Accelerating Corporate Research in the Development, Application and Deployment of Human Language Technologies

David Ferrucci
IBM T.J. Watson Research Center
Yorktown Heights, NY 10598
ferrucci@us.ibm.com

Adam Lally
IBM T.J. Watson Research Center
Yorktown Heights, NY 10598
alally@us.ibm.com

Abstract

IBM Research has over 200 people working on Unstructured Information Management (UIM) technologies with a strong focus on HLT. Spread out over the globe they are engaged in activities ranging from natural language dialog to machine translation to bioinformatics to 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 HLT into IBM's product platforms. Market analyses indicating a growing need to process unstructured information, specifically multi-lingual, natural language text, coupled with IBM Research's investment in HLT, 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 Architecture Goals

In six major labs spread out over the globe, IBM Research has over 200 people working on Unstructured Information Management (UIM) technologies with a significant focus on Human Language Technologies (HLT). These researchers are engaged in activities ranging from natural language dialog to machine translation to bioinformatics to open-domain question answering. Each group is developing different technical and engineering approaches to process unstructured information (e.g., natural language text, voice, audio and video) in pursuit of specific research objectives and their applications.

The high-level objectives of IBM's Unstructured Information Management Architecture (UIMA) are two fold:

1) Accelerate scientific advances by enabling the rapid combination UIM technologies (e.g., natural language processing, video analysis, information retrieval, etc.).

2) Accelerate transfer of UIM technologies to product by providing a robust software framework that promotes reuse and supports flexible deployment options.

UIMA is a software architecture for developing applications which integrate search and analytics over a combination of structured and unstructured information. 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 is a database table. We define *unstructured information* as information whose intended meaning is only implied by its form. The canonical example is a natural language document.

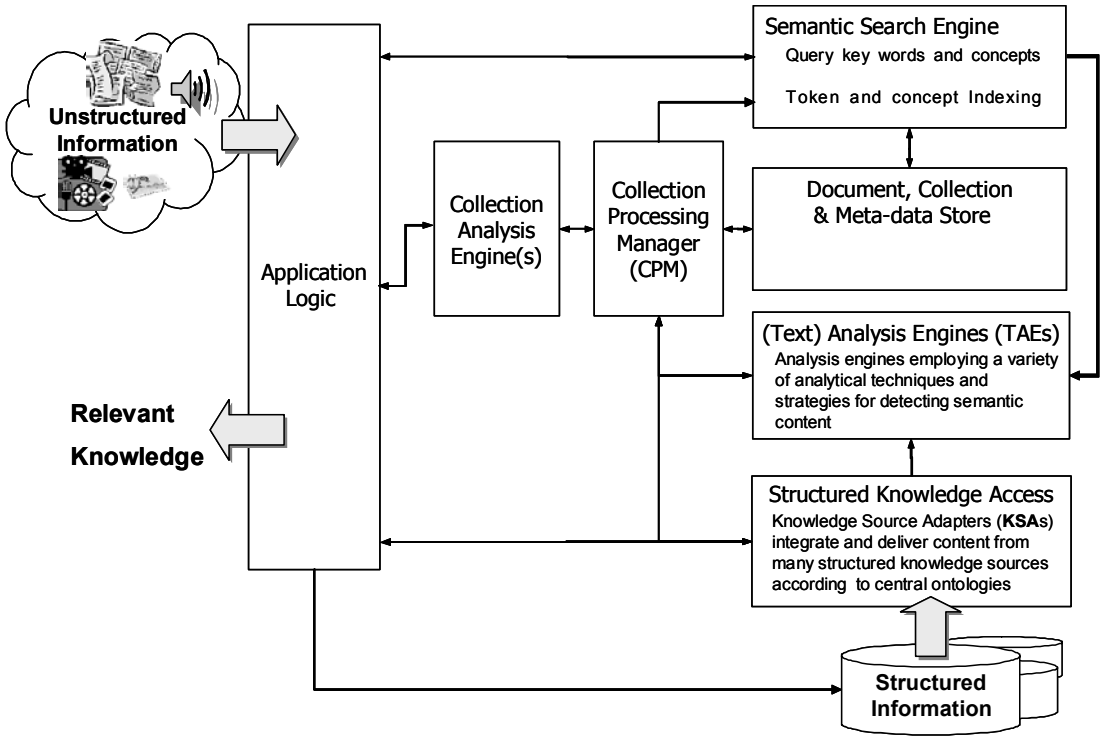

Figure 1: UIMA High-Level Architecture

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.

As part of the UIMA project, IBM is developing different implementations of the architecture suitable for different classes of deployment. These range from lightweight and embeddable implementations to highly scaleable implementations that are meant to exploit clusters of machines and provide high throughput and high availability.

While the architecture extends to a variety of unstructured artifacts including voice, audio and video, a primary analytic focus of current UIMA implementations is squarely on human language technologies.

In this paper we will refer to elements of unstructured information processing as *documents* admitting, however, that an element may represent for the application, a whole text document, a text document fragment or even multiple documents.

2 Generalized Application Scenario and the High-Level Architecture

In this section we provide a high-level overview of the UIMA architecture by describing its component roles in a generalized application scenario.

The generalized scenario includes both analysis and access functions. Analysis functions are divided into two classes, namely document-level and collection-level analysis. Access functions are divided into semantic search and structured knowledge access.

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

2.1 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 et al., 2000). In UIMA, a TAE is a recursive structure which may be composed of sub or component engines each performing a different stage of the application's analysis.

Examples of Text Analysis Engines include language translators, document summarizers, document classifiers, and named-entity detectors. 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 common structure called the *Common Analysis System* or *CAS*. The CAS is conceptually analogous to the *annotations* in other architectures, beginning with TIPSTER (Grishman, 1996).

In general, annotations associate some meta-data with a region in the original artifact. Where the artifact is a text document, for example, the annotation associates meta-data (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). In UIMA, annotations are not the only type of information stored in the CAS. The CAS may be used to represent any class of meta-data element associated with analysis of a document regardless of whether it is explicitly linked to some sub component of the original document. The CAS also allows for multiple definitions of this linkage, as is necessary for the analysis of images, video or other modalities.

The analysis represented in the CAS may be thought of as a collection of meta-data 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 CAS 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.

The UIMA CAS object provides general object-based representation with a hierarchical type system supporting single inheritance. It includes data creation, access and serialization methods designed for the efficient representation, access and transport of analysis results among TAEs and between TAEs and other UIMA components or applications. Elements in the CAS may be indexed for fast access (Goetz et al., 2001). The CAS has been implemented in C++ and Java with serialization methods for binary as well as XML formats for managing the tradeoff between efficiency and interoperability.

2.2 Collection-Level Analysis

Documents are gathered by the application and organized into collections. The architecture defines a *Collection Reader* interface. Implementations of the Collection Reader provide access to collection elements, collection meta-data and element meta-data. UIMA implementations include a document, collection and meta-data store that implements the Collection Reader interface and manages multiple collections and their elements. However, applications that need to manage their own collections can provide an implementation of a Collection Reader to UIMA components that require access to collection data.

Collections are analyzed to produce collection level analysis results. These results represent aggregate inferences computed over all or some subset of the documents in a collection. The component of an application that analyzes an entire collection is considered a *Collection Analysis Engine*. These engines typically apply element-level, or more specifically document-level analysis, to elements of a collection and then considering the element analyses in performing aggregate computations.

Examples of collection level analysis results include sub collections where 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.

In support of Collection Analysis Engines, UIMA defines the *Collection Processing Manager* (CPM) component. The CPM's primary responsibility is to manage the application of a designated TAE to each document accessible through a Collection Reader. A Collection Analysis Engine may provide, as input to the CPM, a TAE and a Collection Reader. The CPM applies the TAE and returns the analysis, represented by a CAS, for each element in the collection. To control the process, the CPM provides administrative functions that include failure reporting, pausing and restarting.

At the request of the application's collection analysis engine, the CPM may be optionally configured to perform functions typical of UIM application scenarios. Examples of these include:

1) **Filtering** - ensures that only certain elements are processed based on meta-data constraints.
2) **Persistence** - stores element-level analysis results in a provided *Collection Writer*.
3) **Indexing** - indexes documents using a designated search engine indexing interface based on meta-data extracted from the analysis.
4) **Parallelization** - manages the creation and execution of multiple instances of a TAE for processing multiple documents simultaneously utilizing available computing resources.

2.3 Semantic Search

To support the concept of "semantic search" – the capability to find documents based on semantic 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.

The key feature of the query interface is that it supports queries that may be predicated on nested structures of annotations and tokens in addition to Boolean combinations of tokens and annotations.

2.4 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 (KIF) encoded in XML.

A key aspect of the KSA architecture is the KSA meta-data 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 meta-data includes the predicate signatures belonging to registered ontologies that the KSA can instantiate and the knowledge sources it consults.

Application or analysis engine developers can consult human browseable KSA directory services to search for and find KSAs that instantiate predicates of a registered ontology. The service will deliver a handle to a web service or an embeddable KSA component.

3 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.

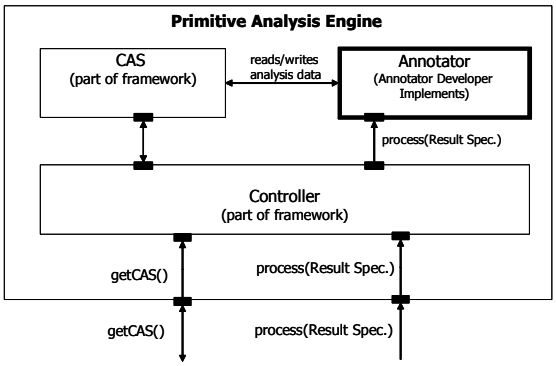

Figure 2: Primitive Analysis Engine

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.

3.1 Encourage and Enable Component Reuse

With many HLT 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

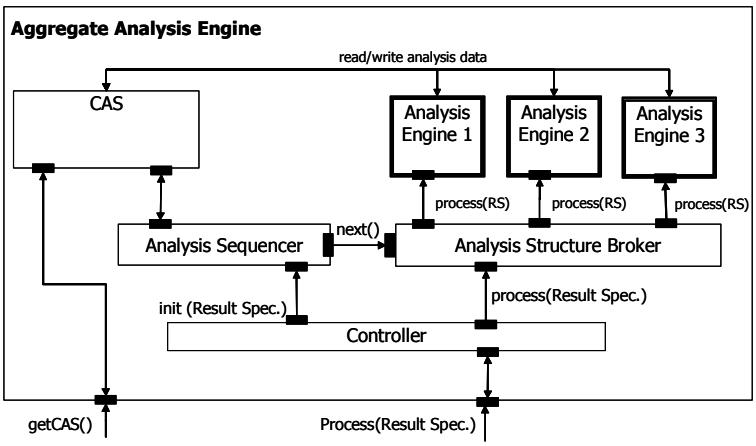

Figure 3: Aggregate Analysis Engine

Recursive Structure. A primitive analysis engine, illustrated in Figure 2, is composed of an *Annotator* and a CAS. The annotator is the object that implements the analysis logic (e.g. tokenization, grammatical parsing, entity detection). It reads the original document content and meta-data from the CAS. It then computes and writes new meta-data to the CAS. An aggregate analysis engine, illustrated in Figure 3, is composed of two or more component analysis engines, but implements 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.

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, whose responsibility is 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-n-play of different Analysis Sequencers. The Analysis Sequencer is a pluggable range from provide 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 ends up in a declarative representation and how much is implemented in the sequencer for these advanced requirements is currently being explored.

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.

Each analysis engine's data model is declared in XML and then dynamically realized in the CAS at runtime, an approach similar to MAIA (Laprun et al., 2002). In UIMA, however, analysis engines publish their input requirements and output specifications relative to this declared data model, 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 will promote reuse, their value is still dependent on establishing common data models (or fragments thereof) to which analysis engine capability descriptions subscribe.

3.2 Support Distinct Development Roles

Language technology researchers that specialize in, for example, multi-lingual machine translation, may not be

highly trained software engineers nor be skilled in the system technologies required for flexible and scaleable deployments. Yet one of the primary objectives of the UIMA project is to ensure that their work can be efficiently deployed in robust and scaleable 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).

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 framework design ensures that the annotator developer need NOT develop code to address aggregate system behavior or systems issues like interoperability, recovery, remote communications, distributed deployment, etc., but instead allow them to focus squarely on the algorithmic logic and the logical representation of their results.

This was 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 and the AnnotatorContext interface to access resources
3) Write Analysis Engine Descriptor
4) 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 meta-data contributed by any analysis engines that have run previously. This meta-data may include annotations over elements of the original document. The CAS input to an analysis engine may reside in memory, be managed remotely, or 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 manifestation 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.

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 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. The assembler inputs this descriptor in the framework's analysis engine factory object and an aggregate analysis engine is created and returned.

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 components 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 that meets their needs.

3.3 Insulate Lower-Level System Middleware

HLT 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, HLT 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.

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.

4 Measuring Success

We have considered four ways to evaluate UIMA in meeting its intended objectives:

1) Combination Experiments
2) Compliant Components and their Reuse
3) System Performance Improvements
4) Product Integration

4.1 Combination Experiments

Our UIMA implementations and associated tooling have led to the design of several combination experiments that were previously unimagined or considered too cumbersome to implement. This work has only just begun but includes collaborative efforts combining rule-based parsers with statistical machine translation engines, statistical named-entity detectors in rule-based question answering systems (Chu-Carroll et al., 2003), and a host of independently developed analysis capabilities in bioinformatics applications.

4.2 Compliant Components and their Reuse

Another 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, 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 will have over 15 UIMA-compliant analysis engines, many of which are based on code developed as part of multi-year HLT research efforts. Engines were contributed from six independent and geographically dispersed groups. They include several named-entity detectors of both 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.

4.3 System Performance Improvements

We expect the architecture's support for modularization and for the insulation of system deployment issues from algorithm development to result in opportunities to quickly deploy more robust and scaleable solutions.

For 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 et al., 2002). With a half day of training, an algorithm developer cast 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. The result was a dramatic improvement in overall throughput.

4.4 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 HLT that does not constrain algorithm invention and that allows for easy extension and integration and support for a wide variety of system deployment options.

5 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 for classifying, describing, developing and combining HLT components in aggregate applications that integrate search and analytical functions.

While our measurements are only just beginning, the adoption of UIMA has clearly improved knowledge transfer throughout the organization. Implementations of the architecture are advancing and are beginning to demonstrate that HLT 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 HLT components. They are realizing the benefits of having Research adopt a standard architectural approach that does not constrain algorithm invention while allowing 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.

Acknowledgements

We acknowledge the contributions of Dan Gruhl and the WF 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 CAS and analysis engine framework.

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

Kaling Bontcheva, Hamish Cunningham, Valentin Tablan, Diana Maynard, Horacio Saggion. 2002. "Developing Reusable and Robust Language Processing Components for Information Systems using GATE." 3rd International Workshop on Natural Language and Information Systems (NLIS'2002), IEEE Computer Society Press.

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

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

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

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

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

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

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

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

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