specification

Logical Architecture refers to a platform-independent model defining data representations and operation types, their interfaces and behaviors. The intent of which is to facilitate interoperability between applications whose interfaces conform to the architecture without specifying how to implement those interfaces on any particular platform.

2.2.3 UIMA Specification

The UIMA Specification refers to an evolving specification for supporting interoperable text and multi-modal analytics. Requirements and elements of the UIMA specification are proposed and discussed in this report.

2.2.4 UIM Application

UIM applications are software systems that apply analytics to assign semantics to unstructured information content in order to solve some business problem. The additional semantic information produced by the analysis may be used, for example, by the application to better organize, store and access the content according to the application's requirements.

2.2.5 UIM Middleware

UIM Middleware refers to software infrastructure that facilitates the definition, composition and deployment of unstructured information analytics in the development of UIM Applications.

2.2.6 UIMA Compliance Point

The UIMA Specification proposed in this report describes different requirements of analytics, applications and/or frameworks in order for them to interoperate at varying levels of compliance.

We believe it is worthwhile to ultimately provide very explicit and precise descriptions of these requirements and the implications of compliance or non-compliance by any implementation. In this report we highlight requirements of the specification we believe are candidates for detailed explication as a formal "Compliance Point" like this

> **Candidate Compliance Point:** Informal description and/or discussion of the requirement.

A final specification should consider these for more formal treatment.

2.2.7 UIMA Framework

A UIMA Framework is a software system designed to facilitate the development and deployment of UIMA-compliant analytics. A UIMA framework binds the architecture specifications to a particular programming language implementation and in so doing defines specific APIs and programming models. A UIMA framework may, for example, provide an implementation in a particular programming language of a data specification defined by the architecture, including a particular set of APIs. A UIMA framework may provide the means for communicating between implemented UIMA interfaces over a particular communications protocol. A UIMA framework may provide hooks for packaging and installing components, sharing resources between components, common logging mechanisms and/or common APIs for managing component life cycles.

2.2.8 Apache UIMA

Apache UIMA is a UIMA Framework, originally implemented by IBM and now undergoing incubation at the Apache Software Foundation [ApacheUIMA1], where IBM and non-IBM committers can contribute to its development. For details on Apache UIMA, we refer the reader to http://incubator.apache.org/uima/.

Apache UIMA Notes

In this report we call out in grey boxes notable highlights regarding how the implementation of Apache UIMA relates to the proposed UIMA specification.

2.3 Levels of Interoperability

In this section we provide practical definitions of different levels of interoperability to help describe design objectives of the UIMA Specification.

The UIMA Specification design goals focus on providing a standard specification for text and multi-modal analysis that support the *data and service levels of interoperability*.

Frameworks that comply with the UIMA specification may provide support for additional levels of interoperability for example at the *programming model and component levels*.

2.3.1 Data Level

UIM systems interoperate at the *data level* if they share standard *analysis data*. This level of interoperability is independent of how the data is produced, consumed or how the sharing is implemented. It is dependent only on the data specification. For example, one application may produce *analysis data* and store it to an agreed-upon location on disk where another application may read it. Another use-case may have two applications share the data over a socket connection. The key to interoperability at this level is that two or more processes share a standard data format.

2.3.2 Services Level

UIM Systems interoperate at the *services level* if they use standard service interfaces and protocols to implement and call analytics. These services may be implemented as, for example, SOAP services but must conform to standard service descriptions and comply with metadata specifications for describing the *analysis operations* that they implement.

This level of interoperability is dependent on the data level since the messages exchanged over the services would conform to the data specification. It adds services descriptions, metadata specifications describing behavior and SOAP bindings but is independent of how the services are implemented (e.g., as Java Classes, Java Beans, C++ programs, Perl scripts etc).

2.3.3 Programming Model Level

UIM systems interoperate at the programming model level if they share a standard set of programming language interfaces for implementing analytics and analysis data.

At this level it would be possible to develop a program in Java for example that calls an analytic whose implementation can be replaced by any other implementation of the specific Java interface. The analytic would comply with the UIMA specification as long is it provided compliant metadata describing the operations it preformed. This level of interoperability is dependent a common commitments to a programming language, API specifications, programming model and whatever other platform dependencies that may entail.

The UIMA Specification will not specify a means for programming-model level of interoperability.

Apache UIMA Notes

The Apache UIMA framework, however, provides a Java-based programming model for implementing UIMA analytics.

2.3.4 Component Level

This programming model level of interoperability does not provide a common means for transparently discovering and deploying components in different application environments. For example a Java class implementing an analytic may be transported to a target application environment other than the one in which is was developed, but will not run without properly aligning the class with the target environment to ensure it will locate, access and configure all its resource dependencies.

UIM systems interoperate at the *component level* if they can share a common means to bundle their programs as deployable objects. These are objects that once registered in a target environment through a standard means will be able to access and configure all their dependent resources without additional involvement by the application developer.

The UIMA Specification will not specify a means for component-level interoperability.

Apache UIMA Notes

The Apache UIMA framework is adopting the OSGi standard [OSGi1] to support this level of interoperability.

3 Design Goals

The UIMA Specification design goals focus on providing a standard specification for text and multi-modal analysis that supports the *data and service levels of interoperability* to facilitate the rapid combination and deployment of analytics in the development of UIM applications.

The proposed standard is intended specifically to facilitate object-oriented modeling and programming for creating component analytics and for the platform-independent discovery and composition of independently-developed analysis components or services.

1. **Data Representation.** Support the representation of *artifacts* and *artifact metadata* independently of *artifact modality* and *domain model*. UIMA should allow for the independent representation of different artifact modalities and domain models and not constrain the developer to any particular modality or domain model. Annotations that make-up artifact metadata and describe whole artifacts or regions thereof should be created and manipulated independently of the unstructured content or subject of analysis – annotations should be represented in a “stand-off” model to allow for multiple, potentially contradictory interpretations of the content and different representations of the same artifact to be manipulated independently.

2. **Data Modeling and Interchange.** Support the platform-independent interchange of *analysis data* in a form that facilitates object oriented modeling and programming based on UML and XMI standards. UIMA should facilitate the use of object oriented modeling and programming standards and tools to make it easy to define domain models for artifact metadata and manipulate artifacts and their metadata as standard object models.

3. **Discovery, Reuse and Composition.** Support the discovery, reuse and composition of independently-developed *analytics* to accelerate UIM applications development. UIMA should facilitate the common definition and interpretation of analytic capabilities to assist in the manual, semi-automatic and automatic discovery and composition of analytics to meet specific application requirements through a common representation of analytic metadata.

4. **Platform Independent Development.** Facilitate the compliance of existing applications or the development of new applications on different platforms and in different programming languages. It should be easy to comply with the UIMA specification by easily transforming XML data representations and/or wrapping existing analysis operations as compliant SOAP services for example.

5. **Out-of-the-Box Service-Level Interoperability.** Support the interoperability of independently developed *analytics* based on a common service description and associated SOAP bindings.

4 Use Cases

This section provides brief use-cases describing examples of the types of problems that a standard for interoperability of text and multi-modal analytics can help address. These use-cases were independently written to conclude with short list of requirements.

They are not intended to be exhaustive nor are all their asserted requirements addressed by the proposed standard. Rather this set of use-cases is intended to give a flavor for the range of applications that can benefit from easily reusing and composing independently-developed text and multi-modal analytics to analyze unstructured content.

We identified a set of requirement types common to these use-cases. We inserted a **bold label** at the beginning of similar requirements mentioned in these use-cases, which we think should be considered by a standard architecture for text and multi-modal analytics. These requirement labels include:

1. **Discovery, Reuse and Integration of Independently Developed Analytics**. Support for describing, discovering and combining into workflows, independently developed analytics based on a common set of component interfaces and component metadata.
2. **Search System Integration and Independence**. The ability to plug-in an arbitrary indexing mechanism for extracting artifact metadata resulting from analysis and indexing it in any independently-developed search engine.
3. **Multi-Source/Format Independence.** The ability to connect to any raw content source and deal with arbitrarily formatted content. The architecture, therefore, should not limit applications to a particular source or format but rather allow plug-ins to adapt any application appropriately.
4. **Multiple-Views**. The ability to generate different representations of the same artifact and to be able to analyze these representations independently.

5. **Muti-Modal**. The ability to support different modalities (e.g., text, video, speech) with a common representation scheme and within the same analysis workflow.
6. **Provenance**. Support for enabling the tracking the provenance of artifact metadata back to the original source data as well as to the analytics that produced it from those sources.
7. **Segmentation and Recombination**. The ability to segment artifacts into derivative artifacts and/or to recombine or merge them into new artifacts at any point in an analysis workflow.
8. **Stand-Off Annotations.** A representation system that allows for representing artifact metadata in a way that references and does not modify the original artifact data, allowing for multiple, possibly overlapping interpretations of an artifact.

4.1 Text analysis of email for SOX compliance

Motivated by several large corporate accounting scandals, including Enron, Tyco International, and WorldCom, the Sarbanes-Oxley Act (SOX) [http://en.wikipedia.org/wiki/Sarbanes-Oxley_Act] was enacted into law in 2002 to provide better oversight of corporate accounting, clearer financial reporting, and better protections for shareholders. Direct results of this law are significant new ethics, reporting, and records management requirements for public companies.

As part of their efforts to comply with SOX, Lots of Stuff, Inc. (LSI) would like to install an internal email analysis, alerting, indexing, and archiving system. LSI has 10,000 employees worldwide that together generate 100,000 email messages a day, with an average of 500 words (4K bytes) per email message. LSI's email system must analyze the 400MB of new email daily, look for activity that potentially violates SOX rules, alert internal auditors of potential violations, index the email and add it to a growing email archive, and periodically purge expired email according LSI's records retention policy. Moreover, the email index must support various queries that LSI may be required to perform. Such queries will come from government agencies that have the authority to request certain searches over LSI's email archive, as well as (perhaps more importantly) from internal LSI auditors who wish to catch potential SOX violations before they become a problem.

To satisfy these information processing requests, LSI needs to deploy text analysis components that can, at a minimum, detect within emails the occurrences of variety of named entities (e.g., persons, places, organizations, etc.) and relationships between these entities. The relationships might be based on named entity co-occurrence statistics, or they might be semantically derived and indicate more meaningful interactions or transactions between the named entities. For external SOX compliance, LSI would like to use off-the-shelf components that will supply the required analysis. For internal auditing, however, LSI would like to develop their own analytics that will work with and enhance the results produced by the off-the-shelf components. Moreover, as compliance regulations change, business needs change, and LSI's products and markets expand, LSI wants the flexibility to easily integrate new analytics and extend the functionality of the overall system.

The requirements are summarized below:

- **Discovery, Reuse and Integration of Independently Developed Analytics**. Off-the-shelf text analytics for named entity detection and relationship extraction, especially if they are already customized for the SOX compliance domain, and a standard mechanism for sharing data between them and integrating them within an application.

- **Search System Integration and Independence.** The ability to integrate a robust, commercial text search system that can index the analysis results and satisfy the query requirements for SOX compliance

4.2 Enterprise Semantic Search

A typical company has an enormous amount of stored information, and most of that information is unstructured. Employees need to be able to find information that is relevant to their jobs. There have been many attempts to employ traditional keyword search (of the sort used in popular WWW search engines) on company-internal documents. However, the WWW is very different from internal document repositories. For example:

1. The WWW is massively redundant; often the same information is expressed many different ways. Thus if a user can think of some words that are relevant to the desired information, there is a reasonable chance that some page with this information includes those words.
2. The WWW has a great deal of structure that is extremely useful in identifying important documents. Thus even if an enormous number of documents include a set of keywords *and* most of those documents are not useful, there is a reasonable chance that a useful document will appear near the top of the list of matches.

Effective searches over internal repositories need to be both broader in one sense (including multiple ways of expressing some concept) and more narrow in another (omitting hits that are useless but include words that are individually relevant). For example, a search for a *bank* that provides *checking* should match a document about some bank's overdraft protection service (even if the words *bank* and *checking* do not appear) and not match a document about checking whether erosion of a river bank poses a flood risk (even if the words *bank* and *checking* do appear).

Consider a fictional insurance brokerage named UICO (Unstructured Insurance Company). UICO employees need access to information from insurance policies, notes from field adjusters, emails from customers, etc. These sources lack the redundancy and structure that make keyword searching relatively effective. To be successful, UICO employees need to be able to specify what type of information they are looking without knowing specific instances, names or keywords. Some of the types of information they will need to find will include, for example, people, places, organizations, vehicles, events, times, accidents, causes etc.. Most documents stored in their databases do NOT include tags that uniquely identify these different types of information. Thus the documents provide neither explicit semantics classifying these elements nor specific instance-level knowledge. Consequently, tasks requiring discovery, vetting or analysis of this information become daunting manual efforts. Analytics or recognizers can however, automatically detect and index instances of these various types. Some of these analytics designed for recognizing people or organization names may be generally available while others targeting more specific types like vehicles, accidents or causes, may need to be developed by UICO's information technology staff.

Thus UICO needs an integrating architecture that provides the following capabilities:

- **Discovery, Reuse and Integration of Independently Developed Analytics**. Enables the easy combination of both custom and off-the-shelf analytics so that they can share data and easily work within the same application.
- **Search System Integration and Independence.** Enables the integration of analysis capabilities with an indexing and search facility that can exploit the metadata generated by the analytics to improve search results.

- **Multi-Source/Format Independence**. Supports processing of a heterogeneous and distributed corpus of documents (multiple file-formats, multiple types of content, located on multiple servers)

4.3 Cross language summarization of multimodal news feeds

In the global economy more and more companies can be directly affected by events happening around the world. Competitive product announcements, changes to the availability of critical resources, even cultural events may influence business decisions.

The Global Intelligent Information Company (GIIC) provides timely business intelligence by monitoring information sources on the web and on television and radio stations for matters of interest to their customers. Foreign broadcasts and web pages are translated into the customer's preferred language and analyzed with respect to customer needs. GIIC delivers summaries of relevant information along with links to specific segments in both the translated and raw content for examination.

In order to provide these services, GIIC must combine off-the-shelf analytics with customized analytics designed to find concepts unique to specific customers. An acceptable architecture must be able to provide the following capabilities:

- **Multi-Source/Format Independence.** Accept and analyze data from multi-modal sources including streaming video or audio sources, as well as web pages.
- **Multiple Views**. Represent different interpretations or views of a single artifact. For example, the video track, audio track and close-captions of a single video segment.
- **Multi-Modal.** Combine analytics for speech transcription, video processing, translation and text analysis.
- **Discovery, Reuse and Integration of Independently Developed Analytics**. Support assembly of different analytic pipelines for each mix of languages and custom analysis desired.
- **Discovery, Reuse and Integration of Independently Developed Analytics**. Easily replace analytics with independently-developed analytics to take advantage of ongoing advancements in audio, video and text processing products.

4.4 Populating a knowledge-base of metabolic pathways

Finding information in the vast volumes of the internet is a challenge. In the sciences today, it may be easier, faster, and cheaper to perform an experiment from scratch than it is to determine if an experiment has already been performed and, if so, to re-use the results. This is mainly a limitation on searchable citation databases, where keyword search does not give the scientist the ability to adequately express his or her information need.

In this use case, the (fictional) Metabolic Pathways Institute (MPI) is maintaining an on-line resource for publishing information about metabolic pathways [http://en.wikipedia.org/wiki/Metabolic_pathway]. By tracking the continuing flood of medical literature, they hope to keep a knowledge-base (KB) up to date with the latest experimental results on this subject. The goal is that scientists and practitioners interested in information on some specific pathway can check the knowledge-base and get immediate answers.

The MPI publishes an ontology in OWL [OWL1] for this domain, which identifies the basic classes (e.g., cell, enzyme, pathway) and relations (e.g., metabolizes, catalyzes, partOf), and

allows queries in SPARQL [SPARQL] over the KB. For example, they imagine being able to easily translate the question "Can lactate be used in Gluconeogenesis?" into a SPARQL query.

There are so many possible sources for this information, so many places that scientists publish results on research into new metabolic pathways, and so much other unrelated information being published through the same sources, that keeping the KB up to date is an impossible task. However, a lot of the linguistic patterns used to describe experimental results are easy to recognize, as long as the basic ingredients (like enzymes, proteins, etc.) can be identified. Using automation to support the production of this KB is an attractive and plausible approach.

From a technical perspective, building such an application requires a variety of capabilities including for example:

- **Discovery, Reuse and Integration of Analytics**. Recognizing enzymes, proteins, biochemical processes, etc., in text. Several analytics exist to recognize chemical formulae and protein names, but these *components must be combined and augmented* to recognize the full set of relevant terms.
- **Discovery, Reuse and Integration of Analytics**. Recognizing the relations between them as described in the text (e.g., lactate convertsTo pyruvate), given that the types (e.g., enzymes, proteins) have been identified. Relation recognition can be implemented as analytic components, but in addition to the text, these *components must take as input the results of other analytics*, in this case the chemical formulae and protein recognition, and *must be able to map between the type systems of different input components*.
- **Stand-Off Annotations.** Recognizing and combining multiple references to the same entities (e.g., lactate, lactic acid) and relations (e.g., lactic acid convertsTo pyruvate) within and across documents. This is a critical requirement for the MPI's KB, if the same result is published in multiple places it should only be in the KB once. Analytics to perform this processing must be able to *consider data produced by other analytics and across different documents*.
- Generation of data in RDF that populates an OWL ontology. Components that perform this generation must be able to *map between the semantics of the analytic type systems and an ontology* in OWL, and they must be able to *store the results persistently*.
- Application of deductive reasoning. Some of the desired information is not explicitly in any text, but is easily deducible from existing knowledge. For example, lactate can be used in Gluconeogenesis, but only after it is converted to pyruvate. This is fairly common knowledge in the medical community, and a system that didn't take this basic knowledge into account would not meet the MPI's requirements. It must be possible to *make use of existing sources of knowledge* to augment the analytic process.
- **Provenance.** Often when an answer is found the customer will insist on knowing where the data came from. It will not be enough for the MPI to publish just the answers, but the articles from which they were extracted. It must be possible to *record and trace the sources of information in all analytics*.

4.5 Video Segmentation for Just-in-Time Learning

Segmenting instructional videos for reuse and repurposing can dramatically facilitate just-in-time learning. Videos require a long time to produce and are expensive. On one hand, educational videos tend to be long - 30 minutes or more - and cover more then one topic. On another hand, classroom teachers prefer to show short, one-topic videos to demonstrate their point. In addition, e-learning course creators often need to create courses in a very limited time to respond to new demands, for example news in pandemics or anti-terrorism procedures. Short, narrow-focused

videos in these cases better target the learning requirement. The learning community recognizes the importance of video segmentation and addresses them in different ways. For example University of North Carolina at Chapel Hill is working on the Open Video project http://www.open-video.org/project_info.php.

"The purpose of the Open Video Project is to collect and make available a repository of digitized video content for the digital video, multimedia retrieval, digital library, and other research communities. Researchers can use the video to study a wide range of problems, such as tests of algorithms for automatic segmentation, summarization, and creation of surrogates that describe video content; the development of face recognition algorithms; or creating and evaluating interfaces that display result sets from multimedia queries."

Currently scientists of UNC manually segment and annotate the videos; this is a laborious task, taking into the account that the Open Video website itself contains hundreds of videos.

There are existing off-the shelf analytics, which can recognize audio and video characteristics of the video, such as "a person", "text", or "music" and "speech", which would be deployed in video segmentation process. In addition, many specialized text analysis tools have been developed, which can be utilized in analyzing video text transcripts. Based on the availability of these kinds of analytics, the following are some requirements that a framework for supporting video processing for just-in-time learning.

- **Segmentation and Recombination.** Support the decomposition of artifacts into component parts where each can be individually analyzed and then recombined.
- **Stand-off Annotations.** Support chronological annotations of segments, identifying unit of time and accuracy, for example the segment number n starts on m-th *second*, lasts l-seconds, where the accuracy is r seconds.
- **Provenance.** Retain segment's source information after the segment is extracted from the original video.
- **Discovery, Reuse and Integration of Analytics**. Find, Reuse and combine (plug-n-play) independently-developed audio, video and text analytics to segment, annotate to recombine videos to take advantage of ongoing advancements in audio, video and text processing products.
- **Multi-Source/Format Independence**. Accept and analyze data from differently formatted multi-modal sources including streaming video, audio and text sources.
- **Multiple Views.** Represent different interpretations or views of a single artifact. For example, the video track, audio track and closed-captions of a single video segment.

5 Proposed Elements of a Standard Specification

In this section we propose eight elements of a standard architecture specification for interoperable text and multi-modal analytics. They are listed in brief below.

1. **Common Analysis Structure (CAS) Specification.** Provides a simple and extensible model for representing *analysis data* as a standard object model that may be easily instantiated and manipulated in object oriented programming systems. This element of the specification is provided as a UML [UML1] model. We propose adopting the XML Metadata Interchange (XMI) specification [XMI1][XMI2] to provide a standard means for representing *analysis data* as an XML document.

2. **Type-System Language Specification.** Provides a standard means for associating object model semantics with *artifact metadata* that complies with object modeling standards. The proposal is to use Ecore as the type system language. Ecore is the modeling language used in the Eclipse Modeling Framework [EMF1] and is tightly aligned with the OMG's EMOF standard[1]. We provide an example illustrating how an Ecore Type-System is represented as an XMI documents to support XML-based representation and interchange of Type-Systems.

3. **Type-System Base Model**. Provides a standard and extensible set of domain-independent types generally useful for analyzing unstructured information. For example we define a type `Annotation` to represent objects that have ***regional 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. **The Behavioral Metadata Specification**. Provides a standard declarative means for describing the capabilities of analysis operations in terms of what types of CASs they can process, what elements in a CAS they can analyze, and what sorts of effects they would have on CAS contents as a result. While we do not provide a complete and formal specification for Behavioral Metadata, we provide a discussion of requirements and a rough proposal appealing to the OCL standard [OCL1]. We demonstrate how this proposal may provide a generalization for the behavioral metadata that has been implemented in Apache UIMA.

5. **Processing Element Metadata Specification.** Provides a standard declarative means for describing identification, configuration and behavioral information about Processing Elements (analytics and flow controllers). This specification is represented as a UML Model, and we use the XMI standard to represent the processing element metadata as XML. This section of the specification refers to the Behavioral Metadata Specification to represent an processing element's behavioral information.

6. **Abstract Interfaces.** Abstractly describes the interfaces to the two different types of Processing Elements, namely, Analytics and Flow Controllers. These abstract interfaces are specified with a UML model.

7. **WSDL Service Descriptions.** Provides a standard means for describing Processing Elements as web services using WSDL [WSDL1]. We also define a standard SOAP binding.

8. **Aggregate Analytic Descriptor Specification.** Provides a standard declarative means for an aggregate analytic to

    a. refer to its constituent analytics
    b. identify a flow controller which determines the order in which the constituent analytics of the aggregate are invoked on a CAS.
    c. define mappings to facilitate the composition of independently-developed analytics.

[1] There are small mostly naming differences between EMOF and Ecore. The Eclipse Modeling Framework transparently reads and writes EMOF.

The Aggregate Descriptor Specification is defined by a UML model, and we again use XMI to represent the descriptor in XML.

We discuss each proposed element in a subsection below.

Formatting Conventions
Formatting conventions used in this specification

- ***Italicize and bold*** first defined use of a term.
- *Italicize and camelCase* model elements -- classes, attributes, and references.
- `CapitalizedCourierFont` for UIMA base and example types
- `courierFont` for XML, code snippets, variables and instance names

5.1 The CAS Specification

The Common Analysis Structure or **CAS** is the common data structure shared by all UIMA analytics to represent the unstructured information being analyzed or 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 provides a domain neutral, object-based representation scheme that is aligned with UML and XML standards.

In this section we describe a formal model for the UIMA CAS.

5.1.1 Basic Structure: Objects and Slots

The Common Analysis Structure (CAS) is a data structure for representing and sharing *analysis data* (i.e., an artifact and its metadata) among *analytics*.

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 CAS.

This approach allows UIMA to adopt general object-oriented modeling and programming standards for representing and manipulating *analysis data*.

In particular we propose adopting the Unified Modeling Language (UML) 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* is 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***.

The relationship between the analysis data represented by a CAS and a *type system* is described in the next section.

5.1.2 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*. The language we use to specify a type system is Ecore [EMF1]. The UIMA Type-System language and how it is used to define the structure of a CAS is described in section 5.2 The Type-System Language Specification.

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 f*eatures* with an *upper bound* greater than one as *multi-valued features.*[2]

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. It is discussed in more detail in Section 5.2 The Type-System Language Specification.

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.

> **Candidate Compliance Point:** UIMA components/frameworks that create or modify a CAS might be required to ensure that the objects in the CAS conform to the type system as determined by the Ecore specification.

---

[2] In a particular programming model there may be a choice as to how a multi-valued feature is implemented, for example as an array or a linked list. This is a framework implementation detail and is not specified in the type system.

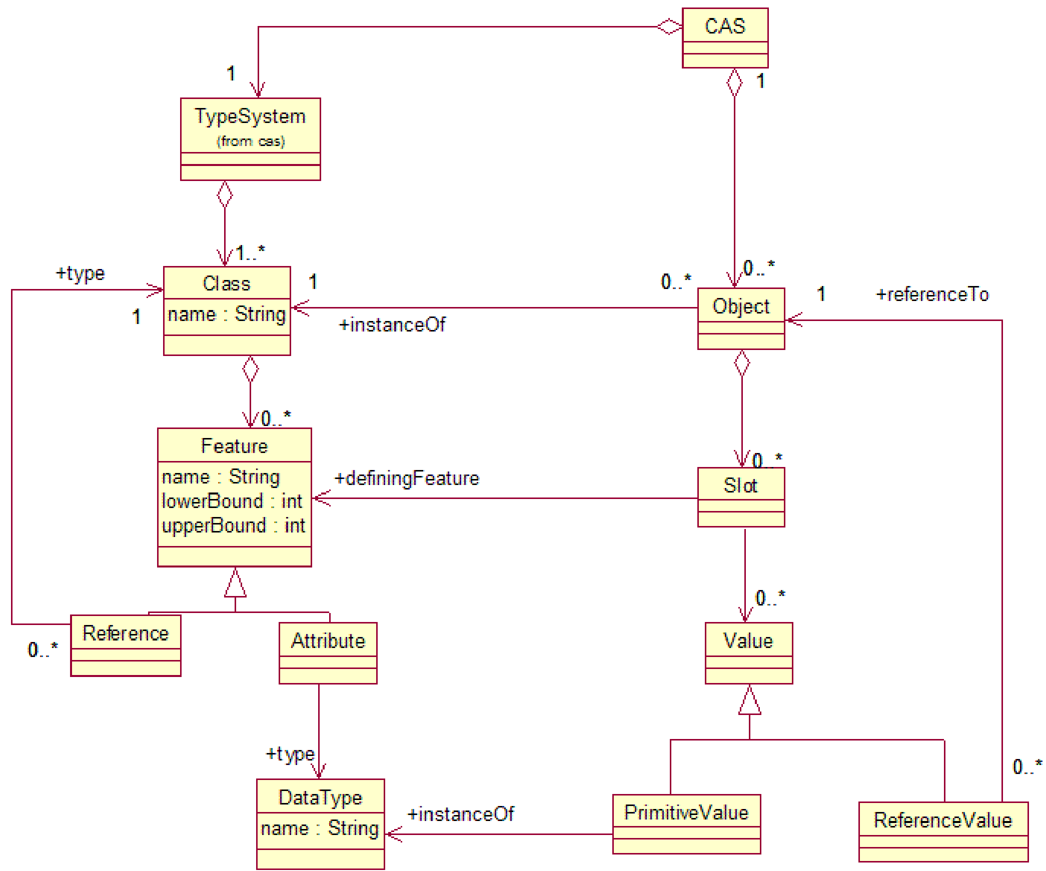

Figure 1: CAS Specification UML

Apache UIMA Notes

Apache UIMA implemented its own type-system representation language prior to adopting Ecore as the UIMA specification. The original Apache UIMA type system representation uses some different terminology:

- Apache UIMA does not use the term "Class", instead using the term "Type" to refer to both primitive and nonprimitive types.
- The type of a feature is referred to in Apache UIMA as the feature's "range type".
- Apache UIMA uses the term "FeatureStructure" to refer to what this specification calls an "Object."

Also, Apache UIMA's original type-system representation language does not directly implement multi-valued features. Instead it has explicit array and list types.

The current version of Apache UIMA now supports the proposed UIMA standard type-system language (i.e., Ecore). We use an Ecore "Annotations" (arbitrary tags attached to Ecore model elements) to record Apache UIMA-specific information such as whether an array or list will be used to implement a multi-valued feature.

5.1.3 Introduction to Annotations and Sofas

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.

Stand-off Annotations

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 the analytic includes 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, and we propose a **"stand-off" model** for annotations.

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 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.

In the remainder of this section we describe in greater detail *annotation*, *subject of analysis* and *regional reference* as part of our discussion of the CAS. These concepts are formally defined as elements of the base type-system model in section 5.3 Type-System Base Model."

An ***Annotation*** is a type of object that has ***regional 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.

The annotated content is referred to as the *annotation's* ***subject of analysis***, abbreviated "***sofa***".

Figure 2 illustrates an example. The CAS contains an *object* of class `Document` with a *slot* `textString` containing the string value, "Fred Center is the CEO of Center Micros."

A `Person` annotation then refers to a particular range of character offsets within that text string, for example, denoting the substring "Fred Center". The string value in the `textString` *slot* in this example is the subject of analysis (sofa) of the `Person` annotation. The substring "Fred Center" is the annotated region within that sofa.

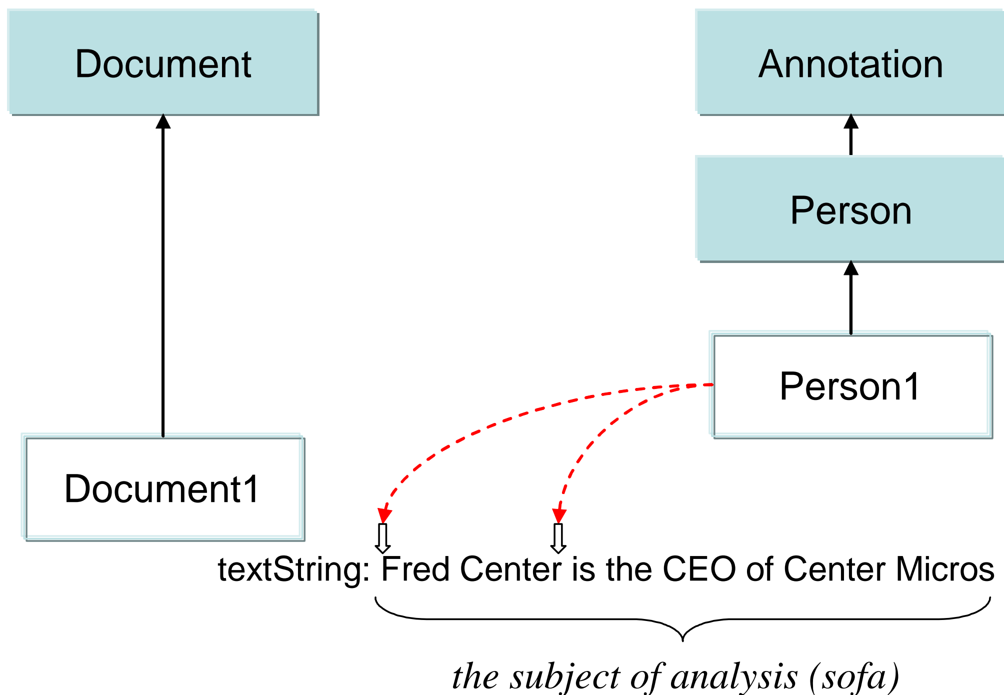

Figure 2: Annotation and Subject of Analysis
Note: solid boxes indicate types (classes), hollow boxes indicate instances (objects).

All *annotation* types must indicate their subject of analysis by providing a ***sofa*** *feature*. 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 provides a base type called `SofaReference` for referring to sofas from annotations. The SofaReference also allows an annotation to refer to a

subject of analysis that is not located in the CAS. See Section 5.3 Type-System Base Model for details.

> **Candidate Compliance Point:** A UIMA component/framework may be "annotation model compliant" if it uses this definition of Annotation and Sofa Reference as defined by the UIMA Type-System base model.

A regional reference provides a mechanism for denoting a region of a sofa. 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 regional references. For example a text-based regional reference may simply define two features *begin* and *end* to indicate a contiguous region in a text string.

UIMA defines a general `Annotation` type in the Type System Base Model. It uses the `SofaReference` to refer to its sofas. Any subtype of the `Annotation` base type may define how its *regional reference* is implemented.

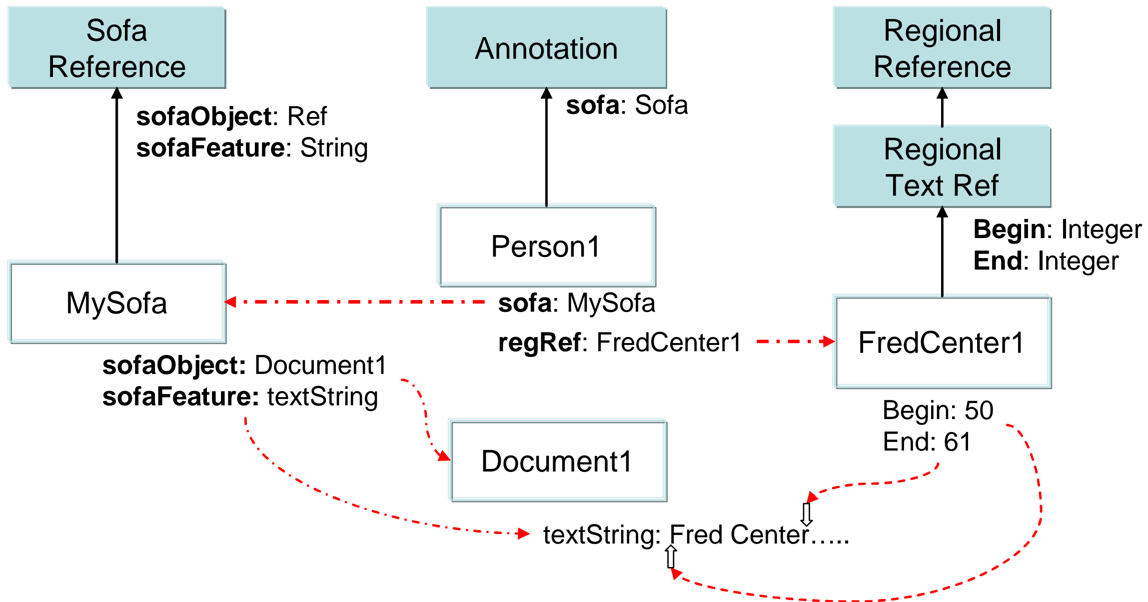

Figure 3: Sofa Reference and Regional Reference
Note: solid boxes indicate types (classes), hollow boxes indicate instances (objects).

Figure 3 expands on the previous example, showing how the Sofa Reference and Regional Reference are represented. The Annotation "Person1" has a subject of analysis is the slot "textString" defined on object "Document1". An instance of the SofaReference type, labeled "MySofa", is used to establish a reference from "Person1" to the text string. The figure also introduces a particular subtype of RegionalReference called "Regional Text Ref", which allows the annotation to point to a particular range of characters within its subject of analysis.

5.1.4 The Standard XMI CAS Representation

In his section we describe how a UIMA CAS is represented as an XML document based on the standard XMI (XML Metadata Interchange) specification [XMI1, XMI2]. XMI is an OMG standard for expressing object graphs in XML.

> **Candidate Compliance Point:** A compliant UIMA component/framework may be required to produce and consume the Standard XMI CAS Representation. Other CAS representations may also be supported, but to allow interoperability between implementations the Standard XMI CAS Representation must be supported.

5.1.4.1 XMI Tag

The outermost tag may be `<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.

5.1.4.2 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:Foo xmi:id="1"/>
  ...
</xmi:XMI>
```

This XMI document contains an object whose class is named `Foo`. The `Foo` 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 do not have 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.

Apache UIMA Notes

In Apache UIMA we follow the EMF convention that a Java-style package name is converted to an XML namespace URI by the following rule:

replace all dots with slashes,
prepend http:///, and
append .ecore.

So for example the Java package name `org.myorg` would be converted to the XML namespace URI `http:///org/myorg.ecore`.

5.1.4.3 Attributes (Primitive Features)

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

For example, an *object* of *class* `Foo`, with slots:

```
begin = 14
end = 19
myString = "bar"
```

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:Foo xmi:id="1" begin="14" end="19" myString="bar"/>
  ...
</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:Foo xmi:id="1">
    <begin>14</begin>
    <end>19</end>
    <myString>bar</myString>
```

```
  </myorg:Foo>
  ...
</xmi:XMI>
```

The attribute serialization is preferred for compactness, but either representation is allowed. Mixing the two styles is allowed; some *features* can be represented as attributes and others as elements.

5.1.4.4 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 `Baz`, with *feature* `myFoo` that is a reference to the `Foo` 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:Foo xmi:id="1" begin="14" end="19" myFeature="bar"/>
  <myorg:Baz xmi:id="2" myFoo="1"/>
  ...
</xmi:XMI>
```

As with primitive-valued *features*, it is permitted to use an element rather than an attribute. However, the XMI spec defines a slightly different syntax for this as is illustrated in this example:

```
<myorg:Baz xmi:id="2">
  <myFoo href="#1"/>
<myorg.Baz>
```

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.

5.1.4.5 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.[3]

---

[3] 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.

Multi-valued *references* serialized in a similar way. For example an *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.

Apache UIMA Notes

As noted previously, Apache UIMA's original type-system representation language does not directly implement multi-valued features. Instead it has array and list types. For a feature whose range type is one of Apache UIMA's array or list types, it is usually appropriate to serialize this to XMI as a multi-valued feature. However, since arrays and lists are first-class objects in Apache UIMA, it is possible to have multiple references to the same array or list, which is not compatible with the multi-valued feature representation.

To address this, the Apache UIMA Type System Description has an additional attribute `multipleReferencesAllowed` that can be set for a feature. An array or list with `multipleReferencesAllowed = false` (the default) is serialized as a multi-valued feature in XMI. An array or list with `multipleReferencesAllowed = true` is serialized as a separate object and referenced from the containing object.

Apache UIMA v1.4 and later support the proposed UIMA standard type-system language (i.e., Ecore). This ambiguity does not arise for type systems developed directly in Ecore.

5.1.4.6 Linking an XMI Document to its Ecore Type System

As mentioned in section 5.1.2, the structure of a CAS is defined by a UIMA type system which is represented by an Ecore model.

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:

```
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].

5.1.4.7 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.

5.2 The Type-System Language Specification

One of the design goals for the UIMA specification is to *support object-oriented modeling and programming paradigms.* In these paradigms objects are described by some schema, specifically a *class model*, which defines the basic structure of every object, determining its attributes and the types of values that may fill them.

To address this design goal, the *artifact metadata*, represented as objects in the CAS, must conform to a user-defined schema compatible with object-oriented modeling and programming. We call this schema 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.

A UIMA design goal is to support an object-oriented representation that is independent of any particular domain model. UIMA Type Systems are user-defined. UIMA does <u>not</u> specify any particular set of types. 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*. These implement *analysis operations*, the behavior of which may be specified in terms of type

definitions expressed in a type system language. For example every *analysis operation* 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 5.4 Behavioral Metadata Specification). 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.

5.2.1 Ecore as the UIMA Type System Language

Rather than invent a language for defining UIMA Type System, 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 [MOF1] and the essential MOF or EMOF [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 5.2.3.

We propose adopting Ecore as UIMA's type system representation language. Ecore addresses the design goal to support object oriented modeling and programming, offers the obvious benefits of using a standard and because of the direct benefits associated with using EMF tooling. For example, EMF includes tooling that will generate an Ecore Type System for a UML diagram and will generate the associated Class definitions.

> **Candidate Compliance Point:** A compliant UIMA component/framework that outputs a CAS may be required to use Ecore to define the types of objects in that CAS.

5.2.2 An Introduction to Ecore

Ecore is well described in 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.

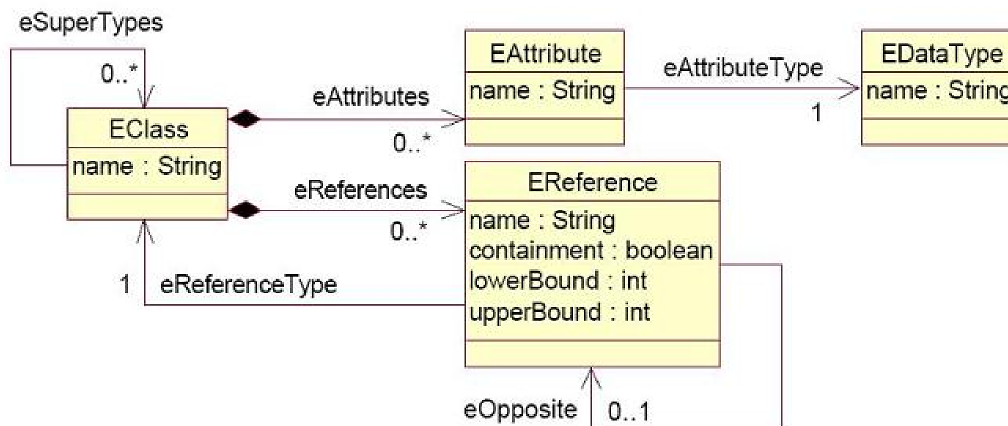

Figure 4: The Ecore Kernel

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.

5.2.3 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].

5.2.4 ECore Examples

Figure 5 shows a simple example of an object model in UML. This model describes two types of Named Entities: `Person` and `Place`. They may participate in an `At` relation (i.e., a `Person` is located at a particular `Place`).

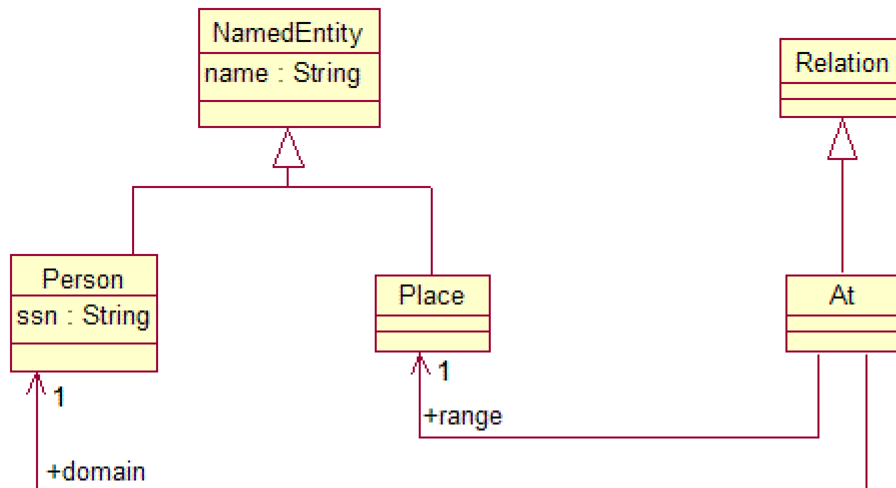

Figure 5: Example UML Model

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:

```
<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="example" nsURI="http:///example.ecore" nsPrefix="example">
  <eClassifiers xsi:type="ecore:EClass" name="NamedEntity">
    <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"/>
  <eClassifiers xsi:type="ecore:EClass" name="Person"
    eSuperTypes="#//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="At"
```

```
    eSuperTypes="#//Relation">
    <eStructuralFeatures xsi:type="ecore:EReference" name="domain"
      lowerBound="1" eType="#//Person"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="range"
      lowerBound="1" eType="#//Place"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Place"
    eSuperTypes="#//NamedEntity"/>
</ecore:EPackage>
```

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

5.3 Type-System Base Model

The UIMA Type-System Base Model is the set of predefined types assumed common across all UIMA-compliant analytics, applications and frameworks.

The XML namespace for types defines 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 5.3.1, whose namespace is defined by Ecore.)

5.3.1 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).

> **Candidate Compliance Point:** A compliant UIMA component/framework may be required to understand this set of primitive types, and may be required to treat `EObject` as the superclass of all classes.

5.3.2 Annotation

`Annotation` is a *class* that has one *feature* named `sofa`. The `sofa` feature of an `Annotation` is intended to point to any slot of any object that acts as the `Annotation`'s *subject of analysis*.

However, since Ecore does not define a way to refer directly to a slot, UIMA defines a base type called `SofaReference` that provides a standard way to refer to *sofas* from annotations. This is depicted in Figure 6. The `SofaReference` type is discussed in more detail in the next section.

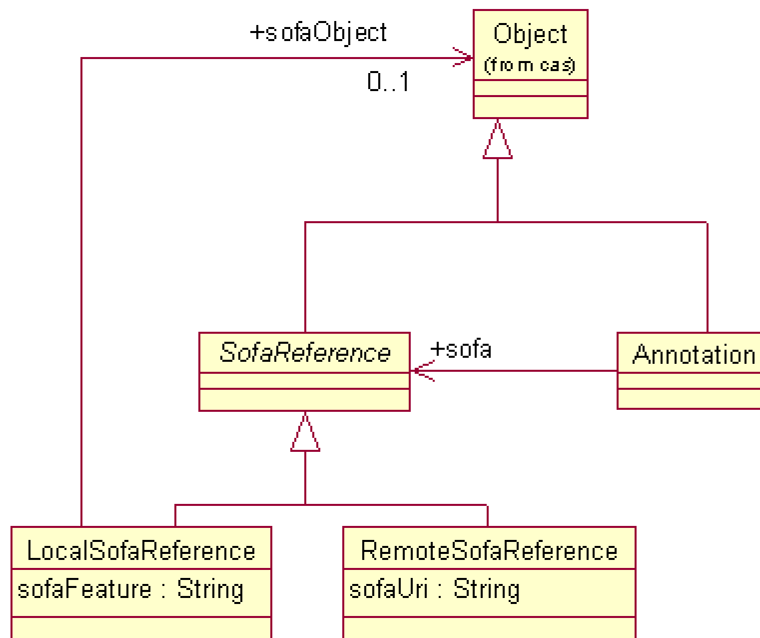

Figure 6: Annotation and SofaReference Base Types

5.3.3 SofaReference

A `SofaReference` is used by an Annotation to refer to its subject of analysis. There are two subtypes: A `LocalSofaReference` is a reference to a slot of another object in the CAS. A `RemoteSofaReference` is a URI to content that is not contained in the CAS.

> **Candidate Compliance Point:** A UIMA component/framework that is "annotation model compliant" may be required to adhere to the constraint that all `Annotation` objects must have a `sofa` slot that holds a reference to either a `LocalSofaReference` or a `RemoteSofaReference`.

5.3.3.1 LocalSofaReference

We first consider local sofa references. As mentioned above, Ecore does not provide a way to point directly to a slot. One straightforward implementation of a local sofa reference would be to identify an *object* and the name of its *feature* as two separate slots on `Annotation`.

For example consider a quotation *object* with two *features* "text" and "author"

```
  <ex:Quotation xmi:id="1"
    text="If we begin in certainties, we shall end with doubts; but if
we begin with doubts and are patient with them, we shall end in
certainties."
    author="Francis Bacon"/>
```

If we produced annotations over this quotation, each annotation would have to not only reference the `Quotation` object but also indicate which *feature* (`text` or `author`) it was annotating. For example:

```
<ex:Clause sofaObject="1" sofaFeature="text" begin="0" end="30"/>
<ex:Pronoun sofaObject="1" sofaFeature="text" begin="3" end="5"/>
<ex:Pronoun sofaObject="1" sofaFeature="text" begin="29" end="31"/>
...
```

In typical use-cases, however, many annotations point to different regions of the same sofa. Repeating the `sofaFeature` slot in each annotation is space inefficient.

To address this issue, UIMA defines the type `LocalSofaReference`. The intent is that it is used as a standard way to bridge between an `Annotation` and the region of the sofa it annotates. It introduces a level of indirection and creates an extra object. The benefit however is that for the typical use cases it reduces the space required for sofa references.

As shown in Figure 6, `LocalSofaReference` defines two *features*, `sofaObject` and `sofaFeature`. The `sofaObject` *feature* can be a reference to any *object* in the CAS, and `sofaFeature` is a string which must name the *feature* of that object which contains the ***sofa data*** (the data into which the annotation points).

The example above using the `SofaReference` becomes:

```
  <ex:Quotation xmi:id="1"
    text="If we begin in certainties, we shall end with doubts; but if
we begin with doubts and are patient with them, we shall end in
certainties."
    author="Francis Bacon"/>

  <cas:SofaReference xmi:id="2" sofaObject="1" sofaFeature="text"/>

  <ex:Clause sofa="2" begin="0" end="30"/>
  <ex:Pronoun sofa="2" begin="3" end="5"/>
  <ex:Pronoun sofa="2" begin="29" end="31"/>
```

5.3.3.2 RemoteSofaReference

The other subtype of `SofaReference` is a `RemoteSofaReference`, which has just one feature, `sofaUri`. This feature contains the URI to sofa data that does not reside in the CAS.

Remote Sofa References are particularly important for large binary artifacts (for example, video frames) that are inconvenient to store and ship around in XMI.

5.3.4 Other Candidate Base Types

Our experience implementing and applying UIMA has suggested other types we found useful for variety of use-cases. We propose that they be considered for inclusion in UIMA Base Type Model. We discuss each in turn.

5.3.4.1 RegionalReference

Annotations point to regions of artifact data using some application-specific mechanism. In this specification we cannot define a single mechanism for implementing regional references for all types of artifact data.

However, an open issue is whether to introduce into UIMA an abstract `RegionalReference` type that could be subclassed by applications. An example of a type system that uses the `RegionalReference` type is illustrated in Figure 7.

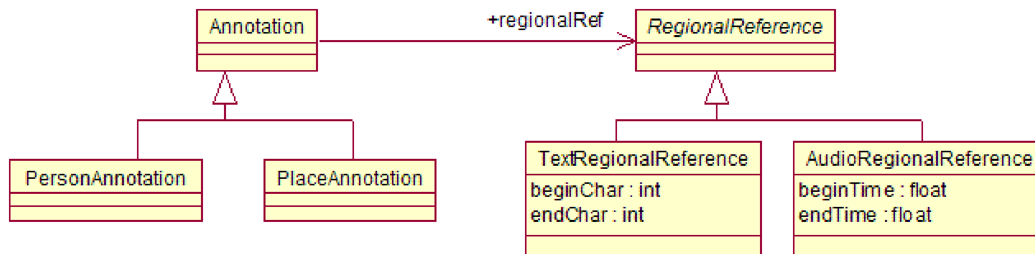

Figure 7: Example Type System with Regional Reference

This type system could be used by an analytic that can detect mentions of Person or Place names within either a text document or a segment of audio speech.

If for example a `PersonAnnotation` refers to a text subject of analysis, then its `regionalRef` feature would point to an instance of `TextRegionalReference` that indicated the span of character offsets where the person's name occurs. If instead a `PersonAnnotation` refers to an audio subject of analysis, then its `regionalRef` feature would point to an instance of `AudioRegionalReference` that indicated the time span where the person's name was spoken.

This kind of indirection between annotations and the method used to denote regions of the artifact data was used in ATLAS [Laprun1].

We can also imagine different types of regional references even within a single type of artifact. For example for text artifacts a framework or application could define a subtype `UTF16RegionalReference` with `begin` and `end` features that hold UTF-16 code unit offsets into a Unicode string, and a separate type `UnicodeCharacterRegionalReference` which uses true Unicode 3.0 character offsets. Regional reference types for noncontiguous spans would also be possible.

The alternative is to not have a separate `RegionalReference` type and instead include the offset features directly on the annotation types. This would lead to a type system such as the one shown in Figure 8.

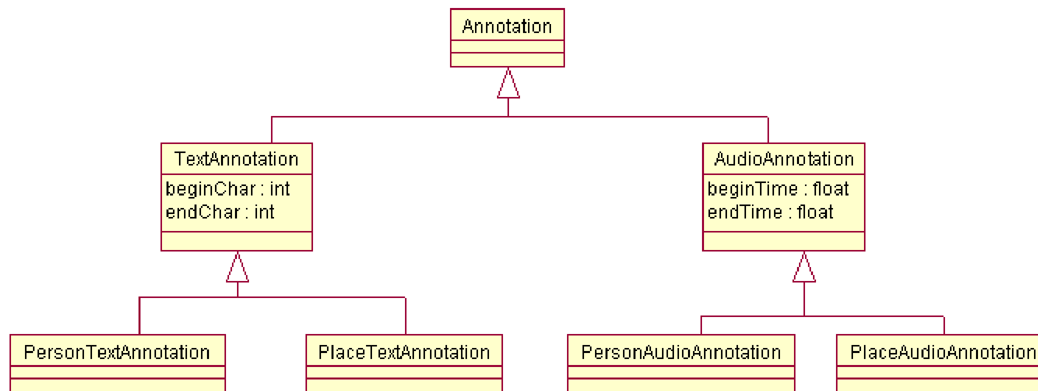

Figure 8: Example Type System without Regional Reference

Each alternative has its advantages and disadvantages.

The main advantage of using the RegionalReference (Figure 7) is that it is a more normalized model, which makes it easy to add more annotation or regional reference subtypes. If we enhance our analytic to detect an additional type (e.g., Organization) we only have to add one additional type in Figure 7, but we would have to add two additional types in Figure 8 (`OrganizationTextAnnotation` and `OrganizationAudioAnnotation`). As we add more regional reference types this explosion becomes much more severe.

On the other hand, there are advantages to having the separate `PersonTextAnnotation` and `PersonAudioAnnotation` types as in Figure 8. The fact that it's known that an annotation is a kind of `TextAnnotation` means that useful operations can be performed on the `PersonAnnotation` type - for example iterating over PersonAnnotations in the order they appear in the document, getting a subiterator that iterates over annotations within the span of the `PersonAnnotation`, or getting the covered text. This seems to be a fairly natural model for our analytic developers. If instead we have a `PersonAnnotation` with a separate `RegionalReference`, then a `PersonAnnotation` now would mean the mention of a person in any Sofa, using any means of identifying offsets. That's a much more general meaning and would make performing operations on the `PersonAnnotation` type more complicated. Perhaps the iteration, getCoveredText, etc. operations are now over RegionalReferences - but now developers have to have a mental model that's a little more complicated - e.g., they might have to call PersonAnnotation.getRegionalReference(), then check if the regional reference is actually to a text document, and if so try to call getCoveredText() on it.

There is also an implementation issue with the Annotation/RegionalReference separation – it would require twice as many objects in the CAS, which could be significant for performance-critical systems.

Because of these issues, we have decided not to mandate the use of separate `RegionalReference` objects in this specification. Developers are free to design their type systems with or without separate `RegionalReference` objects. However, we may include the abstract `RegionalReference` in the base type system and provide some best-practices guidance as to its appropriate usage.

Apache UIMA Notes

Apache UIMA does not implement a separate `RegionalReference` type. Instead, for text annotations Apache UIMA defines a type named `uima.tcas.Annotation` that contains the features `begin` and `end`. These are intended to represents off-sets into the text string specified by the annotation's `sofa` feature. The type `uima.tcas.Annotation`, however, is not extensible to non-text artifacts. Furthermore, the `begin` and `end` features are UTF 16 code units which is not convenient for anyone using UTF-8 for example.

5.3.4.2 View

A `View`, depicted in Figure 9, 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.

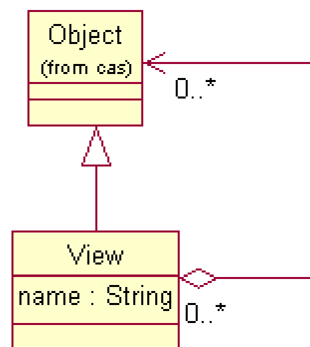

Figure 9: View Type

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*.

In our proposed definition the members of a view are those *objects* explicitly asserted to be contained in the View. Referring to the UML in Figure 9, 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.

The following XMI fragment shows a View that contains a subset of *objects* in the CAS:

```
  <ex:Quotation xmi:id="1"
    text="If we begin in certainties, we shall end with doubts; but if
we begin with doubts and are patient with them, we shall end in
certainties."
    author="Francis Bacon"/>

  <cas:SofaReference xmi:id="2" sofaObject="1" sofaFeature="text"/>

  <ex:Clause xmi:id="3" sofa="2" begin="0" end="52"/>
  <ex:Pronoun xmi:id="4" sofa="2" begin="3" end="5" lemma="6"/>
  <ex:Pronoun xmi:id="5" sofa="2" begin="29" end="31" lemma="6"/>
  <ex:Lemma xmi:id="6" base="I" person="1" number="p"/>

  <cas:View name="AllPronouns" members="4 5"/>
```

In this example the intent is for the View to contain all of the Pronoun annotations found in this subject of analysis . The Lemma *object* is not a member of the view; however, the Lemma *object* is referenced from members of the View , so it is in the *reference closure* of the view.

5.3.4.3 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[4] 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[5] 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.

---

[4] It is still undecided as to whether an Anchored View is best modeled as a subtype of View or that View should contain an optionally instantiated feature sofa. The latter would allow any view to act as an Anchored View; it would be up to the implementation to enforce the constraint on all contained Annotations.

[5] By members we mean as defined above, those Objects directed asserted to be in the View not necessarily Objects that may be in the Views reference closure and not in the View.

Figure 10 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.

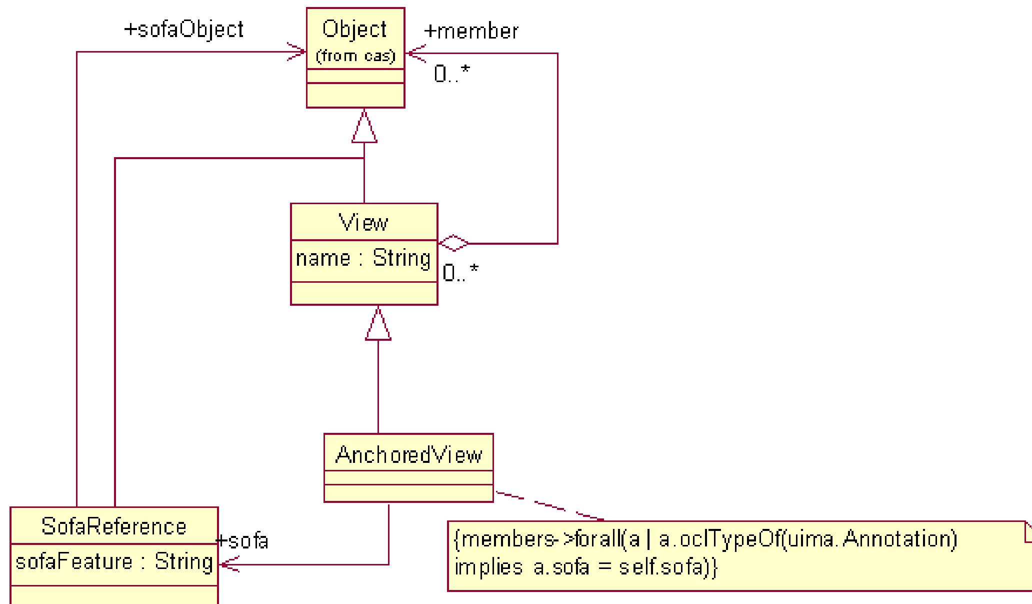

Figure 10: Anchored View Type

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 5.4). 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.

The following XMI fragment shows a CAS with two sofas and two AnchoredViews, one over each sofa. The AnchoredViews provide an efficient way to find all of the annotations that annotate a particular sofa.

```
  <ex:Quotation xmi:id="1"
    text="If we begin in certainties, we shall end with doubts; but if
we begin with doubts and are patient with them, we shall end in
certainties."
    author="Francis Bacon"/>

  <cas:SofaReference xmi:id="2" sofaObject="1" sofaFeature="text"/>
```

```
  <ex:Pronoun xmi:id="4" sofa="2" begin="3" end="5" lemma="6"/>
  <ex:Pronoun xmi:id="5" sofa="2" begin="29" end="31" lemma="6"/>
  <ex:Lemma xmi:id="6" base="I" person="1" number="p"/>

  <uima:AnchoredView sofa="2" members="3 4 5"/>

  <ex:Quotation xmi:id="7"
    text="The only limit to our realization of tomorrow will be our
doubts of today."
    author="Franklin D. Roosevelt"/>

  <cas:SofaReference xmi:id="8" sofaObject="7" sofaFeature="text"/>

  <ex:Pronoun xmi:id="9" sofa="8" begin="18" end="21" lemma="11"/>
  <ex:Pronoun xmi:id="10" sofa="8" begin="54" end="57" lemma="11"/>
  <ex:Lemma xmi:id="11" base="my" person="1" number="p"/>

  <cas:AnchoredView sofa="7" members="9 10 11"/>
```

Both of these views are valid anchored views because all members of the view annotate the view's sofa. In contrast, consider a view that contained all of the `Pronoun` annotations in the CAS. Such a view would not be a valid anchored view.

Apache UIMA Notes

1. In Apache UIMA all views are `AnchoredViews`. The `sofa` feature of a `View` points to an instance of the `Sofa` type. There is exactly one `View` per `Sofa`. The intention is that a `View` contains all objects that are relevant to its `Sofa`.

2. Apache UIMA enforces the anchored view constraint (that all annotations in the view refer to the same sofa as the view itself), and most Apache UIMA analytics rely on the assumption that the constraint is satisfied.

3. Apache UIMA defines CAS APIs that operate specifically on Views. For example there is a method CAS.getView(Sofa) through which an annotator can get the View containing objects relevant to a particular Sofa.

4. Apache UIMA defines *view indexes*, which provide efficient iteration over the members of the view according to a sort order defined declaratively by the user. (Note this is an index over the contents of a single CAS View, and is not the same as for example an inverted file index that indexes the contents of multiple CASes.) For example Apache UIMA annotators frequently use view indexes to iterate over Annotations in a text document, in order from the beginning of the document to the end.

5. In Apache UIMA, the API to a CAS is the same as the API to a View. (This can be done because both are collections of objects.) This is not ideal since it blurs the distinction between a CAS and a View. The intended distinction is that a CAS *contains* Views. Apache UIMA may be made to more closely reflect the proposed UIMA specification by providing a View interface which is distinct from the CAS interface. For example the CAS interface could provide getView() methods but not indexes, while Views do the opposite.

5.3.4.4 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. If the source data represents just a subset of a file, then additional offset information may also be needed. We may wish to consider a possible base type for representing this kind of information.

See also Section 7.3 End of Collection Processing for other requirements on source document information stored in the CAS.

5.4 Behavioral Metadata Specification

We think of an ***analytic*** as implementing an analysis operation. The Behavioral Metadata of an analytic declaratively describes what the analytic does; for example, what types of CASs in 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.[6]

A ***flow controller*** may also have behavioral metadata, although since flow controllers do not modify the CAS[7], this may be limited to preconditions on the types of CASs it can process and what elements in a CAS it will inspect.[8]

We do not yet provide a formal specification for Behavioral Metadata, rather this section discusses design goals and requirements and presents some rough examples for how Behavioral Metadata may be expressed by appealing to the OCL standard [OCL1].

5.4.1 Design Goals

The goals of the Behavioral Metadata Specification are:

1. Support composition either by a human developer or an automated process. 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.
2. 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.
3. An application or framework should NOT be required to evaluate behavioral specifications in order to run an *analytic*. An application or framework may evaluate the behavior specifications to ensure, for example, that input requirements of all analytics in a chain are satisfied by incoming CASs, or that the outputs satisfy some application requirements but this should not be required to simply run the analytics.
4. Behavioral Metadata should express what a component *can* do, not what role it actually ends up playing in any particular workflow. (i.e., behavioral metadata describes the component's *capability* independent of its *application*.)
5. Behavioral Specifications must be *data-driven*. That is, behavioral specifications consist of predicates that can be evaluated over data in the CAS. Components may not declare behavioral specifications over anything other than CAS data. Analytics therefore can not declare their behavior in terms of other components or any particular flow in which they may play a constituent role. For example, the behavioral specification can not assert that that an analytic's outputs should be processed by some other component; this decision is made by the person or process that combines the analytics to accomplish a higher-level task.

---

[6] Note that it is possible for a human to perform the function of an analytic. For example in some systems, automated analytic processes make a first pass at doing the analysis but a human may correct their results. Behavioral Metadata is still useful in this case, in order to capture the types of operations that the human is expected to perform.

[7] Currently the Abstract Interfaces do not define a way for the Flow Controller to return an updated CAS. It has been argued that this might be too constraining. In fact, Apache UIMA Flow Controllers *can* modify the CAS. This issue is worth more consideration.

[8] The focus of this specification is on the discovery and reuse of analytics, not flow controllers. For this reason, and because the behavior of flow controllers may be complex, we do not attempt to define behavioral metadata that describes the operations performed by flow controllers. However, we do allow the flow controller to state what elements of the CAS it will inspect, since this may be useful for optimizing CAS transport.

Discussion Point

Non-Functional Requirements

In addition to these design goals, we believe some consideration should be given to representing non-functional requirements as part of the Behavioral Metadata. Information about quality of service expected CPU, memory or disk requirements, for example, are non-functional requirements that affect behavior and may be important consideration for reusing any given analytic as part of an application.

5.4.2 Requirements

Behavioral metadata should be able to express in a declarative, human and machine readable form the following

1. **Precondition:** 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. For example, if the pre-condition requires that valid input CASs contain `People`, `Places` and `Organizations` and the assembler knows that they will not, then the analytic is clearly not suitable for the intended operation.

2. **Capability:** A description of the intended effects of the analytic's operation on subsets of valid input CASs. The description need not completely specify analytic behavior but rather describe the results that the analytic is capable of providing. The capability description may break down into the following parts:
    a. **Analyzes** A predicate that defines the subjects of analysis (sofas) that the analytic can analyze. For example, an analytic may declare that it analyzes instances of type `Person`, or it may declare that it analyzes instances of type `Sofa` whose `mimeTypeMajor` is "text". This expression may identify a single object or a collection of objects.
    b. **Inspects:** A predicate that identifies the collection of objects which the analytic may consult while doing its analysis. If an object is NOT a member of the `inspects` or `analyzes` predicates, then a framework or application is permitted to filter this information (perhaps as an optimization for remote transport of the CAS). The `inspects` predicate may specify that all content in the CAS will be inspected.
    c. **Creates:** An expression that identifies objects that an analytic may create as a result of its analysis. For example, an analytic may declare it creates instances of type `Organization` with their `sector` feature equal to "Financial".
    d. **Modifies:** An expression that identifies objects and or slots that an analytic may modify.
    e. **Deletes:** An expression that identifies objects that an analytic may delete.

3. **Post-Condition:** An analytic developer should be able to declare a post-condition that the developer asserts will 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.

Candidate Compliance Point: A compliant UIMA component/framework may be required to guarantee that its post-condition holds true on CASes that it outputs. There may be other possible compliance points associated with other behavioral metadata (for example that the component does not create objects that are not declared in its `creates` predicate.

Apache UIMA Note

Apache UIMA capability specifications are able to express the following kinds of conditions in each of the categories:

1. **Preconditions:** The only precondition that Apache UIMA supports is `languagesSupported`, which is a check against the `language` feature of the built-in type `uima.tcas.DocumentAnnotation`.
2. **Analyzes:** can specify multiple "input sofas" to a component. The names declared by the analytic are matched against the `sofaID` feature of the built-in type `uima.cas.Sofa`.
3. **Inspects:** can specify the names of type and features that the analytic will inspect.
4. **Creates:** can specify the names of types that the analytic may create.
5. **Modifies:** can specify the names of features that the analytic may modify.
6. **Deletes:** cannot be expressed
7. **Postcondition:** cannot be expressed

Apache UIMA analytics can declare multiple capabilities. The reason for this is to allow an analytic to declare different creates/modifies statements for different `languagesSupported` preconditions. This may be an issue if the UIMA spec decides not to allow multiple preconditions in a single analytic.

Discussion Point

Sets of Capabilities

We have considered whether we should allow components to specify sets of (Precondition, Capability, Postcondition) declarations. That is, if the CAS satisfies Precondition 1, the component can perform Capability 1; if the CAS satisfies Precondition 2, the component can perform a different Capability 2.

The trouble with this approach is that we would need to specify what happens if a CAS satisfies more than one precondition. Are all operations performed? If so, in what order are they performed? Similarly, what if a CAS originally satisfied only Precondition 1 only, but after the analytic had performed Capability 1 the CAS now satisfied Precondition 2. Should the analytic now perform Capability 2?

These issues hinder composition because it is no longer clear what the precondition and post condition are for *the analytic as a whole*. Therefore we have proposed only a single (Precondition, Capability, Postcondition) declaration for each analytic.

Allowing multiple capability sets would require a more complex behavioral model, and that burden would be transmitted to the application or flow controller that wanted to consider the behavioral metadata.

If we wish to allow multiple sets of capabilities then we need to extend the specification so that it provides explicit answers to the questions in the second paragraph of this discussion point.

5.4.3 Examples

In the following examples, we explore the use of a predicate language for representing behavioral metadata based on UML's Object Constraint Language (OCL) [OCL1]. This is not the only possible language for expressing behavioral metadata, and we may wish to consider other suitable languages such as OWL-S [OWL-S1].

In general, the pre-condition and post-condition may be represented by OCL predicates over the object model contained in an incoming CAS.

Each element of the capability section in the behavior specification may be represented by an OCL `Collection`. The predicates characterizing the collections are assumed to be over the object model contained in an incoming CAS.

The entire specification is assumed to be contained in a single OCL `let` statement. The let statement creates a scope in which all variables bound to these collections are visible.

5.4.3.1 Example 1: Creating Types on Select Sofas

In this example we want to describe the behavior of an analytic that

- will process any CAS. There is no pre-condition.

- analyzes instances of `ex::TextDocument` (a type in the CAS; note we are using the OCL syntax `ex::TextDocument` to refer to type `TextDocument` in package `ex`).
- will inspect instances of `ex::Person` and `ex::Place`
- will create zero or more instances of `ex::At` type
- guarantees the `sofa` feature of each instance of `ex::At` created by the analytic will point to an `ex::TextDocument` instance

The pseudo-OCL might look like this:

```
Let
   ex1PreCondition : true
   ex1Analyzes : select (s | s.oclKindOf(ex::TextDocument) )
   ex1Inspects : select (i | i.oclKindOf(ex::Person) or
   i.oclKindOf(ex::Place))
   ex1Creates :  Collection(ex::At)
   ex1PostCondition: ex1Creates->forAll(a | ex1Analyzes->exists(r |
   a.sofa=r)
```

This may be captured in the proposed behavioral specification as follows:

```
<behavioralSpec>
  <capability>
    <precondition>true</precondition>
    <analyzes handle="ex1Analyzes"> select(s |
      s.oclKindOf(ex::TextDocument)) </analyzes>
    <inspects handle="ex1Inspects"> select (i |
      i.oclKindOf(ex::Person) or i.oclKindOf(ex::Place)) </inspects>
    <creates handle="ex1Creates"> collection(ex::At)</creates>
    <postcondition> ex1Creates->forAll(a | ex1Analyzes->exists(r |
      a.sofa=r) </postcondition>
  </capability>
</behavioralSpec>
```

To form an OCL expression from this XML, the values of the `handle` attributes are mapped to the variable names in the OCL `let` statement. For example, the OCL expression would bind `ex1Analyzes` to the instances of `ex::TextDocument` that satisfy the sofa select statement. The name `ex1sofas` can be referred to in subsequent OCL expressions. In this example it is used in the postCondition to indicate that all created `ex::At` annotations will reference one of these `ex::TextDocument` objects.

5.4.3.2 Example 2: Pre-Conditions enable CAS Filtering

This example extends the previous example by adding a precondition that would allow filtering CASs that do not contain at least one `Person` and one `Place` annotation on a text document sofa.

Pseudo-OCL:

```
Let
   persons : select (p | p.oclKindOf(ex::Person)
   places : select (p | p.oclKindOf(ex::Place)
   ex2PreCondition :
     persons->exists(p | p.sofa.oclKindOf(ex::TextDocument)) and
```

```
     places->exists(p | p.sofa.oclKindOf(ex::TextDocument) )
   ex2Analyzes : select (s | s.oclKindOf(ex::TextDocument) )
   ex2Inspects: persons->union(places)
   ex2Creates :  Collection(ex::At)
   ex2PostCondition: ex1Creates->forAll(a : At | ex2Analyzes->exists(r
   | a.sofa=r)
```

Proposed XML for behavioral specification:

```
<behavioralSpec>
  <let handle="persons">select (p | p.oclKindOf(ex::Person))</let>
  <let handle="places">select (p | p.oclKindOf(ex::Place))</let>
  <precondition> persons->exists(p |
    p.sofa.oclKindOf(ex::TextDocument)) and places->exists(p |
    p.sofa.oclKindOf(ex::TextDocument) ) </precondition>
  <capability>
    <analyzes handle="ex2Analyzes"> select(s |
s.oclKindOf(ex::TextDocument))
       </analyzes>
    <inspects handle="ex2Inspects">persons->union(places)
      </inspects>
    <creates handle="ex2Creates"> Collection(ex::At)</creates>
  </capability>
  <postCondition> ex2Creates->forAll(a | ex2Analyzes->exists(r |
    a.sofa=r) </postCondition>
</behavioralSpec>
```

This example also shows that we would like to support binding arbitrary handles (in the `<let>` elements) that can later be referred to from other OCL expressions, such as the `precondition` and `inspects` elements.

5.4.3.3 Example 3: Multiple Sofas

In this example we describe the behavior of a component that analyzes two distinct types of objects: a `TextDocument` object and a `RawAudio` object. It produces annotations of type `ex::text::Person` over the `TextDocument` and of type `ex::audio::Person` over the `RawAudio`.

Pseudo-OCL:

```
Let
   ex3Precondition : true
   textSofas : select (s | s.oclKindOf(ex::TextDocument))
   audioSofas: select (s | s.oclKindOf(ex::RawAudio))
   ex3Inspects : true  -- This component may inspect all objects in CAS
   createsTextPersons :  Collection(ex::text::Person)
   createsAudioPersons : Collection(ex::audio::Person)
   ex3PostCondition:
     createsTextPersons->forAll(a | textSofas->exists(r | a.sofa=r) and
     createsAudioPersons->forAll(a | audioSofas->exists(r | a.sofa=r)
```

Proposed XML for behavioral specification:

```
<behavioralSpec>
  <precondition>true</precondition>
  <capability>
    <analyzes handle="textSofas"> select (s |
      s.oclKindOf(ex::TextDocument)) </analyzes>
    <analyzes handle="audioSofas"> select (s |
      s.oclKindOf(ex::AudioDocument)) </analyzes>
    <inspects>true</inspects>
    <creates handle="createsTextPersons">
      Collection(ex::text::Person)</creates>
    <creates handle="createsAudioPersons">
      Collection(ex::audio::Person)</creates>
  </capability>
  <postCondition> createsTextPersons->forAll(a | textSofas->exists(r
    | a.sofa=r) and createsAudioPersons->forAll(a |
    audioSofas->exists(r | a.sofa=r) </postCondition>
</behavioralSpec>
```

5.4.3.4 Example 4: Anchored View

Apache UIMA makes use of the `AnchoredView` type to facilitate the processing of collections of annotations that all refer to the same sofa. See 5.3.4.3 Anchored View for the definition of an `AnchoredView`.

This example describes an analytic that:

- Will process any CAS containing at least one `AnchoredView` whose sofa has `mimeTypeMajor` = "text"
- Inspects `ex::Person` and `ex::Place` objects within an `AnchoredView`.
- Creates `ex::At` objects and adds them to an `AnchoredView`.

Pseudo-OCL:

```
Let
   myAnchoredViews : select (v | v.oclKindOf(uima.AnchoredView),
     v.sofa.mimeType="text")
   ex4PreCondition : myAnchoredViews->notEmpty()
   ex4Analyzes : myAnchoredViews->collect(sofa)
   ex4Inspects : myAnchoredViews->collect(members->select (v |
     i.oclKindOf(ex::Person) or i.oclKindOf(ex::Place)))
   ex4Creates :  Collection(ex::At)
   ex4PostCondition: ex4Creates->forAll(a |
     myAnchoredViews->exists(v | v.members->includes(a)))
```

Proposed XML for behavioral specification:

```
<behavioralSpec>
  <let handle="myAnchoredViews">: select (v |
    v.oclKindOf(uima::AnchoredView), v.sofa.mimeType="text")</let>
  <precondition>myAnchoredViews->notEmpty()</precondition>
  <capability>
```

```
    <analyzes handle="ex4Analyzes">
      myAnchoredViews->collect(sofa)</analyzes>
    <inspects> myAnchoredViews->collect(members->select (i |
      i.oclKindOf(ex::Person) or i.oclKindOf(ex::Place)))
      </inspects>
    <creates>Collection(ex::At)</creates>
    <postCondition>ex4Creates->forAll(a : At |
      myAnchoredViews->exists(v | v.members->includes(a)))
      </postCondition>
  </capability>
</behavioralSpec>
```

Apache UIMA Note

In the following Apache UIMA capability specification:

```
<capability>
  <inputs>
    <type>ex.Person</type>
    <type>ex.Place</type>
  </inputs>
  <outputs>
    <type>ex.At</type>
  </outputs>
  <inputSofas>
    <sofaName>SomeInputSofaName</sofaName>
  </inputSofas>
</capability>
```

It is an implicit assumption that the `ex.Person` and `ex.Place` objects will be members of the `View` named "SomeInputSofaName". (Or more precisely, the unique `View` associated with the `Sofa` that has that name.)

We can make the Apache UIMA semantics explicit by specifying the exact mapping from this capability representation to a set of OCL expressions.

5.4.4 Restricting the Subjects of Analysis

Note that the `analyzes` predicate of the Behavioral Specifications qualifies objects the analytic is *capable* of operating on. At runtime, an application or aggregate that calls the analytic may wish to direct the analytic to process only a particular set of objects that satisfy the analytic's `analyzes` predicate.

Handles declared in an Analytic's behavioral specification provide a hook whereby the caller of analytic may bind a specific set of objects to the handle. For example, consider an analytic that declares in its behavioral metadata:

```
<analyzes handle="ex1Analyzes"> select(s |
  s.oclKindOf(ex::TextDocument)) </analyzes>
```

This analytic is declaring that it is capable of processing any instance of `ex:TextDocument`, and that it will use the handle `ex1Analyzes` to refer to the set of `ex:TextDocument` instances that it will analyze.

When we define the Analytic interface (see Section 5.6), we will provide a way for the caller of the analytic to specify that the handle `ex1Analyzes` should be bound to a particular set of `ex:TextDocument` instances that the caller wants the analytic to process.
For each `analyzes` predicate that the analytic defines, it may declare a different *handle*, which is a local name that identifies that `analyzes` predicate to this analytic. This allows the caller to specify a different set of objects to be bound to each `analyzes` predicate.

Apache UIMA Note

Apache UIMA also provides Sofa Mapping as part of its aggregate specification, which is a kind of instance-level CAS data mapping that satisfies this "handle" requirement by allowing the aggregate assembler to map any Sofa in the CAS to the name expected by the analytic. Sofa mappings also allow an aggregate to guarantee unique Sofa names even if analytics create Sofa objects with the identical names.

This binding of handles to objects by the caller serves two primary purposes:

1. It allows a framework or caller to provide a convenience to the analytic developer. Note that a framework may already evaluate OCL expressions in the analytic's behavioral spec in order to determine if the analytic's precondition is met. In that scenario it makes sense for the framework to make the results of that evaluation available to the analytic rather than force the analytic to recompute them.

2. It enables the caller to further restrict the collection bound to a handle. The need for this was discussed in the Requirements section above. For example in the XML above, this component declares that it can analyze any instance of `ex::TextDocument`. The caller may wish, however, to have only one particular instance of `ex::TextDocument` analyzed. The caller can indicate this by binding the ex1sofas handle to just that particular `ex::TextDocument` instance.

It only makes sense for the caller to bind input handles such as `analyzes` or `inspects`. It would not make sense for a caller to bind objects to a `creates` handle, since that handle refers to a set of objects produced by the analytic.

Declaration of handles is optional. If the analytic does not declare any handles then the caller cannot specify bindings. Also it is optional for a caller to provide bindings. If the bindings are not computed and sent along with the CAS, then the analytic must locate the required collections itself (using whatever APIs are provided to the CAS for example).

> **Candidate Compliance Point:** A UIMA component/framework may be required to accept input CASes that do not include handle bindings. However, if handle bindings are provided, a UIMA compliant component/framework may be required to use them (e.g., to restrict its processing to only those objects that the caller has bound to the handle).

5.5 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.

> **Candidate Compliance Point:** A UIMA component/framework may be required to public Processing Element Metadata that conforms to this specification.

5.5.1 Abstract Definition

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. **Reference to a Type System**. Defines types 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.

Also in this section we define the ***Configuration Parameter Settings*** object. While not a subpart of the PE Metadata, it is defined in this section because it is closely related to the Configuration Parameter definitions section of the metadata. The Configuartion Parameter Settings object is used by applications or aggregate analytics to override the default configuration parameter values of a PE.

Figure 11 is a UML model for the PE metadata. We describe each subpart of the PE metadata in detail in the following sections.

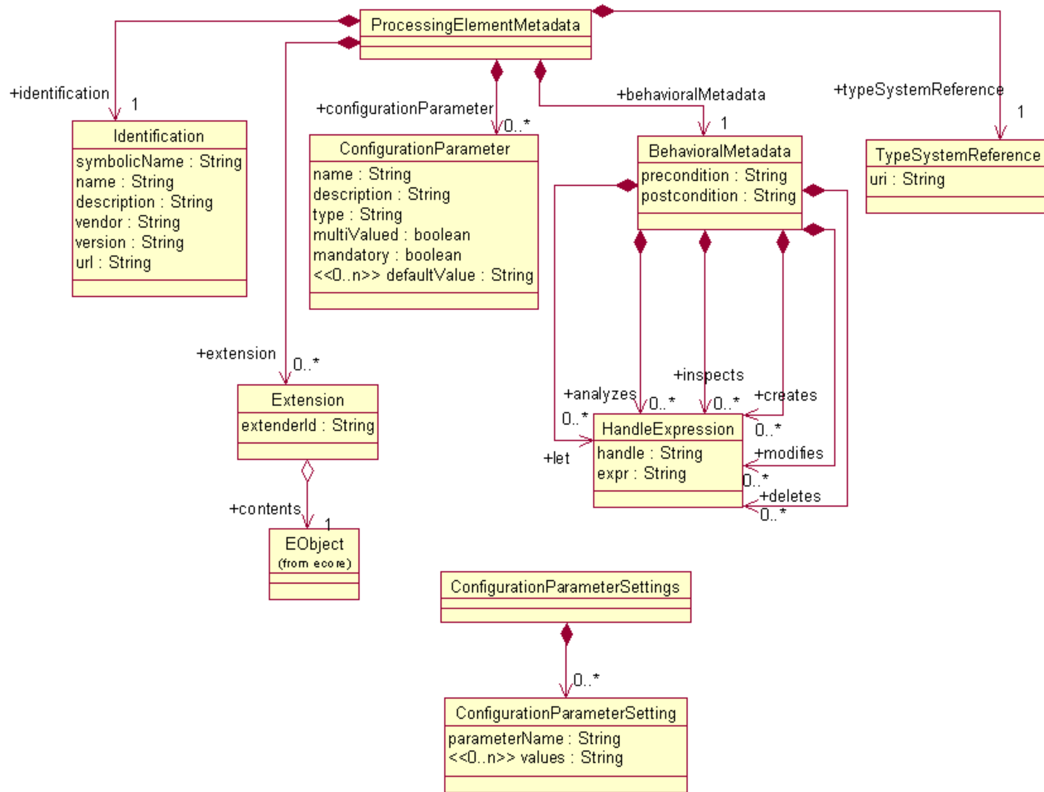

Figure 11: Processing Element Metadata UML Model

Apache UIMA Notes

Currently the Apache UIMA Component Metadata Descriptor includes the following elements that are not part of the proposed UIMA Specification.

1. **Indexes:** Defines the structure of indexes through which the analytic will access data. In some sense the actual indexing design is an Apache UIMA issue and so this may be an extension to the descriptor schema that is specific to Apache UIMA. However if we think of the index definitions as a component declaring the key features that it is going to use to query the data, we can make a case that this should be a UIMA standard, so that any framework could optimize based on this information.
2. **Type Priorities:** These are closely related to the index definitions and should probably be combined with them rather than represented as a separate element
3. **External Resources:** The core concept of external resource dependencies is captured using the "ResourceURL" configuration parameter type, discussed above. Other details of Apache UIMA's external resource mechanism are framework-dependent and not covered in the UIMA spec.
4. **Configuration Parameter Settings:** Default values for parameters are becoming part of the configuration parameter declarations. Specifying non-default values should not be done as part of the descriptor.
5. **Operational Properties:** (modifiesCas, outputsNewCASes, multipleDeploymentAllowed): These should be covered by the Behavioral Specification. The first two are fairly straightforward. The "multipleDeploymentAllowed" property states whether the component is “parallelizable”. Usually components that maintain state across input CASes are not parallelizable and can’t be multiply deployed. Will that be covered by the behavioral spec? We make significant use of this property in the Apache UIMA CPM.

5.5.1.1 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 OSGi standard [OSGi1] also has a set of identification information for its components (called "bundles"). An example is that in OSGi a bundle can be given two names. The "bundle-name" is a human readable name, while the "bundle-symbolic-name" is a unique ID. We have borrowed that terminology for this spec. We have also borrowed the OSGi convention for the syntax of version numbers.

UDDI [UDDI1] may also be a relevant standard for representing identification information for PEs. We intend to explore how we can represent PE identification metadata in a way that is compatible with UDDI.

We have identified the following useful properties:

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

5.5.1.2 Configuration Parameters

Many kinds of PEs may be configured to operate in different ways[9]. 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 5.6 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.

We propose that the type of a parameter must be one of the following:

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

[9] 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.

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.

Discussion Points

1. Are the types String, Integer, Float, Boolean, and ResourceURL sufficient? For example, it might be useful to have a 2D-array or a HashMap as the value of a parameter. Also, should the types of parameters be related to the CAS TypeSystem, or not?
2. In the UML, should the features `type`, `multiValued`, and `mandatory` be required (cardinality 1 instead of the implicit 0..1)?

Apache UIMA Note

Apache UIMA has a more extensive schema that allow for "configuration groups". For example this feature can be used to allow an annotator to use a different pattern file for English documents than for German documents. The annotator's descriptor would declare groups named "en" and "de" each containing a "PatternFile" parameter, like this:

```
<configurationGroup names="en de">
    <configurationParameter>
      <name>PatternFile</name>
      <description>Location of external file containing
additional patterns to search for.</description>
      <type>ResourceURL</type>
    </configurationParameter>
</configurationGroup>
```

The Apache UIMA API then allows an application to set a different value for this parameter in the "en" group than in the "de" group.

This feature does not get much use in Apache UIMA and adds a lot of complexity to framework implementations, so we have proposed leaving it out of the UIMA specification.

5.5.1.3 Type System Reference

PE Metadata does not *include* a type system, it simply refers to it. This specification is only concerned with the format of that reference. For the actual definition of the type system, we have adopted the Ecore/XMI representation. See Section 5.2 The Type-System Language Specification for details.

URIs are used as references by many web-based standards (e.g., RDF), and they are also used within Ecore. Thus we propose using a URI to refer to the type system.

To achieve interoperability across frameworks, this URI should be a URL at which the Ecore/XMI type system data is located.

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 5.1.4.6 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.[10]

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).

Discussion Points

While we believe that referencing a separate type system is the preferred way to structure descriptor files, there may also be use cases for including the type system definition directly in the analytic metadata.

For example, consider a UIMA Analytic deployed as a service. The service defines a getMetadata call that returns the descriptor. Is it our intention that the returned descriptor has a URI for a type system reference and that the caller would always need to initiate a second request to obtain the type system? This may hinder the flexibility of deployment options.

Another drawback of referencing a separate type system is if there are multiple analytics that refer to the same type system file, that type system cannot be updated without affecting all referencing analytics. It may not in general be possible to determine the complete set of analytics refer to a given type system file.

[10] 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 metdata for a PE to declare whether it can accept undefined types.

Apache UIMA Notes

1. Apache UIMA allows type systems to be defined directly inside an analytic descriptor, as well as by reference.
2. For Apache UIMA remote services, references to type systems are resolved during service deployment and "included" directly into the descriptor. When the service sends its metadata to a client, it sends the descriptor that directly includes the entire type system definition. Therefore the client never needs to initiate a second request to obtain the type system.
3. Apache UIMA has an "import" construct that can be used not only for type systems but also many other parts of the descriptor that may be reusable. Imports can by "by location" or "by name". An import by location is a URL; if the URL is relative then it is resolved relative to the descriptor containing the import. An import by name is a dotted name (as in a Java classname) that is looked up in the Java classpath. Several users have found this classpath look up very useful and a natural way to do things in Java. Are we now requiring URIs instead? Perhaps it is sufficient to use relative URLs in Apache UIMA descriptors (thus complying with the UIMA spec), and for Apache UIMA to resolve those relative URLs against the classpath or datapath.

5.5.1.4 Behavioral Specification

The Behavioral Specification is discussed in detail in Section 5.4 Behavioral Metadata Specification. Here we define a UML model that is consistent with the requirements and examples provided in that section.

5.5.1.5 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.

There are two choices for adding an extension to a PE metadata descriptor. The simplest option is to use the XMI extension mechanism (described in Section 5.1.4.7), which allows arbitrary XML content to be added to a descriptor. In this approach there is no explicit class model that defines the structure of the extension.

A second approach is to make the extension part of the class model. To add an extension, a framework must provide an Ecore model that defines the structure of the extension. This has the advantage that a framework implementation could choose to use EMF [EMF1] to parse the PE metadata descriptor, including any extensions. Using Ecore models for extensions is also consistent with our use of Ecore models for defining CAS type systems.

To enable the latter approach, we have defined the `Extension` class in Figure 11. 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.)

5.5.1.6 Configuration Parameter Settings

A `ConfigurationParameterSettings` object is a collection of `ConfigurationParameterSetting` objects, each of which defines the setting of a single parameter. The `ConfigurationParameterSetting` object has a `parameterName` feature that identifies which parameter will be set, as well as a `values` feature that contains zero or more values to assign to that parameter.

The `ConfigurationParameterSettings` object is used by applications or aggregate analytics to override the default values of a PE's Configuration Parameters. We will refer back to this definition of `ConfigurationParameterSettings` in Sections 5.6 Abstract Interfaces and 5.8 Aggregate Analytic Descriptor.

5.5.2 XML Schema

Rather than define a custom XML schema for representing PE metadata, we use the XMI standard [XMI1] to define the XML schema based on the UML Diagram in Figure 11. The Eclipse Modeling Framework [EMF1] tools can automatically generate this XML Schema from the UML model, and can also generate Java code that can be serialized to and deserialized from XML conforming to that schema.

The XML Schema is included as an appendix (Section 9.2.1). In this section we show an example descriptor that conforms to the schema. Each of the major sections of the descriptor is identified by a comment. The contents of each section directly correspond to the features in the UML diagram in Figure 11.

```
<pemd:ProcessingElementMetadata xmi:version="2.0"
  xmlns:xmi="http://www.omg.org/XMI"
  xmlns:pemd="http:///uima/peMetadata.ecore">

  <!-- Identification Information -->
  <identification
    symbolicName="com.ibm.uima.example.PersonTitleAnnotator"
    name="Person Title Annotator"
    description="Detects person titles in a text document."
    vendor="IBM" version="1.0"/>


  <!-- Configuration Parameters -->
  <configurationParameter name="RegExPatterns"
    description="Regular expression patterns to search for"
    type="String" multiValued="true">
    <defaultValue>Mr\.|Mrs\.|Dr\.</defaultValue>
    <defaultValue>Lt\.|Capt\.|Maj\.|Col\.|Gen\.</defaultValue>
  </configurationParameter>
  <configurationParameter name="PatternFile"
    description="Location of external file containing additional
patterns to search for."
    type="ResourceURL">
    <defaultValue>
```

```
      myResources/personTitlePatterns.dat</defaultValue>
  </configurationParameter>

  <!-- Type System Parameters -->
  <typeSystemReference
    uri="http://sith.watson.ibm.com/types/exampleTypeSystem.ecore"/>

  <!-- Behavioral Metadata -->
  <behavioralMetadata precondition="true"
    postcondition="ex1Creates->forAll(a | ex1Analyzes->exists(r |
a.sofa=r))">
    <analyzes handle="ex1Analyzes"
      expr="select(s | s.oclKindOf(ex::TextDocument))"/>
    <creates handle="ex1Creates"
      expr="Collection(ex::PersonTitle)"/>
  </behavioralMetadata>
</pemd:ProcessingElementMetadata>
```

5.6 Abstract Interfaces

The UIMA specification has defined 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 12 on page 63 for a UML model of these interfaces. The abstract definitions in this section lay the foundation for the concrete service specification defined in Section 5.7.

5.6.1 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 (define in Section 5.5.1.6 that contains a set of (name, values) pairs that identify configuration parameters and the values to assign to them.[11]

After a client calls `setConfigurationParameters`, those parameter settings must be "remembered" by the PE and 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. This issue is further discussed in Section 7.2 Supporting Multiple Sessions.

5.6.2 Analytic

For Analytics, we define 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.

[11] Note that Aggregate Analytics (see Section 5.8) are opaque and you can only set parameters exposed from the Aggregate; you cannot drill in and set parameters directly on the constituents.

5.6.2.1 Analyzer

The `Analyzer` interface defines one additional operation:

- `processCas`, which takes a CAS plus *input bindings* and returns the same CAS, possibly updated[12]. Input bindings are assignments of CAS objects to handles declared in the analytic's behavioral specification (see Section 5.4 Behavioral Metadata Specification).

Where we say that two CASes are "the same", we mean that all objects in the first CAS appear in the second CAS, *except* where an explicit delete or modification was performed by the service (which is only allowed if the service declares such operations in its behavioral spec). Also, in the XMI representation we require the `xmi:ids` of objects that appear in both CASes must be the same, so that the recipient of the CAS can determine the correspondence between objects in the two CASes.

The input CAS may contain a reference to its type system (see Section 5.1.4.6). If it does not, then the PE's type system (see Section 5.5.1.3) 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.

5.6.2.2 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 input bindings, but returns nothing.
- `getNext`, 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.

Note that in the `CasMultiplier` interface, there are separate operations to send and to retrieve the input CAS. Typically retreiveInputCas would be called only after all of the output CASes have been generated. However, we would like to leave open the possibility the input CAS may be retrieved *before* all of the output CASes have been generated, if possible.

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.[13]

---

[12] If an Analytic makes only a smaller number of changes to its input CAS, it will be more efficient to respond with a "delta" rather than repeating the entire CAS. See Section 5.7.6.3 Delta Responses.

[13] A CAS Multiplier that merges CASes may want to know when it has received all of the input CASes from a given collection, so that it may produce a final merged CAS. See Section 7.3 End of Collection Processing for a discussion.

Discussion Points

There are several other possible functions that we may want to allow a CAS Multiplier to perform, for example:

1. Return more than one CAS at a time. (The caller may specify a maximum number.)
2. Return an indication that no more CASes are available now, but that the caller should try back later. (The caller may specify the amount of time to wait before returning.)
3. Return an estimate of how many CASes have not yet been retrieved by the caller.

To address these requirements we might want to redefine the `getNext` operation as:

```
GetNextResponse getNext(int maxCASesToReturn, int timeToWait)
```

Where `GetNextResponse` is a structure that contains:

- a collection of CASes (from zero up to `maxCASesToReturn`)
- a flag indicating whether there are any more CASes to be returned
- if there are more CASes, an estimate of how may CASes remain

5.6.3 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 access to the Metadata of the Aggregate Analytic containing this Flow Controller. This operation may not be called if the Flow Controller is not encapsulated in an Aggregate Analytic.
- `getNextDestinations`, which takes a CAS and returns one or more destinations for this CAS.

The application or aggregate framework containing the `FlowController` must call `addAvailableAnalytics` and pass an `AnalyticMetadataMap`, which is a map from String keys to analytic metadata. The keys are arbitrary identifiers that are unique within the set of analytics known to this `FlowController`. If the FlowController is contained in an Aggregate Analytic, the aggregate framework must also call the `setAggregateMetadata` operation.

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.

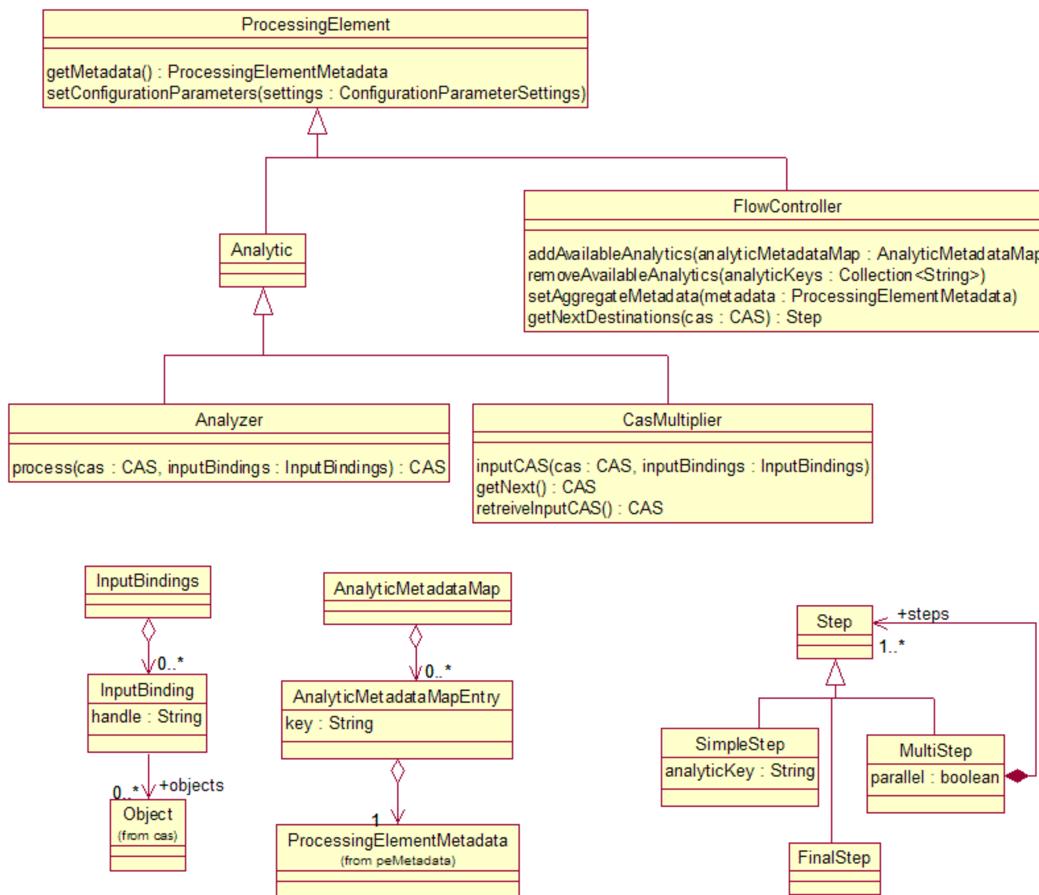

Figure 12: UML for Abstract Interfaces

Apache UIMA Notes

Apache UIMA binds the UIMA abstract interfaces to Java interfaces which may then be implemented by Apache UIMA component developers.

Apache UIMA further specializes the Analytic interfaces into different component types:

1. `Analyzer` is specialized to:
    - `Annotator`, for analyzers that modify the CAS
    - `CasConsumer`, for analyzers that do not modify the CAS
2. `CasMultiplier` is specialized to:
    - `CollectionReader`, for CAS Multipliers that produce CASes that each represent an artifact from a collection.

The Apache UIMA `FlowController` interface also introduces a slightly different programming model. Apache UIMA defines a method `FlowController.computeFlow(CAS)`, which is called when a new CAS first enters the aggregate. The `computeFlow` method returns an object of type Flow. The Flow object is dedicated to routing a particular CAS. The Flow interface defines a `next()` method which returns the next destination for this CAS (it can consult any information in the CAS to make this decision). With this programming model developers of Flow Controllers are insulated from the complexity of multiple CASes potentially flowing through an aggregate at the same time. However, we did not want to mandate the use of this programming model throughout all UIMA implementations, so a simpler `FlowController` interface is defined here. The Apache UIMA implementation can be easily adapted to the proposed standard UIMA interface.

Also, the Apache UIMA `FlowController` interface permits the FlowController to modify the CAS, whereas the Flow Controller interface in this specification does not.

In the following section we appeal to WSDL to provide a more formal definition of these interfaces.

5.7 Service WSDL Descriptions

This section describes a WSDL [WSDL1] document for the UIMA Processing Element Service Interfaces. We also define a binding to the SOAP protocol as an appendix (Section 9.1).

Candidate Compliance Point: To comply with UIMA at the Services Level, a framework may be required to enable Processing Elements to be deployed as Services that implement this WSDL definition and support the SOAP binding.

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.

5.7.1 Header

```
<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">
```

Here we define the XML namespace prefixes used later in the WSDL document.

5.7.2 Types

The `<wsdl:types>` element includes the XML Schema definitions for all of the data that flows to and from the service. Here we are importing the type definitions from two separate schema files:

- `uima.peMetadataXMI.xsd`: Defines the PE metadata XML Schema, as specified by Section 5.5 Processing Element Metadata. This is needed so that services can publish their descriptive metadata to clients.
- `XMI.xsd`: The schema that defines an XMI document, according to the OMG standard. Note that the XMI schema is very general: almost any XML document is valid XMI. We might consider if a specific UIMA service could restrict this so that it only accepts types from its type system. See 5.7.6 PE Service Specification - Open Issues for further discussion.
- `peServiceXMI.xsd`: Additional types that are defined solely for use as part of the WSDL message definitions that follow.

```
  <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.  Note this will allow any arbitrary XMI
    content.  A specific UIMA service could restrict this by supplying
    its own custom schema that only accepts types from its type system.
    EMF can generate such a schema from an Ecore model. -->
    <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>
```

5.7.3 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.

Note that 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).

```
  <!-- Define the messages sent to and from the service. -->
  <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="processCasRequest">
  <wsdl:part element="cas" type="xmi:XMI" name="cas"/>
  <wsdl:part element="inputBindings" type="pe:InputBindings"
    name="inputBindings"/>
</wsdl:message>

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

<wsdl:message name="inputCasRequest">
  <wsdl:part element="cas" type="xmi:XMI" name="cas"/>
  <wsdl:part element="inputBindings" type="pe:InputBindings"
    name="inputBindings"/>
</wsdl:message>

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

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

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

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

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

<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>
```

5.7.4 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 5.6 Abstract Interfaces.

5.7.4.1 The Analyzer Port Type

```
<wsdl:portType name="Analyzer">

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

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

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

</wsdl:portType>
```

5.7.4.2 The CasMultiplier Port Type

```
<wsdl:portType name="CasMultiplier">
```

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

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

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

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

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

5.7.4.3 The FlowController Port Type

```
  <wsdl:portType name="FlowController">

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

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

```
    </wsdl:operation>

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

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

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

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

  </wsdl:portType>
```

5.7.4.4 SOAP Bindings

For each port type, we define a binding to the SOAP protocol. These bindings are included as an appendix (see Section 9.1). The reader may also wish to refer to the example SOAP messages in this appendix.

5.7.5 Supplying a Custom XMI Schema

The proposed WSDL does not restrict the input CAS to contain only instances of types defined in the type system that the service declares in its metadata. As described in Section 5.5.1.3, UIMA does not in general require that CASes conform to a PE's type system. In fact, some PE's may be able to process CASes containing any type of object.

However, if a particular PE Service can only accept CASes that contain a specific set of types, then it may be appropriate to express this in the WSDL for that PE service. This can be done by changing the import of the XMI.xsd file to instead import a custom XML schema. In fact EMF can generate just such a schema from an Ecore model.

Expressing a PE's input specification as XML Schema fits in with common web service practices, and so would make UIMA PE's more understandable to web services practitioners and

would enable web services tooling to interact with UIMA PE's. This could also allow web service containers to do some amount of automatic checking on input CASes. This checking would ensure that input CASes contain appropriate types and features, but it cannot do full validation of all constraints in an Ecore model, not can it enforce the full expressivity of UIMA Behavioral Specifications. Nonetheless, some amount of automated checking may be desirable.

The downside of expressing the PE's input specification in XML schema is a lack of flexibility to handle subtypes. Consider a PE that can accept CASes containing only instances of the `Place` type. A client may have a CAS containing a `City` object. The `City` type may be unknown to the PE, but the client can encode in the CAS a URI that references a type system defining `City` as a subtype of `Place`.

In this scenario, if the PE use used XML Schema to specify that input CASes could only contain `Place` objects, then the PE would reject such a CAS. If the PE did not use XML schema, it could look up the definition of `City`, determine that `City` was a subtype of `Place` and therefore that the CAS did in fact meet the PE's input constraint.

In summary, UIMA allows the use of XML Schema to restrict input CASes, in order to be compatible with web services methodologies. However, we believe that it many scenarios it will be more flexible to not do so, and we leave this decision up to the person who is deploying the PE service.

5.7.6 PE Service Specification - Open Issues

5.7.6.1 Need to define wsdl:fault for each operation

WSDL allows you to specify a message that is returned by an operation in the event of failure. We should specify this for each of our operations.

5.7.6.2 Sending multiple CASes in one request

In cases where each CAS is small and processing is quick, it may improve performance to send multiple CASes in one call to the service. This is something that the WebFountain system does. We may wish to extend the SOAP service spec to allow this.

5.7.6.3 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. The XMI specification includes a way to specify differences between object graphs. To illustrate this, here is an example delta response:

```
<processCasResponse xmlns="">
  <cas xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI">
    <xmi:Difference>
      <target href="input.xmi"/>
      <xmi:Add addition="p1">
      <xmi:Add addition="p2"/>
    </xmi:Difference>
    <ex:Pronoun xmi:id="p1" sofa="2" begin="3" end="5"/>
    <ex:Pronoun xmi:id="p2" sofa="2" sofaFeature="text" begin="29"
end="31"/>
  </cas>
</processCasResponse>
```

You need to specify separate `Add` elements that refer to the elements to be added, so there are twice as many elements as are really needed. The purpose of this appears to be to allow the `Add` element to specify an optional position where the object should be added (for adding elements inside of other "container" elements), which we don't really need. Another issue is the `target` element, which is supposed to be an href to the original XMI file to which these differences will get applied. Here we don't really have a URI for that - it is just the input to the Process CAS Request. The example uses a placeholder `input.xmi` for this.

5.8 Aggregate Analytic Descriptor

An ***Aggregate Analytic*** is a composition of two or more constituent analytics connected in a work flow. An Aggregate encapsulates a workflow of its constituents behind an analytic interface. It does not therefore introduce a different abstract interface.

By supporting the concept of an Aggregate Analytic, UIMA enables developers or automated tools to assemble and encapsulate sets of reusable analytics to perform composite analysis tasks.

Aggregates designate their constituent analytics along with a ***Flow Controller***. The flow controller is a kind of Processing Element that determines the order in which an aggregate's constituent analytics will obtain access to CASes flowing among them.

This section of the UIMA specification defines a UML model for an ***Aggregate Analytic Descriptor*** (from which an XML Schema may be generated). Having a standard specification for the Aggregate Analytic Descriptor allows aggregates to be consistently reused across frameworks.

> **Candidate Compliance Point:** A UIMA framework that implements aggregates may be required to process aggregate descriptors that conform to this specification.

5.8.1 Requirements

This section describes the motivating requirements for defining aggregate analytics as part of the UIMA specification.

5.8.1.1 Encapsulation

A primary motivation for UIMA is to facilitate the reuse of analytics. This motivation is based on the observation that many analysis tasks may be defined by composing existing analytics rather than by developing new algorithms from scratch.

To facilitate the application of these compositions, each composition should not introduce a new and unique interface but rather should project a standard analytic interface.

Aggregates therefore should provide a means for encapsulating a flow of constituent analytics behind a standard analytic interface so that clients of analytics need not be aware of whether or not they are implemented as aggregates.

5.8.1.2 User-Defined Flow Control

Aggregates compose and encapsulate constituent analytics. The functions of these constituents are characterized by their behavioral metadata. The behavioral metadata can be used to compute

a partial ordering among the constituent elements such that the output of one constituent produces results that satisfy the preconditions of the next in the ordering.

In general, however, applications will want to explicitly order constituent analytics based on the content of the CASs flowing through the aggregate or other external data.

Aggregates should therefore support user-defined flow control. This enables the development of application-specific Flow Controllers as well as reusable Flow Controllers that interpret flow control languages such as BPEL [BPEL1].

5.8.1.3 Inter-component Dependencies

Dependencies between components in an aggregate are discussed to a certain extent above under Composition, where we consider behavioral constraints and input/output capabilities that must be satisfied in the aggregate. The question here is if we need to provide a mechanism for specifying explicit dependencies between specific components. Such dependencies might relate to sharing libraries, packaging, installation, configuration, etc. One way to address these kinds of issues is with an existing packaging standard, such as OSGi [OSGi1]. This remains an open issue.

5.8.1.4 Configuration

An aggregate analytic must be able to control the configuration parameter settings of its constituent analytics. There are two types of things that the aggregate may wish to do:

1. Provide specific, static values that override the default parameter settings for constituent analytics.
2. Expose parameters on the Aggregate, where those parameters are mapped to parameters on consistent analytics.

5.8.1.5 Input Specification

An aggregate should be able to specify which objects in the CAS are operated on by each constituent analytic. Each constituent analytics, in its Behavioral Specification, provides a predicate that identifies the objects the analytic is *capable* of operating on. At runtime, an aggregate that calls the analytic may wish to direct the analytic to process only a particular set of objects that satisfy this predicate.

For example, an analytic may specify the following in its Behavioral Specification:

```
<analyzes handle="ex1Analyzes">
  select(s | s.oclKindOf(ex::TextDocument))
</analyzes>
```

This component declares that it can analyze any instance of `ex::TextDocument`. The aggregate assembler may wish, however, for this analytic to only analyze one particular instance of `ex::TextDocument`, say the instance whose `name` feature equals “DetaggedText”.

The aggregate descriptor must allow the assembler to express such a constraint. This can be done by associating the analytic’s handle (`ex1Analyzes`) with a more restrictive predicate that identifies just those instances that should be bound to that handle when the analytic executes.

5.8.2 Abstract Descriptor Definition

Figure 13 shows a UML Model for the Aggregate Descriptor. An Aggregate Descriptor includes the following parts:

1. **Analytic Metadata:** An aggregate descriptor specifies the same Analytic Metadata as any Analytic. This includes its identification information, configuration parameters, type system reference, and behavioral specification. See Section 5.5 Processing Element Metadata.

2. **Constituent Components:** An aggregate descriptor identifies a set of analytics that are the *constituent components* of this aggregate. For each constituent component the aggregate can specify configuration settings/mappings and input specifications.

3. **Flow Controller:** An aggregate descriptor identifies the flow controller component that will decide the workflow for the aggregate. The aggregate can also specify mappings and input specifications for the Flow Controller.

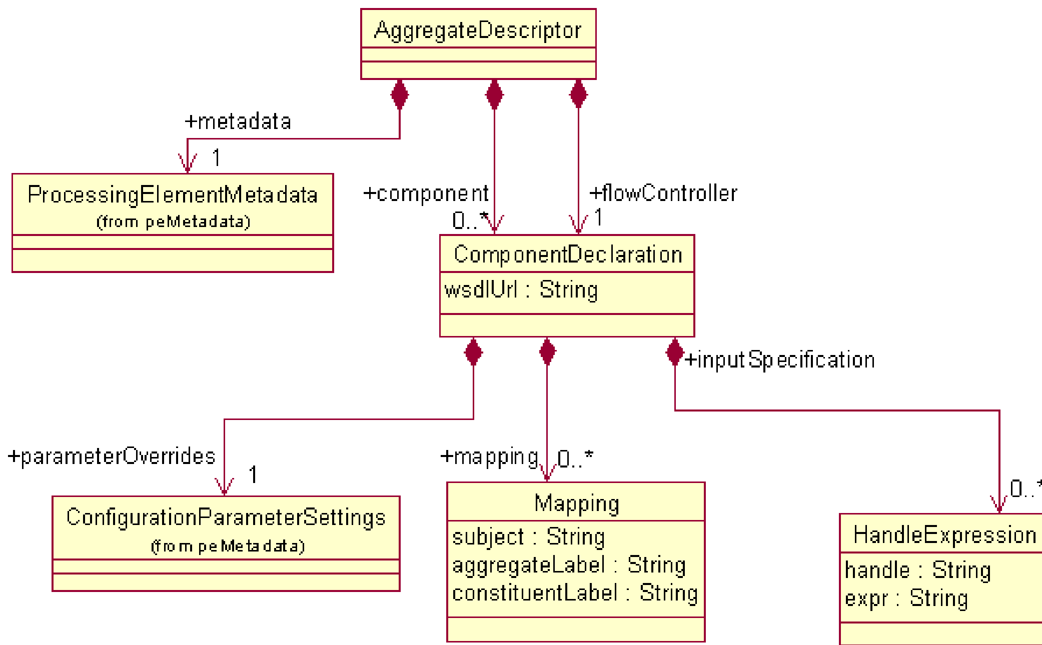

Figure 13: Aggregate Descriptor UML Model

5.8.2.1 Processing Element Metadata

An aggregate descriptor specifies its Metadata in the same format as any Processing Element. This includes its identification information, configuration parameters, type system reference, and behavioral specification. See Section 5.5 Processing Element Metadata.

5.8.2.2 Component Declarations

The aggregate descriptor specifies a `wsdlUrl` for each of the constituent analytics that comprise the aggregate. This is a URL that is expected to point to a WSDL document that describes where to locate the constituent analytic and how to interact with it.

If the WSDL document uses the standard UIMA port types and SOAP bindings defined by the UIMA specification in Sections 5.7 and 9.1 (usually by using WSDL's import feature), then there is the expectation that the aggregate descriptor could be reused in any UIMA framework that implements aggregates. If the WSDL document does not use the standard port type and bindings, UIMA provides no expectation of interoperability.

Note that WSDL allows a service to have multiple ports for the same port type, each with a different protocol binding. In this way the a constituent analytic could declare that it implements the standard SOAP binding as well as another, framework-specific transport. This would allow that constituent component to operate in other frameworks (using the standard SOAP binding) as well as to implement a specific (perhaps optimized) transport when run within its native framework.

5.8.2.3 Parameter Overrides

For each constituent, the aggregate descriptor may include a `ConfigurationParameterSettings` object (formally defined in Section 5.5.1.6). This object specifies values for the constituent's configuration parameters. These values will override any default values specified in the constituent PE's metadata.

This part of the descriptor satisfies the requirement that the aggregate can provide specific, static values that override the default parameter settings for constituent analytics. It does not address the mapping of aggregate parameters to constituent parameters; this is covered in the next section.

5.8.2.4 Mappings

An aggregate descriptor, in its `ProcessingElementMetadata`, may declare configuration parameters that will be exposed to callers of the aggregate analytic. In order for settings of these parameters to affect the aggregate's processing, each parameter declared by the aggregate must be *mapped* to one or more parameters declared by constituent components of the aggregate.

Rather than define a mapping mechanism specific to configuration parameters, to enable future extensibility we propose a more general mapping mechanism that could map any label used in an aggregate to a different label used by a constituent. See Section 7.1 for more discussion of this general mapping mechanism.

The aggregate descriptor specifies a set of `Mapping` element. Each Mapping element includes the following features:

- `subject`, which determines the type of thing being mapped. Currently UIMA only recognizes the value `ConfigurationParameters`, meaning that this mapping applies to the names of configuration.
- `aggregateLabel`, which is the name that the aggregate uses for this element (e.g., parameter)
- `constituentLabel`, which is the name that the constituent uses for this element (e.g., parameter)

Note that if a given constituent configuration parameter has both a mapping and an override (as described in 5.8.2.3), the mapping takes precedence. A framework might wish to report a warning in this case.

5.8.2.5 Input Specification

For each constituent, the aggregate descriptor may declare an `inputSpecification` which is used to determine which objects should be bound to handles declared in the Analytic's Behavioral Specification.

The `inputSpecification` for a constituent is a predicate against the CAS that identifies specific objects in the CAS that the analytic must[14] use as its input. As noted in Section 5.4 Behavioral Metadata Specification, the constituent analytic defines *handles* that the aggregate may bind to objects when the constituent analytic is called.

For example, the constituent component may declare in its behavioral specification:

```
  <analyzes handle="ex1Analyzes"
    expr="select(s | s.oclKindOf(ex::TextDocument))"</analyzes>
```

indicating that it can analyze any CAS object of type `ex::TextDocument`. The aggregate assembler may wish, however, for this analytic to only analyze one particular instance of `ex::TextDocument`, say the instance whose `name` feature equals "DetaggedText". The aggregate could do this using the input specification:

```
  <inputSpecification handle="ex1Analyzes">
    expr="select(s | s.oclKindOf(ex::TextDocument) and
s.name="DetaggedText")</inputSpecification>
```

Also, we need to allow such binding expressions to refer to handles declared in the *aggregate's* behavioral specification. So for example the aggregate may declare in its behavioral specification:

```
  <analyzes handle="mySofas"
    expr="select(s | s.oclKindOf(ex::TextDocument))"</analyzes>
```

This indicates that the aggregate operates on any instance of `ex::TextDocument`. The aggregate assembler may wish to bind the handle `ex1Sofas` in a constituent analytic to the same set of objects that was bound to `mySofas` by the caller of the aggregate. This could be done by the input specification:

```
  <inputSpecification handle="ex1Analyzes">
    expr="mySofas"</inputSpecification>
```

The general rule is that handles declared in the aggregate's behavioral specification are referenceable from the input specifications. We may need to specify what happens in the case of name collisions.

5.8.3 XML Schema

As for the Analytic Metadata, we use the XMI standard to define the XML schema for the Aggregate Descriptor based on the UML Diagram in Figure 13.

The XML Schema is included as an appendix. In this section we show an example descriptor that conforms to the schema. Each of the major sections of the descriptor is identified by a comment. The contents of each section directly correspond to the features in the UML diagram.

```
<?xml version="1.0" encoding="ASCII"?>
<agg:AggregateDescriptor xmi:version="2.0"
```

[14] See Candidate Compliance Point in Section 5.4.4.

```
  xmlns:xmi="http://www.omg.org/XMI"
  xmlns:agg="http://docs.oasis-
open.org/uima/aggregateDescriptor.ecore">
  <metadata>
    <identification
      symbolicName="com.ibm.uima.example.NamesAndPersonTitlesAggregate"
      name="Names and Person Titles Aggregate"
      description="Detects names and person titles in a text document."
      vendor="IBM" version="1.0"/>

    <configurationParameter name="PersonTitlePatterns"
      description="Regular expression patterns for person titles"
      type="String" multiValued="true">
      <defaultValue>Mr\.|Mrs\.|Dr\.</defaultValue>
      <defaultValue>Lt\.|Capt\.|Maj\.|Col\.|Gen\.</defaultValue>
    </configurationParameter>

    <typeSystemReference
      uri="http://sith.watson.ibm.com/types/exampleTypeSystem.ecore"/>

    <behavioralMetadata precondition="true"
      postcondition="ex1Creates->forAll(a | ex1Analyzes->exists(r |
a.sofa=r))">
      <analyzes handle="mySofas"
        expr="select(s | s.oclKindOf(ex::TextDocument))"/>
      <creates handle="ex1Creates"
        expr="Collection(ex::PersonTitle), Collection(ex::Name)"/>
    </behavioralMetadata>
  </metadata>

  <component
    wsdlUrl="http://localhost:8080/services/NameDetector?wsdl">
    <inputSpecification handle="textDocuments" expr="mySofas"/>
  </component>

  <component
    wsdlUrl="http://localhost:8080/services/PersonTitleAnnotator?wsdl">
    <mapping subject="ConfigurationParameters"
      aggregateLabel="PersonTitlePatterns"
      constituentLabel="RegExPatterns"/>
    <inputSpecification handle="ex1Analyzes" expr="mySofas"/>
  </component>

  <flowController
    wsdlUrl="http://localhost:8080/services/FixedFlowController?wsdl"/>
</agg:AggregateDescriptor>
```

6 Interoperability Case Studies

We have worked with several different frameworks and a wide array of analytics for unstructured information. Many elements of this proposal for a standard architecture based on UIMA, emerged from, or were influenced by our experiences adapting or bridging Apache UIMA[15] to

[15] Technically, these interoperability case studies were not carried out with Apache UIMA but instead with IBM's UIMA SDK implementation prior to its donation to Apache. However, to avoid introducing another term we will continue to refer to this implementation as Apache UIMA in the following discussions.

interoperate with other frameworks. In this section we provide a brief overview of these experiences as case-studies highlighting interoperability issues that motivate various aspects of the architecture specification.

6.1 GATE Case Study

GATE, or the General Architecture for Text Engineering is a popular text analysis toolkit developed at the University of Sheffield [GATE1, GATE2]. GATE is a pluggable framework for defining and running text analytics. GATE has a long history in the NLP community. It includes several graphical tools for building components that plug into the framework. The focus and strength of GATE is supporting research in Natural Language Processing and Computational Linguistics. As such, GATE emphasizes interactive function, measurement, and hypothesis testing. The interactive components in GATE allow users to define analysis rules, grammars, and expressions, manually annotate documents for use as training and test sets, and quickly iterate through analysis development and testing. The experimental platform in GATE supports creating repeatable experiments and easily evaluating analytics and applications.

Our goal was to allow GATE analytics to run transparently in Apache UIMA, and Apache UIMA analytics to run transparently in GATE. This required a means to automatically translate back and forth between the Apache UIMA and GATE analysis data representations, and a software layer that mediates between the two frameworks. With these requirements, IBM and the University of Sheffield collaborated to build the Apache UIMA / GATE interoperability layer, which consists of four main parts:

1) a declarative data mapping that specifies how to translate between a CAS and a *GATE Document* (GATE's analog to the CAS),
2) a software component that uses the data mapping specification to translate automatically between the CAS and GATE Document,
3) a wrapper that allows a GATE processing pipeline to be used as an Apache UIMA Analytic.
4) a wrapper that allows an Apache UIMA Analytic to be used as a GATE *Processing Resource*.

The first two parts address sharing analysis data between the two frameworks, and represent the minimum amount of work that must be done to get two frameworks to interoperate. This motivated and supported the definition of *data level interoperability*, the most basic level of interoperability.

The declarative specification of the data mapping between the CAS and GATE Document was based on and enabled by Apache UIMA's declarative type system specification and the base types defined for the CAS. This provided strong evidence that the CAS Specification, Type System, and Type System Base Model are required in the architecture specification to support interoperability.

Additionally, the declarative data mapping for a given analysis component supports the specification of exactly what analysis metadata will be modified or produced. This provides both a way to describe the behavior of the component (i.e., what the component does), and a way to optimize the mapping, since the mapping layer need translate only data that will be used as input or produced as output by the foreign component. This provided evidence for the need to consider a behavioral specification as part of the architectural specification.

The last two parts of the interoperability layer allow an analytic from one system to run transparently in the other system. This went beyond the data level of interoperability and provided an example of interoperability at the *programming model level*, supporting the need for multiple levels of interoperability, and in particular a level of interoperability that enables a tight integration and reuse of components from one framework to another.

It is worth noting that rather than implement the last two parts of the Apache UIMA / GATE interoperability layer as wrappers that tightly integrate the two execution environments, we could have connected the two frameworks via a services protocol, such as SOAP. The declarative data mapping specification and translation component would still be used to provide data level interoperability, but the connection between the execution environments would be provided by SOAP. This would have achieved *services level interoperability* at the expense of creating a dependency on a services environment. Integrating at the programming model level, on the other hand, allows for a self-contained, potentially more efficient interoperability mechanism.

6.2 OpenNLP Case Study

OpenNLP Tools is an open source package of natural language processing components written in pure Java. The tools are based on Adwait Ratnaparkhi's Ph.D. dissertation (UPenn, 1998), which shows how to apply Maximum Entropy models to various language ambiguity problems. The OpenNLP Tools rely on the OpenNLP MAXENT package, a mature Java package for training and using maximum entropy models. The OpenNLP Tools package (as of Version 1.3) includes a sentence detector, tokenizer, part-of-speech tagger, noun phrase chunker, shallow parser, named entity detector, and co-reference resolver. All together these tools provide a rich and powerful set of text analysis capabilities.

Our goal was to make it possible to run the OpenNLP Tools components in Apache UIMA. This includes the ability to run the native OpenNLP components in the Apache UIMA execution environment, and to integrate the OpenNLP components in such a way that they could take input from other non-OpenNLP components, and other non-OpenNLP components could process the output of the OpenNLP components. These capabilities require a way to connect the OpenNLP components to Apache UIMA, and a way to share data between the Apache UIMA environment and the OpenNLP environment.

To connect the OpenNLP components to Apache UIMA, we decided to wrap the OpenNLP components as Apache UIMA Analytics. Since the OpenNLP Tools package already contains a collection of distinct components, we chose to wrap the tools at their existing granularity, i.e., each component was wrapped as a separate Analysis Engine. Each wrapper is responsible for pulling analysis data out of the CAS, converting it into the format expected by the OpenNLP APIs, invoking the OpenNLP component, then putting the results back into the CAS.

This resulted in a form of *programming model level interoperability*. Since the OpenNLP components are a collection of loosely integrated tools without consistent APIs and a formal data structure for analysis data, it was impossible to create a single wrapper that would translate between Apache UIMA and OpenNLP, making it impossible to create general programming model level interoperability. However, the separate wrappers that were implemented for each component, when taken as a whole, provide a tightly integrated mechanism for interoperating between Apache UIMA and OpenNLP.

The second step was to define a type system that would represent the data required as input by the OpenNLP components and the data produced as output. The OpenNLP components already had an implicit type system defined by the in-line tags used to represent analysis metadata. Using this implicit type system, we designed a corresponding UIMA type system that captured the entire domain model and implemented it in Apache UIMA.

Finally, we created a behavioral specification for each wrapper that describes the input and output capabilities of the wrapped component in terms of the UIMA type system. This made it possible to ensure that the wrapped components were assembled into correct processing sequences, and it facilitated combining the OpenNLP components with other analytics.

The type system and behavioral specifications, in combination with each of the wrappers created in the first step, provided *data level interoperability* between Apache UIMA and OpenNLP. Moreover, formalizing interoperability at this level facilitated the combination of multiple OpenNLP components by making the input, output, and ordering dependencies explicit. In other words, it became easier and less error prone to combine multiple OpenNLP components within Apache UIMA than it was in the native OpenNLP environment.

This experience clearly supported the value of data level interoperability, and the specific features of Apache UIMA used to provide that interoperability, including the CAS Specification and Type System. Furthermore, the value of a Behavioral Specification for interoperability was clearly seen due to the improved robustness it provided when combining OpenNLP components with other components.

6.3 WebFountain Semantic Super Computer Case Study

The WebFountain Semantic Super Computer was created to allow ingestion, complex annotation, storage, indexing and query processing to occur over multi-petabyte size corpora such as the Web. As such, it is highly tuned to allow very fast, distributed processing at rates of up to thousands of documents per second. Since there are dozens of analytics running on each document, it is critical that the system be highly fault tolerant, so that no one analytic crashing brings down the system -- while still facilitating a very high performance point.

This system was partially developed when the desirability of interoperating with the Apache UIMA system was identified. Specifically, there was a desire to

1. share analysis data,
2. support "plug-and-play" of analytics developed for Apache UIMA (and vice-versa), and
3. to encapsulate and share the thorny problem of running a chain or pipeline of many poorly behaved analytics.

An effort was launched to enable interoperability between WebFountain and Apache UIMA. This effort co-evolved with and influenced the UIMA specification.

WebFountain was responsible for much of the early thinking on remote service deployment and management. Such remote deployments allow individual analytics to run in a highly compartmentalized (or sand-boxed) fashion, especially since many of the analytics developed for the WebFountain project were authored in a multitude of programming languages, and/or where imported from pre-existing analytics of 3rd party developers. This "at a distance integration" via

light-weight XML allowed analytics written in C, C++, Perl, Python and Java to all run in the same analytics chain. This is especially important since, as noted above, some of these analytics crash on a rather regular basis due to, for example, many mal-formed documents on the web.

The key to interoperability was an agreement on the concept of an "annotation" as a "stand off" reference to a (possibly undefined) span of characters in the subject of analysis (as opposed to the in-line model some systems require).

The common heritage of blackboard systems can be seen in the notion that analytics operate independently (and in many cases idempotently) while making use of the results of other analytics -- in fact this is almost always the case in an SSC, since at the minimum format normalization of source documents to UTF8 is performed on every document (and most all subsequent stages make use of this normalized form).

This came together in two main points of intersection between the systems - the XMI CAS specification, and the Analysis Service interface and metadata specification.

Additionally, interoperability with the core UIMA spec is achieved by wrapping (and in part co-developing) the Apache UIMA CPM (an already UIMA compliant flow engine) and empowering it to draw documents out of and return them to the main SSC object store. Since pulling the whole CAS every time is not always needed (and can be quite slow in any event) the concept of "delta CAS" processing was introduced (and used in almost all calls). With these three areas of agreement, common annotators can be run in either of these two very different platforms.

Storage-wise, the SSC may interoperate with other UIMA-Compliant systems in that it stores its documents as CASes (without embedded index) as its native document format, which allows for very rapid additional annotation as needed (quite frequently as it turns out, as people are always performing new experiments and adding new features). These annotations are indexed with a high performance distributed index which supports search as well as other, more complex analytics.

In short, the co-evolution of the SSC and Apache UIMA via UIMA has been extremely profitable for pushing the UIMA standard to support very high speed distributed environments, and for pushing SSC to support a flexible, pluggable annotation approach.

6.4 System S Case Study

System S is a distributed stream processing platform designed to host applications for analysis and information extraction from a large number of potentially high volume digital data streams. System S provides a runtime, programming model, and services for analysis of a variety of data types, including text, transactional data, digital audio, video and image data, network packet data, instant messages, and sensor data. Examples of services include distributed scheduling and lifecycle management, automated resource (CPU, bandwidth, storage) management based on the assessed value of workload, automated/semi-automated application composition, privacy and security, and high performance content routing.

Similar to Apache UIMA and GATE, System S has a pipeline processing model where *processing elements* operate on data objects (referred to as stream data objects or SDOs in System S), which conform to UIMA analysis data in that they consist of a artifact and artifact metadata

in the form of a CAS. A processing element accepts input and produces output as a stream of SDOs. The PE operates on the SDO according to the PE role.

For example, similarly to Apache UIMA and GATE, processing elements operates on the SDO subject of analysis and any existing annotations, and produces new annotations for consumption by down-stream processing elements. Alternatively, the output SDOs of a transform PE are created by transforming (e.g., aggregating or transcoding) of information obtained from one or more incoming SDOs. This requirement to aggregate or transcode multiple system S SDO's is addressed by the UIMA CAS Multiplier.

System S provides a native programming model and Eclipse-based integrated development environment. Design objectives for System S include the ability to host processing elements developed under different analytic frameworks or programming models.

In order for System S to host Apache UIMA analytics, System S can process UIMA CASs and has introduced wrappers for Apache UIMA Analytics based on the UIMA Specifications for **type-system language** and base model, **analysis data**, **artifact metadata** and **analytic interfaces**.

As with Apache UIMA and GATE, an application (or analytic) is specified as a collection of processing elements.

However, the application composition, or flow of content amongst those processing elements follows a different paradigm in System S. In traditional workflow approaches, the data flow is defined for a single job submission, and content does not flow across multiple jobs (analytics). Dataflow may be declared, or it may be controlled programmatically. However, in System S data flow is declared and System S allows data flow across multiple application specifications or job submissions. This allows System S to adapt the data flow as jobs are submitted or canceled, without interrupting jobs already in progress.

7 Discussion Topics

7.1 Mapping Between Type-Systems

UIM applications that work with independently-developed analytics each with their own type systems typically need to map data from one type system to another. We have considered using the mapping mechanism discussed in Section 5.8.2.4 to map between type systems, but chose not to include that in this specification because it only addresses very trivial mappings. In practice, there are many complexities that may arise when mapping between type systems, for example:

- Classes in the type systems have features with different names but the same meaning. For example, one type system has a *Circle* class with a *radiusInInches* feature and another type system has a *Circle* class with an equivalent *inchesOfRadius* feature.
- Classes in the type systems have features that have different meaning but are computable from each other. For example, one type system has a *Circle* class with a *radiusInInches* feature and another type system has a *Circle* class with an *areaInSquareCentimeters* feature; there are equations to compute each from the other.
- Classes in one type system are more general than classes in another type system. For example, one type system has classes representing circles, squares, and triangles while another type system only has a class representing shapes.

- Information that is encoded as different classes in one type system is encoded as different feature values in another. For example, one type system could have a *Triangle* class with a multi-valued *lengthOfSides* feature, while another could have an *EqualateralPolygon* class with a *numberOfSides* feature; mapping from the former to the latter would take instances of *Triangle* for which all of the *lengthOfSides* are equal and map them to *EqualateralPolygon* with *numberOfSides* set to 3.
- Sets of instances in one type system correspond to different sets of instances in another type system. For example, one type system has classes describing lines while another type system has classes describing shapes; a mapper could encounter five lines that share certain end points in the former type system and could map those instances to two triangle instances in the latter type system.

In general, we expect complex mappings to be managed by application developers, not by the architecture. The architecture could provide built-in capabilities that manage some kinds of mappings but certainly cannot enable every possible mapping that a user might want

For those mappings that the architecture does not provide explicit support, it is possible to develop an analytic that implements a mapping, i.e., takes in a CAS with instances in one type system and outputs a CAS with instances in another type system. It is also possible to implement mappings in a flow controller; when the flow controller decides what the next delegate in the flow is, it would map whatever data was currently in the CAS into the type system of that next delegate[16].

There are some important best-practices issues regarding which choice a developer makes under different circumstances. Detailed best practices are outside the scope of the architectural specification; however, decisions regarding the architectural specification need to be based on the expected usage of the capabilities provided; thus it is probably worthwhile to carefully consider the advantages and disadvantages of the various mapping mechanisms available.

Another question is how to handle mapping outside of the context of an aggregate. A user may want to make UIMA components interoperate without including them in the same aggregate. One solution to this problem is for the user to create a “wrapper” aggregate on either or both components to introduce mappings. The wrapper aggregate would input or output CAS data in its type system, not the type system of its delegate. However, this solution seems potentially confusing; if it is not acceptable as a common practice, then it might be useful for the architecture to provide some other mechanisms for handling this case. On the other hand, it would be reasonable for the architectural specification to declare that interoperation of components that are not in the same aggregate is outside the scope of the architecture and thus are entirely the responsibility of the application developer.

7.2 Supporting Multiple Sessions

If a ProcessingElement service serves multiple clients concurrently, there are some important issues that rise:

1. The PE needs to maintain a different set of configuration parameters for each client.
2. A stateful PE that is aggregating information over a collection of CASes may need to have separate state for each of its clients (since they may be processing separate collections).

[16] This assumes that a Flow Controller is allowed to modify a CAS, which is an open issue.

3. A FlowController service that is shared between aggregates needs to keep separate information about each aggregate (for example the set of analytics that a CAS may be routed to depends on which Aggregate is processing that CAS).

This can be enabled by the use of sessions, where each client has a distinct session and the configuration settings or other state information are stored as data associated with that session.

We propose that UIMA does not specify how this should be done. Middleware such as Application Servers or SOAP Containers provide session management features that can be used in scenarios where this is needed.

7.3 End of Collection Processing

Some UIMA components may want to take some action when they have received the last CAS from a particular collection of CASes. Here are a few use cases:

1. A CAS Multiplier merges CASes by accumulating data from its input CASes until the data reaches 1MB in size, at which point it is output as a new CAS. This is deployed inside a collection processing engine that reads documents from a collection. Say that after the last document has been processed, the CAS Multiplier has accumulated 743KB of data towards its 1MB threshold. No more data will be received by that CAS Multiplier, but we still want the CAS Multiplier to output a CAS containing the buffered 743KB of data.
2. An aggregate that, for each input CAS it receives, segments it into pieces, annotates each piece, and then merges the results. So this could be an aggregate of 3 components: "Segmenter", "Annotator", and "Merger". The Merger component (an implementation of the CAS Multiplier interface) accumulates data from all CASes that were segments of the original CAS input to the aggregate. The Merger needs to know when it has seen the last segment of that original CAS, at which point it will output the merged CAS.
3. Building on scenario #2, if an error occurs on processing the last segment, it will never reach the CAS Multiplier. We would like the CAS Multiplier to be aware of this so that it can choose to either throw away the accumulated data, or else output a new CAS containing the incomplete data.
4. An aggregate containing a custom Flow Controller, a CAS Multiplier, and a CAS Consumer requires that when the end-of-collection event is reached the CAS Multiplier is informed first, giving it a chance to produce new output CASes. These output CASes will then be routed by the Flow Controller to the CAS Consumer. Only after all of these output CASes have been processed by the CAS Consumer should the CAS Consumer receive the end-of-collection event. (This use case can be extended to an arbitrary number of components, any number of which may be CAS Multipliers, with arbitrary order dependencies between the components.)

To address these use-cases, we had considered adding a `collectionProcessComplete` API to the Analytic interface, which would allow it to take some action at the end of the collection. However, there are a number of issues with such a design:

1. What exactly is meant by a *collection* is arbitrary and is defined by the application. In use case #1, only the application that is feeding data to the CAS Multiplier may know when there is no more data available. In use case #2, a *collection* means all CASes that were derived from one original CAS.

2. As required by use case #4, the FlowController must be able to decide the order in which the `collectionProcessComplete` event would be delivered the constituents of an aggregate analytic. This may put too great a burden on developers of Flow Controllers.
3. In multithreaded implementations, it may be difficult to ensure that calls do not get out of order, such that a CAS is delivered to an Analytic after that Analytic has already received the `collectionProcessComplete` call for the collection to which that CAS belongs. Such ordering problems can occur, for example, if a CasMultiplier can respond to the `collectionProcessComplete` call by outputting additional CASes.

Because of these complexities we are proposing that end-of-collection notification will not be provided by the framework and that applications may implement this by putting additional information in each CAS. For example, the application could define its type system so that the following information is recorded in each CAS:

- The ID or URI of the Collection from which this CAS originated
- The sequence number of this CAS within its Collection
- A flag indicating whether this is the last CAS in its Collection

This would be enough information for an Analytic to determine when it has seen all of the CASes from a given collection, even if they arrive out of order.

We may wish to include this information as part of the UIMA Base Type System Model, perhaps as part of the `SourceDocumentInformation` type proposed in Section 5.3.4.4.

Candidate Compliance Point: A framework implementation may provide support for automatically populating these fields – this is a possible compliance point.

7.4 Provenance

Given a set of analysis results, a user or a system may wish to know where those results came from. There are a wide variety of uses for that information: debugging, security, automatically generated explanations, machine learning, regulatory compliance auditing, etc.

There is a virtually limitless range of provenance information that users might want for a given object, for example:

- What is the source of this object?
  - What component created this object?
  - What time was this object created?
  - In which computer/process/thread was this object created?
  - Which rule in a rule-based component motivated the creation of this object?
  - Which other object(s) motivated the creation of this object?
    - What component(s) created the object(s) that motivated the creation of this object?
    - What time was the object that motivated the creation of this object created?
    - …
- How has this object been modified?
  - What components have set the value of features in this object?
  - What components have set the value of a specified feature in this object?
  - At what times were the values of features in this object set?

    - In what computers/processes/threads have the values of features in this object been set?
- How has this object been accessed?
    - What components have accessed the value of features in this object?
    - What components have accessed the value of a specified feature in this object?
    - At what times were the values of features in this object accessed?
    - In what computers/processes/threads have the values of features in this object been accessed?

In addition, one might want provenance for an entire CAS, e.g.:

- What is the source of this CAS?
    - What artifact (document/stream/etc.) was used to populate this CAS?
    - What component created this CAS?
    - What time was this CAS created?
    - In which computer/process/thread was this CAS created?
    - What CAS is this CAS a segment of?
        - What artifact was used to populate the CAS that this CAS is a segment of?
        - What component created the CAS that this CAS is a segment of?
        - ...
- How has this CAS been modified?
    - What components have added objects to this CAS?
    - What components have set values of features of objects in this CAS?
    - ...
- How has this CAS been accessed?
    - What components have received this CAS as an input?
    - What components have accessed objects in this CAS?
    - ...

Some kinds of provenance listed above would be specific to a particular component or set of components. For example, "Which rule in a rule-based component motivated the creation of this object?" would involve information that was internal to a particular analysis algorithm. We refer to these kinds of provenance as ***application-specific provenance***. We do not expect the UIMA architectural specification to provide any explicit support for handling application-specific provenance; instead developers of type systems and components are welcome to develop their own encoding of this information and to store and use that information in the CAS.

However, much of the provenance information listed above is generally applicable to the proposed architecture. A framework could potentially record this ***architecture-general provenance*** automatically. For example, the question "What time was this CAS created?" could be addressed automatically by recording a time stamp within a CAS whenever it is created. Questions about access and modification of data by components could be addressed by recording a component name whenever CAS data is accessed or manipulated (assuming that a framework provides some sort of mechanism for obtaining the name of the currently executing component). An advantage of this over provenance recorded by the application is that the framework can be accountable for ensuring accuracy of this provenance metadata. This automatic recording is relatively easy to implement if all access to the CAS is managed by the framework through a CAS API. However, some compliant frameworks may allow developers to supply their own implementations of the CAS or may simply supply the raw data in some standard form such as an

XMI character string. Those frameworks would need to add provenance to the CAS in a distinct post-processing step after the execution of a given component (which could be problematic).

Recording provenance takes time and consumes memory. Extremely detailed provenance (e.g., recording every access to every feature of every object in a CAS) can be extremely expensive. Consequently, a framework that provides a mechanism for automatically recording provenance should make this mechanism configurable; it should be possible to run a given analysis process with or without automatic provenance recording. It may even be desirable to enable different levels of detail for provenance recording.[17] This configuration information could be specified in the appropriate descriptors; for example, an aggregate analytics descriptor could have a tag indicating that recording of provenance should be enabled during the execution of that aggregate. It would also be useful to allow a component that requires provenance information to state this requirement in its descriptor (perhaps with specific details about what provenance is needed). Thus aggregate assemblers would be able to easily discover what provenance is required by the components they are using and to configure provenance recording as needed.

One way to encode provenance for objects in a CAS would be to add extra features to all CAS classes. Under this design, a CAS implementation would need to either (A) change the internal structure of the classes depending on whether provenance recording was enabled or (B) allocate space for provenance even if no provenance recording is enabled. The former may be prohibitively complex, and the latter may be prohibitively inefficient.

Alternatively, a new class added to the base model can store a record of operations performed on a given CAS. In some cases, this may make accessing provenance information relatively expensive; some of the questions above might only be addressed by iterating through a whole sequence of steps in a trace. However, framework implementations could mitigate this cost using lookup tables, etc. The content and structure of provenance traces is an open issue. Ideally, these traces should include enough information to answer many different questions about the provenance of analysis results without being too expensive to store and/or query.

7.5 Privacy and Security

Privacy and security related to the development and use of analytics or related to data that passes among them is an important concern for many applications.

We suggest that modeling methods and system-level middleware should be used to address these concerns in any UIMA-compliant implementation. We propose that the UIMA specification need not provide specialized representation features to address privacy and security. However, we acknowledge this matter would benefit from more detailed exploration.

7.6 Configuration Parameters Affecting Behavioral Metadata

Analytics are associated with configuration parameters. These are named variables devised by the analytic developer and intended for a client or user of the analytic to set. They may indicate for example, location or name of a resource used by the analytic, but there is currently no constraint on what they may indicate or how the analytic may use them.

Besides the obvious benefits offered by facilities that can help manage and apply sets of configuration parameters to analytics at development time or run-time, a concern arises if

[17] Extremely detailed logging that is used only for debugging might be better implemented using standard logging toolkits, which provide ways to adjust the level of recording.

configuration parameters settings affect the behavior of a component such that its behavioral specification is not longer valid. In this case, then an analytic's behavioral specification would have to somehow be conjoined with different combinations of configuration parameters. Each combination and pairing, in effect, would produce a unique analytic with different behavioral specifications.

We have made a simplifying assumption that configuration parameter settings would not affect the behavioral specification. We have assumed best practices would prevail. This issue deserves closer attention.

7.7 Efficient Stream Processing

The XMI model assumes a DOM like access pattern to the objects represented. While this makes sense in a traditional object oriented environment, it is not always the best access pattern in an application space where the objects may grow to hundreds of kilobytes, and a particular stage in processing may only need a few of the slots. Such an environment suggests a SAX-like approach where unneeded slots can be skipped without the need to parse and allocate them.

This problem becomes particularly acute in the case of very large CAS stores (e.g., one holding a CAS per web page on the web) where there is the need for throughput on the scale of hundreds of thousands of CASes per second.

The challenge with the SAX-like approach is that in general, XMI makes extensive use of ID and IDREF to record references from one object to another. Forward references are allowed, which results in problems for streaming mode processors. Backward references can also be problematic since a streaming processor would not know which objects to keep track of in case they were referenced from a later object.

7.7.1 Views: An Example Streaming Problem

The proposed *View* Base type is representative of the efficient stream processing problem.

Our XMI CAS Spec defines the serialization of a View like this:

```
<xmi:XMI>
  <myproj:Person xmi:id="1" .../>
  <myproj:Person xmi:id="2" .../>
  <myproj:Person xmi:id="3" .../>
  <myproj:Person xmi:id="4" .../>
  <cas:View name="foo" members="1 2"/>
  <cas:View name="bar" members="3 4"/>
</xmi:XMI>
```

If an analysis service wanted to look at only the `Person` objects in the `foo` view (a primary use case for views), it would not be able to do so in a streaming manner, since the information about View membership is not known at the time the `Person` object is being processed.

7.7.2 Possible Solutions

7.7.2.1 Record the view name directly on each Object

In this example, the view name is a value of an attribute on each object. So as objects are processed the view information is immediately accessible.

```
<xmi:XMI>
  <myproj:Person xmi:id="1" _view="foo" .../>
  <myproj:Person xmi:id="2" _view="foo" .../>
  <myproj:Person xmi:id="3" _view="bar" .../>
  <myproj:Person xmi:id="4" _view="bar" .../>
  <cas:View name="foo"/>
  <cas:View name="bar"/>
</xmi:XMI>
```

Unfortunately, this `_view` attribute would not in general be part of the user-defined type system (e.g., Ecore model) and the XMI-spec doesn't permit us to add such "system-level" metadata as an attribute.

7.7.2.2 Reorder things so that the Views come first

In this example, the view information comes first facilitating the computation by allowing it to store the view membership before processing each object.

```
<xmi:XMI>
  <cas:View name="foo" members="1 2"/>
  <cas:View name="bar" members="3 4"/>
  <myproj:Person xmi:id="1" .../>
  <myproj:Person xmi:id="2" .../>
  <myproj:Person xmi:id="3" .../>
  <myproj:Person xmi:id="4" .../>
</xmi:XMI>
```

For example, a streaming processor could build a map from ID to view name when it saw the View objects, and then refer to that map when processing each Person object to determine if the object belonged to a particular view.

One issue with this is that the XMI specification doesn't appear to require an XMI-complaint processor to maintain the element order. However, the UIMA specification may note that XMI CASes in a particular order (say, `Views` first) are most efficient for streaming services and that it is recommended that data be serialized in that form.

7.7.2.3 A Streaming CAS Representation

In addition to the XMI CAS representation UIMA may specify a separate streaming representation for the CAS that has different goals.

7.7.3 Workarounds for Inefficiencies in Streaming Processing

Even without a specially designed streaming CAS representation, there are a few solutions that a CAS Store might implement in the interest of performance:

7.7.3.1 View Specific Rewrite Support

A CAS store might silently process data so as to add the `_view` element suggested above, and strip it on return. This addition and deletion is not that expensive (since it can be done in the write-out and read-in streams).

7.7.3.2 Side index of Views

Alternatively, header could be placed in the stream holding much of this information. Again, this moves somewhat away from the ability to use pure XML stores, but may allow them to be used at speed with minimal changes.

7.7.3.3 Take the hit

If view specific calls are not the norm, a system can simply switch to a DOM approach when views are used. This is less satisfying as it requires two access modes, but again may require minimal programming on the CAS Store implementer's part. If Views are popular in a system, this approach works less well.

The problem will become more acute if other pointer chasing approaches are needed in an application (e.g., "pull this attribute with closure of everything it references"). The effective processing of such queries may require memory-based XML databases - even XML enabled DBs, however, may have trouble dealing with such recursive calls.

8 References

[ApacheUIMA1] http://incubator.apache.org/uima/

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

[CAT1] Mardis, S., Burger, J., Anand, P., Anderson, D., Griffith, J., Light, M., McHenry, C., Morgan, A., and Ponte, J. 2001. Qanda and the Catalyst Architecture. In *Proceedings of the Tenth Text REtrieval Conference (TREC-10)* Gaithersburg, MD.

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

[EJB1] http://java.sun.com/products/ejb/

[EMF1] http://dev.eclipse.org/viewcvs/indextools.cgi/*checkout*/org.eclipse.emf/doc/org.eclipse.emf.doc/references/overview/EMF.html

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

[GATE1] http://gate.ac.uk/

[GATE2] K. Bontcheva, H. Cunningham, V. Tablan, D. Maynard, and H. Saggion. "Developing Reusable and Robust Language Processing Components for Information Systems using GATE." In Proceedings of the 3rd International Workshop on Natural Language and Information Systems (NLIS'2002), IEEE Computer Society Press (2002).

[Laprun1] C Laprun et al. "A Practical Introduction to ATLAS." In *Proceedings LREC 2002*, Las Palmas.

[Mal1] McCallum, Andrew Kachites. "MALLET: A Machine Learning for Language Toolkit." http://mallet.cs.umass.edu. 2002.

[MOF1] http://www.omg.org/docs/ptc/04-10-15.pdf

[OCL1] http://www.omg.org/technology/documents/formal/ocl.htm

[ONLP1] http://opennlp.sourceforge.net/

[OSGi1] http://www.osgi.org/

[OWL1] http://www.w3.org/TR/owl-ref

[OWL-S1] http://www.w3.org/Submission/OWL-S/

[SystemS1] Multi-site cooperative data stream analysis. ACM SIGOPS. Volume 40 , Issue 3 (July 2006)

[SPARQL] http://www.w3.org/TR/rdf-sparql-query/

[TAL1] Neff, Mary, Byrd, Roy, and Boguraev, Branamir, 2004. The Talent System: Textract Architecture and Data Model. Natural Language Engineering. Volume 10 , Issue 3-4 (September 2004)

[TIP1] Grishman, Ralph, editor. Tipster Phase II Architecture Design Document Version 1.52, July 1995. http://cs.nyu.edu/cs/faculty/grishman/tipster.html

[UDDI1] http://www.uddi.org/

[UIMASrc1] http://uima-framework.sourceforge.net/

[UML1]
http://www.omg.org/technology/documents/formal/uml.htm

[WSDL1] http://www.w3.org/TR/wsdl20/

[WF1] D. Gruhl, L. Chavet, D. Gibson, J. Meyer, P. Pattanayak, A. Tomkins, and J. Zien. How to build a WebFountain: An architecture for very large-scale text analytics. In *IBM Systems Journal*, 43:1. 2004. http://www.research.ibm.com/journal/sj/431/gruhl.html.

[WF2] Dill, Stephen, Eiron, Nadav, Gibson, David, Gruhl, Daniel, Guha, R., Jhingran, Anant, Kanungo, Tapas, Rajagopalan, Sridhar, Tomkins, Andrew, Tomlin, John A., and Zien, Jason Y. 2003. SemTag and Seeker: Bootstrapping the semantic web via automated semantic annotation. In Proceedings of the Twelfth International World Wide Web Conference (WWW2003), Budapest, Hungary.

[XMI1] http://www.omg.org/docs/formal/03-05-02.pdf

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

[XML1] http://www.w3.org/TR/xml-names11

[XMLFrags1] D. Carmel, Y. Maarek, M. Mandelbrod, Y. Mass and A. Soffer. "Searching XML Documents via XML Fragments." Proceedings of the 26th annual international ACM SIGIR conference on Research and development in information retrieval. Toronto, Canada (2003).

9 Appendices/Attachments

9.1 SOAP Bindings and Example SOAP Messages

This section specifies the WSDL bindings that bind the abstract service descriptions in Section 5.7 Service WSDL Descriptions to the SOAP protocol.

There is a separate `<wsdl:binding>` element for each port type.

9.1.1 The Analyzer SOAP Binding

```
<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>
```

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.

9.1.2 The CasMultiplier SOAP 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: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="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: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: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:operation>
</wsdl:binding>
```

9.1.3 The FlowController SOAP 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: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="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: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: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: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:operation>
  </wsdl:binding>
```

9.1.4 SOAP Service Example

To complete the WSDL Definitions we need to specify a *service*, which is a collection of *ports*. Each port refers to one of the SOAP bindings in the previous section, and associates with it an actual network address.

This part of the WSDL is not specified exactly under the UIMA specification. Obviously the network address would vary for each instance of a service. Also a service can choose which port types to implement. The followings example shows that a service can implement both the `Analyzer` port type and the `CasMultiplier` port type. It is equally acceptable for a service to implement only one of the port types.

```
  <!-- Define an example service as including both portTypes -->
  <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/CasMultiplierPor
t"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

9.1.5 Example SOAP Messages

The following are examples of SOAP messages that conform to this spec, provided for illustrative purposes.

9.1.5.1 Get Metadata Request

```
<?xml version="1.0" encoding="utf-8"?>
<soapenv:Envelope
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <getMetadata xmlns=""/>
  </soapenv:Body>
</soapenv:Envelope>
```

9.1.5.2 Get Metadata Response

```
<?xml version="1.0" encoding="utf-8"?>
<soapenv:Envelope
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <getMetadataResponse xmlns="">
      <metadata xmlns="http://docs.oasis-
open.org/uima/peMetadata.ecore">
        <identification
          description="Detects person titles in a text document."
          name="Person Title Annotator"
          symbolicName="org.example.PersonTitleAnnotator"
          vendor="IBM" version="1.0">
          <version2 xsi:nil="true"/>
        </identification>
        <!-- Other information (Configuration Parameters,
          Type System, Behavioral Spec.) would go here:
          see Processing Element Metadata section.-->
      </metadata>
    </getMetadataResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

9.1.5.3 Process CAS Request

```
<?xml version="1.0" encoding="utf-8"?>
<soapenv:Envelope
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <processCas xmlns="">
      <cas xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
        xmlns:cas="http://docs.oasis-open.org/uima/cas.ecore"
        xmlns:ex="http:/example.org/example.ecore">
        <ex:Quotation xmi:id="1"
          text="If we begin in certainties, we shall end with doubts;
but if we begin with doubts and are patient with them, we shall end in
certainties."
          author="Francis Bacon"/>

        <cas:LocalSofaReference xmi:id="2" sofaObject="1"
          sofaFeature="text"/>

        <ex:Clause sofa="2" begin="0" end="30"/>
      </cas>
      <inputBindings xsi:nil="true"/>
    </processCas>
  </soapenv:Body>
</soapenv:Envelope>
```

9.1.5.4 Process CAS Response

```
<?xml version="1.0" encoding="utf-8"?>
<soapenv:Envelope
```

```
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <processCasResponse xmlns="">
      <cas xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
        xmlns:cas="http://docs.oasis-open.org/uima/cas.ecore"
        xmlns:ex="http:/example.org/example.ecore">
        <ex:Quotation xmi:id="1"
          text="If we begin in certainties, we shall end with doubts;
but if we begin with doubts and are patient with them, we shall end in
certainties."
          author="Francis Bacon"/>

        <cas:LocalSofaReference xmi:id="2" sofaObject="1"
          sofaFeature="text"/>

        <ex:Clause sofa="2" begin="0" end="30"/>
        <ex:Pronoun sofa="2" begin="3" end="5"/>
        <ex:Pronoun sofa="2" sofaFeature="text" begin="29" end="31"/>
      </cas>
    </processCasResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

9.1.5.5 Alternative Process CAS Response, using XMI "differences"

```
<?xml version="1.0" encoding="utf-8"?>
<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <processCasResponse xmlns="">
      <cas xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI">
        <xmi:Difference>
          <target href="input.xmi"/>
          <xmi:Add addition="p1">
          <xmi:Add addition="p2"/>
        </xmi:Difference>
        <ex:Pronoun xmi:id="p1" sofa="2" begin="3" end="5"/>
        <ex:Pronoun xmi:id="p2" sofa="2" sofaFeature="text" begin="29"
end="31"/>
      </cas>
    </processCasResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

Note that the XMI Differences Schema is actually bit cumbersome to use for this purpose. See 5.7.6 PE Service Specification - Open Issues for a discussion.

9.2 Referenced XML Schemata

9.2.1 PE Metadata (uima.peMetadataXMI.xsd)

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:ecore="http://www.eclipse.org/emf/2002/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/peMetadata.ecore">
<xsd:import namespace="http://www.eclipse.org/emf/2002/Ecore"
  schemaLocation="ecoreXMI.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="pemd: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="pemd: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="pemd:TypeSystemReference"/>
<xsd:complexType name="BehavioralMetadata">
  <xsd:choice maxOccurs="unbounded" minOccurs="0">
    <xsd:element name="let" type="pemd:HandleExpression"/>
    <xsd:element name="analyzes" type="pemd:HandleExpression"/>
    <xsd:element name="inspects" type="pemd:HandleExpression"/>
    <xsd:element name="modifies" type="pemd:HandleExpression"/>
    <xsd:element name="deletes" type="pemd:HandleExpression"/>
    <xsd:element name="creates" type="pemd:HandleExpression"/>
    <xsd:element ref="xmi:Extension"/>
```

```
    </xsd:choice>
    <xsd:attribute ref="xmi:id"/>
    <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
    <xsd:attribute name="precondition" type="xsd:string"/>
    <xsd:attribute name="postcondition" type="xsd:string"/>
  </xsd:complexType>
  <xsd:element name="BehavioralMetadata"
    type="pemd:BehavioralMetadata"/>
  <xsd:complexType name="HandleExpression">
    <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="handle" type="xsd:string"/>
    <xsd:attribute name="expr" type="xsd:string"/>
  </xsd:complexType>
  <xsd:element name="HandleExpression"
    type="pemd:HandleExpression"/>
  <xsd:complexType name="ProcessingElementMetadata">
    <xsd:choice maxOccurs="unbounded" minOccurs="0">
      <xsd:element name="configurationParameter"
        type="pemd:ConfigurationParameter"/>
      <xsd:element name="identification"
        type="pemd:Identification"/>
      <xsd:element name="typeSystemReference"
        type="pemd:TypeSystemReference"/>
      <xsd:element name="behavioralMetadata"
        type="pemd:BehavioralMetadata"/>
      <xsd:element name="extension" type="pemd: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="pemd: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="pemd:Extension"/>
  <xsd:complexType name="ConfigurationParameterSettings">
    <xsd:choice maxOccurs="unbounded" minOccurs="0">
      <xsd:element name="ConfigurationParameterSetting"
        type="pemd:ConfigurationParameterSetting"/>
      <xsd:element ref="xmi:Extension"/>
    </xsd:choice>
    <xsd:attribute ref="xmi:id"/>
    <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
  </xsd:complexType>
```

```
  <xsd:element name="ConfigurationParameterSettings"
    type="pemd:ConfigurationParameterSettings"/>
  <xsd:complexType name="ConfigurationParameterSetting">
    <xsd:choice maxOccurs="unbounded" minOccurs="0">
      <xsd:element name="values" 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="parameterName" type="xsd:string"/>
  </xsd:complexType>
  <xsd:element name="ConfigurationParameterSetting"
    type="pemd:ConfigurationParameterSetting"/>
</xsd:schema>
```

9.2.2 Aggregate Descriptor (uima.aggregatedescriptorXMI.xsd)

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema
  xmlns:agg="http://docs.oasis-open.org/uima/aggregateDescriptor.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/aggregateDescriptor.ecore">
  <xsd:import namespace="http://www.omg.org/XMI"
    schemaLocation="XMI.xsd"/>
  <xsd:import
    namespace="http://docs.oasis-open.org/uima/peMetadata.ecore"
    schemaLocation="uima.peMetadataXMI.xsd"/>
  <xsd:complexType name="AggregateDescriptor">
    <xsd:choice maxOccurs="unbounded" minOccurs="0">
      <xsd:element name="metadata"
        type="pemd:ProcessingElementMetadata"/>
      <xsd:element name="component"
        type="agg:ComponentDeclaration"/>
      <xsd:element name="flowController"
        type="agg:ComponentDeclaration"/>
      <xsd:element ref="xmi:Extension"/>
    </xsd:choice>
    <xsd:attribute ref="xmi:id"/>
    <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
  </xsd:complexType>
  <xsd:element name="AggregateDescriptor"
    type="agg:AggregateDescriptor"/>
  <xsd:complexType name="ComponentDeclaration">
    <xsd:choice maxOccurs="unbounded" minOccurs="0">
      <xsd:element name="mapping" type="agg:Mapping"/>
      <xsd:element name="inputSpecification"
        type="pemd:HandleExpression"/>
      <xsd:element name="parameterOverrides"
        type="pemd:ConfigurationParameterSettings"/>
      <xsd:element ref="xmi:Extension"/>
    </xsd:choice>
    <xsd:attribute ref="xmi:id"/>
    <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
```

```
    <xsd:attribute name="wsdlUrl" type="xsd:string"/>
  </xsd:complexType>
  <xsd:element name="ComponentDeclaration"
    type="agg:ComponentDeclaration"/>
  <xsd:complexType name="Mapping">
    <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="subject" type="xsd:string"/>
    <xsd:attribute name="aggregateLabel" type="xsd:string"/>
    <xsd:attribute name="constituentLabel" type="xsd:string"/>
  </xsd:complexType>
  <xsd:element name="Mapping" type="agg:Mapping"/>
</xsd:schema>
```

9.2.3 Base XMI Schema (XMI.xsd)

```
<?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>
```

9.2.4 PE Service (uima.peServiceXMI.xsd)

```
<?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>
```