Design and implementation of the UIMA Common Analysis System

by T. Götz
O. Suhre

The Common Analysis System (CAS) is the subsystem in the Unstructured Information Management Architecture (UIMA) that handles data exchanges between the various UIMA components, such as analysis engines and unstructured information management applications. CAS supports data modeling via a type system independent of programming language, provides data access through a powerful indexing mechanism, and provides support for creating annotations on text data. In this paper we cover the CAS design philosophy, discuss the major design decisions, and describe some of the implementation details.

The Unstructured Information Management Architecture (UIMA) defines a framework for implementing systems for the analysis of unstructured data.[1,2] In contrast to *structured information,* whose meaning is expressed by the structure or format of the data, the meaning of *unstructured information* cannot be so inferred. Examples of data that carry unstructured information include natural language text and data from audio or video sources. More specifically, an audio stream has a well-defined syntax and semantics for rendering the stream on an audio device, but its music score is not directly represented.

In UIMA, Common Analysis System (CAS)[3] is the subsystem that handles data exchanges between the different components and unstructured information management (UIM) applications. UIMA components known as *analysis engines* receive analysis results from other components and produce new results that include their own contribution. Similarly, all results of an analysis engine are contained in CAS and extracted from there by the invoking application. Note that although we refer to UIMA as a "component architecture" for systems that perform analysis of unstructured data, it is not a general component architecture such as CORBA**[4] or J2EE**.[5] An important constraint is that UIMA components do not share or exchange code; all they exchange is data.

The functionality of CAS can, on a high level, be compared to that of a database engine. The data model is defined by a type system that corresponds to the table and column definitions of a relational database. The objects licensed by the type system, called *feature structures,* correspond to rows in a database table. Unlike databases, however, feature structures created by one component cannot be accessed directly by another. A component makes its feature structures accessible to other components by explicitly placing them in an index (not to be confused with a database index). Accessing a CAS index can be compared with accessing database records through views, which may hide certain rows or change their ordering. The application programming interface (API) for accessing CAS indexes is based on the iterator pattern, in the same way as views (or tables, in general) can be traversed by cursors.[6]

0018-8670/04/$5.00 © 2004 IBM

The current UIMA work is the successor and a generalization of the IBM Text Analysis Framework (TAF),[7] based, as its name suggests, upon text analysis. Nevertheless, many ideas in TAF on componentization and data modeling have been adapted and generalized in UIMA. Today, TAF is the C++ implementation of UIMA.

There are other UIM frameworks similar to UIMA, most notably GATE (see Reference 8 and references cited therein) and ATLAS.[9] GATE is specific to text and does not appear to provide a specialized data layer, using instead native data structures (the current version of GATE is implemented in Java**).

Although the ATLAS project has an effort underway to add a data abstraction layer, its basic approach is somewhat different from ours. On the one hand, it has a richer built-in data model, one that is suited to specific tasks; that is, all data in ATLAS appear to be (a generalization of) annotation graphs as developed in Reference 10. In CAS, such data structures would need to be built out of the basic building blocks. On the other hand, the added convenience for application developers comes at the price of reduced generality. In our case, we opted for increased flexibility in the core—more specific data models and support layers can always be added on top of CAS.

The rest of the paper is structured as follows. In the next section, we discuss why something like CAS is needed in UIMA, and we justify the requirements for CAS by describing typical usage scenarios in UIMA. Then, we describe the key concepts of CAS and in particular the type system, the feature structures, and the index repository. We also include an example illustrating the APIs for implementing these concepts. In the following section, we discuss aspects of the implementation, namely the implementation of feature structures and the index repository. We also include a section on serialization in CAS, a topic of interest when using CAS over a network or when interoperability between Java and C++ is an issue. In the concluding section we provide a brief summary.

Motivation

Individual components in UIMA may use not only data provided by external applications, but also data from upstream components. The question is the kind of data that should be passed between components and between these components and external applications.

In principle, there is a continuum of approaches that can be taken on this issue. At one extreme data modeling could be left entirely to the individual components, and data could simply be transferred between components without any knowledge about the structure of the data. This is the most flexible approach because there are no restrictions on the kind of data that can be modeled. But there are drawbacks. The framework can offer no support for handling the data. Moreover, APIs and tools for dealing with the data must be provided by the individual components. In the Web world, this can be compared to XML (eXtensible Markup Language) without validation, i.e., without DTDs (document type definitions) or schemas.[11] Such an XML document might contain any kind of data, and for communication to be possible, the data format must be known both to the sender and the receiver of the XML message.

At the other extreme we could have the framework precisely define the kind of data that may be used and exchanged by individual components. Then, the framework provides APIs to define and manipulate data. Clearly the advantage is that many generic data services can be provided by the framework. However, this approach is not flexible because the data model is defined beforehand and cannot be easily extended. This approach works well only if the data model is unlikely to change. In our Web analogy, this might correspond to HTML (HyperText Markup Language); that is, data with a fixed structure and meaning.

In a third alternative, we try to steer a middle course between the two extremes by using a data model that is understood by the framework but can be freely modified. The data model itself is treated as data. In our Web analogy, this might correspond to XML schemas that define the structure of XML documents. Schemas are used by XML parsers to validate the syntax of XML data and constitute data objects in their own right. This works although there are an infinite number of different XML schemas.

In this third approach, the one adopted for UIMA and CAS, we try to combine the advantages of the two extremes: the flexibility of the first approach and the framework support for data services of the second approach. On the negative side, there is increased complexity; for the framework designers it is more difficult to get the details right, and for the application developers, it is more difficult to master the programming environment.

The usability issue is addressed by the Java class model (JCas) Java APIs. A precompilation step can be used to generate Java source code that provides users with an intuitive, bean-like view of their data. This can be compared to the Java API for XML Binding (JAXB),[12] which provides a similar service for XML data. See the sidebar in the subsection, "Code Example," under the section, "Main Concepts," for more detailed information on JCas.

In the rest of this section, we explain why we consider control over the data so important that we are willing to live with the disadvantages mentioned above. The main considerations in our decision involved *runtime component assembly, data-driven inference,* and *data exchange.* We will address each of these in turn.

Component assembly at runtime. Probably the most important reason to build UIMA was component reuse. In the absence of such an architecture, the average team finds building its own tokenizer easier than interfacing to one of the myriad of tokenizers that are already available. Thus one of the fundamental requirements of UIMA is that one team should be able to easily integrate another team's components. To this end, we need to be able to write components without full knowledge of the data that those components consume. We now consider some examples that further illustrate this point.

Example: Get analysis results; take only what you need. Team A has written a lexical analysis component that does tokenization, part-of-speech tagging, and dictionary look-up for 30 languages, including Chinese. Team B also needs a tokenizer. They have written their own for English, but their customers also require one for Chinese, which they are unable to do themselves (other required functionality is language independent). Although they would like to use team A's tokenizer for Chinese, they prefer not to adopt Team A's entire data model, because it goes beyond the tokenization function they are interested in. In a few months, they will need to add Arabic, which Team A doesn't provide, and then they may have to get that tokenizer from yet another team. Thus, they would like to be able to declare only that their component consumes tokens. The fact that team A's component produces much more than just tokens should not matter. Furthermore, they should be able to keep using their own English tokenizer, and later integrate an Arabic tokenizer, with few or no code changes.

This kind of scenario requires that UIMA be able to handle different tokenizers whose data models are compatible, but not necessarily identical. Replacing one tokenizer by a compatible one should not require coding changes or even recompilation (this might be done even at deployment time, when no source code is involved). To enable this, the data model must not be hard-coded in component implementations. Rather, the data model must itself be implemented as data that UIMA can inspect and has control over. When an instance of an analysis engine is created at runtime, the framework checks that all data models are compatible and provides the consumer of data with the appropriate view, without the consumer having to "know" the exact internal structure of the data.

Example: Add part-of-speech tagging to an existing tokenizer. Team A has written the ultimate tokenizer for some language, and Team B would like to extend the tokenizer data model to include part-of-speech information. Because Team B does not have access to the tokenizer source code, they write their own component that adds part-of-speech information to the tokens they consume, and thus declaratively extend the tokenizer data model. As before, this is possible only if the tokenizer implementation does not fix its data model, but uses a mechanism that allows downstream components to extend its data model without the need to change the code.

These examples show the kind of flexibility required of UIMA as a component architecture, which makes the introduction of a data abstraction layer very useful. The data engine that drives such a layer must be powerful enough to merge compatible data models, while still providing individual components with customized views of their data.

Data-driven inference. UIMA may be viewed as a data-driven architecture in which the components, that is, analysis engines, are consumers and producers of data. An analysis engine specifies what kind of data it needs to work properly, and also, what data it produces as the result of its analysis. This information is described declaratively in the XML specifier of each component. The UIMA Analysis Structure Broker (ASB) uses this information to verify aggregate text-analysis-engine (TAE) specifiers and to dynamically decide which TAEs to call, and which not. In the future, the ASB will automatically assemble TAEs based on output requirements and known component specifications.

To be able to perform this kind of inference, we must have simple and well-defined semantics for the data specification. General-programming-language constructs are too powerful and unconstrained for this purpose, even if we are willing to ignore platform issues (this is particularly true if one wants CAS to be programming-language and operating-system independent and, even more important, to be able to interoperate between UIMA implementations in different programming languages and operating systems). Our data-specification language must be flexible enough to allow component providers to express their data models, and constrained enough to allow the ASB and other UIMA subsystems to draw meaningful inferences from the data specifications.

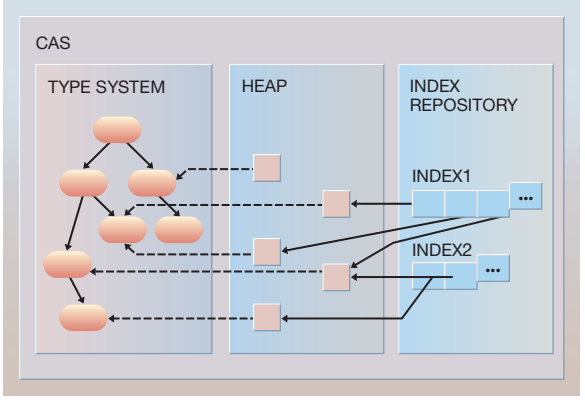
This architecture makes it possible to assemble existing TAEs in a declarative manner without the need for their source code. In fact, the UIMA defines an "analysis engine assembler" as a person who assembles existing TAEs and thus creates new applications without writing any code. Also, for TAE developers who want to reuse a component written by others, the integration is performed in a declarative way—no recompiling or other programming-language-specific tasks of any sort are necessary.

Example: Tokenization and part-of-speech tagging for multiple languages. Team A implements an application that requires tokenization and part-of-speech tagging for multiple languages. Because part-of-speech taggers for some languages are hard to find, they use a grab bag of components from various sources, which includes a tokenizer for Italian and Spanish, a combined tokenizer and part-of-speech tagger for French, and a part-of-speech tagger that can handle Italian, Spanish, and French. An Italian document is to be processed. The system determines that it needs to call the tokenizer for Italian, followed by the part-of-speech tagger. Next, a French document is analyzed. The engine calls the combined tokenizer and part-of-speech tagger for French, and automatically figures out from the output specification of that component that it need not call the part-of-speech tagger separately, because the part-of-speech information has already been added.

Although the example is not very complex, it illustrates the kind of data-driven inference that can be done automatically by the engine based on the data that is being exchanged by the components.

Data exchange. The UIMA components do not work in isolation, but as producers of data for downstream components and as consumers of data from upstream components. The location of the downstream component may be as close to the producer of data as the same machine, or as far as a machine halfway around the globe. The producer of data does not know where the data are going to, and the consumer does not know where the data are coming from.

Figure 1 A conceptual view of CAS



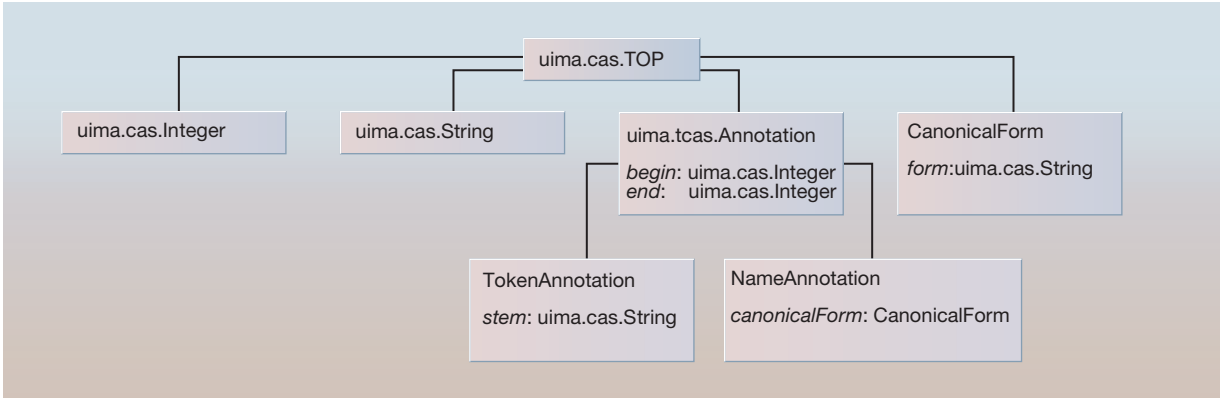
To address this requirement, UIMA implementations must have complete knowledge and control over the data that are passed between components (data used internally by a component are not included). Thus, data passed to other components must be "understood" by the UIMA subsystem that performs the data transfer, whether they are parameters to a function call, serialized objects to be transported to a remote machine, or something in between.

Main concepts

CAS entities fall into three major groups. First, we have a type system that has a dual role—it is responsible for creating the data model from the TAE specifiers at start-up, and it provides information about the data model at runtime. The second group is concerned with creating and supporting feature structures, that is, data according to the type system. Finally, a group of APIs specifies indexing information. Data can be indexed in various ways; indexing determines how data may be accessed.

Figure 1 shows a conceptual view of the system. The type system is shown as a tree-shaped type hierarchy, the heap is shown as a container where all feature structures reside (these are represented by small squares), and the index repository is shown as a con-

Figure 2 An example type hierarchy



tainer for indexes. The indexes contain references to feature structures on the heap. Note that the relationships between the objects in different groups are not one-to-one. In particular, there can be more than one feature structure of one type, and one feature structure can be associated with different indexes (more details are provided later). We now examine these different concepts in more detail and conclude the section with a brief code example that illustrates the APIs.

Type system and typed feature structures. UIMA requires that CAS data structures accommodate all kinds of information. In particular, data structures should support development of applications beyond text analysis. Thus, the concept of text annotation could not play the role of a building block. The CAS data structures should at the same time be constrained enough so that the framework can draw meaningful inferences. For example, a TAE (more precisely its ASB) should be able to determine which annotators should be called when processing a CAS with a specified resulting data structure (an annotator is a UIMA analysis component; see Reference 1).

This led to the decision to choose *typed feature structures* as the CAS data structures. These are well-known and understood data structures, widely used in natural language processing and artificial intelligence.[13] One can simply think of them as attribute-value structures with an object-oriented-like type system (without multiple inheritance). This type system defines the inheritance relationship between types and introduces so-called "features" (attributes) of types that are inherited to all subtypes. These features also have a "range type" that indicates which type the value of this attribute must have. The range information is checked at runtime when feature values are set through the API. (We make use of the "type" and "feature" concepts here for historic reasons.) The most general type is called uima.cas.TOP, and it serves as a root of the type tree.

The type system supports a simple name-space concept, similar to Java packages. The uima.cas and uima.tcas namespaces are reserved for built-in types. As mentioned earlier, there are only a small number of built-in types: basic types, arrays, lists, and annotations.

The basic types are integers, floats (floating-point numbers), and (Unicode) strings. Arrays and lists of all the basic types, as well as arrays of feature structures, are supported. Finally, types are defined to support text processing, namely annotations (spans of text) and documents. The document annotation contains information about the language in which a document is written, if known.

There is no user-level API to create or modify a type system. UIMA developers create type system information exclusively through XML specifications. There is, however, a user-level API to query an existing type system. Through this API, one can get information on existing types and their features.

Consider the example type system in Figure 2 (all types prefixed with uima are actually part of the predefined type system). This hierarchy states that there is a type uima.tcas.Annotation, which inherits from uima.cas.TOP and has two features called *begin* and

end, which take integers as values. There are also types TokenAnnotation and NameAnnotation, which are subtypes of uima.tcas.Annotation, and thus have the *begin* and *end* features. In addition, TokenAnnotation has a feature *stem* whose value is a string (indicating the stem of the token, e.g., "house" for "houses"); whereas, NameAnnotation has a feature *canonicalForm* whose value is of type *CanonicalForm*. Although we interpret the *begin* and *end* features as "begin" and "end" positions in a text, this is not an assumption built into CAS, but simply an interpretation. We also note that annotations are just a special form of feature structure and have no special status.

We can view the type system as similar to a class hierarchy in Java, with each type resembling a Java class with no methods and only public data members. These members are then inherited to subclasses. The uima.cas.TOP type is the equivalent of the java.lang.Object class. However, it is important to remember that these objects are pure data and thus are not proper objects in the object-oriented sense. We use the notions subtype and supertype to refer to types higher up or lower down in the type tree, respectively.

A feature structure then is an instance of a given type. This object can have values for all the features defined on it, that is, features inherited from supertypes or features introduced at the type itself. Using again the Java analogy, we think of a feature structure as analogous to an object of a given class. This object has all the members it inherits plus those defined in the definition of its class.

We consider the following example of a feature structure that is licensed according to the type system (i.e., well-formed with respect to the type system) in the example above. The notation we use here is called attribute-value-matrix (a notation also used in Reference 13). Every structure within brackets is a feature structure.

```
[ NameAnnotation
    begin: 37
    end: 40
    canonicalForm: [ CanonicalForm
                       form: "IBM Corp." ]
]
```

This example denotes a feature structure of type NameAnnotation with values 37 and 40 for features *begin* and *end,* where the value of feature *canonicalForm* is, in turn, another feature structure of type CanonicalForm. This embedded feature structure has the string value "IBM Corp." for its *form* feature, as required by the type system.

We use a restricted version of typed feature structures here. In Reference 13, for example, the type system also allows multiple inheritance and the overwriting of the range type of a feature.

These feature structures fulfill the requirements mentioned earlier, as follows:

- The feature structures are broad enough to accommodate almost all kinds of information. The added type system gives the user the ability to group objects by the kind of information these structures represent.
- The type system makes it possible to specify what kind of feature structure an annotator consumes and produces so that the TAE can determine which annotators must be called during processing.
- The type system is extensible by the user through a declarative XML syntax. Because no specific type system is provided by CAS (except the built-in types), users have complete freedom to develop a type system of their own.

As shown next, these feature structures can be implemented in a very efficient way.

Sharing analysis results through indexes. All annotators and UIMA applications use CAS feature structures to store or read information they are interested in and nothing else. In other words, all data that a UIMA component uses are modeled as feature structures. But, for example, how does an annotator access feature structures created by an upstream annotator? For this purpose, CAS provides indexes and an index repository. An index is a specialized container for (references to) feature structures of a certain type. Access to previously created feature structures is only possible through the index repository. Note that there is no one-to-one correspondence between annotators or TAEs and indexes. An annotator or TAE can define an index itself (in its specifier) or can use indexes defined elsewhere (in the specifier of another TAE, for example). Also note that indexes always contain only references to feature structures; an index never "owns" a feature structure.

To share their analysis results with the outside world annotators must (1) create feature structures that contain all the relevant information, and (2) add these feature structures to the index repository. The index repository adds every new feature structure to all the appropriate indexes. We note that a feature structure F can be accessible to other annotators without being in an index itself, provided another feature structure that is contained in an index has a feature which points to F.

An index declaration can be inserted into the TAE specifier and comprises all of the following:

1. A name
2. A type (*T*): the most general type for which the index is intended
3. A comparison criterion: a list of features defined for *T* and the way the features should be compared
4. The kind of index: sorted, set, or bag (The comparison criterion is used only for the first two, sorted or set; in bag indexes features are inserted in the order in which they were received.)

For instance, a straightforward annotation index would have the following declaration:

1. Name: AnnotationIndex
2. Type: uima.tcas.Annotation
3. Comparison criterion:
   a. Feature: *begin,* comparison: standard
   b. Feature: *end,* comparison: reverse
4. Kind: sorted

We interpret this definition as follows. The index called AnnotationIndex contains only feature structures of type uima.tcas.Annotation. The index is sorted. For two feature structures F1 and F2, F1 is considered less then F2 when its begin value is smaller (standard!) than that of F2; if the two begin values are equal, F1 is less then F2 when its end value is larger (reverse!) than that of F2. If the two begin values are equal and the two end values are equal, then F1 and F2 are considered equal for the purposes of this index. (Our UIMA implementation includes this annotation index as defined here.)

There are three different kinds of indexes: sorted, set, and bag indexes. Because feature structures are inserted in the order in which they are received, bag indexes should only be used when random access to data is not required and the order of the data does not matter.

In the more commonly used sorted and set indexes, the sort order is determined through the use of the comparison criteria. In set indexes a check for duplicates is performed when new feature structures are inserted. If the set index already contains a feature structure that, according to the comparator, is equal to the new feature structure, the new feature structure will not be added (even if the feature structures have different values for non-key features). In sorted indexes there is no check for duplicates. If there is a second request to add the same data item to the index, then a duplicate entry is entered in the index.

To retrieve feature structures, one must specify the index (using its name) and the type of these feature structures. This type must be either a subtype of the type specified when the index was created (the index type), or the index type itself. The API then provides iterators over the index. For example, you can create an iterator over all feature structures in the index named AnnotationIndex of type TokenAnnotation. We note that this is possible because we assume that TokenAnnotation is a subtype of uima.tcas.Annotation. The order in which the feature structures are returned by the iterator corresponds to the comparison criterion. For the built-in annotation index, an annotation is returned ahead of all those that start to the right of it and also ahead of those that start at the same position but span a shorter area of text. For example, we have the sentence, "UIMA rules." It consists of three tokens ("UIMA", "rules", and "."). The iterator first returns the complete sentence, then the three tokens from left to right.

Now, we consider two annotators: one which tokenizes text, and one which performs named-entity (NE) recognition. The tokenizer, which is called first, creates feature structures of type TokenAnnotation. All those feature structures are also added to the index repository. Next, how does the NE recognizer read all those tokens? The answer is that it creates an iterator over all TokenAnnotation feature structures of the built-in annotation index via the API. In addition to the fact that this iterator only returns feature structures of type TokenAnnotation, the defined order assures that the tokens are returned in a way that the NE recognizer interprets as "from left to right." The recognizer iterates over all tokens and determines whether a token or a number of consecutive tokens form a name such as "IBM" or "International Business Machines." If so, it creates a new feature structure of type NameAnnotation with the appropriate begin and end values and adds this feature struc-

Figure 3 Code example

```
01  // get cas from somewhere
02  CAS cas = …;
03  // type of feature structures we want to retrieve
04  Type retrieveType = cas.getTypeSystem().getType("MyType");
05  // feature1 is defined on MyType
06  Feature feature1 = type.getFeature("myFeature");
07  // type of feature structures we want to create
08  Type creationType = cas.getTypeSystem().getType("MyOtherType");
09  // feature2 is defined on MyOtherType
10  Feature feature2 = creationType.getFeature("myOtherFeature");
11  // label of the index
12  String indexLabel = "MyIndex";
13  // get desired index
14  FSIndex index = cas.getIndexRepository().getIndex(indexLabel, retrieveType);
15  // create iterator over index
16  FSIterator it = index.iterator();
17  int i=0;
18  // for each feature structure fs in the index
19  for (it.moveToFirst(); it.isValid(); it.moveToNext()) {
20      FeatureStructure fs = it.get();
21      i++;
22      // analyze fs (here: check if the value of myFeature is 42)
23      if (fs.getIntValue(feature1) == 42) {
24          // create new feature structure
25          FeatureStructure newFS = cas.createFS(creationType);
26          // modify newFS
27          //   (here: set value of myOtherFeature to i)
28          newFS.setIntValue(feature2, i);
29          // add newFS to the index repository
30          cas.getIndexRepository().add(newFS);
31      }
32  }
```

ture to the index repository (so that subsequent annotators have access to these names).

We point out that yet another kind of index might be needed; for instance, one similar to a set index could ensure that feature structures of type CanonicalForm occur at most once with the same *form* feature. In this case, when "IBM" and "International Business Machines" occur in the same text, they both refer to the same canonical form feature structure.

Apart from iterating over feature structures, indexes can also be used to search for feature structures using the comparison criterion of the index. For example, we could query the mentioned set index over canonical forms whether a CanonicalForm with "IBM" as *form* is found there; if so, we reuse this structure, otherwise we create it.

Code example. Figure 3 shows what the typical process method of an annotator may look like (although this code is Java, the C++ API looks exactly the same apart from the fact that it uses the UnicodeString class of ICU (International Components for Unicode)—a C++ library for Unicode support—rather than java.lang.String).[14]

Lines 3 to 10 show how the type system is used, in particular how access to types and features works. In line 14, an index with a specified label, containing only feature structures of the specified type, is retrieved from the index repository, and an iterator over this index is created (line 16). In the for-loop between lines 19 and 32, the current feature structure to which the iterator points is retrieved (line 20), and if its value of feature1 (an integer) is equal to 42 (line 23), a new feature structure is created with a new type (line 25), the value of the feature feature2 is set to *i* (we currently examine the *i*th feature structure in the index we iterate over), and this newly created feature structure is added to the index repository (line 30).

The sidebar shows the same example coded in JCas.

CAS vs. TCAS. Because many UIMA users work in the field of text analysis, there exists a CAS implementation specialized for text processing, named TCAS. As the namespace uima.tcas indicates, it introduces several types, features, and indexes useful for processing text, including some convenient API functions. We note, however, that it simply provides a convenience layer and does not add any basic functionality. Thus, using just the CAS APIs of TCAS is sufficient for building any text analysis task. The only TCAS function not contained in CAS allows document text to be set as part of the TCAS, so that a TCAS is self-contained in the sense that the begin and end values of annotations are interpreted with respect to this very document text.

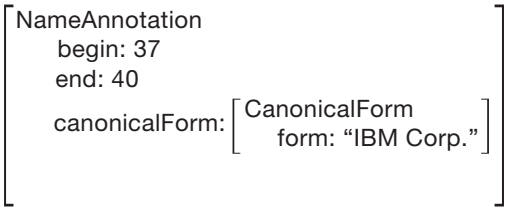
Implementation

We show in this section how the previously described concepts can be implemented efficiently. There are two reference implementations of UIMA TAEs, namely JEDII in Java and TAF in C++. This section applies to both. In fact, one of our design objectives is to minimize the use of programming-language-specific features.

Type system. Types and features are created during the initialization of a TAE, and thus the type system remains fixed in normal operation. Applications and annotators can always cache any types or features at initialization, and, therefore, the performance of the corresponding API calls is not an issue here. It is easy to represent types and features as integers. Otherwise, the type system implementation is straightforward; that is, it consists simply of a tree of types, in which each node in the tree has information as to which features are defined on this type.

Feature structures. Our implementation of feature structures was strongly influenced by the Warren Abstract Machine (WAM),[15] the standard implementation of Prolog. The restrictions we put on the type system allow for very efficient representation of feature structures (see Reference 16 for an account of how feature structures with less restrictive type hierarchies can be efficiently implemented). Feature structures can be represented as an array of integers (as mentioned before, we assume that our types and feature identifiers are represented as integers). Following the WAM conventions, we call this array the *heap*.

We consider again the same example of feature structure.



This feature structure (plus the embedded one) looks as follows on the heap:

Index	Value
1	NameAnnotation
2	37
3	40
4	5
5	CanonicalForm
6	1

The name of the types NameAnnotation and CanonicalForm are actually integer identifiers and are written here as strings only for expository purposes. The NameAnnotation feature structure starts at cell number 1, and the CanonicalForm feature structure starts at cell number 5. Cell number 2 is the value of the begin feature, cell number 3 is the value of the end feature, and cell number 4 is the value of the canonicalForm feature. For the integer-valued features, the values are stored directly on the heap, whereas the feature with a “real” reference points to the heap cell with the actual feature structure value (here cell number 4). The value points to cell number 5, which is the start of a feature structure of type CanonicalForm (as cell 5 indicates). The value of cell 6 represents the value of the form feature that we interpret as an entry into a separate symbol table. In this example, we assume that the first entry of this table is the string “IBM Corp.”

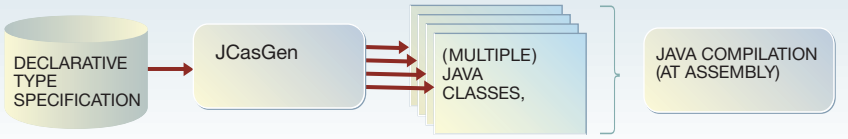
We note that no feature names appear anywhere in this example. Thus, in order for this to work the type system, in addition to mapping types and features to integer identifiers, must also compute the offsets for all features. In the example, the begin feature has offset 1, the end feature has offset 2, and canonicalForm has offset 3.

A feature structure with n features consists of $n+1$ heap cells (1 for the type and n for the feature val-

AN EFFECTIVE, JAVA-FRIENDLY INTERFACE TO CAS

Java programmers who began using the Common Analysis System (CAS) saw a natural mapping from the CAS type system to Java classes. Some of them implemented their own mappings. We recognized we could use the declarative specification of the CAS type system to generate a Java class model that closely follows the CAS types. In this model, named JCas, CAS types have corresponding Java classes of the same name; "get" and "set" methods on these types are generated for all the CAS features. For instance, a feature structure (see the section "Type system and typed feature structures") with the name **com.ibm.demo.Token** would create a corresponding Java class **Token** in the Java package **com.ibm.demo**. If **Token** had features **begin** and **end** defined as integers, these would appear in the Java class definition as **getBegin()** and **setBegin(int value)** method definitions.

In the figure, a utility, **JCasGen**, takes the declarative specification for a collected set of components, some of which may be extending type specifications of other components (see the section "Component assembly at runtime"), and generates the proper Java class definitions. When compiled, these class definitions provide at runtime a high-performance type-checked interface to the underlying CAS data. This approach allows moving some of the runtime type checking to compile time, using the strong-typing features of Java.



Java programs created with JCas hide some of the complexity of using CAS. Using the JCas approach, the code in the section "Code example" becomes the following:

```
 1  // get jcas from somewhere
 2  JCas jcas = …;
 3  // no need to access the type system or
 4  // to declare and initialize variables to hold Type or Feature values
 5  // label of the index
 6  String indexLabel = "myIndex";
 7  // We use Java classes instead of strings to reference types
 8  // Supports compile-time checking of misspelled type names
 9  // get desired index
10  FSIndex index = jcas.getJFSIndexRepository() .getIndex(indexLabel,MyType.type);
11  // create iterator over index
12  FSIterator it = index.iterator();
13  int i=0;
14  // for each feature structure fs in the index
15  for (it.moveToFirst(); it.isValid(); it.moveToNext()) {
16     MyType fs = (MyType)it.get();
17     i++;
18     // analyze fs (here: check if the value of myFeature is 42)
19     if (fs.getMyFeature() == 42) {
20           // create new feature structure
21           MyOtherType newFS = new MyOtherType();
22           // modify newFS
23           //       (here: set value of myOtherFeature to i)
24           newFS.setMyOtherFeature(i);
25           // add newFS to the index repository
26           newFS.addToIndexes();
27     }
28  }
```

In line 16, Java variables holding references to CAS feature structures are typed with the specific feature structure (or a supertype of that feature structure). In line 19, the call to get the value of a feature named "**MyFeature**" is the strongly typed **getMyFeature** method, generated by the convention of appending the feature name following the word **get**. This generated method is strongly typed to return an integer in this case, and the type of the feature in the general case. Likewise, in line 24 the **setMyOtherFeature** method is strongly typed to accept arguments that are appropriate for that type. Line 21 shows a new object created in CAS via the familiar Java **new** operator. Line 26 shows the operation to add an instance of a type to the CAS indexes by using the **addToIndexes**() method.

Marshall Schor
IBM Research Division, Yorktown Heights, N.Y

Reference

M. Schor, *An Effective, Java-Friendly Interface for the Unstructured Management Architecture (UIMA) Common Analysis* System, IBM RC23176, IBM T.J.Watson Research Center, Yorktown Heights, N.Y. (2004).

ues). Because, following initialization, all types and features in a TAE are fixed, the following holds:

- The size of a feature structure of a certain type (in heap cells) is fixed before the first feature structure is ever created (and, by the way, can thus be stored in the type system). This ensures that the heap is "dense" in the sense that there is no wasted space.
- The offset for a feature is the same for the type at which it was introduced, and all its subtypes. This means that a feature structure of type *t* can just be treated as a feature structure of a supertype of *t* (in matters concerning the features of this supertype).

Every feature structure is, in effect, an integer that is an index into the heap array. As claimed earlier, we can now easily see why feature structure operations are fast.

- Creating a feature structure involves mainly incrementing the top-of-heap counter by a number depending on the cell type (which need not be computed at runtime).
- Setting or getting feature values involves just looking up the offset of the feature (which is computed at initialization) and a simple array access.
- The size of the feature structure is minimal in the sense that there is no padding.

All operations on a feature structure, such as creation and getting or setting feature values, are very fast, independent of the usage scenario (this is different in the index repository, as we show later).

Index repository. The index repository supports two main operations: (1) adding a feature structure, and (2) retrieving feature structures of a specified index via an iterator. Adding a feature structure means the feature structure is added to all indexes that are defined on the type of the feature structure or on one of its supertypes. When retrieving feature structures of a specified index via an iterator, one must also specify a type; all feature structures returned by the iterator should be of this type or one of its subtypes.

An implementational question arises directly from the second function. Should an index be implemented as a single (physical) container or multiple containers, say one for each type? The exact nature of this container depends on the programming language; for C++ this may be an STL (Standard Template Library) vector or set,[17] for Java an instance of java.util.Vector or java.util.HashSet. A single container allows fast access when iterating over all feature structures that are contained in an index, and there are no constraints on the types (e.g., we traverse all uima.tcas.Annotation feature structures in the annotation index), but access is slower when retrieving only feature structures of a subtype (e.g., TokenAnnotation) because we have to filter out the feature structures of a type we do not want. The multiple-containers approach behaves somewhat inversely, in that when all feature structures of an index are retrieved, the access is slow because all containers for all subtypes for this index must be merged, but the access is very fast on "leaf" types (i.e., types without subtypes) because the iterator required is simply an iterator over a physical container.

Which implementation should we choose? What usage scenarios require the level of performance associated with each of these alternatives? The implementation should be very fast in performance-critical scenarios (such as search engines) and reasonably fast in more complex analysis scenarios. Apart from the manipulation of feature structures, for the design of the index repository one has to decide the application scenario for which the data structure is optimized. In a scenario involving tokenization for a search engine, the main operations are adding (and retrieving) same-type feature structures to (and from) a specified index. In particular, this type may be TokenAnnotation, and the index is the default annotation index. Because it is reasonable to assume that TokenAnnotation is a leaf type, we chose the second of the two approaches above.

Let us consider an index with name *I* is defined on type *T*. For every subtype *T′* of *T* (including *T* itself) there exists a physical container that represents all feature structures contained in *I* of exactly type *T′* (excluding subtypes). So if we assume that uima.tcas.Annotation has a subtype NameAnnotation, which in turn has a subtype PersonNameAnnotation, the annotation index is implemented as three physical containers, say, vectors: one that contains only feature structures of type uima.tcas.Annotation (which is neither NameAnnotation nor PersonNameAnnotation—rather unlikely in common usage scenarios), one that contains only NameAnnotation (and no PersonNameAnnotation), and one that contains only PersonNameAnnotation.

When a feature structure of some type *T″* is added to *I*, the feature structure is inserted into the physical container of *T″*, and nowhere else. In our ex-

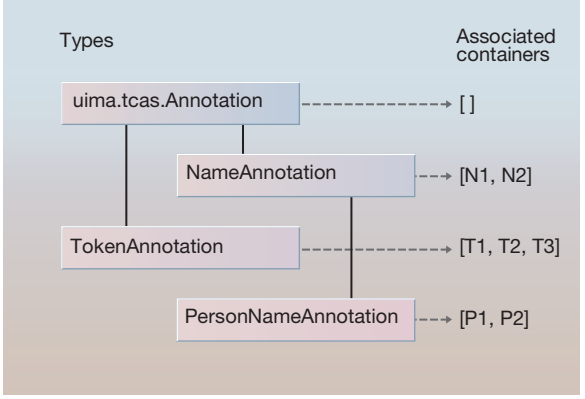
ample, a PersonNameAnnotation is added only to its container, not to those for NameAnnotation or even uima.tcas.Annotation; therefore, adding a feature structure is fast as required by the search engine scenario. In fact, it is fast in every scenario because no matter how complex the type system is, the feature structure is only added to a single physical container.

When we iterate over all feature structures of type T'' of I, the index repository must merge all physical containers of all subtypes of T'' (including T'') itself. If the index is sorted (as our annotation index is), we have to perform what amounts to a merge-sort on all these containers before or while iterating. If, however, T'' has no subtypes, the iterator becomes, in effect, an iterator over the native container. In our example, iterating over NameAnnotation requires merging the containers for NameAnnotation and PersonNameAnnotation, whereas iterating only over PersonNameAnnotation reduces to using the iterator over its physical container. Thus, iterating over feature structures of non-leaf types performs worse than iterating when leaf types are involved, but the latter is, in turn, very fast.

The attentive reader might have noticed that we have not yet mentioned how the various index kinds (sorted, set, bag) are implemented. This depends on the kind of physical container used. Thus, for a sorted index the physical container could be a data structure such as a red-black tree, and for bag indexes, the physical container could simply be a vector. (The bag index implementation works also for sets, with the minor change that no duplicate feature structures with the same type are allowed.) We have previously assumed that inserting into, and iterating over, such physical containers is fast. Whereas iterating is fast, regardless of whether we use vectors, lists, or sets, inserting may indeed depend on the kind of the index. (Performance when an element is added to a vector is different from the same operation on a set.)

Figure 4 shows an example of an index repository in which the physical containers for the annotation index contain a number of feature structures denoted by offsets into the heap (note that these offsets have nothing to do with begin or end positions). We view these containers as nodes in a subtree of the type hierarchy rooted at uima.tcas.Annotation. Thus, there are no feature structures of type uima.tcas.Annotation, there are two of NameAnnotation, two of PersonNameAnnotation, and three of TokenAnnotation in this index. If we wanted to add a new PersonNameAnnotation (P3) to this index, we could add it to the appropriate container, which would then contain [P1, P2, P3].

Figure 4 An example index repository



For iterating over all NameAnnotation subtypes, the two containers [*N1, N2*] and [*P1, P2*] should be merged according to the begin and end positions of those feature structures. Note that they all have those features because they are subtypes of uima.tcas.Annotation. Iterating over all TokenAnnotation subtypes, however, does not require any merging; we can simply return the feature structures *T1, T2,* and *T3,* in that very order—adding a feature structure already inserts it correctly with respect to the physical container of its type. In other words, the tokens are already sorted from left to right. Of course, all those implementation details are completely hidden behind the API and are not visible to the user.

Serialization. The CAS heap layout allows not only feature structures within one TAE to be represented efficiently, but also CAS entities to be serialized and deserialized efficiently. Note that CAS serialization does not mean Java built-in serialization but a special way of "flattening" complex data structures of the CAS into data structures which are simple, easy to transport over a network, and easy to use for interoperating between different programming language (typically arrays of integers and the like). In particular, it is enough to use an integer array (the heap as such) and a string array (a table for string-valued features). Because CASs usually contain many data objects, the ability to serialize and deserialize a CAS without examining and rebuilding each feature structure it contains is crucial. There are a number of scenarios where this might be necessary:

1. Deploying TAEs in a distributed environment (currently only supported by JEDII)
2. Crossing a language-framework boundary (JEDII to TAF or vice versa)
3. Saving data to disk

Note that although it would be possible to just use the Java built-in serialization, our CAS serialization mechanism is also used in scenario 1 for performance reasons. Scenario 2 means, in effect, that you can use a TAF annotator in JEDII or a JEDII annotator in TAF by serializing the CAS in one language (e.g., C++/TAF), crossing the language boundary only once (passing the serialized CAS to Java), and deserializing the CAS in the other language (Java/JEDII). This is more efficient than having a Java "proxy" object to a C++ CAS which has native implementations for all CAS functionality so that basically all calls to this Java proxy cross the language boundary. Moreover, a Java CAS which is the result of processing a TAF TAE is a pure Java object (similarly for a TAF CAS the other way round) and thus completely lives in the environment where it is used and obeys its rules.

From a user perspective, all this happens "under the covers," that is, only the framework implementation (namely, TAF or JEDII) need be indicated in the descriptor of the TAE, and the framework used creates an instance of a TAF/JEDII TAE automatically.

Conclusion

We have demonstrated that for a data analysis component architecture such as UIMA, it is advantageous to implement a data model and a data container that enables the framework engine to (1) monitor the data produced and consumed by various components and use this knowledge to (among other things) instrument the work flow, and (2) control the data that is being created as a result of analysis work (so the data can, for example, be serialized for storage or network transport).

Having a separate data layer puts an additional burden on users of the architecture because they need to understand its concepts in order to use its APIs. On the other hand, this allows many services to be provided by the framework, such as serialization for storage, that would otherwise have to be implemented by each application. Our data layer does not overly constrain developers in their choice of data structures because typed feature structures are general enough to model most data. Moreover, we showed that an implementation of such a data layer need not be slow. Using well-known techniques from compiler construction, data containers can be designed that allow for fast data creation and access.

**Trademark or registered trademark of Object Management Group, Inc. or Sun Microsystems, Inc.

Cited references and note

1. D. Ferrucci and A. Lally, "UIMA: An Architectural Approach to Unstructured Information Processing in the Corporate Research Environment," *Journal of Natural Language Engineering* (to appear).
2. D. Ferrucci and A. Lally, "UIMA by Example," *IBM Systems Journal* **43,** No. 3, 455-475 (this issue, 2004).
3. The acronym CAS in Reference 1 has a different meaning than the one used in this paper - it stands for Common Analysis Structure.
4. Object Management Group, http://www.omg.org/.
5. *Java 2 Platform, Enterprise Edition (J2EE),* Sun Microsystems, Inc., http://java.sun.com/j2ee/.
6. E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns*, Addison-Wesley Publishing, Reading, MA (1995).
7. T. Hampp, "Beyond Text Representation," *Proceedings of the 18th International Unicode Conference*, Unicode Consortium, Hong Kong (2001).
8. H. Cunningham, *Software Architecture for Language Engineering*, Ph.D. Thesis, University of Sheffield, UK (2000).
9. 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 (LREC)*, Evaluations and Language Resources Distribution Agency, Paris (2002).
10. S. Bird and M. Liberman, *A Formal Framework for Linguistic Annotation*, Technical Report MS-CIS-99-01, Department of Computer and Information Science, University of Pennsylvania, PA (1999).
11. *Extensible Markup Language (XML) 1.0,* World Wide Web Consortium (W3C) (2000), http://www.w3.org/TR/REC-xml.
12. Java Architecture for XML Binding (JAXB), Sun Microsystems, Inc., http://java.sun.com/xml/jaxb/.
13. B. Carpenter, *The Logic of Typed Feature Structures*, Cambridge University Press, New York (1992).
14. M. Davis and S. Loomis, *International Components for Unicode (ICU) Version 2.4.*, (2003), http://oss.software.ibm.com/icu/docs.
15. D. H. D. Warren, *An Abstract Prolog Instruction Set*, Technical Note 309, SRI International, Menlo Park, CA (October 1983).
16. G. Penn, "Generalized Encoding of Description Spaces and its Application to Typed Feature Structures," *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics (ACL 2002)*, Philadelphia, PA (2002).
17. M. H. Austern, *Generic Programming and the STL*, Addison-Wesley Publishing, Reading, MA (1999).

Thilo Götz *IBM Germany, P.O. Box 1380, Boeblingen, Germany (tgoetz@de.ibm.com).* Dr. Götz joined IBM in 1997 at the Watson Research Center in Yorktown Heights, New York, where he worked on text analysis. In 2003 he moved to the IBM Development Lab in Boeblingen, where he is a software engineer in the Data Management department. He received an M.A. degree in linguistics and computer science from the University of Tübingen in 1994, and a Ph.D. degree in computational linguistics from the same university in 2000.

Oliver Suhre *IBM Germany, P.O. Box 1380, Boeblingen, Germany (suhre@de.ibm.com).* Mr. Suhre received his M.Sc. degree in computer science from the University of Tübingen in 1999. Since joining IBM in 2000, he has worked as a software engineer in the areas of text analysis and information integration.