Building an example application with the Unstructured Information Management Architecture

by D. Ferrucci
A. Lally

IBM's Unstructured Information Management Architecture (UIMA) is a software architecture for developing and deploying unstructured information management (UIM) applications. In this paper we provide a high-level overview of the architecture, introduce its basic components, and describe the set of tools that constitute a UIMA development framework. Then we take the reader through the steps involved in building a simple UIM application, thus highlighting the major UIMA concepts and techniques.

Structured information may be characterized as information whose intended meaning is unambiguous and explicitly represented in the structure or format of the data. The canonical example of structured information is a relational database table. *Unstructured information* may be characterized as information whose intended meaning is only loosely implied by its form and therefore requires interpretation in order to approximate and extract its intended meaning. Examples include natural language documents, speech, audio, still images, and video.

One reason we focus on deriving implied meaning from unstructured information is that 80 percent of all corporate information is unstructured.[1] An even more compelling reason is the rapid growth of the Web and the perceived value of its unstructured information to applications that range from e-commerce and life science applications to business and national intelligence.

An *unstructured information management (UIM) application* may be generally characterized as a software system that analyzes large volumes of unstructured information in order to discover, organize, and deliver relevant knowledge to the end user.[2] An example is an application that processes millions of medical abstracts to discover critical drug interactions. Another example is an application that processes tens of millions of documents to discover evidence of probable terrorist activities. We have seen a sharp increase in the use of UIM analytics (the analysis component of UIM applications), in particular text and speech analytics, within the class of applications designed to exploit the large and rapidly growing number of sources of unstructured information.[3]

In analyzing unstructured content, UIM applications make use of a variety of technologies including statistical and rule-based natural language processing (NLP), information retrieval, machine learning, ontologies, and automated reasoning. UIM applications may consult structured sources to help resolve the semantics of the unstructured content. For example, a database of chemical names can help in focusing the analysis of medical abstracts. A database of terrorist organizations and their locations can help in analyzing documents for terror-related activities. A UIM application generally produces structured information resources that unambiguously represent content derived from unstructured information input. These structured resources are made accessible to the end user through a set of application-appropri-

ate access methods. A simple example is a search index and query processor that makes documents quickly accessible by topic and ranks them according to their relevance to key concepts specified by the user. A more complex example is a formal ontology and inference system that, for example, allows the user to explore the concepts, their relationships, and the logical implications contained in a collection consisting of millions of documents.

IBM is in the business of enabling on-demand delivery of information. This involves providing hardware, middleware, and database management and collaboration software. In the rapidly growing world of natural language text, speech, audio, and video, these traditional strengths must be coupled with systems that enable the analysis and integration of unstructured information. IBM's Unstructured Information Management Architecture (UIMA) is a component-based software architecture for developing UIM applications. The UIMA project, which originated at the IBM Research Division, was intended to leverage a wealth of research results in the area of unstructured information analysis toward advancing science and impacting IBM products and services. Although UIMA was designed for the entire range of unstructured information types, the implementation discussed in this paper focuses on natural language text.

UIMA has many features in common with other software architectures for language engineering such as GATE[4,5] and ATLAS.[6] Each of these systems isolates the core algorithms that perform language processing from system services such as storing of data, communication between components, and visualization of results. However, UIMA's emphasis on transferring UIM technologies to products has led to a richer architecture that allows integrating applications with a host of enterprise products (e.g., WebSphere* Portal Server, Lotus* Workplace) and a variety of middleware and platform options.

The rest of the paper is structured as follows. In the next section, we provide a high-level overview of the architecture, introduce the basic components, and discuss the component architecture of UIMA. Then we describe the UIMA development frameworks and the developer roles defined for building UIM applications. Next, we take the reader through the steps involved in building a simple UIM application, *Meeting Finder,* designed to highlight the main UIMA concepts and its methodology. We present the implementation results of our project and relate this work to similar efforts elsewhere. We conclude with a summary of the work and some ideas for future research.

Architecture overview

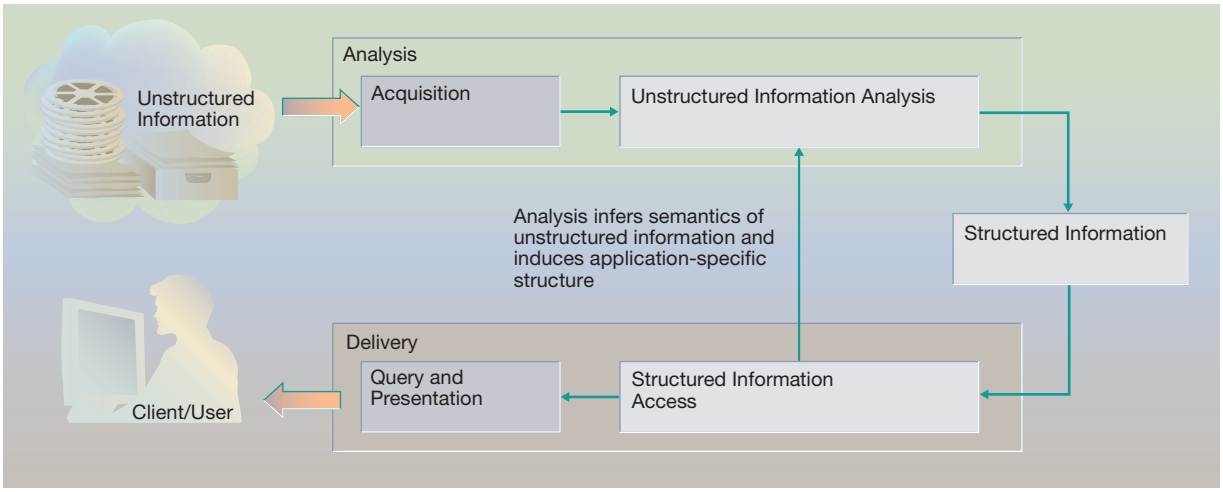
A UIM application may be thought of as comprising two phases: analysis and delivery. In the analysis phase, collections of documents are acquired and analyzed. The results are stored in one or more forms as needed for the delivery phase. In the delivery phase, the analysis results, possibly together with the original documents or other structured information, are made accessible to the application's end user through application-appropriate access methods and interfaces.

For example, the analysis phase of the application illustrated in Figure 1 may include tokenization and semantic class detection on input documents. Consulting structured sources such as dictionaries or ontologies, the application may find and annotate classes of entities such as organizations, persons, locations, and events. In addition, the application may annotate classes of relationships such as *located_in, attends_event,* or *employee_of.* The result of the analysis phase would be a search engine index that contains tokens as well as the detected entities and relationships. In the delivery phase, the application might present a query interface and allow the user to search for documents that contain some Boolean combination of tokens, entities, and relationships through a semantic search engine.

UIMA is focused on providing the conceptual foundation and component infrastructure for supporting the discovery, development, composition, and deployment of unstructured information analysis capabilities and their integration with structured information sources. Delivery of the information to the application's end users is open-ended and typically very specialized. Consequently, although the delivery phase is a critical aspect of a UIM application, it is not directly addressed by the architecture.

For a broader description of the UIMA high-level architecture, we refer the reader to a general UIMA overview paper.[2] In an attempt to provide the reader with a deeper sense of how UIMA development frameworks support the design, development, integration, and deployment of unstructured information analysis functions, we focus here on using UIMA to implement an example application. In the remainder of this section we discuss unstructured informa-

Figure 1 Example UIM application



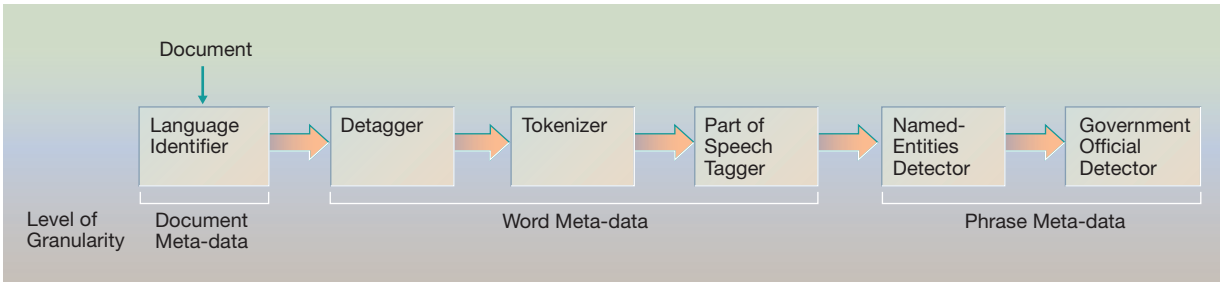
tion analysis at two levels: analysis of a single document (document-level) or of a collection of documents (collection-level).

Document-level analysis. In *document-level analysis,* the focus is on an individual document (as opposed to a collection of documents). The analysis component takes that document as input and outputs its analysis as meta-data describing portions of the original document. These may refer to the document as a whole or to any sized region of the document. In general, we use the term *document* to refer to an arbitrarily grained element of unstructured information processing. For example, for a UIM application, a document may represent an actual text document, a fragment of such a document or even multiple such documents. Examples of document-level analyses include language detection, tokenization, syntactic parsing, named-entity detection, classification, summarization, and translation. In each of these examples, the analysis component examines the document and associated meta-data and produces additional meta-data as a result of its analysis.

An analysis component may be implemented by composing a number of more primitive components, as illustrated in Figure 2. In this figure a number of components are assembled in series in order to implement a "government official detector" function. The output of each stage consists of the document with the result of the analysis. For example, the output of the language identifier component consists of the document annotated with a label that specifies its language; the output of the detagger component consists of the document with HTML tags identified and content extracted, and so on. Composition of analysis components is a valuable aspect of UIMA because implementing each of the analysis components may require specialized skills. Thus, when complex programs are built by uniquely skilled individuals or teams, reuse through composition becomes particularly valuable in reducing redundant effort.

The fundamental processing component for document analysis in UIMA is the *analysis engine,* which for text documents is known as the *text analysis engine* (TAE). TAEs have a standardized interface and may be declaratively composed to build aggregate analysis capabilities. TAEs built by composition have a recursive structure—the primitive TAEs may be core analysis components implemented in C++ or Java**, whereas aggregate TAEs are composed of such primitive TAEs or other aggregate TAEs. Because aggregate TAEs and primitive TAEs have exactly the same interfaces, it is possible to recursively assemble advanced analysis components from more basic elements while the implementation details are transparent to the composition task. Composing text processing modules to form aggregate analysis capabilities is not new to the field of natural language engineering. For example, TAEs are functionally analogous to Processing Resources in the GATE architecture.[4,5]

Figure 2 An analysis component as a series of primitive components



The document analysis operates on a common data structure that consists of the original document (the subject of analysis) and its associated meta-data. A container object is also defined that provides interfaces for indexing, accessing, and updating the common data structure. This data structure and container are collectively referred to as the Common Analysis System (CAS).[7] We should point out that different interfaces to the CAS are possible. Whatever interface is used, during analysis the common data structure, representing the subject of analysis and its meta-data, is made accessible to an application-specified sequence of TAEs. In this paper we refer to both the common data structure shared by a sequence of TAEs and a container object that provides an interface to this data structure as the CAS.

Each TAE in a specified sequence operates on the input CAS, performs its analysis function, and then updates the CAS with additional meta-data. The updated CAS is the result of that analysis. For example, see Figure 3, which depicts a government official detector TAE recursively composed from core analysis components. After the "Aggregate Analysis Engine: Named-Entity Detector" completes its analysis, the CAS contains meta-data including the Persons, Places, and Organizations detected in the document. The "Government Official Detector" then reads the CAS, considers the Persons, identifies which of those Persons are Government Officials, and records this information in the CAS.

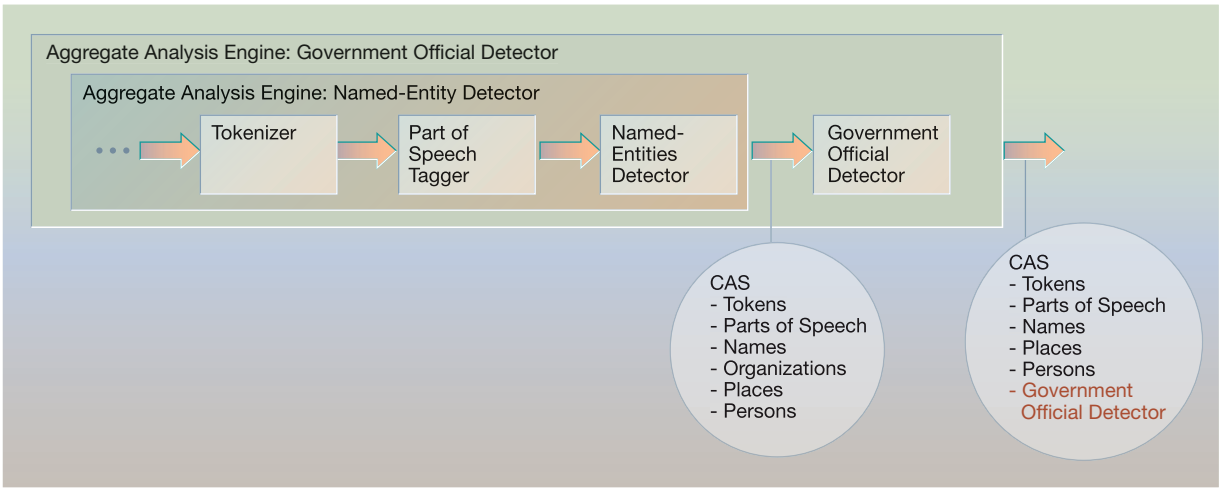
The CAS contains meta-data in the form of *annotations,* which are similar to the annotations used in other systems, beginning with TIPSTER.[8] An annotation associates meta-data (e.g., a label) with a region in the artifact that is being analyzed; for text documents, this is done by identifying start and end positions within the text. An example would be the label "token" being associated with each word in the document. Another would be the label "person" annotating the mentions of people in the document. The CAS uses *stand-off* annotations rather than inline markup, as this achieves greater flexibility.[9] UIMA specifies an eXtensible Markup Language (XML) representation for CAS data models (i.e., schemas), a similar concept to the Meta-Annotation Infrastructure for ATLAS (MAIA).[6] UIMA also defines a CAS interface for indexing, accessing, and updating the contents of a CAS.[7]

A number of development frameworks have been implemented for the UIMA architecture. Two major ones provide implementations of the CAS interface in both C++ and Java. A native object-oriented programming interface to the CAS has been developed for Java, named JCas.[10] JCas automatically generates Java classes from developers' annotation models, allowing them to use plain Java to interact with annotation data models, and thus reducing their learning curve for using UIMA. The process of using JCas is described in more detail as we discuss the example UIMA application later in this paper.

UIMA specifies component interfaces for implementing, composing, and deploying analysis engines and for representing, routing, and accessing analysis results. The architecture specifies three varieties of technical TAE interfaces to support a broad set of implementation and deployment options. These are the C++ interface, the Java interface, and the network service interface (when analysis components are distributed over a network).

Collection-level analysis. In *collection-level analysis* the UIM application processes an entire collection of documents. Whereas each document may undergo document-level analysis as part of collection process-

Figure 3 An encapsulation and composition of TAEs



ing, the results of collection-level analysis represent inferences made over the entire collection. Examples of collection-level results are glossaries, dictionaries, databases, search indexes, and ontologies. UIMA does not restrict the content, form, or technical realization of collection-level results; they are treated generally as structured information resources.

The fundamental UIMA processing element for collection analysis is the *collection-processing engine* (CPE). CPEs are responsible for controlling the application of TAEs to elements of a collection and managing the routing of results to *CAS consumers.* These components are the final stage in a CPE, and thus, they consume but do not produce CAS objects. CAS consumers may perform a wide variety of specialized functions ranging from simply storing CASs for subsequent access by the application to computing global inferences that consider the analyses of all the documents in the collection.

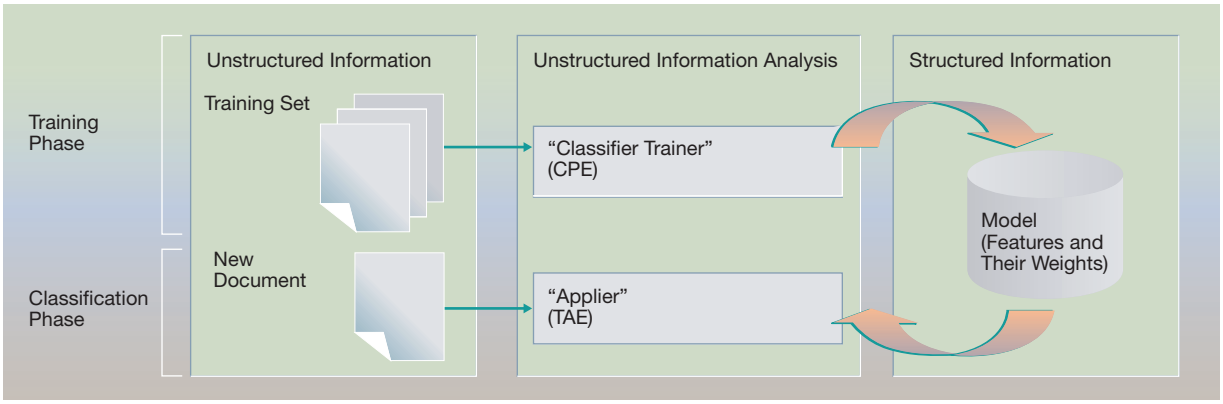
The CPE includes APIs that allow the application to specify the document-level analysis that should be performed and the CAS consumers that should have access to the results. The API includes methods to support the following features:

1) Control—start, stop, pause, resume, restart from checkpoint.
2) Filtering—allows processing of a subset of the collection, specified by constraints on the metadata within the CAS.
3) Error handling—the user may specify how errors should be handled. For example, failed documents may be automatically retried some number of times, then skipped and added to a list of failed documents that the user can manually review.
4) Performance monitoring—monitors and reports statistics for each component such as processing time, memory usage, and error rates.
5) Parallelization—may replicate TAEs and CAS consumers, utilizing available computing resources to simultaneously process multiple documents.

The structured information feedback loop. A common feedback loop present in UIM applications produces structured resources from collection-level analysis and then uses these resources to enable subsequent unstructured information analysis. A classic example of this feedback loop is present in a type of application we refer to as the *classifier-trainer* application, depicted in Figure 4. This type of application may be decomposed into two phases: the training phase (the upper part of the figure) and the classification phase (the lower part).

In the training phase, a collection of documents, the *training set,* is analyzed by a CPE. The CPE invokes a set of TAEs and statistically analyzes the results, producing a structured resource called the *model.* The model explicitly assigns weights to the features detected by the TAEs. The classifier-trainer, in ad-

Figure 4 Feedback loop: The classifier-trainer example



dition to the model, produces a new TAE called the *applier.* The purpose of the classifier-trainer application is to create and train this new analysis engine to analyze new documents in a way that is consistent with the training set.

In the classification phase, the model is the structured resource built by the collection-level processing step and consulted by the applier when it analyzes a new document. If the algorithms used in the classifier-trainer are appropriate and the input TAE annotations sufficiently predictive of those in the training set, then the applier may accurately analyze a new document, even rivaling human performance in the analysis task.[11–13]

Other examples of this feedback loop include the construction of ontologies where a collection of documents must first be analyzed to detect concepts and the many possible mentions of the concepts throughout the documents. Looking at the entire collection improves the accuracy of concept determination. The result might be a "glossary"—a database of concepts with links to definitional resources. This database would then be used as a structured resource in subsequent analysis to detect relations between concepts required to fill out the ontology. UIMA supports the managed deployment and access of structured information resources from analysis-engine components.[2]

Semantic search. Common to many UIM applications is the integration of search technology with unstructured information analysis. Traditional search engines index the tokens or words that make up a document and then process queries as Boolean combinations of tokens. They then return a ranked list of documents that contain the combinations of tokens specified in the query. UIM applications, however, often require *semantic search,* which UIMA defines as the capability to issue queries for documents based on not just key words that appear in the document, but also any concept derived from the text by the applied analysis engines. Such concepts are represented in CAS by annotations over specified spans of text. Therefore, a UIMA-compliant search engine indexer must be capable of indexing annotations as well as tokens. Such an indexer must also support *cross-over* annotations, which are annotations whose spans intersect. Cross-over annotations can allow multiple interpretations of a sentence to be easily represented and indexed. Representing multiple parses, for example, is important for deferring sense disambiguation until more information is available.

Similarly, the query interface to a UIMA-compliant search engine must support queries of annotations as well as tokens. It must also be possible to query for nested and intersecting annotations. These UIMA indexing and query requirements are met by the Juru XML[14] search engine, which is used in the example application discussed in this paper.

Component overview. As mentioned earlier, the two primary components that UIMA specifies for addressing document-level and collection-level analysis are TAEs and CPEs. Developing analysis capabilities for UIM applications revolves around implementing, deploying, and controlling combinations of these primary components.

Drilling down to the next level of detail, UIMA specifies component architectures for implementing TAEs and CPEs from more basic components. As is the case with all component architectures, the goal is to identify a component type for each conceptually distinct function performed by the system and to ensure that the components can be easily reused and combined with one another. To support this reuse and interoperability, all UIMA components are required to be *data-driven* and *self-descriptive.*

Being data-driven means that each component's function must depend only on its input. This is important because it ensures that the only requirement for reusing a component within an aggregate system is that the component's input requirements are met. Components that have other dependencies, such as the presence of specific other components or a particular sequence of execution, are not suitable for reuse and are therefore not permitted in UIMA.

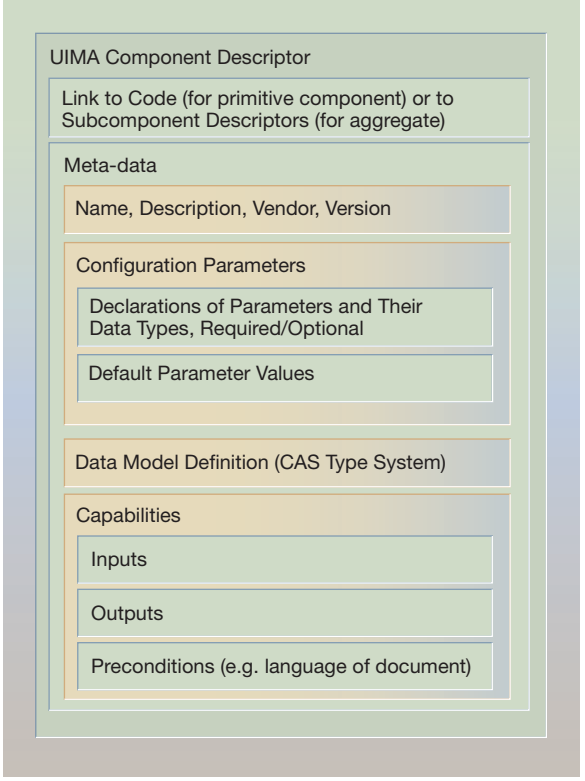
All UIMA components must also be self-descriptive, meaning that they publish declarative meta-data about themselves. The most important piece of component meta-data is the component's *capabilities,* which are its input requirements and output specification. Components may be registered in a browseable repository based on this meta-data, allowing developers to easily find components that meet their needs. This reduces accidental duplication of effort, which is a serious issue in an organization as large and dispersed as the IBM Research Division. Capability specifications must be machine-readable, enabling UIMA development frameworks to intelligently assemble components and produce TAEs and CPEs.

The declarative meta-data of a component is captured in component *descriptors.* Figure 5 shows the logical content of a descriptor. UIMA specifies an XML schema for component descriptors. In addition to capabilities, descriptors also contain configuration parameters to which values may be assigned by applications.

The remainder of this section provides a brief overview of the component interfaces that a UIMA developer must implement to create a UIM application. The UIMA development framework is used to connect these components to form TAEs and CPEs as illustrated in Figure 6.

TAE components. The primary TAE component is the *annotator.* The annotator interface implements a particular analysis function (e.g., tokenization, language detection, classification). In its simplest form the interface may be described as "process CAS." It is in the implementation of this method that the developer encodes an analysis algorithm. The UIMA framework takes an annotator and its descriptor and constructs a primitive TAE.

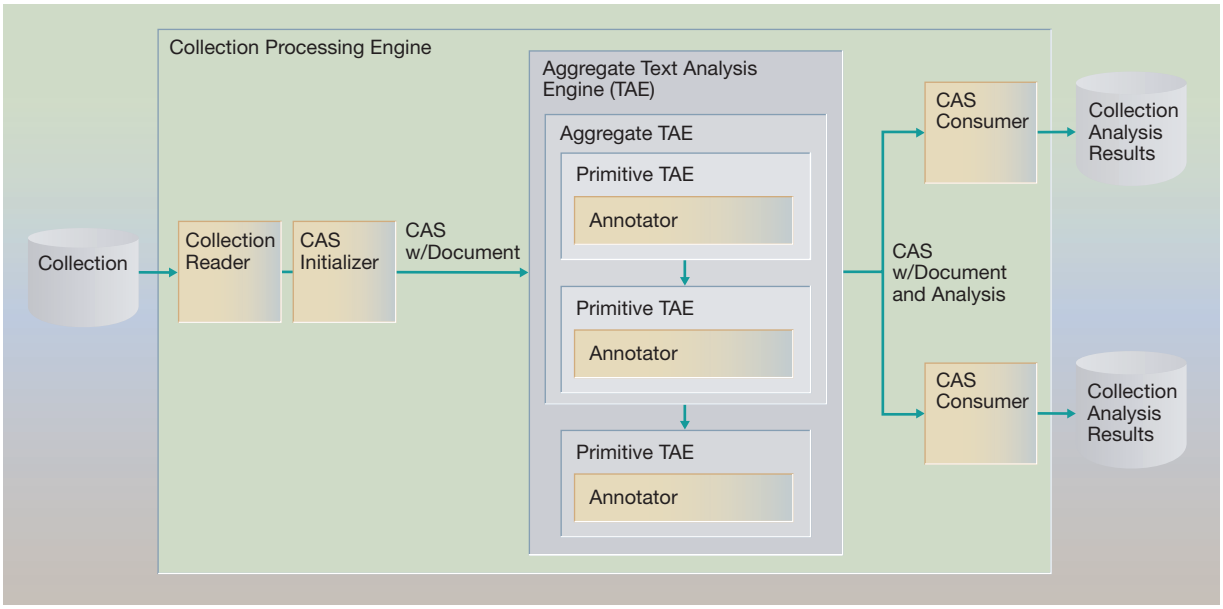
Figure 5 A UIMA component descriptor



An aggregate analysis capability may be specified from component TAEs by providing the framework with a special descriptor called an aggregate TAE descriptor. Such a descriptor identifies a set of other TAE descriptors (either primitive or aggregate) and specifies the order in which those components should execute. The UIMA framework takes this descriptor, instantiates each component TAE, and hooks them together in the proper order. The result is an aggregate TAE. At runtime the framework ensures that each component TAE obtains access to the CAS in the specified order.

CPE components. CPEs consist of a configuration of TAEs, *collection readers, CAS initializers,* and CAS con-

Figure 6 Assembling a CPE from primitive components



sumers. The UIMA framework orchestrates instances of these components to implement and manage the collocated or distributed execution of a CPE. As previously mentioned, a CAS consumer takes a CAS as input; it may produce an arbitrary application-specific data structure, such as an index or database.

Collection readers have a simple conceptual interface: *Get next document.* A collection-reader implementation is responsible for interfacing with a collection of documents, determining the next document in the collection, getting the next document from the collection, and initializing a CAS with its contents and any original document meta-data deemed appropriate for downstream processing.

To support a wide variety of input document formats, collection readers may use pluggable components called CAS initializers to populate a CAS with a form of the original document appropriate for input to the CPE TAEs. The CAS initializer may optionally extract meta-data from the original document and store it in the CAS. For example, a significant number of UIMA TAEs are written to analyze plain text documents, rather than documents in markup languages such as XML. To accommodate these analysis engines, a reusable CAS initializer has been developed that takes an XML document as input, removes the tags, inserts the detagged document as the subject of analysis in the CAS, and maps a specified set of the inline XML tags to stand-off annotations in the CAS. The initialized CAS is returned. This generally reusable CAS initializer may be configured with an arbitrary mapping of inline XML tags to stand-off CAS annotations.

UIMA development frameworks and developer roles

Like GATE,[5] UIMA is not only an architecture but also an implementation, known as a framework. More precisely, there are two major frameworks, one for C++ developers and one for Java developers. A framework handles all the details for efficient interoperability between components developed in the two languages; this interoperability has been a critical factor in enabling the use of IBM Research Division's existing C++ text analytics components by Java application developers.

Prior to UIMA, text analytics researchers at IBM were often expected not only to develop innovative new analysis algorithms but also to determine how best to combine their work with that of other researchers and finally to deliver a coherent, well-engineered solution to an IBM product or service organization.

This was a formidable requirement, given that each of these tasks requires a different set of specialized skills. We have addressed this issue by identifying five distinct *development roles* in UIMA, each with a separate set of responsibilities.[2] Each role deals with a different subset of framework interfaces and facilities, so no one person needs to understand the details of the entire process. These roles are as follows: annotator developer, analysis-engine assembler, collection-processing-component developer, CPE assembler, and component deployer. The UIMA development roles and the framework facilities designed to support them are described in the following sections.

Annotator developer. An annotator developer produces core analysis algorithms, delivering them as annotators. These algorithms can be statistical or rule-based, and can perform any type of analysis, such as categorization, named-entity detection, or relation extraction. The framework insulates annotator developers from technical interoperability and deployment issues.

Annotator developers must define the data model for representing the annotations they create, code the analysis algorithm, and declare meta-data about their component in its descriptor. The UIMA development framework supports annotator developers by providing:

- A simple Java interface (the annotator interface), which the annotator developer's code must implement.
- Interfaces for reading from and writing to CAS, such as the native Java JCas interface, which generates simple Java classes for the annotation data model.
- Interfaces for accessing framework facilities such as logging and resource management.

UIMA also provides the following tools that assist annotator developers:

- A *descriptor builder* tool allowing the creation of UIMA component descriptors without directly editing XML documents. The tool performs consistency checking and allows the developer to focus on the concepts present in the descriptor rather than on syntax.
- A *document analyzer* tool that is useful for testing annotators and TAEs. A user of the document analyzer selects the TAE component descriptor from a component browser. The user also selects a set of input documents. The document analyzer then passes the descriptor to the UIMA analysis-engine factory, which takes care of instantiating the TAE. After each input document has been processed by the TAE, the document analyzer converts the resulting CAS to an HTML view to illustrate the annotations produced by example TAEs using a web browser.

Analysis engine assembler. The analysis-engine assembler considers an analysis task and composes a solution from existing annotators and analysis engines. Such a composition is referred to as an aggregate analysis engine. The analysis-engine assembler may identify gaps where new annotators need to be developed and communicate this to an annotator developer.

In UIMA, analysis-engine assemblers do not need to write any code. They simply use the descriptor builder tool to create a simple descriptor identifying the sequence of component annotators, and the framework provides the rest:

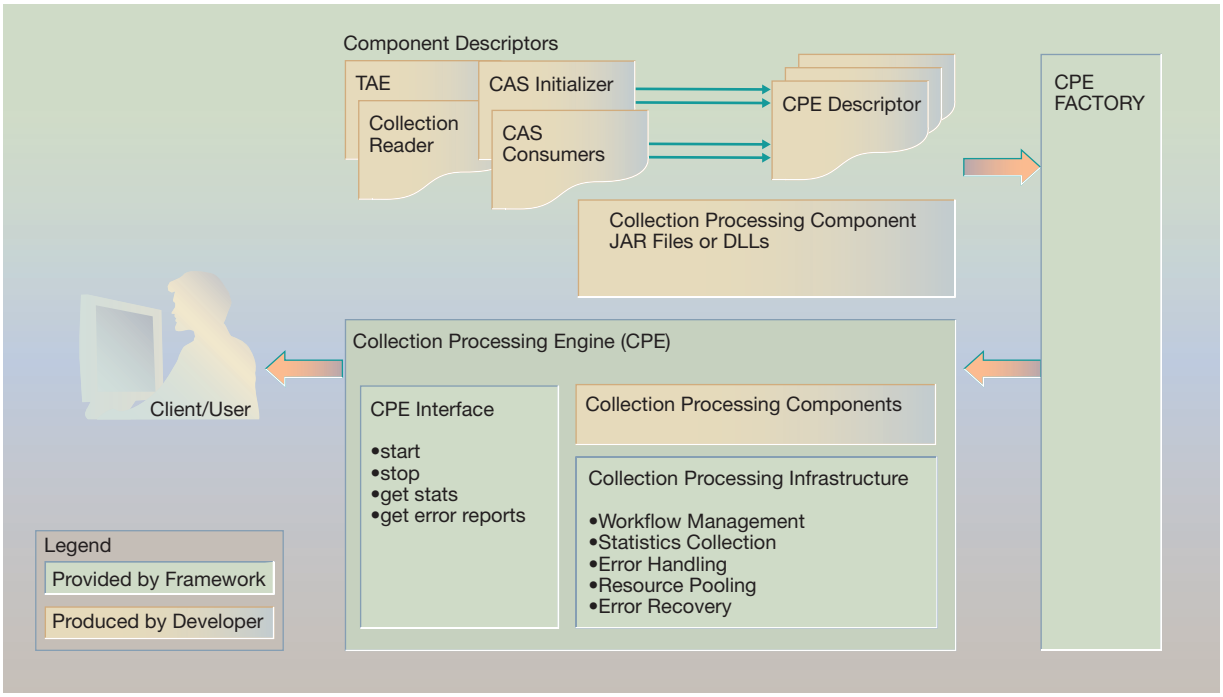
- A TAE factory, which, given a TAE descriptor, constructs TAEs from component annotators, automatically adapting for C++, Java, or network-based TAE implementations.
- A set of infrastructure components that implement requirements common to many TAEs. The UIMA framework includes infrastructure components implementing shared resource management and control of processing flow.[2] These components are used by the TAE factory to construct TAEs from component annotators.

Collection-processing-component developer. A collection-processing-component developer produces collection readers, CAS initializers, and CAS consumers for use in collection analysis. This role is similar to the annotator developer in that it involves creating an algorithm and packaging it as a class that implements a simple framework interface (the collection reader, CAS initializer, or CAS consumer interface in this case, rather than the annotator interface). However, this role is more often focused on data processing than on text analytics research.

Like annotator developers, collection-processing-component developers make use of the JCas interface of the framework and the descriptor builder tool.

CPE assembler. A CPE assembler declaratively specifies a configuration of analysis engines and collec-

Figure 7 Using the UIMA framework to generate a CPE



tion-processing components to accomplish a particular collection-processing task. As with analysis-engine assembly, this is done by using the descriptor builder tool to identify the components and their workflow; no coding is required. The framework provides:

- A CPE factory, which, given a CPE descriptor, constructs CPEs from collection-processing components.
- Infrastructure components for workflow management, performance monitoring, and error recovery, which are used by the CPE factory to construct CPEs from components (see Figure 7).

UIMA also provides a CPE interface or GUI (graphical user interface), which the client uses to configure and run CPEs. A user selects the component descriptors for a collection reader, CAS initializer, analysis engine, and one or more CAS consumers. The configuration parameters for each component are then displayed, and the user can override the default values before starting the engine. The user can then monitor progress and review performance statistics and will be notified of any errors that occur.

Component deployer. A component deployer decides how TAEs or CPEs, along with their required resources, are deployed on particular hardware and system middleware. A component deployer chooses appropriate middleware and produces a *deployment descriptor* that specifies options, such as what distributed communication protocol to use and how many simultaneous requests should be supported.

The UIMA development framework supports the component deployer via its *adapter framework.* The adapter framework is an extensible set of two types of components:

1. *Service wrappers,* which permit components to be deployed as distributed services.
2. *Client adapters,* which permit component services to be used as if they were located on the same machine as the caller.

The UIMA distribution includes service wrappers and client adapters for the standard Web services Simple Object Access Protocol (SOAP) protocol as well as an IBM-developed, lightweight protocol called

Vinci.[15] The CAS is transmitted over the network in a UIMA-standard XML format. The adapter framework is extensible, in that support for new service protocols can be added by developing appropriate service-wrapper and client-adapter components and plugging them into the framework. The developer and assembler roles do not need to be aware of the ways in which their analysis engines are deployed.

An example UIM application: Meeting Finder

Consider a simple UIM application named Meeting Finder. This application should allow e-mail users to search their e-mail based on references to meetings, their times, locations, or subjects, and to extract or highlight the found meeting information. The Meeting Finder should perform the following tasks:

1. *Access documents*. Read e-mail documents from a collection of e-mail.
2. *Analyze documents*. Analyze each e-mail looking for meeting references and identifying time of meeting, location, and subject.
3. *Build collection analysis results*.
   - Build a search engine index of the e-mail based on keywords as well as the detected meeting content.
   - Create a structured database containing a list of all mentions of meetings and explicitly associating the meeting content with the subject and author of the containing e-mail.
4. *Deliver results*. Provide the user with a query interface that allows the user to search the collection of e-mail based on a combination of keywords and meeting elements. Present the found e-mail, highlighting the meeting content.

A simple query, for example, would allow the user to find all e-mails referring to a meeting with the subject related to UIMA. A more complex query may allow the user to find all UIMA-related meetings in building HAWTHORNE1 during which refreshments would be served.

For the purposes of the example UIM application the following assumptions apply:

1. The e-mail documents are stored in the Lotus Notes e-mail export format.
2. Available for reuse in the implementation of Meeting Finder are UIMA TAEs that perform tokenization, sentence detection, and date/time detection.
3. Juru XML is used as the search engine and is capable of indexing and querying tokens and annotations.
4. A CAS consumer is available that indexes CAS annotations for the Juru XML search engine.

Meeting Finder design. We implement the Meeting Finder tasks outlined above using UIMA components as follows:

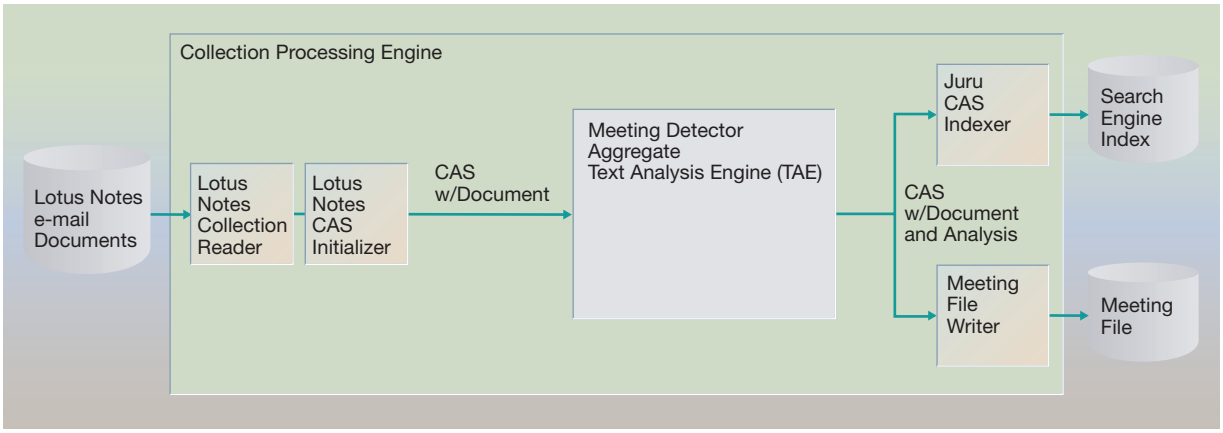
1. For the access-task-documents task, a collection reader retrieves documents from a collection of e-mails. It makes use of a CAS initializer capable of taking a Lotus Notes e-mail document in its original form and initializing a CAS with the message text as the subject of analysis and the e-mail's Subject, To, and From fields as CAS meta-data.
2. For the analyze-documents task, an aggregate TAE, capable of analyzing an e-mail document and annotating meetings, their times, locations, and subjects,[16] is assembled reusing available TAEs, which include a tokenizer, a sentence detector, and a date/time detector.
3. For the build-collection-analysis-results task, a CPE is assembled from the collection reader and the TAE described above and the following CAS consumers (the CPE data flow is illustrated in Figure 8):
   - Meeting file writer, which writes all discovered meeting mentions to a structured text file.
   - The Juru CAS indexer (an existing CAS consumer), which indexes e-mails based on tokens and annotated meeting content.
4. For the deliver-results task, the existing Juru XML query processor and the front end are employed to provide a query interface that searches the index for documents based on a combination of keywords and found meeting components.

We now discuss the implementation of the Meeting Finder application using UIMA components. This discussion is organized according to the UIMA developer roles introduced previously.

TAE development. We first examine the assembly of the analysis engine and then we discuss the annotators needed to implement the TAE.

TAE assembly. The analysis-engine assembler's task is to produce a TAE that detects meeting mentions in e-mail documents. This *meeting detector* TAE first locates room numbers, dates, and times. The TAE

Figure 8 Meeting Finder CPE data flow



then looks for co-occurrences of these entities within a certain distance of one another and annotates such co-occurrences as meetings. For example, the following sentence contains a meeting that would be detected by this algorithm:

> "UIMA 101: The new UIMA tutorial, will be held Tuesday August 26 9:00AM-4:30PM in GN-K35."

Sentence information is also useful for determining what span of text to annotate. For the preceding example the entire sentence should be annotated as the meeting; therefore, the TAE needs to first detect sentence boundaries.

The meeting detector TAE is implemented by assembling four annotators, as illustrated in Figure 9. These are:

1. Tokenizer and sentence detector annotator. The IBM Research Division has developed a library of NLP functions that includes tokenization and sentence detection capabilities. This library has been cast as a UIMA annotator, using the C++ development framework.
2. Date/time annotator. A simple date/time annotator has been previously developed using the UIMA Java development framework.
3. Room number annotator. This annotator detects occurrences of room numbers in text.
4. Meeting annotator. This annotator uses the information produced by the previous three annotators to detect and annotate meeting mentions.

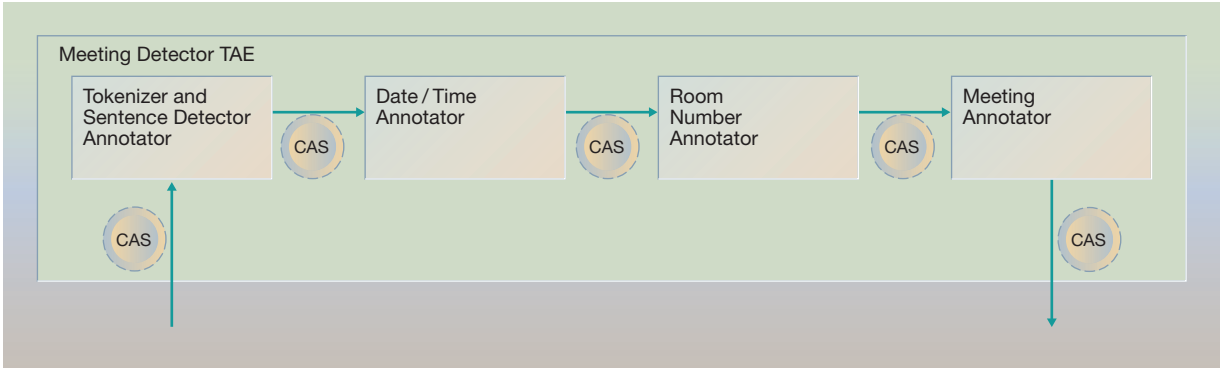
Components 1 and 2 exist as UIMA annotators and are reused without modification in the meeting detector TAE. Implementation of the room number and meeting annotators is discussed below. When these annotators are implemented, the analysis-engine assembler specifies, by means of a declarative descriptor, that these four components comprise the aggregate meeting detector TAE, and that their sequence of execution is as illustrated in Figure 9.

The current UIMA implementation requires an explicit sequence of execution. The UIMA framework architecture, however, is extensible and designed to support more powerful sequencing algorithms that can compute the execution sequence based on the inputs or outputs of each component and additional constraints supplied by the assembler.

The implementation of the meeting detector TAE, as described, contains both C++ and Java annotators. UIMA C++, Java, and network-analysis engines can easily interoperate. They share a common descriptor format, a common CAS representation, and very similar APIs. In fact, the UIMA framework can recognize the type of engine to which a descriptor refers and automatically use its built-in adapters to transparently serve as an interface between these technical variants.

Annotator development. As noted in the previous subsection, the Meeting Finder TAE requires two new annotators to be implemented:

Figure 9 Assembling the Meeting Detector TAE



1. A room number annotator, which detects room numbers in text.
2. A meeting annotator, which detects mentions of meetings based on sentences that contain dates, times, and room numbers.

The steps performed by the annotator developer are: (1) create a data model, (2) develop an analysis algorithm, and (3) declare meta-data (develop a component descriptor).

Create data model. The first step in developing an annotator is to decide on a data model for its input and output. The data model of the room-number annotator should contain RoomNumber annotations that are linked to a span of text in the document and that indicate the name of the building in which the room number is located.

In UIMA, data is stored in the CAS, and data models are declared in a Java-like CAS Type Specification (CTS) file. From a CTS file, the framework automatically generates Java classes that correspond to the types in the data model.[10]

The CTS file of the room number annotator declares a single annotation type named RoomNumber with a property named building. From this CTS file, the framework automatically generates a Java class RoomNumber with the following methods:

- getBegin, setBegin—get and set the position in the text at which the room number begins. (All annotations have begin and end positions.)
- getEnd, setEnd—get and set the position in the text at which the room number ends.
- getBuilding, setBuilding—get and set the name of the building in which the room is located.

We next discuss how these generated Java classes are used in the implementation of the room-number-annotator analysis algorithm.

Develop analysis algorithm. The annotator algorithm is written as a Java class, which is then plugged into the UIMA framework. To enable this, the annotator class must implement a standard framework interface called JTextAnnotator. The most important method on this interface is process, which invokes the annotator's analysis logic on a document.

In its process method, the room number annotator searches for room number occurrences by using, for example, regular expressions. When the annotator is to record a room number annotation in JCas, it simply creates an instance of the generated RoomNumber class, and assigns values to its begin, end, and building features by calling standard Java set methods:

```
RoomNumber room = new RoomNumber (jcas);
room.setBegin (matcher.start());
room.setEnd (matcher.end());
room.setBuilding ("Hawthorne I");
```

Declare meta-data. UIMA requires that each component publish its meta-data in a descriptor, the general structure of which was introduced in Figure 5. The room-number-annotator descriptor contains:

1. Configuration parameter declarations—the annotator should be configurable to support dif-

ferent regular expression patterns for detecting room numbers in different buildings. This is done by declaring configuration parameters, perhaps named "Room Number Patterns" and "Buildings," in the descriptor and referring to those parameters from the annotator code.

2. Capabilities—the room-number annotator declares that it requires no input (other than the document, which is assumed), produces RoomNumber annotations, and assigns values to the building property of each annotation.

Having completed the room number annotator, the annotator developer can use the UIMA document analyzer tool to run it on sample documents and visually inspect its output. The next task for the annotator developer is to develop a meeting annotator, which uses the annotations produced by the room number annotator and the existing sentence detector and date/time annotators to detect and annotate meeting mentions. The steps in the development of the meeting annotator are analogous to those for the room number annotator:

1. Create data model. Write a CTS file declaring a Meeting annotation type, from which JCas generates the corresponding Java class.
2. Develop analysis algorithm. Write a MeetingAnnotator Java class that implements the JTextAnnotator interface. The process method of this class needs to access annotations produced by the sentence detector, room number, and date/time annotators.
3. Declare meta-data. Write a descriptor of the same form as the room-number-annotator descriptor. The capabilities section of this descriptor declares that Sentence, RoomNumber, Date, and Time annotations are required as input and that Meeting annotations are produced as output.

Because the meeting annotator requires input from other annotators, it cannot be run in isolation. First, the analysis-engine assembler must combine it with the tokenizer/sentence detector, date/time annotator, and room-number annotator to produce the meeting-detector aggregate TAE. That aggregate TAE can then be run in the document analyzer tool, producing the screen shot shown in Figure 10. In this view, each annotation type is highlighted in a particular color, and the right pane shows the details of the annotations over which the mouse is currently positioned.

CPE development. The meeting detector TAE developed so far is a useful and reusable component in itself, and could be directly embedded into an application. However, the TAE is just one component of the CPE, whose design was presented in Figure 8. In this section, we discuss the implementation of the three remaining types of components: a collection reader that can read documents from a Lotus Notes* export file, a CAS initializer that populates a CAS from a Lotus Notes e-mail, and the CAS consumers that aggregate the document-level analysis results to produce a search engine index and a structured database.

Collection-processing components, just as annotators, consist of two parts: a Java class that implements a framework interface and a descriptor.

Collection reader development. The Lotus Notes export file stores multiple messages in a single file, separated by form feed characters. The Lotus Notes collection reader must extract individual e-mail messages from such a file.

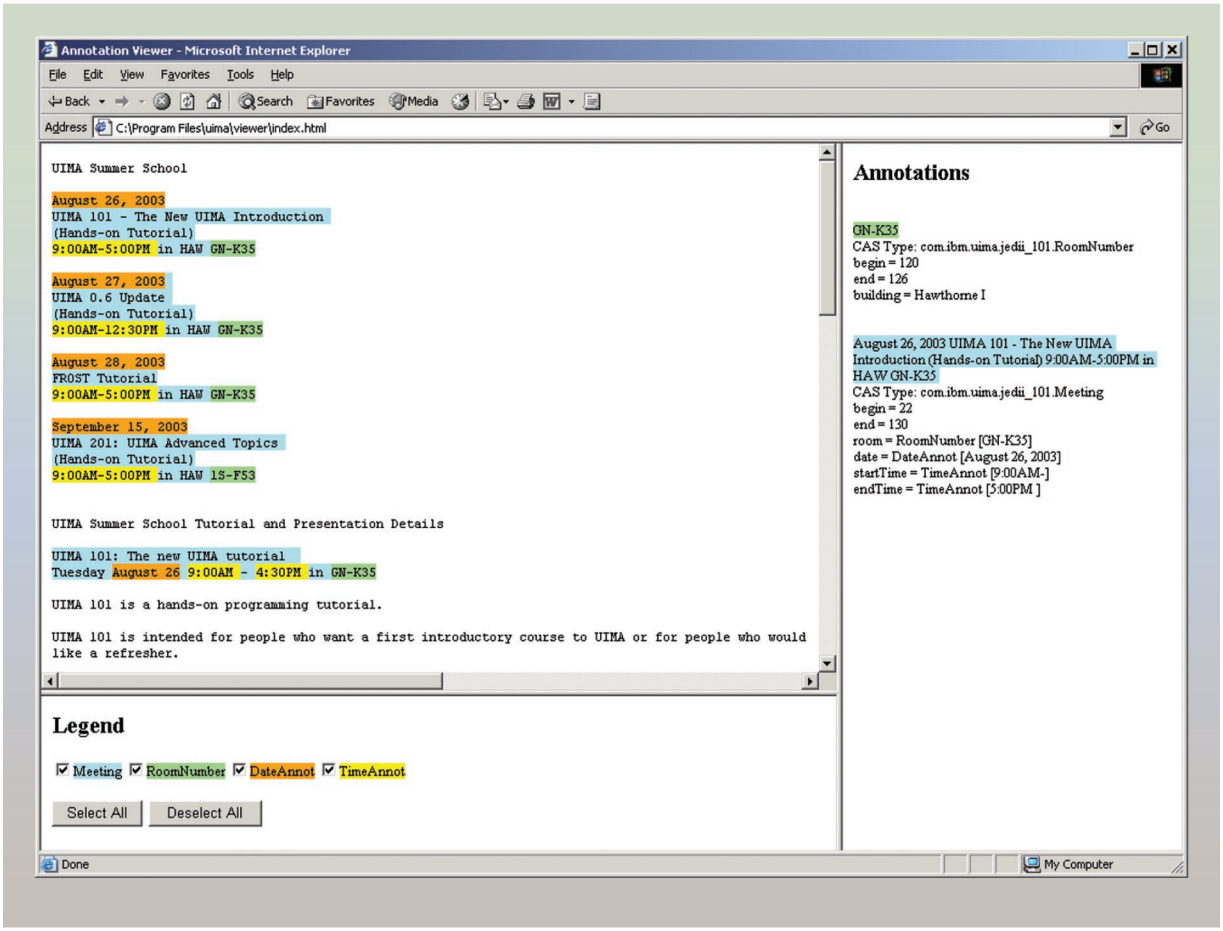
The UIMA collection reader interface requires that we implement hasNext and getNext (CAS) methods. For the Lotus Notes collection reader, the hasNext method returns TRUE if we have not reached the end of the file. The getNext (CAS) method reads from the current file position to the next form feed character and passes that text to a CAS initializer, which populates the CAS.

This collection reader also has a descriptor that is very similar to the annotator descriptors we have previously discussed. For example, an InputFile configuration parameter is defined that determines the Lotus Notes export file to be read from.

CAS initializer development. An example e-mail message in a Lotus Notes export file is shown in Figure 11. There are several header lines, followed by a blank line, and then followed by the text of the message.

The Lotus Notes CAS initializer parses the headers and stores relevant information, such as the Subject and Author of the message, as CAS meta-data. When the headers have been parsed, the rest of the document is assumed to be the text of the message, and the CAS initializer stores it in the CAS as the subject of analysis. It is this text, and not the entire raw message, which is passed on to the meeting detector TAE for analysis.

Figure 10 Screen shot of the meeting detector TAE results



The descriptor of the Lotus Notes CAS initializer includes, in addition to other descriptive information, the CAS initializer's output specification: it adds Subject and Author meta-data to the CAS.

CAS consumer development. The Meeting Finder application requires the implementation of two CAS consumers: the meeting file writer and the Juru CAS indexer.

The meeting file writer is a CAS consumer that writes all discovered meeting mentions to a structured text file. It implements the required methods of the UIMA CAS consumer interface as follows:

1. initialize—opens an output file specified by a configuration parameter.
2. processCas—takes a CAS produced by the meeting detector TAE, iterates over the Meeting annotations, and writes them to the file in some structured format.
3. collectionProcessComplete—closes the output file.

The meeting-file-writer descriptor declares that it requires Meeting annotations, Subject, and Author as input. The descriptor also declares a configuration parameter for the name of the output file.

The Juru CAS indexer is an existing UIMA CAS consumer that indexes CASs with the Juru XML search engine. The Meeting Finder application will reuse the Juru CAS indexer to build a searchable index of e-mails that is aware of Meeting annotations discov-

Figure 11 Lotus Notes e-mail message

```
Principal:   CN=Adam Lally/OU=Watson/O=IBM
InetSendTo:          ferrucci@us.ibm.com, ...
$Mailer:     Lotus Notes Release 6.0 September 26, 2002
$MessageID:   <OF17AC3DE3.40525AB8-ON85256D80.005E759E-85256D80.005EBA6D@LocalDomain>
INetFrom:    alally@us.ibm.com
Subject:    New UIMA Framework Version Released
$UpdatedBy:   CN=Adam Lally/OU=Watson/O=IBM
$Revisions:      08/12/2003 01:17:49 PM

Hello Everyone,

Version 0.6.0 of the UIMA Java Development Framework has been
posted to the downloads section of the UIMA project website at
http://uima.watson.ibm.com.

The new version is backwards compatible with version 0.5.

Please note that if you are planning to attend the UIMA Summer School
later this month you will need to download and install this version prior
to attending.

Regards,
  -Adam
```

ered by the meeting detector TAE. Like any CAS consumer, the Juru CAS indexer publishes configuration parameters in its descriptor, for example, the file system directory where the index is built and the types of annotations that should be indexed. These parameters are used to customize the Juru CAS indexer for the Meeting Finder application.

CPE assembly. The CPE assembler produces a descriptor that identifies and configures the UIMA components implemented to comprise the Meeting Finder CPE. The implementation of these components was described earlier and includes: the Lotus Notes collection reader, the Lotus Notes CAS initializer, the meeting detector TAE, which is an assembly of several annotators, and two CAS consumers: the meeting file writer and the Juru CAS indexer.

For each component, the CPE descriptor also declares deployment options: whether that component will be run in the same process as the application using the CPE, as a child process on the same machine, or on a remote machine.

Component deployment. Although it is not a necessary step towards building the Meeting Finder application, for completeness we will describe how the meeting detector TAE may be deployed as a Web service that can be called from other machines over the network by using the UIMA adapter framework. The meeting detector TAE is deployed as a Web service by using the following steps, which are performed by the component deployer:

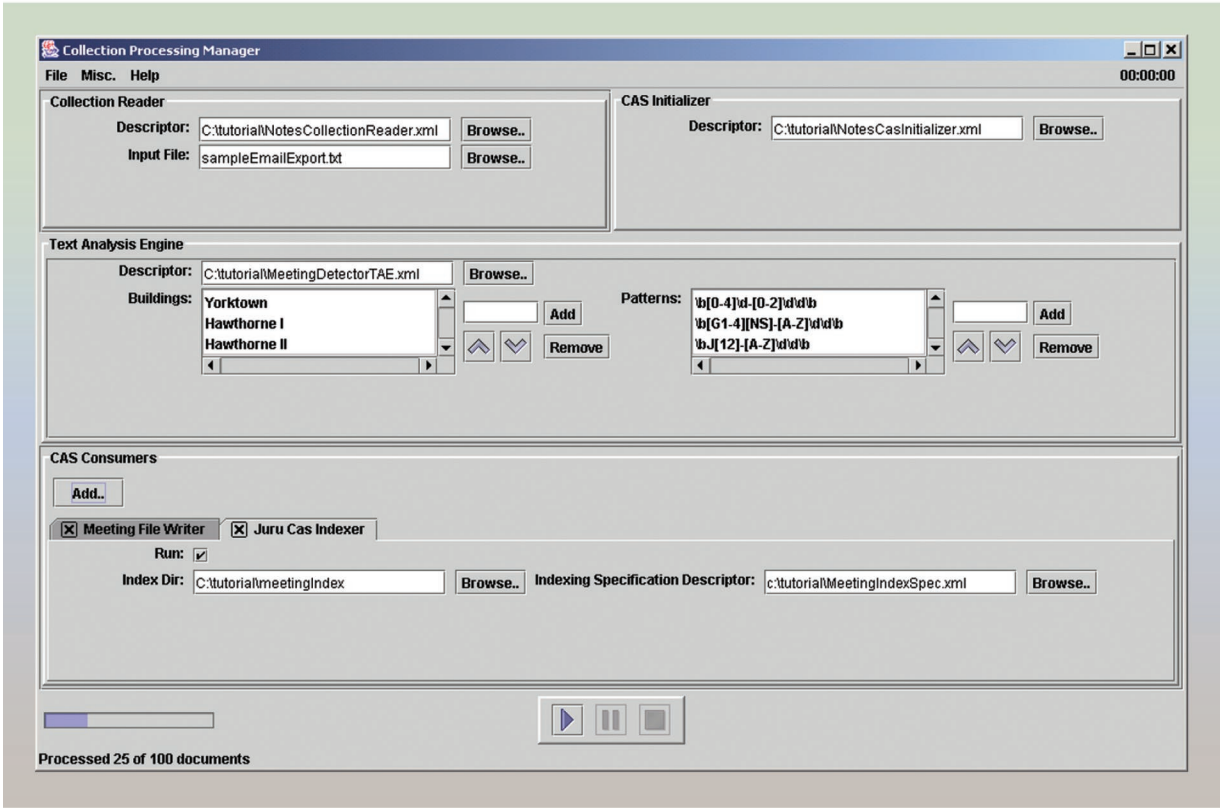
1. Specify deployment options. Develop a deployment descriptor specifying deployment options. For example:

   TAE = meeting detector TAE
   Protocol = SOAP
   Service Name = uima.tutorial.MeetingDetectorTAE
   Number of simultaneous requests supported = 3
   Timeout period = 10 seconds

2. Deploy. Use the deployment descriptor to deploy in a Web services container, such as IBM WebSphere Application Server.

In step 2, the SOAP service wrapper is instantiated. This wrapper implements the SOAP service interface by making calls to the TAE API.

Figure 12 GUI for configuring and running CPEs



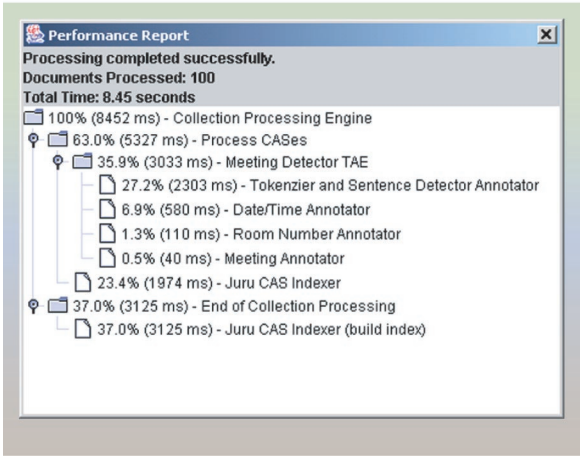
The deployed meeting detector TAE SOAP service can now be used as part of a CPE that is running anywhere on the network. To do this, the CPE assembler uses, in lieu of a full TAE descriptor, a simple *service client descriptor* that indicates the service name (uima.tutorial.MeetingDetectorTAE), host, and protocol (SOAP). The UIMA TAE factory recognizes this descriptor and automatically instantiates the appropriate client adapter to handle the details of communicating with the remote TAE.

Running the application. After all the necessary components have been developed or acquired and a CPE descriptor has been written, only two lines of code are required to invoke the UIMA development framework in order to create and run the Meeting Finder CPE. The produceCPE method of the framework invokes the CPE factory, which constructs the CPE from the components specified in the descriptor. The configuration settings for each component are the defaults specified in its descriptor. The full CPE interface of the framework provides many methods for controlling the execution and for retrieving progress and performance reports, but to simply process the entire collection, all that is needed is to invoke the start method.

UIMA includes a GUI for configuring and running CPEs; it is shown in Figure 12. This tool simply translates user actions to calls to the CPE interface. The CPE GUI allows the user to override default configuration settings. The buttons at the bottom control the processing, and a progress bar is shown at bottom left. When processing is complete, performance statistics are displayed as shown in Figure 13.

The result of the CPE execution is a Juru XML index. This index contains both the tokens from the original document and the Meeting annotations. Queries can now be issued against this index using Juru XML span-based query language.[14] The syntax of this

Figure 13 CPE performance report



language is based on XML fragments. Some sample queries follow:

- *UIMA*—returns any message containing the term "UIMA."
- <*Meeting*/>—returns any message with a Meeting annotation in it.
- <*Meeting*>*UIMA*</*Meeting*>—returns any message with a Meeting annotation where the annotation's span includes the term "UIMA."
- <*Meeting*>*UIMA*</*Meeting*> *refreshments*—returns messages matched by the previous query which also contain the term "refreshments" anywhere in the message.

The Meeting Finder application can return hyperlinks to the returned e-mail documents that bring up the HTML annotation view similar to that shown in Figure 10, highlighting the meeting information.

Results and discussion

We described in this paper a purely pedagogical application of UIMA in an attempt to highlight the UIMA concepts and methodologies from the point of view of component and application developers. The architecture has been applied for developing more practical applications, however, as a result of its widespread adoption by IBM research teams worldwide. As we noted in Reference 2, evaluation of the architecture independent of the host organization is nontrivial, and we identified three criteria to use in such an evaluation:

1. Compliant components and their reuse. At the time of this writing, we have a registry of over 25 UIMA-compliant TAEs produced by ten different groups in five IBM research laboratories in four countries. These TAEs include named-entity detectors for English, Arabic, and Chinese, deep and shallow parsers for English and Japanese, classifiers, a summarizer, an Arabic-to-English translator, and relation extractors. Several of these components evolved from legacy systems components that were not previously compatible; they can now be easily assembled and deployed in an application. For example, a collection of linguistic-processing components including a shallow parser, all developed in C++, have been combined with Java-based biological entity detectors and relation extractors, as well as collection readers and CAS consumers, to form a powerful CPE for the life science domain.[17]
2. System performance. A thorough analysis of the effect of UIMA on overall system throughput has yet to be undertaken. Preliminary experiments, however, suggest that the overhead imposed by the architecture is minimal. Throughput is also high enough not to represent an obstacle to product deployment. As noted in Reference 2, we have already reaped some throughput benefits from the ability of the framework to easily deploy multiple instances of components across a set of machines, a result much more difficult to accomplish with legacy systems.
3. Product integration. UIMA has become the established platform for delivering and integrating text analytics into IBM products and services, and it is success in this arena that will validate the merits of the architecture. UIMA has already been used to integrate summarization and categorization components into WebSphere Portal Server, and UIMA components are being used in several service offerings. More product deployments are currently underway.

UIMA has many similarities to other software architectures for language engineering such as GATE[4,5] and ATLAS.[6] All these systems isolate the core algorithms that perform language processing from support services such as data storage, communication between components, and visualization of results.

UIMA, however, supports a richer set of architected interfaces for document and collection processing in order to meet the requirements of a substantial pre-existing asset base in text analytics and UIM ap-

plications. Further emphasis was placed on scalability and middleware and platform independence to enable extensibility, product integration, and a wide variety of deployment alternatives. These goals were given priority over development tools and a prepackaged set of integrated analytics. This is largely because with over 200 researchers at IBM working in this area, there was an extensive collection of existing capabilities that needed to be integrated and deployed in order to bring value to IBM product lines in data, content, and knowledge management. As just one example, a large C++ text analytics code base developed within the IBM Research Division for high-performance analysis, coupled with the Java focus of the IBM WebSphere products, led to the development of efficient Java/C++ interoperability features within UIMA.

Collection-level analysis may involve complex workflows, large volumes of documents, different deployment architectures (e.g., Web services, message-based architectures), different document formats, and a wide variety of collection-level results. While GATE provides out-of-the-box support for different document formats, UIMA defines a rich collection-processing architecture that is extensible along all these dimensions. For example, UIMA's highly scalable and recoverable collection-processing architecture supports the processing of very large volumes of documents for IBM's WebFountain*.[18] WebFountain is a cluster of thousands of nodes dedicated to mining the Web and other very large corpora. For WebFountain, scalability and robustness are essential qualities. WebFountain architects and UIMA architects collaborated in extending the UIMA framework to process very large collections in a service-oriented, managed-cluster environment. The result is that WebFountain can deploy any combination of UIMA TAEs and CPEs on its cluster.

One area in which GATE has focused more effort to date than UIMA is in its development environment. GATE provides an integrated set of graphical tools that assist users in building, modifying, and debugging language engineering systems. UIMA currently has only a base set of standalone tools including a descriptor builder, annotation viewer, and collection-processing-manager GUI. We are beginning efforts to develop and integrate these and other tools into a coherent development environment based on the Eclipse open source, extensible IDE.[19]

UIMA analysis components have to be able both to stand alone and to be embedded in products and service solutions including Lotus Workplace,[20] WebSphere Portal Server, DB2* Content Manager,[21] WebFountain, and Enterprise Search.[22] Each of these carrier environments may rely on different system middleware. The UIMA middleware adapter framework extends to accommodate and integrate with these environments while insulating the UIMA component developer. For example, UIMA includes a robust middleware adapter for the standard Web services SOAP protocol. This adapter can be used to easily deploy any type of UIMA component as a Web service, without writing any additional code and without requiring detailed knowledge of Web services. The original developer of the component need not be aware that the component might later be deployed as a Web service.

UIMA also includes middleware adapters for deployment in IBM's WebFountain environment, and, before the end of this year, we expect to have similar adapters for deployment in the other major products indicated above.

Finally, the UIMA JCas interface is a novel extension to the annotation model in ATLAS and GATE. By automatically generating Java code from a developer's data model definition, we allow Java developers to easily create annotations by simply instantiating Java classes, while the framework maintains tight control over the data representation. Maintaining control over the data representation supports efficient C++/Java interoperability and preserves serialization options for transport to and from remote services. Also, by allowing methods to be declared on these annotation Java classes, the UIMA JCas interface enables data model designers to leverage the power of the object-oriented-programming paradigm to abstract away many implementation details of a complex data model.[23]

Conclusion

UIM applications must distill relevant content from implicit and often ambiguous unstructured information sources and deliver it in explicit, structured and highly accessible forms to their end users. With huge volumes of unstructured information including text documents, video, and speech being generated at ever increasing rates, the requirements to analyze, discover, and deliver valuable content are overwhelming industry and government. UIM applications are being developed to assist in e-commerce, life science, business, and National Security intelligence.

IBM's UIMA project is focused on developing a software architecture and associated development frameworks for supporting the development, integration, and deployment of unstructured information analysis capabilities. In this paper we provide a high-level conceptual overview of UIMA and illustrate its use to design and implement a simple UIM application. UIMA has been widely adopted throughout the IBM Research Division and the IBM Life Sciences Division. UIMA has become the established platform for delivering and integrating text analytics into IBM products. Its use in the IBM Research Division has encouraged and facilitated advances in text analytics based on technologies that would have otherwise remained untried due to barriers such as lack of skills, lack of interoperability, and lack of a common architecture.

In plan for the UIMA project at IBM are investigations into advanced control structures, pluggable workflow, stronger support for the disciplined integration and maintenance of ontology resources, support for multiple modalities, and the development of Eclipse-based tooling for the UIM application developer.

Acknowledgments

We acknowledge the efforts by Alfred Spector and Arthur Ciccolo to make a unifying architecture the focus of the NLP and UIM activities at the IBM Research Division. We acknowledge the contributions of Dan Gruhl and Andrew Tomkins of IBM's WebFountain project to the development of UIMA. In addition, we acknowledge David Johnson, Thomas Hampp, Thilo Götz, and Oliver Suhre for their role in the development of IBM's Text Analysis Framework, and Roy Byrd and Mary Neff for their role in the design of the TALENT system. Their work continues to influence the UIMA designs and implementations. Starting this project and implementing UIMA would not have been possible without the rapid prototyping skills and committed software development efforts of Jerry Cwiklik. Finally, we acknowledge Marshall Schor for his invaluable contributions to the design and implementation of the UIMA analysis engine framework and JCas.

This work was supported in part by the U.S. government's Advanced Research and Development Activity (ARDA) Advanced Question Answering for Intelligence (AQUAINT) Program under contract number MDA904-01-C-0988.

Cited references and note

1. C. Moore, "Diving into Data," *InfoWorld* (October 25, 2002), http://www.infoworld.com/article/02/10/25/021028feundata_1.html.
2. D. Ferrucci and A. Lally, "UIMA: An Architectural Approach to Unstructured Information Processing in the Corporate Research Environment," *Natural Language Engineering* (2004, to appear).
3. W. Roush, "Computers that Speak Your Language," *Technology Review* **106,** No. 5, 32 (2003).
4. H. Cunningham, K. Bontcheva, V. Tablan, and Y. Wilks, "Software Infrastructure for Language Resources: A Taxonomy of Previous Work and a Requirements Analysis," *Proceedings of the Second Conference on Language Resources Evaluation*, Athens, European Language Resources Association, Paris (2000).
5. K. Bontcheva, H. Cunningham, V. Tablan, D. Maynard, and H. Saggion, "Developing Reusable and Robust Language Processing Components for Information Systems Using GATE," *Proceedings of the 3rd International Workshop on Natural Language and Information Systems (NLIS'2002)*, IEEE Computer Society Press, New York (2002).
6. C. Laprun, J. Fiscus, J. Garofolo, and S. Pajot, "A Practical Introduction to ATLAS," *Proceedings of the Third International Conference on Language Resources and Evaluation*, European Language Resources Association, Paris (2002).
7. T. Götz and O. Suhre, "Design and Implementation of the UIMA Common Analysis System," *IBM Systems Journal* **43,** No. 3, 476–489 (2004, this issue).
8. R. Grishman, *Tipster Architecture Design Document Version 2.2*, Technical Report, Defense Advanced Research Projects Agency (DARPA), U.S. Department of Defense (1996).
9. S. Mardis and J. Burger, "Qanda and the Catalyst Architecture," *AAAI Spring Symposium on Mining Answers from Text and Knowledge Bases*, American Association for Artificial Intelligence (2002).
10. M. Schor, *An Effective, Java-Friendly Interface to the CAS*, Research Report RC23176, IBM T.J. Watson Research Center, Yorktown Heights, N.Y (2004).
11. D. Lewis, "Text Representation For Text Classification," P. Jacobs, Editor, *Text-Based Intelligent System*, Lawrence Erlbaum Associates, Mahwah, N.J. (1992).
12. C. Apte, F. Damerau, and S. Weiss, "Automated Learning of Decision Rules for Text Categorization," *ACM Transactions on Information Systems* **12,** No. 3, 233–251 (1994).
13. Y. Yang, "An Evaluation of Statistical Approaches to Text Categorization," *Information Retrieval Journal* **1,** No. 1–2, 69–90 (1999).
14. 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, ACM, New York (2003).
15. R. Agrawal, R. Bayardo, D. Gruhl, and S. Papadimitriou, "Vinci: A Service-Oriented Architecture for Rapid Development of Web Applications," *Computer Networks* **39,** No. 5, 523–539 (2002).
16. To accurately detect informal meeting references expressed in natural language indicating times, locations, and subjects

may be nontrivial. Our focus in this paper is to describe how to structure a UIM application based on UIMA, rather than to present the most accurate algorithms.
17. R. Mack, S. Mukherjea, A. Sofer, N. Uramoto, E. Brown, A. Coden, J. Cooper, A. Inokuchi, B. Iyer, Y. Mass, H. Matsuzaws, and L. V. Subramaniam, "Text Analytics for Life Science Using the Unstructured Information Management Architecture," *IBM Systems Journal* **43,** No. 3, 490–515 (2004, this issue).
18. S. Cass, "A Fountain of Knowledge," *IEEE Spectrum* **41,** No. 1, 60–67 (2004), http://www.spectrum.ieee.org/WEBONLY/publicfeature/jan04/0104comp1.html.
19. eclipse.org, http://eclipse.org.
20. Lotus Workplace, IBM Corporation, http://www.lotus.com/engine/jumpages.nsf/wdocs/ondemand.
21. DB2 Content Manager Family, IBM Corporation, (http://www.ibm.com/software/data/cm/cmgr).
22. See for example the presentations C. Wolpert, "Lotus Workplace: Search," Lotus Software, IBM Corporation, (http://media.lotus.com/lotusphere2004/id/id505.pdf, and A. Soffer, "IBM Search Technologies—The Haifa Perspective," IBM Haifa Labs, IBM Corporation, http://www.haifa.il.ibm.com / Workshops / searchandcollaboration2004 / papers/SearchTechnologies.pdf
23. R. Basili, M. Di Nanni, and M. Pazienza, "Engineering of IE Systems: An Object-Oriented Approach," M. Pazienza, Editor, *Information Extraction*, Lecture Notes in Artificial Intelligence 1714, Springer-Verlag, Berlin (1999), pp. 134–164.

Accepted for publication March 16, 2004.

David Ferrucci *IBM Research Division, Thomas J. Watson Research Center, 19 Skyline Drive, Hawthorne, NY 10532 (ferrucci@us.ibm.com).* Dr. Ferrucci is a research staff member and senior manager at IBM's Thomas J. Watson Research Center. He manages the Semantic Analysis and Integration department which develops technologies, architectures, and solutions for mining unstructured information and applies them to information processing systems. His team includes world-class researchers and software engineers in NLP and knowledge representation. Dr. Ferrucci is a co-architect of IBM's Unstructured Information Management Architecture (UIMA), a project that spans six IBM research labs and provides the foundation for Research Division work in text analytics. Dr. Ferrucci has published in the fields of logic and knowledge representation, architectures for natural language engineering, and automated question answering. His research interests include advanced technologies for representing and applying knowledge about logic and language. He has eight patents pending in search and text analysis, interactive document configuration, and automatic story generation. Dr. Ferrucci received a Ph.D. degree from Rensselaer Polytechnic Institute, Troy, N.Y., in the area of knowledge representation and automated reasoning.

Adam Lally *IBM Research Division, Thomas J. Watson Research Center, 19 Skyline Drive, Hawthorne, NY 10532 (alally@us.ibm.com).* Mr. Lally is an advisory software engineer in the Information Management Solutions group at IBM's Thomas J. Watson Research Center. He is the lead engineer for the UIMA Java development framework, where his work focuses on applying software architecture and engineering principles to accelerate research and to facilitate integration of research into IBM products. Mr. Lally received a B.S. degree in computer science from Rensselaer Polytechnic Institute, Troy, N.Y.