# UIMA: An Architectural Approach to Unstructured Information Processing in the Corporate Research Environment

**David Ferrucci and Adam Lally**

IBM T.J. Watson Research Center
Yorktown Heights, NY 10598
[ferrucci, alally]@us.ibm.com

## Abstract

IBM Research has over 200 people working on Unstructured Information Management (UIM) technologies with a strong focus on Natural Language Processing (NLP). These researchers are engaged in activities ranging from natural language dialog, information retrieval, topic-tracking, named-entity detection, document classification and machine translation to bioinformatics and open-domain question answering. An analysis of these activities strongly suggested that improving the organization's ability to quickly discover each other's results and rapidly combine different technologies and approaches would accelerate scientific advance. Furthermore, the ability to reuse and combine results through a common architecture and a robust software framework would accelerate the transfer of research results in NLP into IBM's product platforms. Market analyses indicating a growing need to process unstructured information, specifically multilingual, natural language text, coupled with IBM Research's investment in NLP, led to the development of middleware architecture for processing unstructured information dubbed UIMA. At the heart of UIMA are powerful search capabilities and a data-driven framework for the development, composition and distributed deployment of *analysis engines*. In this paper we give a general introduction to UIMA focusing on the design points of its analysis engine architecture and we discuss how UIMA is helping to accelerate research and technology transfer.

## 1 Motivation and Objectives

We characterize a UIM application as a software system that analyzes large volumes of unstructured information while consulting structured information sources to discover, organize and present knowledge relevant to the application's end-user. We define *structured information* 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. We define *unstructured information* as information whose intended meaning is only loosely implied by its form. The canonical example is a natural language document; other examples include voice, audio, image and video.

Market indicators suggest a rapid increase in the use of text and voice analytics to a growing class of commercially viable applications (Roush 2003). National security interests are driving applications to involve an even broader set of UIM technologies including image and video analytics. Emerging and commercially important UIM application areas include life sciences, e-commerce, technical support, advanced search, and national and business intelligence.

In six major labs spread out over the globe, IBM Research has over 200 people working on Unstructured Information Management (UIM) technologies with a primary focus on Natural Language Processing (NLP). These researchers are engaged in activities ranging from natural language dialog, information retrieval, topic-tracking, named-entity detection, document classification and machine translation to bioinformatics and open-domain question answering. Each group is developing different technical and engineering approaches to process unstructured information in pursuit of their specific research agenda in the UIM space.

While IBM Research's independent UIM technologies fare well in scientific venues, accelerated development of integrated, robust applications is becoming increasingly important to IBM's commercial interests. Furthermore the rapid integration of different alogorthims and techniques is also a means to advance scientific results. For example, experiments in statistical name-entity detection or machine translation may benefit from using a deep parser as a feature generator. Facilitating the rapid integration of an independently produced deep parser with statistical algorithm implementations can accelerate the research cycle time as well as the time to product.

Some of the challenges that face a large research organization in efficiently leveraging and integrating its (and third-party) UIM assets include:

- **Organizational Structure.** IBM Research is largely organized in three to five person teams geographically dispersed among different labs throughout world. This structure makes it more difficult to jointly develop and reuse technologies.
- **Skills Alignment**. There are inefficiencies in having PhDs trained and specialized in, for example, computational linguistics or statistical machine learning, develop and apply the requisite systems expertise to deliver their work into product architectures and/or make it easily usable by other researchers.

- **Development Inefficiencies**. Research results are delivered in a myriad of interfaces and technologies. Reuse and integration often requires too much effort dedicated to technical adaptation. Too often work is replicated.
- **Speed to product**. Software Products introduce their own architectures, interfaces and implementation technologies. Technology transfer from the lab to the products often becomes a significant engineering or rewrite effort for researchers.

A common framework and engineering discipline could help address the challenges listed above. The Unstructured Information Management Architecture (UIMA) project was initiated within IBM Research based on the premise that a common software architecture for developing, composing and delivering UIM technologies, if adopted by the organization, would facilitate technology reuse and integration, enable quicker scientific experimentation and speed the path to product. IBM's Unstructured Information Management Architecture (UIMA) high-level objectives are two fold:

1) Accelerate scientific advances by enabling the rapid combination of UIM technologies (e.g., natural language processing, audio and video analysis, information retrieval, etc.).
2) Accelerate transfer of UIM technologies to product by providing an architecture and associated framework implementations that promote reuse and support flexible deployment options.

## 2 The Unstructured Information Management Architecture (UIMA)

UIMA is a software architecture for developing UIM applications. The UIMA high-level architecture, illustrated in Figure 1, defines the roles, interfaces and communications of large-grained components essential for UIM applications. These include components capable of analyzing unstructured artifacts, integrating and accessing structured sources and storing, indexing and searching for artifacts based on discovered semantic content resulting from analyses. A primary design point for UIMA is that it should admit the implementation of middleware frameworks that provide a component-based infrastructure for developing UIM applications suitable for vastly different deployment environments. These should range from lightweight and embeddable implementations to highly scalable implementations that are meant to exploit clusters of machines and provide high throughput, high availability serviced-based offerings. While the primary focus of current UIMA framework implementations is squarely on natural language

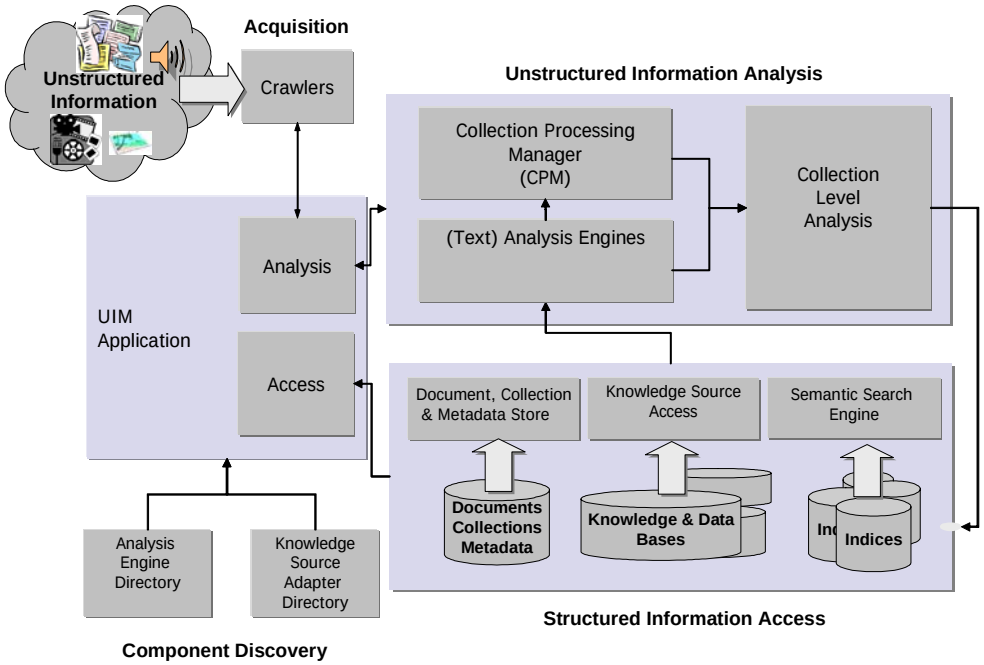


Figure 1: UIMA High-Level Architecture

text, the architecture has been designed to extend to different unstructured artifacts including voice, audio and video. In general, we will refer to elements of unstructured information processing as *documents* admitting, however, that an element may represent for an application a whole text document, a document fragment or even multiple documents.

In the remainder of this paper we provide a high-level overview of UIMA leading to a more detailed discussion of its analysis engine framework. We discuss some of the issues related to evaluating the architecture independently of the UIMA project and close with a status update and comments regarding future directions.

## 3 High-Level Architecture Overview

In this section we provide a high-level overview of UIMA by describing its component roles in a generalized application scenario. The generalized scenario refers to four primary component services provided by UIMA implementations. They are as follows:

1) Acquisition

2) Unstructured Information Analysis

3) Structured Information Access

4) Component Discovery

In this section we refer to the software program that employs UIMA components to implement some end-user capability as the *application* or *application program*.

### 3.1 Acquisition

Acquisition services gather documents from one or more sources to produce particular collections for the application. An acquisition service may be implemented, for example, with a web crawler. These services may range from simple to highly parameterized functions. UIMA does not specify the interface to acquisition service components but requires that ultimately acquired collection data is provided to the application through a *Collection Reader* interface. Collection Readers provide access to a collection's elements (e.g. documents), collection metadata and element metadata. UIMA implementations include a storage facility that contains documents, collections and their metadata and that implements the Collection Reader interface. However, applications that need to manage their own collections can provide their own implementation of a Collection Reader to UIMA components that require access to collection data.

### 3.2 Unstructured Information Analysis: Document-Level

Unstructured information analysis is further divided into two classes, namely document-level and collection-level analysis. In this section we discuss document-level analysis.

Document-level analysis is performed by component processing elements named *Text Analysis Engines* (TAEs). These are extensions of the generic analysis engine, specialized for text. They are analogous, for example, to Processing Resources in the GATE architecture (Cunningham, Bontcheva, Tablan, and Wilks 2000). In UIMA, a TAE is a recursive structure which may be composed of sub or component engines each performing a different stage of analysis.

Examples of Text Analysis Engines include language translators, grammatical parsers, named-entity detectors, document summarizers, and document classifiers. Each TAE specializes in discovering specific concepts (or *semantic entities*) otherwise unidentified or implicit in the document text.

A TAE takes in a document and produces an analysis. The original document and its analysis are represented in a structure called the *Common Analysis Structure*, or CAS. The CAS is conceptually analogous

to the *annotations* in other architectures, beginning with TIPSTER (Grishman, 1996). In general, annotations associate some metadata with a region in the original artifact. Where the artifact is a text document, for example, the annotation associates metadata (e.g., a label) with a span of text in the document by giving the span's start and end positions. Annotations in the CAS are stand-off, meaning that the annotations are maintained separately from the document itself; this is more flexible than inline markup (Mardis and Burger, 2002).

The analysis represented by the CAS may be thought of as a collection of metadata that is enriched as it passes through successive stages of analysis. At a specific stage of analysis, for example, the CAS may include a deep parse. A named-entity detector receiving this common analysis structure may consider the deep parse to identify named entities. The named entities may be input to an analysis engine that produces summaries or classifications of the document.

Our CAS annotation model is similar to that of the ATLAS architecture (Laprun, Fiscus, Garofolo, and Pajot 2002) in several ways:

1. Metadata associated with an annotation may be a primitive type such as a string or arbitrarily complex metadata element composed of primitive types. The CAS provides a general object-based representation with a hierarchical type system supporting single inheritance and references between objects.
2. Because the definition of a region within an entity is dependent on the type of entity, this definition is not part of the CAS specification. Instead, specializations of the CAS (e.g. Text CAS) fix the entity type and provide an appropriate definition of a region.
3. We have developed an XML representation for CAS data models (i.e. schemas), a similar concept to the *Meta-Annotation Infrastructure for ATLAS* (MAIA). In addition to enabling structural integrity and consistency checking, this also allows UIMA TAEs to publish their input requirements and output specifications, which facilitates reuse.

Once a TAE has produced a CAS it may be returned to the application that invoked it or it may be forwarded by the UIMA framework, for example, to a CAS consumer for collection level processing.

## 3.3 Unstructured Information Analysis: Collection-Level

### 3.3.1 Collections

In UIMA, document collections (or simply collections) are entities that contain individual documents. Collections are identifiable objects that may be associated with their own metadata in the form of key-value pairs. Collection content and metadata are accessible through the UIMA Collection Reader interface.

Collection level analysis considers an aggregation of document level analyses to produce collection level analysis results. These results represent aggregate inferences computed over all or some subset of the documents in a collection. Examples of collection level analysis results include sub-collections where all elements contain certain features, glossaries of terms with their variants and frequencies, taxonomies, feature vectors for statistical categorizers, databases of extracted relations, and master indices of tokens and other detected entities.

TAEs, which perform document-level analysis, consistently produce CASs and do not maintain any state between documents. This is a requirement which ensures that TAEs can be restarted in the event of failure and more effectively managed in a distributed, multi-processing deployment. In contrast, programs that do collection level analysis vary widely in terms of the content and form of their results and the extent and form of their intermediate computations. They must maintain state between documents and therefore are intrinsically more complicated to manage in a distributed environment. UIMA takes a very general approach to programs that perform collection level processing – the architecture considers these simply as *CAS Consumers*. A CAS Consumer is any program that takes in a CAS as part of its input. Unlike TAEs, however, they are not assumed to update the CAS altering the result of document-level analysis. CAS Consumers are responsible for managing their own stateful environment and providing their own mechanism for restarting their computation. It is typical for CAS Consumers to further process document-level analysis results and update structured sources, dictionaries, indices, etc. that may be subsequently consulted by TAEs or more generally by UIM applications.

### 3.3.2 Collection Processing Manager

In support of collection level analysis, UIMA defines the *Collection Processing Manager* (CPM) component. The CPM's primary responsibility is to manage the application of a designated TAE to each docu-

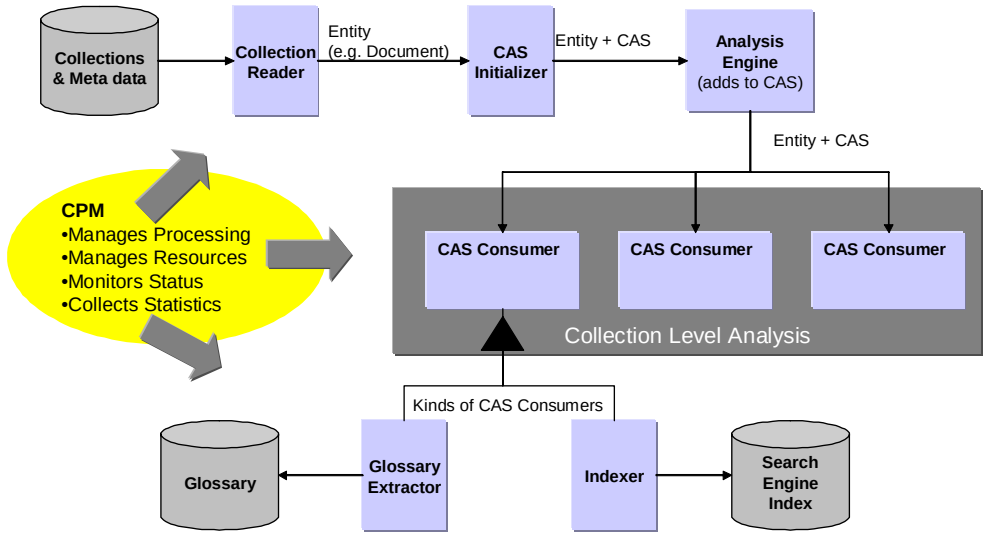


Figure 2: Collection Processing Manager

ment accessible through a Collection Reader and pass the resulting analyses (CASs) to a set of CAS Consumers (see Figure 2).

An application provides, as input to the CPM:

1) a Collection Reader
2) a CAS Initializer
3) a Text Analysis Engine
4) a set of CAS Consumers

The CPM obtains raw documents from the Collection Reader and passes them to the *CAS Initializer*, a component responsible for ingesting the raw document and producing the initial CAS on which the TAE will operate. For example, a Collection Reader acquiring documents from a Web Crawler will return HTML documents, but TAEs generally operate on plain text documents with standoff annotations. It is the job of a CAS Initializer to parse the HTML and populate a CAS with the detagged plain text document and, optionally, standoff annotations that correspond to the HTML markup, e.g. paragraph boundaries. Other reusable CAS Initializer implementations could be developed to handle particular XML dialects or other document formats such as RTF or PDF.

Once the CAS Initializer has completed, the CPM applies the TAE and passes the resulting analysis, represented in the CAS, to the designated CAS Consumers. This process is repeated for each entity in the

collection. CAS Consumers typically aggregate the document-level analyses in an application-dependent data structure, such as a search engine index, a glossary of terms, or a taxonomy.

The CPM may also provide the application with additional administrative functions such as:

1) **Filtering** - ensures that only certain elements are processed based on metadata constraints.
2) **Error Handling** – reports errors to user and may retry failed documents and/or maintain a list of failed documents to be manually reviewed at a later time.
3) **Performance Monitoring –** measures the processing time of each component and presents performance statistics to the user.
4) **Parallelization** - manages the creation and execution of multiple instances of a TAE for processing multiple documents simultaneously utilizing available computing resources.

### 3.4 Structured Information Access

Generally, a UIM application will process unstructured information to produce structured information sources such as indices, dictionaries, ontologies or relational databases, which the application is designed to use to deliver value to its end user. This structured information may also be consulted by analysis engines to assist with the analysis of unstructured sources. This relationship between unstructured information analysis and structured sources is the primary feedback loop in the UIM application – the results of unstructured information analysis provide structured information sources reused by unstructured information analyses. There are of course other sources for structured information including a plethora of existing databases and knowledge bases that may be useful to the UIM application.

Structured information access in UIMA is further divided into semantic search, structured knowledge access and document metadata access.

#### 3.4.1 Semantic Search

UIMA refers to *semantic search* as the capability to query for and return a ranked list of documents based on their containing content discovered by document or collection level analysis and represented as annotations. UIMA specifies search engine indexing and query interfaces.

A key feature of the indexing interface is that it supports the indexing of tokens as well as annotations and particularly cross-over annotations. Two or more annotations *cross-over* one another if they are linked

to intersecting regions of the document. Figure 3 illustrates a sentence with spans of text marked by annotations. The ‘inhibits’ and ‘activates’ annotations span regions of text which a TAE considers mentions of chemical inhibition and activation relations respectively. In this example, the token ‘aspirin’ is in the intersection of these two cross-over annotations.

The key feature of the query interface is that it supports queries that may be predicated on nested and overlapping structures of annotations (which are not naturally represented as inline XML markup) and tokens in addition to Boolean combinations of tokens and annotations.

### 3.4.2 Document and Collection Metadata

UIMA defines a component interface to a document, collection and metadata store. Implementations are used to store documents and collections of documents as entities. Arbitrary metadata may be defined by the application and associated with entities as sets of key-value pairs. Applications may use the store to associate analysis results with documents and/or collections. The store is assumed to be the repository of documents indexed by the semantic search engine defined above. The search engine’s query results deliver identifiers used to retrieve entities and their metadata from the store. UIMA does not specify any particular set of metadata elements.

### 3.4.3 Structured Knowledge Access

As analysis engines do their job they may consult a wide variety of structured knowledge sources. To increase reusability and facilitate integration, UIMA specifies the *Knowledge Source Adapter* (KSA) interface.

KSA objects provide a layer of uniform access to disparate knowledge sources. They manage the technical communication, representation language and ontology mapping necessary to deliver knowledge encoded in databases, dictionaries, knowledge bases and other structured sources in a uniform way. The primary interface to a KSA presents structured knowledge as instantiated predicates using the Knowledge Interchange Format (Geneserrth and Fikes 1992) encoded in XML.

A key aspect of the KSA architecture is the KSA metadata and related services supporting KSA registration and search. These services include the description and registration of named ontologies. Ontologies are described by the concepts and predicates they include. The KSA is self-descriptive and among other

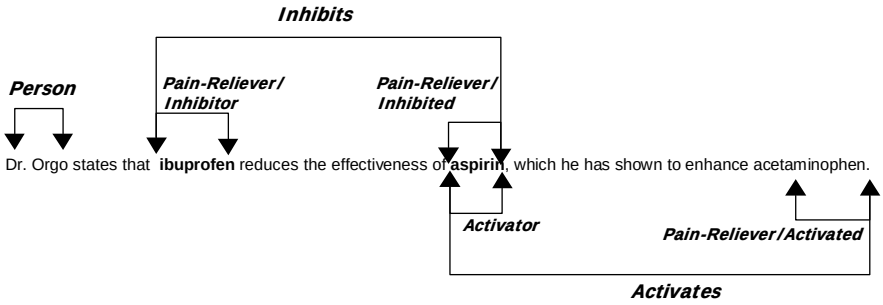


Figure 3: Cross-Over Annotations

metadata includes the predicate signatures belonging to registered ontologies that the KSA can instantiate and the knowledge sources it consults.

### 3.4.4 Component Discovery

UIMA component directory services may be used by the application or analysis engine developers. These components provide human browseable directory service for finding KSAs and TAEs. Among other criteria including the knowledge sources consulted, KSAs may be searched for based on the predicates they instantiate for a registered ontology. TAEs may be found based on their input and output requirements among other metadata features.

Component directory services will deliver a handle to a web service and/or an embeddable KSA or TAE component archive.

## 4 Analysis Engine Framework

This section takes a closer look at the analysis engine framework.

UIMA specifies an interface for an analysis engine; roughly speaking it is ‘CAS in’ and ‘CAS out’. There are other operations used for filtering, administrative and self-descriptive functions, but the main interface takes a CAS as input and delivers a CAS as output.

Any program that implements this interface may be plugged in as an analysis engine component in an implementation of UIMA. However, as part of UIMA tooling we have developed an analysis engine framework to support the creation, composition and flexible deployment of primitive and aggregate analysis engines on a variety of different system middleware platforms.

The underlying design philosophy for the Analysis Engine framework was driven by three primary principles:

1) Encourage and enable component reuse.

2) Support distinct development roles insulating the algorithm developer from system and deployment details.

3) Support a flexible variety of deployment options by insulating lower-level system middleware APIs.

### 4.1 The Common Analysis Structure (CAS)

As part of UIMA an XML specification for the CAS has been defined. It is possible to develop analysis engines that directly ingest and emit this XML. However, to support analysis engines that produce or consume complex data structures, the UIMA analysis engine framework provides a native language interface to the CAS. This has been implemented in both C++ and Java.

For example the Java interface to the CAS, termed the JCAS, reads the CAS data model (schema) and automatically generates the corresponding Java classes. Each object in the CAS is exposed as a Java object. A developer can create a new object by simply using the Java `new` operator and can access and modify an object's properties using standard Java `get` and `set` methods. Inheritance and relations between classes are also intuitively represented using the corresponding Java constructs. The JCAS provides serialization methods for both the standard UIMA CAS XML specification and a more efficient binary representation; the binary serialization is compatible with the C++ CAS implementation.

The primary advantage of the JCAS is that it reduces the learning curve for Java programmers to use the UIMA analysis engine framework while automatically providing interoperability with analysis engines developed using the XML or C++ manifestations of the CAS. Also, developers may customize the automatically generated Java classes by adding their own methods; these customized classes may then be used by other Java developers. Closely associating behavior with data in this way is one of the tenets of the Object Oriented Programming paradigm and offers numerous advantages to engineering natural language systems (Basili, Di Nanni, and Pazienza 1999). For instance, if the CAS data model is complex, those

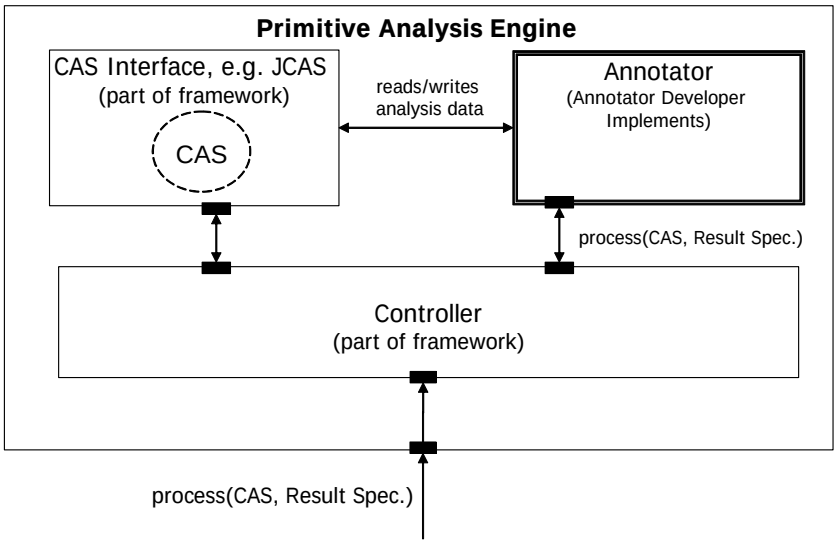


Figure 4: Primitive Analysis Engine

who wish to use the data are faced with a difficult task of learning the data model and developing their own code to extract relevant linguistic information. Using the JCAS, however, the designer of the data model can provide methods that abstract away many of the implementation details, reducing the burden on other developers and thereby encouraging reuse.

In both the Java and C++ implementations, elements in the CAS may be indexed for fast access (Goetz, Lougee-Heimer, and Nicolov 2001). The CAS specialization for Text includes a default index that orders annotations based on their position in the document, but developers can also define custom indexes keyed on other information in the CAS.

### 4.2 Encourage and Enable Component Reuse

With many NLP components being developed throughout IBM Research by independent groups, encouraging and enabling reuse is a critical design objective to achieve expected efficiencies and cross-group collaborations. Three characteristics of the analysis engine framework address this objective:

1) Recursive Structure
2) Data-Driven
3) Self-Descriptive

#### 4.2.1 Recursive Structure

A primitive analysis engine, illustrated in Figure 4, is composed of an *Annotator* and a native-language CAS interface, described in the previous section. The annotator is the object that implements the analysis

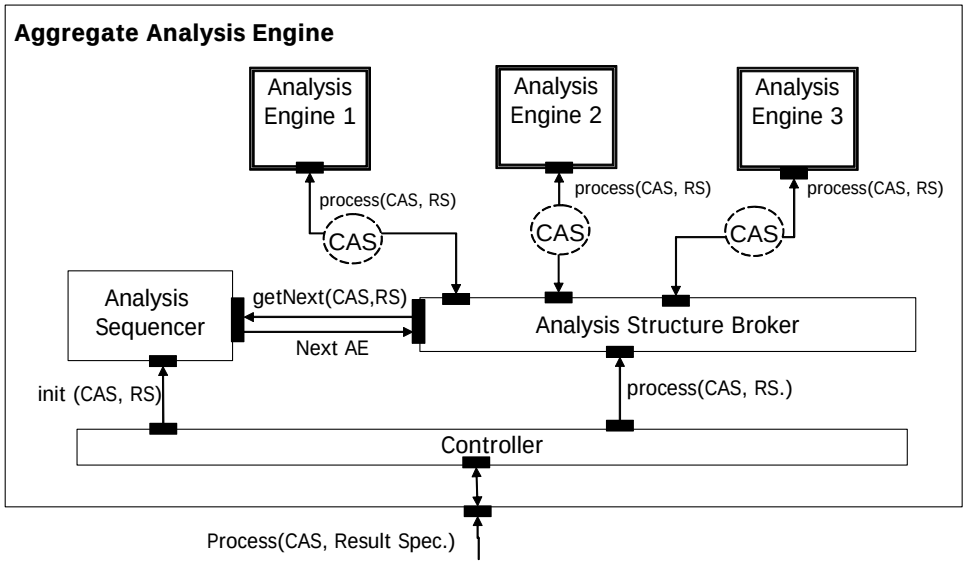


Figure 5: Aggregate Analysis Engine

logic (e.g. tokenization, grammatical parsing, or entity detection). It reads the original document content and metadata from the CAS. It then computes and writes new metadata to the CAS. An aggregate analysis engine, illustrated in Figure 5, is composed of two or more component analysis engines, but exposes exactly the same external interface as the primitive engine. At run-time an aggregate analysis engine is given a sequence in which to execute its component engines. A component called the *Analysis Structure Broker* ensures that each component engine has access to the CAS according to the specified sequence. Like any nested programming model, this recursive structure ensures that components may be easily reused in combination with one another while insulating their internal structure.

### 4.2.2 Data-Driven

An analysis engine's processing model is strictly data-driven. This means that an annotator's analysis logic may be predicated only on the content of its input and not on the specific analysis engines it may be combined with or the control sequence in which it may be embedded. This restriction ensures that an analysis engine may be successfully reused in different aggregate structures and different control environments as long as its input requirements are met.

The *Analysis Sequencer* is a component in the framework responsible for dynamically determining the next analysis engine to receive access to the CAS. The Analysis Sequencer is distinct from the Analysis Structure Broker, which has the responsibility to deliver the CAS to the next analysis engine whichever it

is wherever it may be located. The Analysis Sequencer's control logic is separate from the analysis logic embedded in an Annotator and separate from the Analysis Structure Broker's concerns related to ensuring and/or optimizing the CAS transport. This separation of concerns allows for the plug-and-play of different Analysis Sequencers. The Analysis Sequencer is a pluggable component, and implementations may range from providing simple iteration over a declaratively specified static flow to complex planning algorithms. Current implementations have been limited to simple linear flows between analysis engines; however more advanced applications are generating requirements for dynamic and adaptive sequencing. How much of the control specification should be in a declarative representation and how much should be implemented in the sequencer for these advanced requirements is currently being explored.

#### 4.2.3 Self-Descriptive

Ensuring that analysis engines may be easily composed to form aggregates and may be reused in different control sequences is necessary for technical reusability but not sufficient for enabling and validating reuse within a broad community of developers. To promote reuse, analysis engine developers must be able to discover which analysis engines are available in terms of what they do – their capabilities. In UIMA, analysis engines publish their input requirements and output specifications, and this information is used to register the analysis engine in an analysis engine directory service. This service includes a human-oriented interface that allows application developers to browse and/or search for analysis engines that meet their needs.

While self-description and related directory services can help promote reuse, having analysis engines conform to common data models (or fragments thereof) is essential for facilitating easy plug-and-play. While the CAS and UIMA remain theory neutral, related efforts at IBM are directed at establishing common data models.

### 4.3 Support Distinct Development Roles

Language technology researchers that specialize in multilingual machine translation, for example, may not be highly trained software engineers nor be skilled in the system technologies required for flexible and scalable deployments, yet one of the primary objectives of the UIMA project is to ensure that their work can be efficiently deployed in robust and scalable system architecture.

Along the same lines, researchers with ideas about how to combine and orchestrate different components may not themselves be algorithm developers or systems engineers, yet we need to enable them to rapidly create and validate ideas through combining existing components.

Finally, deploying analysis engines as distributed, highly available services or as collocated objects in an aggregate system requires yet another skill.

As a result we have identified the following development roles and have designed the architecture with independent sets of interfaces in support of each of these different skill sets. Our separation of development roles is analogous to the separation of roles in Sun's J2EE platform (Sun Microsystems, 2001).

### 4.3.1 Annotator Developer

The annotator developer role is focused on developing core algorithms ranging from statistical language recognizers to rule-based named-entity detectors to document classifiers. The annotator developer

1) Focuses on the development of algorithms and the logical representation of their results
2) Is insulated from: technical interoperability, external control, distribution, and deployment concerns.

This insulation is achieved through the analysis engine framework by requiring the annotator developer to understand only three interfaces, namely the *Annotator*, *AnnotatorContext*, and *CAS* interfaces. The annotator developer performs the following steps:

1) Implement Annotator interface
2) Encode analysis algorithm using the CAS interface to read input and write results
3) Use the AnnotatorContext interface to access external resources (e.g. dictionaries)
4) Write Analysis Engine Descriptor
5) Call Analysis Engine Factory

To embed an analysis algorithm in the framework, the annotator developer implements the Annotator interface. This interface is simple and requires the implementation of only two methods: one for initialization and one to analyze a document.

It is only through the CAS that the annotator developer accesses input data and registers analysis results. The CAS contains the original document (the subject of analysis) plus the metadata contributed by any analysis engines that have run previously. This metadata may include annotations over elements of the

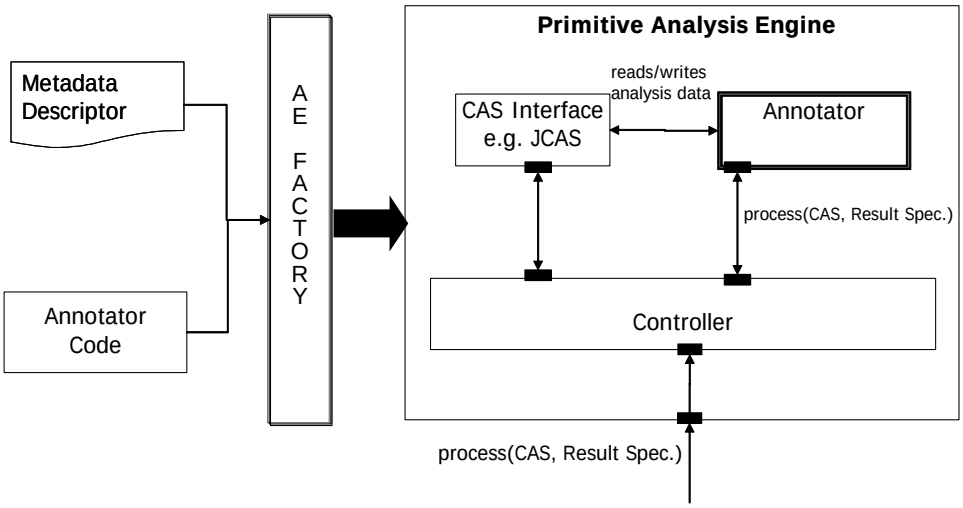


Figure 6: AE Factory builds Primitive Engine for Annotator Developer

original document. The analysis engine framework provides an easy-to-use native language interface to the CAS; for Java developers this is the JCAS interface described in section 4.1. The actual CAS data that are input to an analysis engine may reside in memory, be managed remotely, or be shared by other components. These issues are of concern to the analysis engine deployer role, but the annotator developer is insulated from these issues.

All external resources, such as dictionaries, that an annotator needs to consult are accessed through the Annotator Context interface. The exact physical instantiation of the data can therefore be determined by the deployer, as can decisions about whether and how to cache the resource data.

The annotator developer completes an XML descriptor that identifies the input requirements, output specifications, and external resource dependencies. Given the annotator object and the descriptor, the framework's Analysis Engine Factory returns a complete analysis engine (see Figure 6).

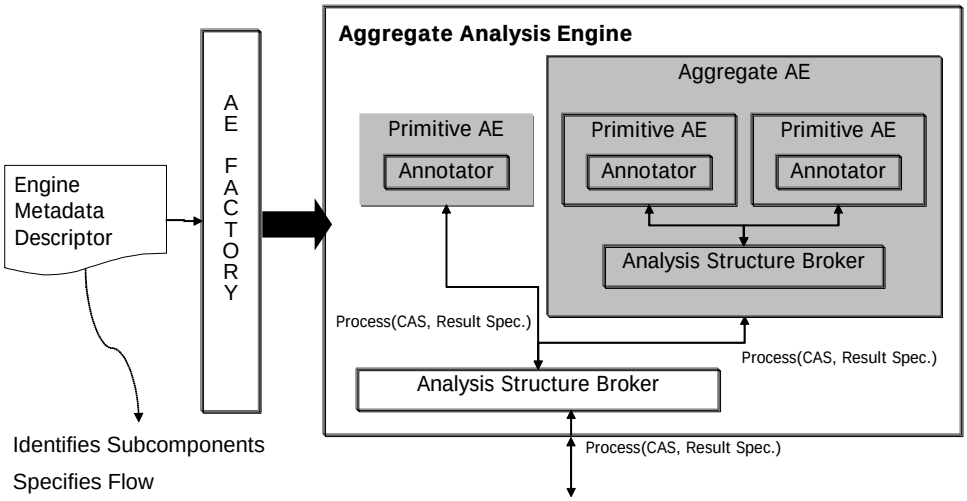


Figure 7: AE Factory builds Aggregate Engine for Analysis Engine Assembler

### 4.3.2 Analysis Engine Assembler

The analysis engine assembler creates aggregate analysis engines through the declarative coordination of component engines. The design objective is to allow the assembler to build an aggregate engine without writing any code. The analysis engine assembler

1) Combines existing components to deliver aggregate analysis capability
2) Is not required to write code (only declarative specifications)
3) Is insulated from: algorithm development, technical interoperability, external control, distribution and deployment concerns.

The analysis engine assembler considers available engines in terms of their capabilities and declaratively describes flow constraints. These constraints are captured in the aggregate engine's XML descriptor along with the identities of selected component engines. As depicted in Figure 7, the assembler inputs this descriptor into the framework's analysis engine factory object and an aggregate analysis engine is created and returned. The framework's Analysis Structure Broker and Analysis Sequencer components manage the run-time sequencing and CAS communication between component engines.

### 4.3.3 Analysis Engine Deployer

The analysis engine deployer decides how analysis engines and the resources they require are deployed on particular hardware and system middleware. UIMA does not provide its own specification for how com-

ponents are deployed, nor does it mandate the use of a particular type of middleware or middleware product. Instead, UIMA aims to give deployers the flexibility to choose the middleware options that meets their needs.

The analysis engine deployer

- Selects deployment options
    - System middleware, scalability/embeddable
    - Target environment, computing resources
- Uses framework-provided *Service Wrappers* (described in the section 4.4) for deploying analysis engines on common types of system middleware. No coding is required.
- Insulated from: algorithm development and component capabilities/composition

The analysis engine deployer role benefits from the annotator developer's use of XML descriptors and the AnnotatorContext interface for external resource access. For example, an annotator that needs to access a dictionary does not do so by directly reading it from a file on disk. Instead, the annotator developer declaratively defines the dependency on the dictionary resource in the XML descriptor and associates a key with that resource. The annotator code then accesses the dictionary by providing this key to the AnnotatorContext. Because of the declarative specification of resource dependencies, the analysis engine deployer is able to easily discover what resources are required by an analysis engine. The level of indirection provided by the AnnotatorContext allows the deployer to manage the dictionary data in whatever way make sense for the currently deployment – for example an in-memory cache or a service interface rather than the file system.

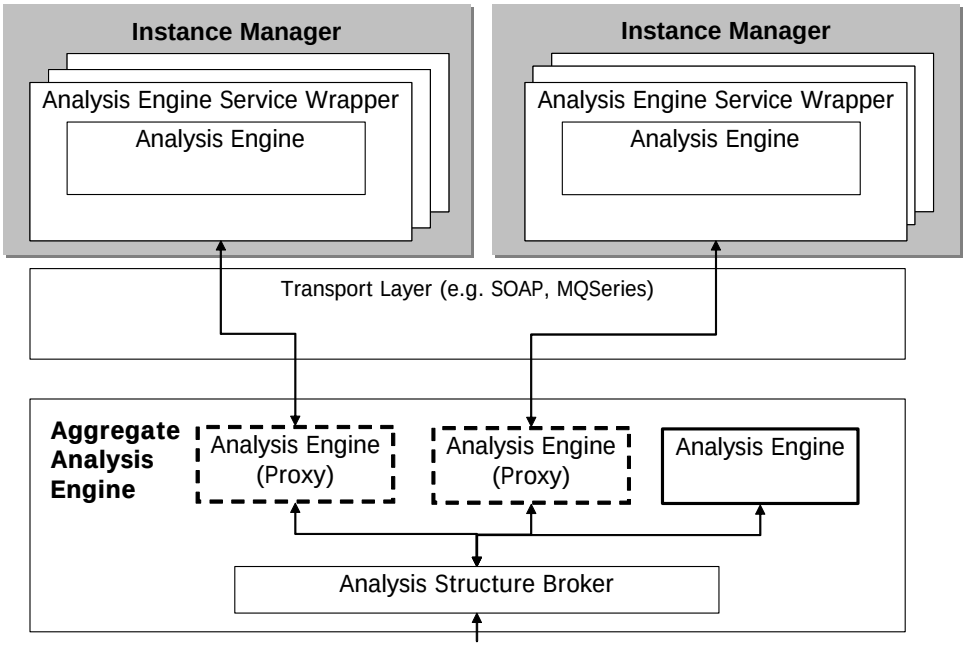


Figure 8: Aggregate analysis engine with two distributed component analysis engines

### 4.4 Insulate Lower-Level System Middleware

UIM applications can share many requirements with other types of applications – for example, they may need scalability, security, and transactions. Existing middleware such as application servers can meet many of these needs. On the other hand, UIM applications may need to have a small footprint so they can be deployed on a desktop computer or PDA or they may need to be embeddable within other applications that use their own middleware.

One design goal of UIMA is to support deployment of analysis engines on any type of middleware, and to insulate the annotator developer and analysis engine assembler from these concerns. This is done through the use of *Service Wrappers* and the Analysis Structure Broker. The analysis engine interface specifies that input and output are done via a CAS, but it does not specify how that CAS is transported between component analysis engines. A service wrapper implements the CAS serialization and deserialization necessary for a particular deployment. Within an aggregate Analysis Engine, components may be deployed using different service wrappers. The Analysis Structure Broker is the component that transports the CAS between these components regardless of how they are deployed (see Figure 8).

To support a new type of middleware, a new service wrapper and an extension to the Analysis Structure Broker must be developed and plugged into the framework. The Analysis Engine itself does not need to be modified in any way.

For example, we have implemented Service Wrappers and Analysis Structure Broker on top of both a web-services and a message queuing infrastructure. Each implementation has different pros and cons for deployment scenarios.

## 5 Evaluating UIMA's Impact

The UIMA project was initiated at IBM based on the premise that an architectural foundation for developing, integrating and deploying UIM technologies, if adopted, would facilitate reuse and integration of natural language technologies under development at IBM and elsewhere with each other and within IBM products. The project has received organizational support through a variety of incentives that have assisted in the adoption of the architecture.

Measuring success of the architecture independently of the project and the surrounding organization is itself more an art than a science. We believe that traditional quantitative measures of component accuracy do not necessarily reflect on the value of the architecture. This is not to suggest, however, that UIMA frameworks can not play a significant role in enabling the integration of components which in turn results in better accuracy.

Adoption is an important measure. One may argue that if the architecture is adopted by the target community then it must be considered valuable. But this is an indirect and potentially inaccurate measure that is affected by other organizational or project incentives. Without rigorous user studies we can not develop a precise evaluation. In the short-term, it may not be practical to entirely separate the evaluation of the architecture from the project and the surrounding organization that has helped it adoption. As a result we have considered three ways to begin to evaluate if UIMA is meeting its intended objectives:

1) Compliant Components and their Reuse. Is UIMA being used to produce and reuse components within Research to advance scientific agendas?

2) System Performance. Has the use of the architecture provided a means to improve overall system throughput for UIM applications? What are some of the tradeoffs?

3) Product Integration. Have product groups accepted UIMA as the foundation for plugging in natural language or other unstructured information analysis capabilities?

### 5.1 Compliant Components and their Reuse

One way to measure the impact of UIMA is to look at its adoption by the target community. This may be measured by the number of compliant components and the frequency of their reuse by different projects/groups.

We have developed a registry and an integration test bed where contributed components are tested, certified, registered and made available to the community for demonstration and download.

The integration test bed includes a web-based facility where users can select from a collection of corpora, select from a collection of certified analysis engines, and run the analysis engine on the corpus. Performance statistics breaking down the time spent in analysis, communications between components, and framework overhead are computed and presented. The system generates analysis results and stores them. Analysis results may be viewed using any of a variety of CAS viewers. The results may also be indexed by a search engine, which may then be used to process queries.

While we are still instrumenting this site and gathering reuse data, within six months of an internal distribution we have over fifteen UIMA-compliant analysis engines, many of which are based on code developed as part of multi-year research efforts. Engines were contributed from six independent and geographically dispersed groups. They include several named-entity detectors of the rule-based and statistical varieties, several classifiers, a summarizer, deep and shallow parsers, a semantic class detector, an Arabic to English translator, an Arabic person detector, and several biological entity and relationship detectors. Several of these engines have been assembled using component engines that were previously inaccessible for reuse due to engineering incompatibilities.

A large project underway at IBM is based on providing a large-scale capability for mining the web for various types of semantic content. Some of the results based on this capability have been recently published (Dill, Eiron, Gibson, Gruhl, Guha, Jhingran, Kanungo, Rajagopalan, Tomkins, Tomlin and Zien 2003). Much of the implementation predates UIMA; however, the project is adopting UIMA's analysis engine architecture for creating and deploying analysis capabilities. In addition UIMA interfaces are being

adopted and this project is contributing robust implementations of framework components including the document and collection metadata store.

### 5.2 System Performance

We expect the architecture's support for modularization and for the separation of algorithm development from analysis assembly and system deployment issues to result in opportunities to quickly deploy more robust, reliable and scalable solutions. We have observed interesting tradeoffs. UIMA's support of inter-operability based on distributed deployment options using a serviced-based infrastructure has led to transparent reuse of existing components, but at a the price of CAS serialization/deserialization and network transmission. This tradeoff has enabled a migration path uniquely useful in the research community. For example, analysis engine developers have first delivered their components as service-based TAEs embedded in unique and often quirky environments. Clients, either aggregate engine developers or application developers, using the UIMA framework can compose their solutions independently of how the component TAE is deployed. If they have found a component useful and have a keen interest in avoiding the remote service call overhead, a tightly-coupled version may be created. The application code would not change at all. Only a new location of the component would be provided.

As another example, IBM's Question Answering system, now under development for over three years, includes a home-grown answer type detector to analyze large corpora of millions of documents (Prager, Chu-Carroll, Brown, and Czuba 2002). An analysis algorithm developer wrapped the answer type detector as a UIMA Annotator and embedded it in the UIMA Analysis Engine framework. The framework provided the infrastructure necessary for configuring an aggregate analysis engine with the answer type detector down-stream of a tokenizer and named-entity detector without additional programming. Within a day, the framework was used to build the aggregate analysis engine and to deploy multiple instances of it on a host of machines using the framework's middleware facilities for creating and managing TAE instances. The result was a dramatic improvement in overall throughput.

### 5.3 Product Integration

IBM develops a variety of information management, information integration, data mining, knowledge management and search-related products and services. Unstructured information processing and language

technologies in particular represent an increasingly important capability that can enhance all of these products.

UIMA will be a business success for IBM if it plays an important role in technology transfers. IBM Research has engaged a number of product groups which are realizing the benefits of adopting a standard architectural approach for integrating NLP that does not constrain algorithm invention and that allows for easy extension and integration and support for a wide variety of system deployment options.

## 6 Conclusion

The UIMA project at IBM has encouraged many groups in six of the Research division's labs to understand and adopt the UIMA architecture as a common conceptual foundation and engineering framework for classifying, describing, developing and combining NLP components in applications for processing unstructured information. The unique challenges present in a corporate research environment drove key design points for UIMA. Perhaps the most significant are reflected in UIMA's commitments to the explicit separation of development roles and the flexible support for a variety of deployment options. These commitments resulted in a framework that supports the insulation of the natural language processing logic and its developers from systems concerns and allows tightly coupled deployments to be delivered as easily as service-oriented distributed deployments. We are finding these features critical for facilitating transfer of Research results to product, which has in turn created a further incentive for adoption of the architecture by our research organization.

While our measurements of UIMA's impact are only just beginning, the adoption of UIMA has notably improved knowledge transfer throughout the organization. Implementations of the architecture are advancing and are beginning to demonstrate that NLP components developed within Research can be quickly combined to explore hybrid approaches as well as to rapidly transfer results into IBM product and service offerings.

IBM product groups are encouraged by Research's effort and are committed to leverage UIMA as a vehicle to embed NLP capabilities in established product platforms. They are realizing the benefits of having Research adopt a standard architectural approach that does not constrain algorithm invention and allows for a wide variety of system deployment options. Product and service groups are seeing an easier

path to combine, integrate and deliver these technologies into information integration, data mining, knowledge management and search-related products and services.

Next steps for the UIMA project at IBM include the investigation of advanced control structures, stronger support for the disciplined integration and maintenance of ontology resources and a closer relationship with the open-source Eclipse Project (www.eclipse.org) as a potential vehicle for making UIMA frameworks generally available as part of a software development tool-kit.

## Acknowledgements

We acknowledge the management efforts of Alfred Spector and Arthur Ciccolo for making the focus on architecture and engineering in the NLP and UIM area an executive-level commitment. We acknowledge the contributions of Dan Gruhl and Andrew Tomkins of IBM's Web Fountain project to the development of UIMA. In addition we acknowledge David Johnson, Thomas Hampp, Thilo Goetz and Oliver Suhre in the development of IBM's Text Analysis Framework and the work of Roy Byrd and Mary Neff in the design of the Talent system. Their work continues to influence the UIMA designs and implementations. UIMA would have never gotten off the ground nor would quick advances be made without the rapid prototyping skills and committed software development efforts of Jerry Cwiklik. Finally, we would like to acknowledge Marshall Schor for his invaluable contributions to the careful design and implementation of the UIMA analysis engine framework and to the writing of this paper.

Portions of this paper describing elements of the same work appeared in the proceedings of the Software Engineering and Architecture of Language Technology Systems workshop, published by the Association for Computation Linguistics (Ferrucci and Lally 2003).

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

## References

Basili, Roberto, Di Nanni, Massimo, and Pazienza, Maria Teresa. 1999. Engineering of IE Systems: An Object-Oriented Approach. In Pazienza (Ed.), *Information Exctraction*, LNAI 1714, pp. 134-164. Berlin Heidelberg: Springer-Verlag.

Bontcheva, Kalina, Cunningham, Hamish, Tablan, Valentin, Maynard, Diana, and Saggion, Horacio. 2002. 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.

Chu-Carroll, Jennifer, Ferrucci, David, Prager, John, and Welty, Christopher. 2003. Hybridization in Question Answering Systems. *Working Notes of the AAAI Spring Symposium on New Directions in Question Answering*. AAAI Press.

Cunningham, Hamish, Bontcheva, Kalina, Tablan, Valentin, and Wilks, Yorick. 2000. Software Infrastructure for Language Resources: a Taxonomy of Previous Work and a Requirements Analysis. In *Proceedings of the Second Conference on Language Resources Evaluation*, Athens.

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. WWW2003, Budapest, Hungary.

Ferrucci, David and Lally, Adam. 2003. Accelerating Corporate Research in the Development, Application and Deployment of Human Language Technologies. In *Proceedings of the HLT-NAACL 2003 Workshop on Software Engineering and Architectures for Language Technology Systems*, pp. 68-75, Edmonton, Canada.

Genesereth, Michael R. and Fikes, Richard E. 1992. Knowledge Interchange Format Version 3.0 Reference Manual. Technical report, Logic Group, Stanford University, CA.

Grishman, Ralph. 1996. Tipster architecture design document version 2.2. Technical report, DARPA.

Goetz, Thilo, Lougee-Heimer, Robin, and Nicolov, Nicolas. 2001. Efficient Indexing for Typed Feature Structures. In *Proceedings of Recent Advances in Natural Language Processing*, Tzigov Chark, Bulgaria.

Laprun, Christophe, Fiscus, Johnathan, Garofolo, John, and Pajot, Sylvain. 2002. A Practical Introduction to ATLAS. In *Proceedings of the Third International Conference on Language Resources and Evaluation (LREC)*.

Prager, John, Brown, Eric, Coden, Anni, and Radev, Dragomir. 2000. Question-answering by Predictive Annotation. In *Proceedings of ACMSIGIR*.

Prager, John, Chu-Carroll, Jennifer, Brown, Eric and Czuba, Krzysztof. 2003. Question Answering Using Predictive Annotation. In Strzalkowski, T. and Harabagiu, S. (eds.), *Advances in Open-Domain Question Answering*, Kluwer Academic Publishers, to appear.

Mardis, Scott and Burger, John. 2002. Qanda and the Catalyst Architecture. *AAAI Spring Symposium on Mining Answers from Text and Knowledge Bases*.

Roush, Wade. 2003, Computers that Speak Your Language, *Technology Review*, 106 (5): 32.

Sun Microsystems, Inc. 2001. Java™ 2 Platform Enterprise Edition Specification, v1.3. http://java.sun.com/j2ee/1.3/docs/.