MTR-MNI-000-012

MITRE TECHNICAL REPORT

Implementation Recommendations for MOSAIC: A Workflow Architecture for Analytic Enrichment

Analysis and recommendations for the implementation of a cohesive method for orchestrating analytics in a distributed model

Ransom Winder
Nathan Giles
Joseph Jubinski

July, 2010 (updated, February, 2011)

Contract No.: DAAB07-01-C-C201
Project No.: 0710N7AZ-SF

MOSAIC – Implementation Recommendations
The MITRE Corporation, 2010-2011

Contents

Introduction

This is a companion document to *MOSAIC: A Workflow Architecture for Analytic Enrichment* that describes the current need for integration of document analytics and a general approach to solving this problem. This document directly addresses the implementation issues of the candidate architecture, with specific frameworks for the different architectural subcomponents analyzed and compared. Ultimately, recommendations are offered.

Architectural Goal of MOSAIC

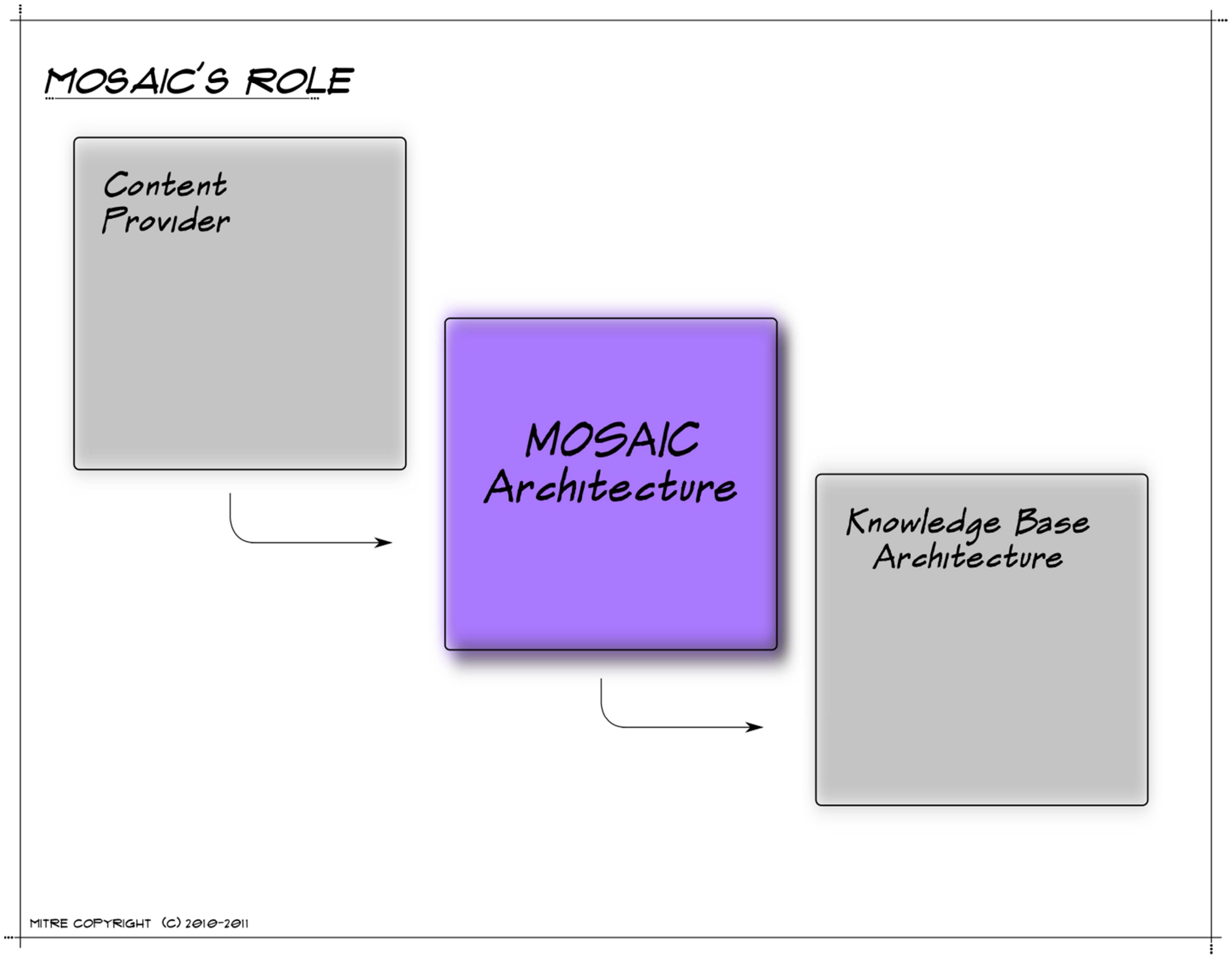

Figure 1. The MOSAIC Architecture's role in a larger system that delivers it input from a Content Provider and consumes its output in a Knowledge Base Architecture.

The goal of this effort is to develop a Natural Language Processing architecture to be used by subject matter experts who are researchers and engineers, termed domain expert engineers here. This architecture, titled MOSAIC, is intended to be shared across multiple projects and hosted in the sponsor's environments and is intended to be compatible with and facilitate a streaming document flow as opposed to execution on a static batch of documents, which would require an entire corpus be present before processing could commence.

In order that the overall goal is addressed, several high level requirements need to be met in order for the system to be considered successful. These requirements are specified here:

1. The system shall maintain a consistent overall structure with evolving components, where the consistent structure is the relationship of the framework built around the analytics, which in turn are the chief evolving components, though the individual products that make up the framework shall be replaceable without deteriorating the interoperability.
2. The system shall at the least be able to accommodate a pipeline of analytics such that they can be run in sequence, potentially using the output of one as input to the other,
3. Ideally the system should be able to handle more complex workflows that make use of splits, joins, and decision points.
4. The analytics involved in the system shall be stateless, specifically in that they make no assumptions about what happens in any workflow prior to their execution or subsequent to their execution.
5. The system shall have the ability to handle streaming documents that arrive asynchronously and in large quantities (generating and executing appropriate workflow instances for each).
6. The system shall have the capability for a “debug” mode allowing a user to specify execution of a workflow instance on a particular document.
7. The system shall leave audit trails for the purposes of provenance, in order that results are verifiable and repeatable.

Figure 1 depicts the larger system in which the MOSAIC architecture can fit. This system can be essentially broken into three key components, from the perspective of the MOSAIC architecture, which takes its input from one and provides its output to the other. MOSAIC’s input arrives from a Content Provider and this input is expected to be a document stream. This is a separate input avenue from a user-specified “debug” mode which accommodates ad hoc input. MOSAIC’s output, which is some subset of relevant document artifacts produced by workflow instances operating on the input, is ingested by a Knowledge Base Architecture.

Architectural Options for MOSAIC

There are multiple approaches available for the design of an architecture that achieves the goal requirements specified above. Here this document puts forth some of these possibilities and makes recommendations on which architecture most appropriately suits different end users’ needs, the end users identified as being the domain expert engineers.

There appear to be two separate environmental needs that can be addressed from a system of integrated analytics such as this, and these are identified as a *production environment* and a *research environment*. The production environment is characterized by a specified activity that needs to be accomplished in a highly efficient manner where artifacts collected throughout processing are insignificant compared to the actual output. The research environment is characterized by an emphasis on flexibility of activity to account for newly developed and evolving workflows, provenance and repeatability for experimental purposes, debugging in order to correct errors in newly developed and evolving analytics, and parameter tuning for the analytics to refine the results of the executing workflows. These two environments are not incompatible. Indeed, it is likely that what is developed, explored, and tuned in a research environment would eventually be migrated into a production environment where it can be used on high volumes of data and require less modification and greater efficiency.

Given these two different environments, it stands to reason that two different approaches to the architecture are likely to suit one or the other of them. It was determined that a production environment would fit best with a tightly integrated architecture using an off-the-shelf technology, such as UIMA (Unstructured Information Management Architecture), which is specifically considered here, as an overarching framework since efficiency concerns are paramount in this environment. It is further recommended that the research environment would be best addressed by a different architecture, one consisting of discrete processes that are less directly integrated but are more flexible when it comes to their addition by researchers as new workflows and analytics are frequently crafted. These two different approaches to the different environments are detailed below, and Table 1 lays out the differences between the approaches, also showing which features best align with the requirements of a research environment or those of a production environment. These alignments with research and production environments for the two examined approaches are equivalent to the advantages contingent on the final use case environment. A lack of alignment between a use case environment and a framework option does not necessarily constitute a disadvantage, but in the instances where this is true, it is indicated in the discussion below.

	Tightly integrated Architecture	Discrete Process Architecture	Required by
Memory managed	**Yes**	No	Production
Tied to a specific architecture	**Yes**	No	Production
Tuned/optimized for targeted tasks	**Yes**	No	Production
Low barrier to entry for domain expert engineer	No	**Yes**	Research
Flexibility across analytics written in multiple languages	No	**Yes**	Research
Ease of integration of new analytics	No	**Yes**	Research
Adaptability to evolving technology	No	**Yes**	Research

Table 1. Examination of features required by either production or research environments and supported by certain architectures.

The first option considered for architectural design makes use of a tightly integrated architecture of analytics. The architectural software tools found in UIMA are ideal for this manner of architecture as they allow for complex Natural Language Processing applications to be decomposed into their incremental individual tasks and provide a framework to manage these components and the flow of data from one to another. Users specify XML descriptor files for the transfer of data between the components, and the tight integration of the components in the framework means that UIMA can execute a workflow across multiple components without having to write to file, instead performing the task entirely within memory.

When looking at the requirements that a production environment imposes, the virtues of this tightly integrated architecture are readily apparent. The emphasis in such an environment is on the efficiency of the overall system, and therefore the ability to perform the task in memory is a benefit as it can forgo the expense of writing to files and the overhead of having to perform the file management of documents and document artifact collections over the life of a workflow instance. The tasks in this environment should be quite specific and therefore a consistent architecture with individual implementations that are each highly tuned and optimized for particular tasks fits well. The memory usage and analytic input/output can be optimized for the specific task in advance.

When considering a research environment though, the lack of alignment with certain requirements reveals certain limitations to a UIMA-based tightly integrated framework. One of these limitations arises naturally from the tight integration of components. UIMA components are written typically in Java. The integration of other languages such as C++, Perl, and Python are supported, but these are suboptimal when considering that the intended language is Java. In a research space though, it is highly likely that legacy components will exist that were composed in other languages while at the same time researchers might want to rapidly create components outside the Java language for convenience on a case by case basis.

In a research environment, a wide variety of analytic components should be available for the purposes of workflows designed to conduct experiments, and this requirement reveals another limitation of using a tightly integrated UIMA architecture, specifically in terms of the efficiency. Because there will be many different workflows that use the components, it can occur that artifacts will be generated for files by different analytics that will not be used further in an executed workflow. The size of document artifact collections can be quite large when compared to the size of the original file, which means that the memory can become strained with excessive information for a document on a particular workflow. If this pipeline of information should be stateless, it is not possible to identify for any analytic what artifacts will be necessary for later points in any given workflow, which means that all the artifacts must be maintained, potentially bloating memory further and further with successive analytics. For analytics that have to be accessed outside UIMA, this efficiency concern extends to the time required for marshalling all analytic output into UIMA's common data format, CAS (Common Analysis System), when there might only be some smaller subset of the artifacts that are necessary for a given workflow. If most of the artifacts were stored in a document management system and the framework were less tightly integrated, then only the information needed for any given workflow would need to be handled by the executive and passed to the analytics. Additionally, it is possible that multiple workflow instances will execute simultaneously and asynchronously in the same system, which creates a further burden for the memory.

There is also a potential risk in the research environment, where flexibility and extensibility are highly valued, in committing to a specific tightly integrated architecture such as UIMA. Because there are many competing and newly emerging alternatives to UIMA (for example, GATE, OpenPipeline), it is possible that UIMA could be abandoned, leaving a final architecture stagnant at that point, although the tight integration of the components means that the commitment level to this specific technology would need to be high.

A more appropriate alternative in a research environment would be to create an architecture that is more loosely coupled where components such as an executive and a data bus are effectively separated from the analytics, which are treated as discrete components, such that they are more easily replaceable as technology evolves.

A discrete process architecture would fulfill the same role as an architecture developed entirely within UIMA, namely executing workflow instances of analytics on incoming documents for the purpose of annotation and information extraction in a systematic and repeatable manner of operation. This architecture is characterized by the more modular nature of its components, the major components being the inbound gateway where documents arrive for processing by workflows, the executive that orchestrates the activity of the executing workflows, the independent analytics, the adapters that allow for communication between analytics, and the data bus where document artifacts are stored between their use in different analytics. Key to this architecture is the separation of the workflow executive from the analytic data, which is not part and parcel of the executive. Instead the executive simply needs to know where to find the data and where to send it. Figure 2 shows a representation of this architecture's layout.

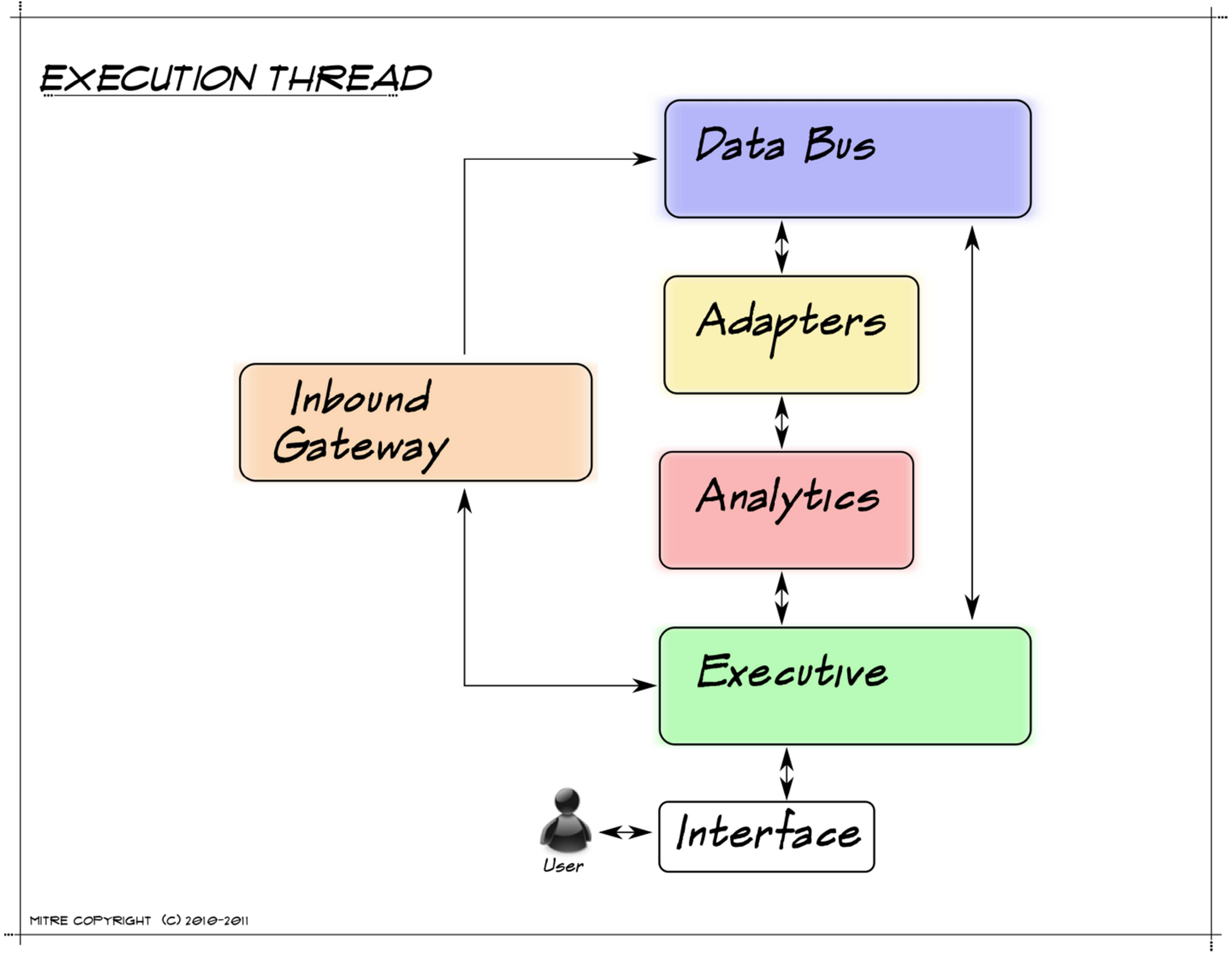

Figure 2. A high-level depiction of the thread of interactions between the services in a discrete process architecture. The user accesses analytics to run workflows on a source document via an interface attached to an executive. Adapters convert analytic output data to a common interchange format (as well as converting from this format back into specific analytic-usable formats) and this data persists over the life of the task in the data bus, which can route the data to other analytics or to the executive.

The advantages of a system like this directly address the issues raised when considering a more tightly integrated system using a product like UIMA as the overarching framework for a research environment. Compatibility issues that arise with non-Java-based analytics are mitigated here as the executive need not impose a particular language on the analytics, each of which can be handled as a discrete process acting independently of other analytics, consuming input and producing output as the analytic was originally specified to do. This makes the integration of current analytics much less onerous and more flexible as they need not be rewritten to accommodate the CAS format and operate on input and output in memory.

This last point indicates another advantage of this system when considered in a research environment. Because the input and output of the system (that is, document and document artifact collections) are stored on the data bus, it is unnecessary to transfer all data from the data bus to any individual analytic that requires it. This means that what data is actively processed can be specified online on an analytic by analytic basis, while the bulk of any document artifact collection can remain in storage until required at some point in the workflow. This allows for a greater efficiency of what enters memory, whereas if an entire document artifact collection was held in memory at all times, a significant amount of unnecessary information would have to be maintained over the life of the workflow. It is also the case that there will be many ongoing workflow instances operating concurrently, so there may be many requests for memory for each individual process that could operate on the same virtual machine.

Finally in a discrete process architecture with more loosely coupled components, adaptability is much easier over the long term, something established as important to a research environment. As parts of the architecture are potentially out-of-date or no longer supported by the community, they can be more easily replaced with components that become popular or new standards, making the architecture capable of a natural evolution as technologies change and new options become available.

There are potential downsides to the discrete process architecture though. The most significant of these is that this manner of architecture is optimized to handle the diversity of analytics that appear in a research environment, and therefore is not as suitable for tools used in a production environment. In a production environment, a full-featured system such as this supports is less applicable than something more targeted to a specific task and therefore optimized in terms of resource costs to that particular task.

Which solution is appropriate really depends on the intended use case. From an initial analysis it appears that a tightly integrated environment (such as would use UIMA) is a good choice for a production environment, bearing in mind certain caveats. These include recomposing current analytics that are incompatible with the architecture and composing the new analytics such that they plug-in to UIMA, consuming and producing CAS, as well as optimizing the CAS's content to make it lightweight enough to maintain the desired efficiency of a memory-based solution. This last point indicates that analytics and workflows written in this framework need to be more targeted to specific tasks. Although we have focused our discussion on UIMA, alternatives in this role are considered below.

On the other hand it appears that a discrete process architecture that makes use of more highly modular components is more suited to a research environment, where the issues of flexibility and generality are more important. This approach requires more investigation and description, and the bulk of this document is devoted to providing the recommendations for the products to be used for the different identified sub-architectures and how they would interact, in particular emphasizing what must be a common interchange model and format that will allow different analytics to adapt their input and output to a *lingua franca* which requires definition in a separate document.

While there has been an emphasis on the differences between research and production environments, it is worthwhile to note that targeted tools for production environments can naturally evolve out of the more general and flexible tools generated in research environments. While a research environment's overall structure is not ideally something that is tightly integrated, the individual analytics of that architecture are excellent candidates for being highly optimized and targeted tools, such as can eventually become parts of the production environment. This is reflected in Figure 3, which demonstrates the potential relationship between frameworks for these two environments and how analytics developed as components in a research environment can be merged and emerge as tightly integrated production tools. A more flexible architecture in a research environment means that the analytics can be matured individually and potentially merged as necessary, creating individual tools that can then be deployed in a production environment where there is less interest in further modification, as this refinement can be carried on in the research environment beforehand and in parallel with a tool in current production use.

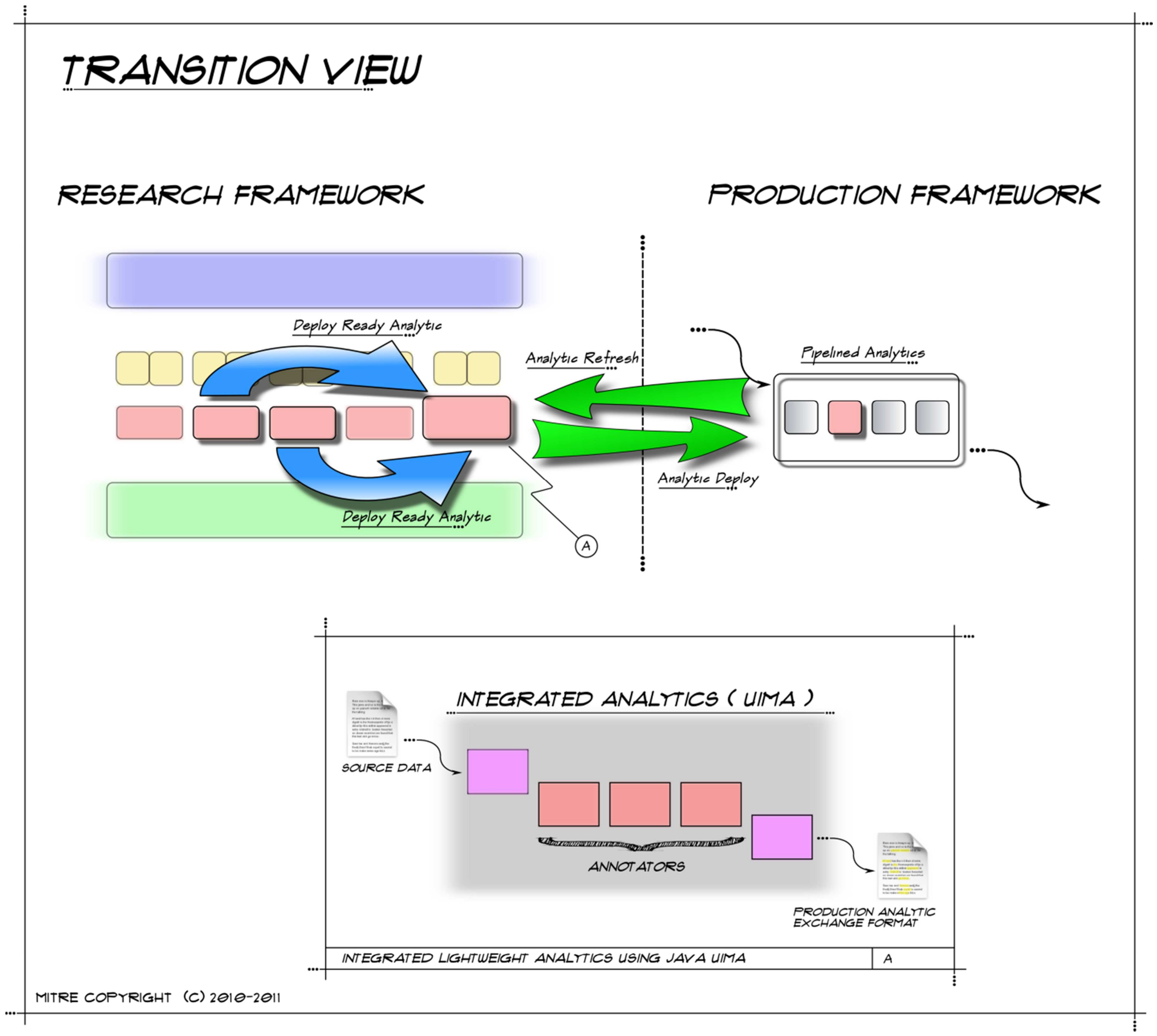

Figure 3. Potential relationship between research and production frameworks, where tightly integrated analytic components developed in the research environment can eventually transition to a production environment, just as existing production analytics can be leveraged by the research framework. UIMA stands as an example of a basis for a given production architecture.

Case Study: METEOR

In the discussion of different architectures for these complex analytic tasks for research, it is helpful to examine an actual workflow used in practice that could become part of either framework proposed in this document. The chosen example is METEOR, a system for capturing and reasoning over meeting and other events appearing in raw text documents. Figure 4 depicts the overall workflow and gives some indication of the complexity involved in carrying out this task with the various required analytic tools.

Breaking down this task, the ultimate goal is to provide reasoning about storyboards of larger events from a set of smaller events directly extracted from raw text as its source. This involves a pre-processing followed by four different phases of analytic work in the workflow. The pre-processing of the source file, such as an email, generates the text file that will be the input to the extractor. This pre-processing parser is a discrete process which the domain expert engineers in this case do not control, other than its execution being a necessary first step in their workflow to ready the documents for consumption by the analytics in their system.

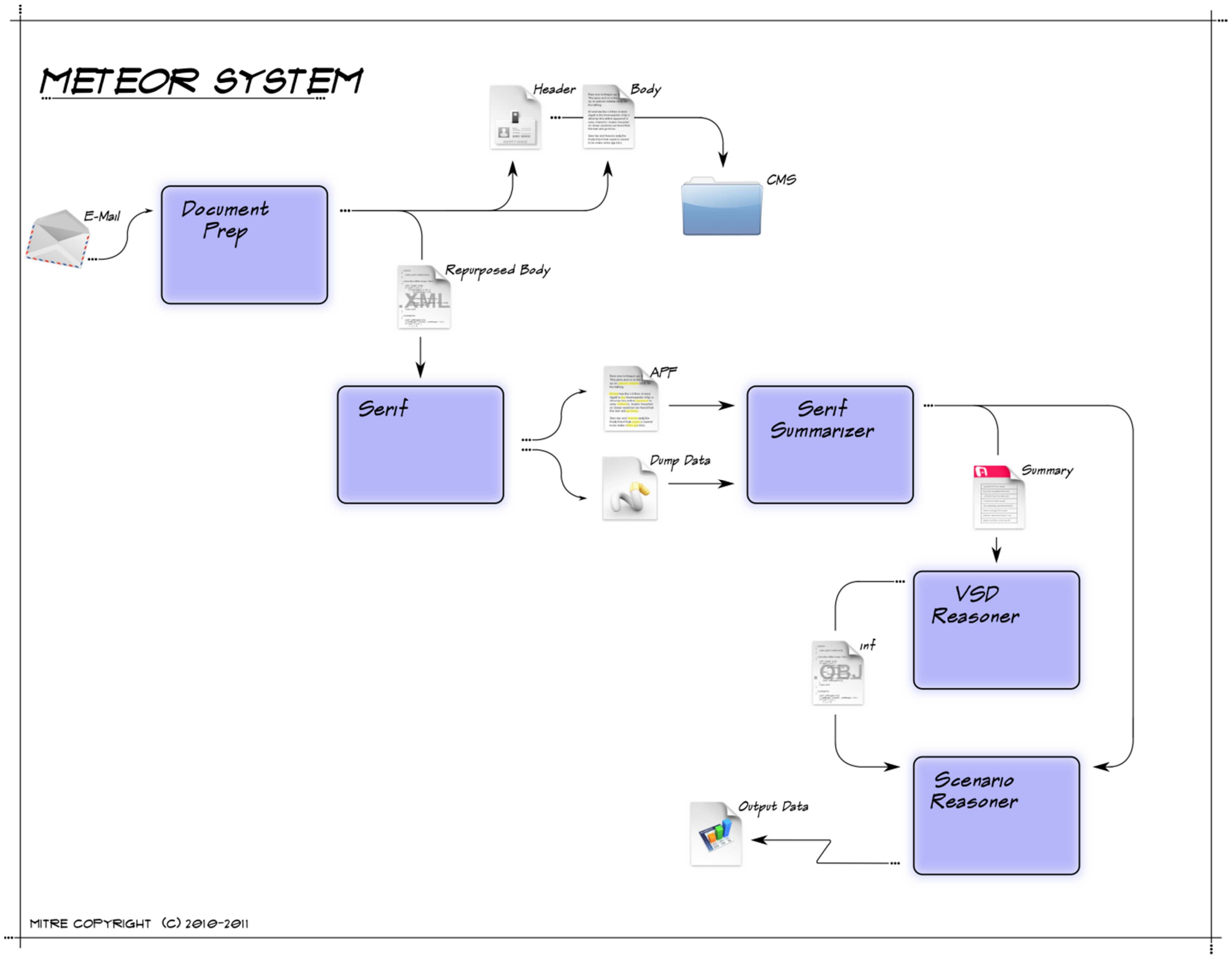

Figure 4. Overview of the workflow in the METEOR system.

The first phase after the pre-processing uses the Serif program for automated information extraction to produce a set of interim output. This output includes the customary Serif dump data that includes the parse tree, annotations, etc. as well as Serif's ACE (Automatic Content Extraction) output represented in APF (ACE Pilot Format), which provides additional information not found in the dump data such as numerical expressions and Timex annotations.

These artifacts, along with a WordNet to SUMO mapping that captures types absent from ACE, are then processed further in the second phase by a program called SerifSum, which aggregates the artifact collection and enriches it in another set of SSP format output, where annotations that capture verb tense, attitude, entity recognition and resolution, and cardinality information are introduced.

The third phase of this workflow involves METEOR's VSD Reasoner, which takes the enriched SSP from SerifSum as input, along with appropriate wordlists and lexicons, and produces INF files that include the interpretations and nominalizations of verbs as well as their arguments and thematic roles. This in total amounts to capturing specific event types. The events of interest were originally meeting events, but have evolved into events covering travel and criminal behavior as well.

The fourth and final phase of this workflow is the Scenario Reasoner, which takes as input the INF files in order to produce a final output that recognizes whether a particular document contains events of different types (e.g., meeting, criminal activity) as well as recognizing larger event scenarios using a set of established "storyboard templates" that specify larger events that while perhaps not directly stated in the raw text of the document are typically characterized by a set of smaller events more likely to appear explicitly in the text. These templates are further input to this phase of the workflow.

One can further add to this information the fact that the applications to perform these different actions are all discrete processes, most of which are written in C++ making use of F-logic interpreters written in Prolog. Excluding time between the execution of these processes, the time for the system per document is on the order of approximately 1 to 2 minutes for Serif (the system bottleneck), 1 to 2 seconds for SerifSum and the VSD Reasoner each, and 10 seconds for the Scenario Reasoner.

From this detailed description, some of the difficulties in the crafting of a workflow architecture around this process become apparent. This is a continually evolving piece of research work, and the flexibility to retune and improve the process is an important element that must be preserved in the architecture and made as flexible as possible. Further, a straightforward transition into UIMA would prove difficult for METEOR, which is not composed in Java but in C++, which while capable of being handled by UIMA is not its primary compatible language. More important than either of these issues though is the fact that the different parts of the workflow are discrete processes and not directly integrated, which would require recoding these components for more tightly integrated systems should one be used as opposed to an architecture that has the discrete processes executed as is by an executive and has their output handled by separate adapters that prepare information for storage in a data bus. Using a tightly integrated UIMA architecture would also require further rework on the analytics such that their output and input is compatible with CAS. Additionally, a data bus allows for some permanence of artifact collections which allows for provenance, traceability, and repeatability of the work, crucial for verification in a research setting, where METEOR is intended to be used.

In contrast, were the intent of METEOR to be used in a production environment then a specific tightly integrated architecture would make sense, where the tradeoff of provenance and flexibility with the efficient processing of incoming data disappears in favor of efficiency. This would ideally be an architecture though that supports and is optimized for only this task, perhaps matured in a research environment.

This distinction between a research and a production environment is crucial for determining which architecture is more appropriate. In situations akin to the complexity of METEOR and its need for flexibility and to accommodate domain expert engineers who are interested in collecting repeatable and traceable results, a discrete process architecture as described below appears to be the most suitable option as its use case best fits a research environment.

Tightly Integrated Architecture Technology Analysis

It is worthwhile to detail some of the options as the different architectures are considered. Although the consideration of a tightly integrated architecture up until this point has been limited to UIMA, which is intended for just such a purpose, here the discussion is opened up to include discussion of an alternative, GATE (General Architecture for Text Engineering), for comparison.

Both the GATE and UIMA frameworks take a similar approach in their fundamental design. Both frameworks define a common data format for a document based around the concept of an annotation. In UIMA, this takes the form of the CAS, which allows for the creation and storage of data types and provides a base data type for annotations that contain start and end offsets. A CAS object has one or more views of its document, and each view is associated with a unique SOFA (Subject of Analysis) for that view. A view of a CAS represents the abstract notion of an interpretation, and the SOFA for a view represents the actual data associated with that interpretation. In GATE, the data format takes the form of GATE Documents, which store annotations that contain start and end offsets and a table of features. Although these data formats are based on annotations, they are both general enough to represent nearly any type of analysis data.

Both frameworks also take a similar approach to incorporating analyses. In UIMA, analysis engines are Java classes with an associated XML file describing configuration parameters. Analysis engines expose methods to get and set configuration parameters, and a process method which takes a CAS as input and modifies its annotations as output. In GATE, language analyzers are also Java classes with an associated XML file describing configuration parameters. Language analyzers expose methods to get and set configuration parameters, and an execute method which takes a GATE Document as input and modifies its annotations as output. Additionally, both frameworks reuse this approach to handle output. For example, either framework could support an analysis which only writes its input to disk without modifying it; placing such an analysis at the end of a workflow would effectively save the results of previous analyses.

UIMA additionally provides a more general notion of analyses which GATE does not, called CAS multipliers. A CAS multiplier is an analysis which may produce any number of output documents (including zero) for each input document. CAS multipliers can be used to filter input documents by conditionally returning no output, to segment an input document into many smaller output documents, or to aggregate many input documents into one larger output document. All output documents produced by a CAS multiplier continue through the remainder of the workflow.

Both frameworks expose an interface for skilled developers to programmatically define robust workflows, and they also provide their own implementations of a simple, serial workflow which users who lack significant programming experience can use. With regards to workflow input, the frameworks take slightly different approaches. A UIMA workflow must include a collection reader as its first element, which exposes a next method that returns CAS objects. The UIMA framework repeatedly calls the next method until the collection reader indicates that input is exhausted, and the collection reader is responsible for reading input into the CAS format and returning it. A GATE workflow must be given a list of documents to process as input, and the GATE framework provides built-in support for reading common formats into GATE Documents with metadata (such as tags in HTML) represented as annotations.

Beyond what has been mentioned, the major advantages of both GATE and UIMA take the form of tools which are not essential to the underlying framework. GATE comes with an extensible GUI for loading documents and creating corpora, viewing and editing annotations, and loading and running analyses. This user interface can be considered superior to UIMA's user interface, which consists of a plug-in for Eclipse and a collection of shell scripts. GATE also defines the JAPE (Java Annotation Patterns Engine) language, which is similar to regular expressions for annotations. Developers can express a series of transformation rules based on annotations in the JAPE language, and then automatically generate a GATE analyzer which performs those transformations on its input. Finally, GATE is distributed with ANNIE (A Nearly-New Information Extraction system), a set of information extraction analyses for GATE developed by the University of Sheffield.

Excluding support for CAS multipliers, the major advantage of UIMA over GATE is scalability. UIMA provides support for deploying analyses to remote nodes as part of a distributed workflow, and support for duplicating a workflow on a single node using threads. Further, there is an addition to the base UIMA framework called UIMA AS (Asynchronous Scaleout) which is integrated with middleware to allow remotely deployed analyses to be duplicated across several nodes and work in parallel as part of a single workflow. Finally, there is also an addition to the base UIMA framework called UIMA C++ which allows analyses to be written in C++. By using SWIG, an open source interface compiler for C++, this framework can also be used to write analyses in languages with which SWIG can interface, particularly Perl, Python, Ruby, and Tcl.

Recommendation

Out of the two chief competing architectures that are intended for a tightly integrated architecture, there appear to be greater advantages to using UIMA, so specifically in a production environment the recommendation is to use UIMA as this architecture given its more general notion of analyses and greater scalability.

Discrete Process Architecture Technology Analysis

When considering a research environment as the intended venue for a Natural Language Processing analytic architecture, a discrete process architecture as suggested is the best fit for domain expert engineers who are crafting workflows of analytics. The following sections discuss in detail the specific subcomponents of the architecture which must be developed along with an examination of the available technologies for implementing each subcomponent. Recommendations for specific technologies are made for each as well following this analysis.

Discrete Process Architecture Technology Analysis: Interface

The interface provides the user with access to the architecture, allowing for definition of workflows and execution of specific instances of these. This access can be provided in different ways, including the user composing a configuration file that establishes the workflow and its input requirements. Another possibility is a graphical user interface (GUI) that can specify both the restrictions on the documents to be analyzed and worked on as well as the particulars of the workflow that will operate on the data taken from these documents, including which analytics are called and what data from different sources (in consideration that an analytic might need to execute on the output of a previous analytic) will serve as input.

While it is anticipated that most workflow instances will be generated as relevant documents arrive in the inbound gateway, this interface provides the opportunity for a user to run a single ad hoc workflow instance on a particular document. The user can also specify where the data is located or how to collect it rather than actually working from an initial document or corpus of documents on hand. This layer of abstraction means the user can avoid both the need to organize and maintain where the mid-workflow data is kept and avoid executing each analytic in the workflow manually, simplifying the input to the raw source document and the output to whatever the user specifies in the interface.

Discrete Process Architecture Technology Analysis: Inbound Gateway

The inbound gateway is responsible for handling the input stream of documents that arrive for processing. Based on the interactions with active workflow instances specified in and deployed by the executive, the inbound gateway will submit documents to the data bus for processing by those workflow instances.

By way of document triage, this transfer of documents can be filtered before it ever reaches the data bus, such that only particular documents (such as documents that match the active workflow instances' specification or those that are not corrupt) reach the data bus for processing. Alternatively, there might be workflows that intend to work on all incoming documents, in which case everything that passes through the inbound gateway will enter the data bus.

In addition to this document triage, there is a set of document interrogation procedures that all documents must undergo, including such analysis as language recognition and genre detection. This analytic work must happen before any of the active workflows execute on the incoming document, so it is necessarily a part of the inbound gateway, which prepares the documents for use by any of the workflows. As suggested above, workflows may be targeted to documents of a particular language or genre and this information must be made available to all of them up front. Any further or deeper analysis should not be part of the inbound gateway as the key role here is not meant to be content extraction or annotation, even though it is necessary in the identified cases.

An issue that the inbound gateway must accommodate is the limitations on the systems of the amount of parallelism possible. There are going to be low and high water marks in the capacity of threads being executed, which require different input rates on a given instance of the system. When a high water mark of capacity is reached, the inbound gateway should inform the Content Provider to throttle back on documents being sent to it and when a low water mark is reached, this is also indicated to the Content Provider so that the stream of input can be resumed.

This prevents the thrashing of the executive between high and low activity, keeping a steady wave of activity.

It should be noted that the inbound gateway is not a core competency of this system, but this description has been included for completeness sake.[1]

Discrete Process Architecture Technology Analysis: Executive

This section considers and examines several technology options for an executive to a discrete process architecture. The role of this executive is to orchestrate all the user-specified behavior in the execution of a workflow. Ideally, an executive would also automatically handle some of the data or memory management issues for the user, but it at the least needs to provide the opportunity for the user to specify the coordination and flow of documents and document artifacts throughout the life of a workflow instance. In this architecture, the executive will either receive input from an inbound gateway that supports the input stream or as a specified file when operating in a "debug" or ad hoc execution mode. This input is moved into the data bus and will be used as input to the analytics, between which the executive is tasked with handling the data flow.

Workflows must be capable of persistent deployment by the executive, and communication with the inbound gateway determines the documents relevant to the workflow that will be processed in a workflow instance. This allows for multiple instances of the same workflow to operate on different documents that arrive for processing. Alternatively, workflows can be deployed temporarily to execute on user-specified documents in an ad hoc manner.

Workflow information, namely the succession of and parameter settings for analytics, should be retained among the data in a document artifact collection in the data bus and past the life of the workflow in the resulting output. This allows for keeping track of the provenance of what is produced by a collection of analytics making results traceable and repeatable.

Among the possible architectural technologies considered that can fill this role examined here are UIMA, OpenPipeline, Mule, and Ptolemy, the applicability and pitfalls of each discussed in turn. Each discussed technology is open source, unless otherwise specified. The initial analysis was conducted for each of these in a simple use case workflow of analytics. This task involves two simple analytics. The first is a *tokenizer* that separates text based on whitespace and puts each token in a document on a separate line in an output file. The usage is "java –jar Tokenizer.jar <input file name> <output file name>". The second is a *decoder* that outputs two files, the first a concatenation of the first letter of each line of its input and the second a concatenation of the last letter of each line of its input. Its usage is "java –jar Decoder.jar <token file name> <output1 file name> <output2 file name>".

The following directory structure is on the disk of a single, isolated machine. There is a top level directory called "Task", which has four subfolders: "Input", "Tokens", "First", and "Last". The input directory has some number of named text files processed with two toy analyses. For each input file, the *tokenizer* output is sent to the "Tokens" directory but with the added extension ".tokens", and the decoder output is sent into the respective "First" and "Last" directories with the added extensions ".first" and ".last".

In each case the task is to execute the *tokenizer* first and then the *decoder* on its output to end up with two output files. This allows a reasonable comparison for the difficulty in crafting a simple workflow using each considered technology as the executive.

[1] A description of the current implementation of the inbound gateway is provided in Appendix F. (Feb. 2011)

UIMA as Executive

As discussed in the earlier examination, UIMA is a software architecture that supports developing and deploying analytics that execute on potentially large volumes of unstructured information. The ultimate intent of these analyses is knowledge discovery. The most relevant use case at hand which UIMA addresses is the consumption of raw text in order to enrich the source material and extract elements such as entities, relations, and events. While intended to orchestrate components that plugged in to be part of the same application, it is also capable of executing a workflow that makes use of outside components. The example case was implemented in UIMA with this in mind.

UIMA was examined in the role of an executive for discrete processes, the details of which are discussed here with the code implementation provided in Appendix A. As described above, two "black box" toy analyses ran on input documents and produced output for, the *tokenizer* and the *decoder*.

In building this system in UIMA, the code necessary to create input CAS objects is first described. The collection reader at the start of a UIMA workflow is responsible for reading the input files and formatting them into CAS objects. In this example, the initial CAS object creation is trivial as the UIMA framework is packaged with a collection reader that can transform input text files into CAS objects where the body of the text is the sole SOFA. It is worth noting that this procedure is not as simple for more complex documents such as PDF files, or HTML files that could require de-tagging. Additional UIMA collection readers for some common file formats exist as open source code, but it may occasionally be necessary to develop a new collection reader. In consideration of this, a collection reader for this task was written for the purpose of demonstration.

The first step in building a collection reader is to write an XML type descriptor. This file and its details are described in Appendix A.1. Once the type descriptor file is created using the UIMA Eclipse plug-in, the plug-in will automatically generate Java classes for each type one defines. This makes these Java classes useable in the collection reader Java class written next. This class and its XML descriptor are further detailed in Appendix A.1. Next, similar XML descriptors and Java classes are written to encapsulate each analytic. The necessary XML type descriptors and wrapper Java code for the discrete process, or black box, components in the task, the *tokenizer* and the *decoder,* are detailed in Appendices A.2 and A.3, respectively. Finally, an output component is written to save the analysis data to disk. This XML file and Java code is similar to that for the analytics, and is not detailed for brevity.

With all the analysis engines defined, the next step is to define a flow. Since only a simple linear flow is required, the built-in flow control in UIMA can be used. However, UIMA does provide an interface for more complicated workflows to be defined if necessary. Both the full flexibility of UIMA workflows and the flow controller used for this task are detailed in Appendix A.4. After writing another XML file to specify this workflow, it can be executed by starting a script in the UIMA distribution, which launches a Java GUI allowing a user to select the designed workflow and run it once as a UIMA application.

While the task was successfully accomplished, it makes clear some of the severe inefficiencies of using discrete processes in a tightly integrated framework. The collection reader reads the input from a file into the in-memory CAS object. The *tokenizer* wrapper then writes the CAS to disk so that the *tokenizer* black box can take it as input. The *tokenizer* black box then has to read the input file into memory, do its work, and write the output back out to disk. The *tokenizer* wrapper reads the output back into memory in the CAS; however, the returned format cannot be immediately and unambiguously converted to annotations, so extra processing must be done.

After the *tokenizer* wrapper performs a linear search through the document in order to format the token list as annotations, the *decoder* wrapper ignores that work by reconstructing the token list, and writing it out to disk so the *decoder* black box can take it as input. The black box has to read that file, and then write out two output files. Then, the *decoder* wrapper has to read both output files back into the CAS. Finally, the output engine reads the CAS and writes everything in it back out into files.

This constant exchange between the in-memory CAS object and on-disk black box formats is representative of a trade-off between efficiency and modularity. Having sacrificed efficiency by repeatedly moving data in and out of memory, the *tokenizer* analytic can now be feasibly replaced with some other UIMA analytic which also produces token annotations, and this would not alter the function of the *decoder*. However, this does not eliminate the need to conform to a specific data model to facilitate communication between components, something discussed below when examining adapters.

Alternatively, it is unnecessary to use the CAS to store all data. Consider that in the collection reader, one could have only put the path to the input file in the CAS, and not bothered to read the file data in to memory. Then, the *tokenizer* wrapper could simply pass the input file path to the black box *tokenizer*, and then add the file path of the *tokenizer* output to the CAS. The *decoder* wrapper could use this file path to send input to the *decoder* black box. If the outputs from each black box are then stored in their intended final destinations, an output engine would not be necessary, and extra memory-to-disk or disk-to-memory penalties would not be incurred. However, this system lacks modularity because it does not transform the inputs and outputs into a common format which might be recognized or produced by other analytics.

Various middle grounds could also be considered, such as keeping the original artifact data on disk and only storing a file path to it but storing actual analysis data in the CAS. If UIMA were to be used for the executive, this balance between efficiency and modularity would be determined by the developer.

OpenPipeline as Executive

An alternative technology to UIMA, though it covers much of the same capabilities in terms of acting as an executive, OpenPipeline is a software architecture intended for analyzing documents. It has pre-built components, but it can integrate external modules.

In order to make a fair comparison with UIMA, it was attempted to recreate the same workflow with the same black box components using OpenPipeline. One of the unfortunate limitations of OpenPipeline is the lack of a Developer's Guide at present and a spotty Javadoc. Most of the analyses are gleaned from examining the source code.

OpenPipeline is tightly integrated with a web based GUI for creating and executing pipelines. As a result of this, OpenPipeline is not easily embeddable within another application or in a separate UI. Further, because the format of required XML parameters for workflow definitions is not clearly specified and the generation of these files is tied to the GUI, users are forced to use the provided interface.

An OpenPipeline workflow is a Java class which exposes an execute method that is responsible for the entire execution of the workflow. In this method, a document crawler examines a data source specified in the workflow XML file for raw input and creates an Item object from each document. The Item class is OpenPipeline's common data format, and stores data in a tree structure of Nodes. It is rich enough to support any analysis data, but unclearly separates data into artifact data, artifact metadata, and analysis data. This could potentially cause issues if analyses intended for data are performed on metadata or vice-versa.

OpenPipeline next routes Item objects through a series of Java classes based on a list of analytics also provided in the XML file. Each analytic provides an execute method which examines the Item object as input and modifies it as output.

The lack of documentation makes building a document crawler ourselves quite onerous, so an implementation provided by OpenPipeline is used. Concerning the analytics, the details of the creation of the wrapper for the *tokenizer* are listed in Appendix B.1 and the details for the creation of the wrapper for the *decoder* are listed in Appendix B.2. This work is followed by the output engine, the code for which is detailed in Appendix B.3. In order for these analytics to appear in OpenPipeline's web interface, a JavaServer Page must be written as described in Appendix B.4.

The class files created for this task are then put into a JAR file with a service specification and added to the install directory for OpenPipeline. The JSP page is placed into OpenPipeline's web directory together with the pages for other stages. After doing this, one can start OpenPipeline and create and execute the workflow. It was found that it executed and produced the correct output.

The lack of any Developer's Guide at present makes OpenPipeline an unattractive option as a solution for developers to create new workflows with relative ease, something that is key to a research environment's requirements, and having a focused Developer's Guide would be necessary to fit the customer needs.

OpenPipeline has the same spectrum of modularity found in UIMA, and the trade-off in efficiency and modularity that is present in UIMA is present here as well. The efficiency drops when users are forced to map analyses in and out of the common format.

Mule as Executive

Mule is looked at next as an option for an executive to the discrete process architecture. Mule is an enterprise service bus, a less tightly integrated alternative to the options discussed thus far but one that scales well. Again for a fair comparison with the earlier examples, the same workflow involving the *tokenizer* and *decoder* discrete processes is recreated using the Mule ESB in the role of executive. Appendix C.1 shows the XML file with embedded scripts, the full extent of the code necessary to make this run. While Mule normally uses Java classes as its components, this demonstrates how some scripting languages can be embedded in the XML.

This implementation of Mule indicates that there is potentially a greater simplicity in developing workflows using multiple discrete processes. Whereas the complexity of crafting the XML and Java code was somewhat heavy when using UIMA and OpenPipeline, this is reducible to a single XML specification file in this simple case, making Mule much more manageable.

Unfortunately, this does not address the problem that a user would still have to learn the specification for the XML in order to craft the workflows and it is not as trivial as simply providing a list of analytics to the front end that need to be run in sequence.

Mule is also not quite comparable to the other examined candidate executives in how it performs its workflow. Mule is built around a data-driven workflow, which leads to a slightly more intuitive model for accommodating simple decision making and decision points (e.g. if-then statements). However, complex decisions still require some programming experience, meaning that as workflows begin to require these conditions, more work will be entailed to implement them. This issue is not unique to Mule and will be discussed at length in a subsequent section examining achieving decision points in executives.

Another advantage of Mule is that it is more flexible in terms of how data is represented between analytics, which means that different formats likely to emerge from different analytics do not pose a compatibility problem from this executive's perspective, although they would still need to be resolved to communicate to one another. However, this removes some of the added burden imposed by other executives.

It is also noteworthy that while Mule has a version that is available as open source, there is an enterprise version of the software that must be paid for. This analysis has examined the open source version, but there might be features available to this version not available in the open source. Though Mule of course states that this version is intended for production use, it is likely that the open source Mule is sufficient for the needs of this architecture.

LONI or Ptolemy/Kepler (Scientific Workflow Projects) as Executive

There exist scientific workflow products that are applicable in this use case too. Here, the LONI Pipeline is examined as an option for an executive to the discrete process architecture. This is a file-driven scientific workflow application. This section also provides an examination of Kepler, another project for creating scientific workflows built to be wrapped around the Ptolemy II open source workflow project. Ptolemy II provides the foundation for Kepler.

Both LONI and Kepler are intended for end-users who are not expected to write code, and each provides a GUI that is intended for use in creating workflows in each. Appendix D shows sample graphical workflows constructed using either LONI or Kepler. This provides some added convenience over the required coding that is necessary for other workflow alternatives. From the analysis of both of these, LONI has the simpler and more user-friendly interface than Kepler, but neither of these make the assumption that their users need to be developers.

In terms of the data format and data transfer, LONI stores data in files passed through input and output while Kepler can pass data in memory between its analytics but also have the capability to write to and read from the file system or network.

LONI and Kepler are capable of parallel execution of workflow instances, though Kepler does not support the simultaneous duplication of an actor (that is a specific analytic). Both workflow projects allow for using grid computing.

While LONI and Kepler are each documented for users, a key limitation on LONI is that, while free, it is closed source, unlike the other technologies seriously considered here. This means that it cannot be executed programmatically. This does present a long-term limitation to using LONI as the executive as it eliminates any ability to extend it to be more robust if necessary, while Kepler is extensible and its user interface could be extended to be more user friendly to a research environment where components.

Another potential limitation of LONI is that it places a restriction on its analytics in that they must take input as files and produce output as files. This, along with the closed source of LONI, makes Kepler the more attractive alternative of the two. A further examination was made of the open source workflow project around which Kepler was built, namely Ptolemy, and it was found this had a greater generality while capable of the same features of interest described above.

Decision Points in Workflows across Possible Executive Options

Naturally, the examined workflow instance is a trivial example, and one might wish to consider workflows of greater complexity as well. What has been considered up until this point has been really more in the manner of single sequence pipelines, as opposed to the more complex structures that can make up workflows. This will be considered next across candidate executives.

Workflows are distinguished from simple sequences of activities by a capability for more complex behaviors past simple chaining of analytics. While far from all inclusive, the most crucial of these capabilities is to provide for decision points, that is to say places in the workflow where the thread of execution can follow one of multiple branching paths based on some condition being met or not met. Although many workflows that are likely to be executed will not make use of this and are sufficiently captured by a completely sequential list of activities, there could be others that do require decision points or they could be created in the future if the executive is capable of handling these.

The difficulty in making use of the robustness of UIMA has been previously discussed, but it bears mentioning again. Developers are required to have a certain degree of skill and experience in Java in order that they can make use of the workflow options such as conditionals, splits, and joins that advance the process past being a sequential pipeline. In the simplest case this would involve implementing at the minimum two non-trivial Java classes.

With regard to OpenPipeline, this limitation is even more severe, in that anything other than a linear workflow is not achievable at the executive level and would have to be embedded into the analyses themselves, which undoes the crucial benefit of modularity.

This is not quite the case with Mule though. Filters can be placed on the input and output to the components that effectively act as decision points, selectively dropping messages and determining what destination they reach. This allows for at the very least simple conditionals by filtering messages and message properties that return from the analytics (via regular expressions, for instance). This issue of decision points in the different alternatives for the executive are detailed further in Appendix E.

While this might not handle highly complex decisions, those can be composed in a script or Java class to make the decision. It is also worth noting that while messages might not be passed from analytics directly to Mule as the executive, they could be wrapped in a manner similar to what has been demonstrated for other technologies to provide this interaction if necessary, avoiding recoding the analytics. These instances would entail more work than just a single XML file, but this work would be not more difficult than what is required by UIMA and OpenPipeline to execute a single linear pipeline of analytics, a functionality that Mule would be overtaking and one that would require even further work in UIMA or OpenPipeline to match. Therefore when comparing equal levels of capability (a linear pipeline, a workflow with decision points, etc.), Mule requires less work than UIMA or OpenPipeline.

In comparison to UIMA, OpenPipeline, and Mule, the scientific workflows such as LONI and Ptolemy/Kepler are more robust with less effort in creating decision points. In LONI, workflows are created by matching input and output files of different executables and splits can be created by duplicating files while basic conditionals could be achieved based on the existence or length of a file. In Ptolemy and Kepler, workflows are created by chaining together the input and output ports of actors, and it is rich enough to support splits, joins, and conditionals.

A BPEL Engine Executive?

One of the considerations that arose in examining the different alternatives for an executive in a discrete process architecture intended for a research environment was using a BPEL (Business Process Execution Language) engine. BPEL workflows are composed in XML and invoke services, operating on both their input and what the services return. Unlike most of the other workflows, which lean more to being pipelines in their implementation, BPEL provides a sufficiently rich workflow to manage the situations that would arise in this field. It has an added advantage that it is becoming a de facto standard in web service workflow. However, this very fact raises the immediate concern with BPEL as an alternative for capturing workflow. It is intended to invoke web services, something that one cannot presume about the legacy analytics. This could require a heavy amount of conversion of this existing software and further stipulations on writing new software in the future, the minimization of which is intended for the sake of the analytic composers in the research space. Indeed, BPEL is likely far too heavyweight a technology for what is required in the executive of this system. Because of this need to scale it down significantly, it was not considered further here.

Other Options for Executive

The Blackbook project back in 2005-6 examined the existing technologies available for performing workflow with the intent to use the most appropriate single technology, but this resulted in no suitable solution being discovered. From their list of technologies that might show some future promise, a more up-to-date investigation was made for this document. These technologies included BigBross Bossa, Ruote, con:cern, YAWL, Zebra, Syrup, Dalma, and GridAnt. As of now these technologies are outdated, still too immature, or not applicable to this project, and they were not considered further.

Recommendation

The difficulty in examining technologies for this role is that there does not appear to be a perfectly ideal choice, but the trade-offs between these different options appear clear and distinguish them from one another. Out of the examined possibilities, the strongest alternatives for this context of a research environment appear to be Ptolemy, Kepler, or Mule, particularly in the simplicity of integrating analytics, creating workflows, and more easily allowing for more complex behaviors within the workflow, such as handling decision points. They provide the flexibility that a system for a research environment needs, and they should be replaceable should an alternative prove more suitable in the future. Of these options, Ptolemy is most attractive given its built-in graphical user interface while configuration files still are required to be written by hand in Mule and given Ptolemy's greater generality than Kepler, which makes use of it. This replacement, if necessary, could be affected with minimal to no change in the analytics themselves.

Discrete Process Architecture Technology Analysis: Data Bus

In the modular discrete process architecture, an inherent limitation on efficiency is that because an attempt is being made to reuse analytics with the minimum amount of redesign and recoding, most of the analytics as they are written and as they are likely to be written in the future are or originate as standalone applications that are not intended to integrate with one another. This limitation means that the desire to have these analytics communicate directly via information stored exclusively in memory during the life of a workflow instance is not feasible in this system design. It bears mentioning that this memory-based data bus model *is* feasible in a more tightly integrated architecture, but this entails the loss of the flexibility afforded by what is being described here.

Therefore from the perspective of a discrete process architecture, this section proceeds facing the limitation that there needs to be some file-based content management system acting as a data bus for the overall architecture. At this point we examine some of the alternatives available for storage of documents and document artifact collections over the life of a workflow instance as well as potentially over a longer term. The discussed options include simple file systems (specifically, a flat file system), Alfresco, and ObjectStore. Other alternatives are Documentum and FileNet, but these are heavy cost options and not investigated here.

Flat File System as Data Bus

The simplest solution for handling the documents and document artifacts when passed between analytics is a flat file system. This system eschews the need for hierarchies of folders as this involves essentially a directory. This leads naturally to the requirement that all the different files produced by the analytics must have different names, names which should distinguish the files based on uniquely identifying information. This information should include versioning as well so that provenance can be established.

Despite its simplicity for storage, it creates some requirements that must be met by either the analytics or wrappers created for the analytics. These must be coded to both uniquely identify the document artifact collections and correlate them with the correct currently executing workflow instance (or past workflow instances if data is preserved for provenance and repeatability). The analytics must also have their output redirected to this data bus's location. In the cases where analytics are hardcoded to send their output to a specific location or give the output a specific name, then transferring analytic output (and potentially input to further analytics that have hardcoded input paths) must be accomplished either via a wrapper or by recoding the analytics to support parameters that specify where to send their output and what to name it. Because it lacks any kind of structure except for what is imposed in the naming scheme though, it seems evident that this would grow and become unmanageable very quickly, especially as many workflow instances begin to operate simultaneously and leave behind artifacts. A solution here though is to only keep the version document artifact collections in this data bus over the life of each instance, transferring them elsewhere to maintain the long term provenance after an instance is finished executing.

Alfresco as Data Bus

Alfresco is an open source enterprise content management system. The following is a rundown of key elements of Alfresco that are of interest to developers and end users. While the system is established and running in the Amazon cloud, Alfresco is also downloadable and is capable of running locally on a user's machine. Alfresco has the capability to be deployed across multiple platforms and is highly scalable. Therefore, at the first examination it seems to aptly fit what is envisioned as being an appropriate content management system to support the data bus in the discrete process architecture.

Alfresco operates by managing spaces, essentially smart folders or containers that have a hierarchical structure. A straightforward implementation would have a single space in Alfresco associated with the discrete process architecture. Different workflow instances would create sub-spaces in this space that are identified by uniquely identifying names. These sub-spaces can have any hierarchy that fits the definition provided when discussing the adapters and the common interchange format, so the key point at this stage is that there is a great deal of flexibility afforded by such an architecture that reshapes itself to the different instances that are created and disappear. Users can be associated with the space and with individual sub-spaces such that workflow instances and users can be easily correlated. This allows any number of uniquely identified runs to be created and maintained during and even after the lives of specific workflow instances (necessary, for example, in the cases where this data needs to be maintained for repeatability comparisons and provenance).

Three additional benefits to using Alfresco are its capability for versioning, its capability for indexing, and facilitation of a central repository. Versioning of document artifact collections and analytic support data is an inherent requirement of this system. Indexing facilitates granular searching of the content, another capability that could be leveraged in the research environment. A central repository allows for an easy distribution of artifacts in a consistent location that is easily accessible from many locations.

ObjectStore as Data Bus

ObjectStore stands as an example of technology for object data management, specifically providing an embedded database that is targeted to provide data storage for object-based languages, in particular C++ and Java. One of the key advantages offered by this data bus is that it offers object data to be delivered in-memory, achieving a greater level of efficiency between different applications. It is also capable of concurrent access by multiple applications.

Among the potential pitfalls of following ObjectStore as the basis for the data bus is that because it is dedicated to storing objects for C++ and Java, this makes integration more awkward for any legacy analytics written in other languages, Perl and Python for instance. Even integration with C++ or Java-based analytics would require those to be recoded to communicate with ObjectStore. Following the option that allows these analytics to act independently and then coordinates a mapping of their output into the ObjectStore representation seems wasteful as it in many cases will involve writing to disk only to then write again to memory, losing any of the benefits of efficiency ObjectStore would offer.

Recommendation

Out of the examined technologies, Alfresco appears to be the best suited to the discrete process architecture data bus for a research environment. Alfresco is a fairly mature content management system and is widely used and well understood. It appears to provide the level of detail and malleability required by this architecture while being straightforward to use.

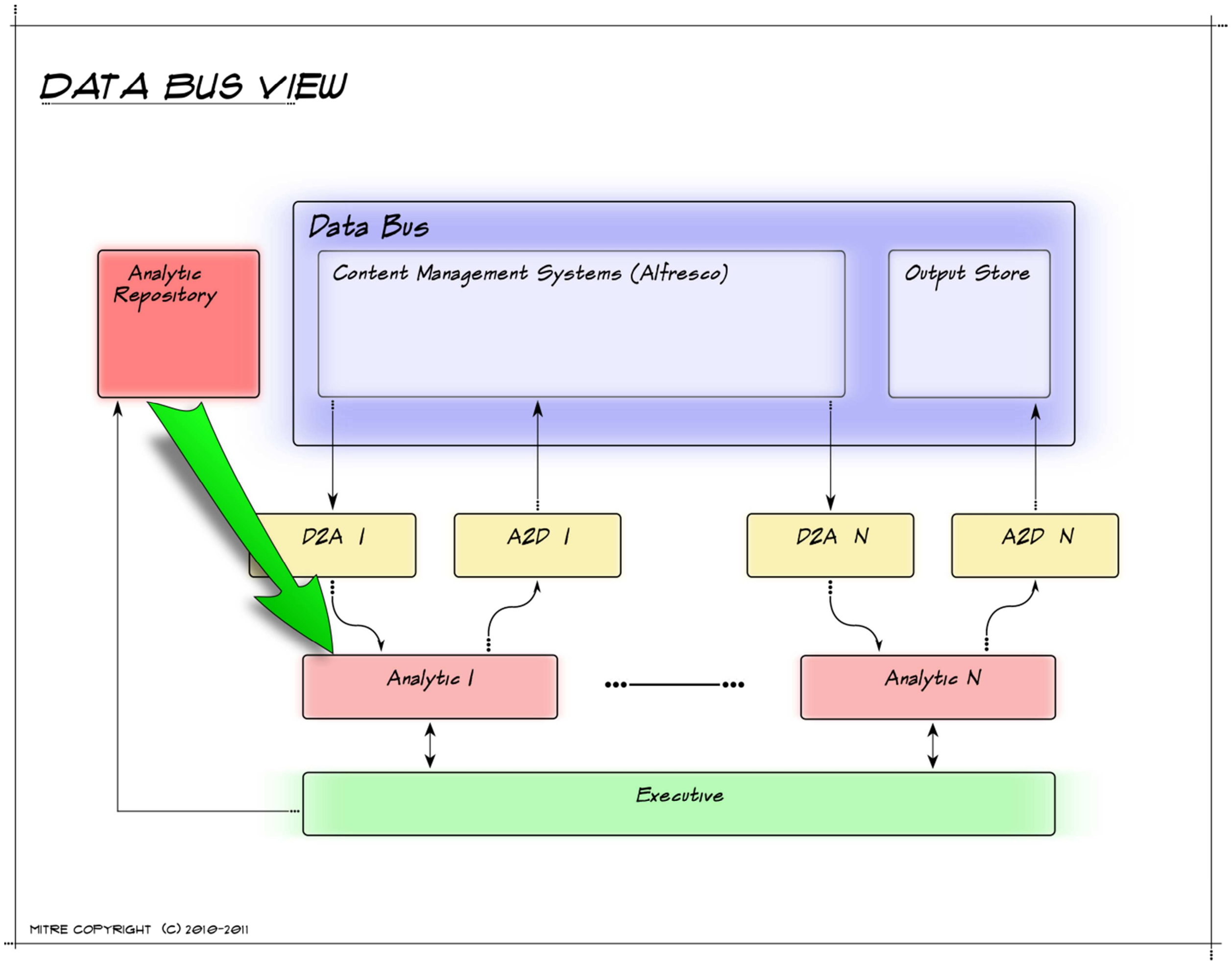

Figure 5. The data bus of the research environment MOSAIC architecture broken into expected sub-components, including a content management system and an output store for the final results. Also depicted is a long-term storage for all the analytics used by the system.

It should be noted that this recommendation is specifically for the data as stored between the analytics during the life of a workflow instance. Data must be maintained somewhat persistently, even if the analytics themselves are stateless and unaware of what processes have preceded and will follow them. It is also necessary for maintaining the various results that are not the intended output but required for the purposes of provenance and traceability or repeatability. Once a workflow instance has terminated though, one remains faced with the issue of storage of the results that will be picked up by any subsequent phase of processing outside the purview of the discrete process architecture. This could be stored in Alfresco, but it seems that for this use case, a faster access of the objects might be useful, so an alternative to Alfresco could be used here.

Figure 5 shows a more detailed view of the data bus that reflects this potential division of function. In this depiction the data bus contains a CMS (Content Management System) that handles the documents and document artifact collections over the life of the different workflow instances as well as an output store where the output to be collected by the Knowledge Base Architecture that ingests the output of MOSAIC could be kept in a quickly-accessible cache.

In addition to these elements of the data bus, it is also recognized that given the potentially large number of analytics that can be a part of the discrete process architecture that a long-term storage for those not in use should be included. Frequently or recently used analytics would be in a cache outside this storage, but those that are called upon which are not presently deployed can be brought out from this long-term storage.

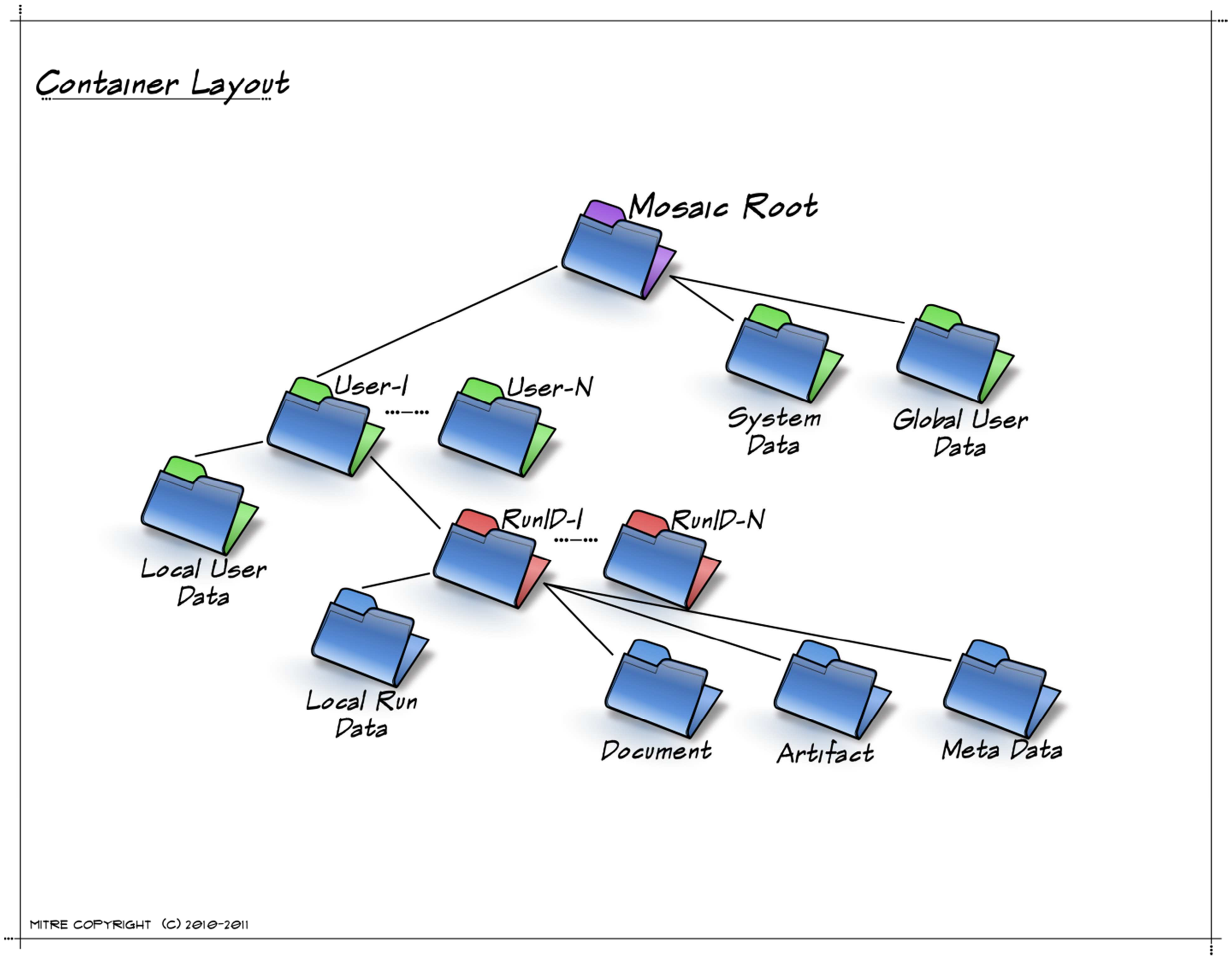

Figure 6. Suggested hierarchy for the content management system.

Figure 6 examines the anticipated hierarchy within the data bus's content management system. The root folder is associated with the entire discrete process system, including a folder that maintains all the required global information to the system. In subfolders below the root, each user has a folder containing the user's local information along with subfolders for each executed or executing workflow instance associated with that user. These subfolders also include the necessary local information along with subfolders for the source document, its document artifacts, and document metadata.

Discrete Process Architecture Technology Analysis: Analytics

In a modular discrete process architecture, it is preferable that there not be any requirements at the outset in terms of what languages are allowed or disallowed. This is due to the need to support legacy analytics that were written to be independent of other analytics, but could potentially integrate into a larger system. By virtue of the flexibility of this discrete process architecture, analytics written across many different languages, even those written without the intent to have been integrated, can be added as subcomponent analytics. In many cases, the appropriate action for including these pieces of software is to run them with adapters that can handle conversion into a common interchange format as well as accessing the data bus for both retrieval and inter-analytic storage.

Analytic Data Models	**Description**
Chunking	Identifying sentence constituents (noun phrases, verb phrases, etc.)
Concept analysis	Characterizing documents by concepts explicitly/implicitly expressed in them
Co-referencing	Matching multiple textual mentions of the same entity/relation/event
Document classification	Categorizing a document based on its content
Document metadata	Data about the document as opposed to its textual content
Extract entities	Objects or sets of objects in the world, often people, organizations, locations
Extract entity mentions	Textual instances of entities
Extract event mentions	Textual instances of events
Extract events	Specific occurrences involving participants
Extract relation mentions	Textual instances of relations
Extract relations	An ordered pair of entities that indicates a relationship between them
Language recognition	Identifying the language in which the text is written
Morphological analysis	Analysis of the structure of words
Part-of-speech tags	Categorization of words in a text to their grammatical tag
Semantic role labels	Recognizing roles nouns have in relation to the actions stated in a sentence
Sentences	Parsing documents into sentences
Sentiment	Recognizing the attitudes of a document's author
Sequences	Parsing documents into a specified recognizable sequence of words usually
String transliteration	Transfer of text in one writing system to another
Syntactic parse	Determining grammatical structure of the text
Time	Points or durations of time that appear in the text; possible subset of entities
Tokens	Breaking documents down into tokens, usually words
Value	Further information about or characterization of an entity

Table 2. Growing list of required analytic output the common interchange format must support.

This flexibility does not exclude the utility of a best-practices guideline for future analytics that can be written with the intention of integrating with the overall architecture. Indeed it is expected that analytics here can eventually become operational in a production environment after being implemented, debugged, and refined in a research environment.

An examination of the more tightly integrated frameworks that can support what are considered modular analytics as well as accounts of their features and their inherent advantages and disadvantages in this use case was examined earlier in the section entitled "Tightly Integrated Architecture Technology Analysis." Specific analytics required for the system, many with existing implementations, are listed in Table 2, and some key analytics are discussed further in the following section.

The overall recommendation arrived in this previous discussion was composition of individual analytics in UIMA, but this does not mean that all analytics need to be written in this language for research purposes. Because of the heavier weight that UIMA has as a framework, it is very likely that initial analytics can be more easily composed and tested outside that framework so as to be verified as useful before more time is committed to them and integrating them into what is more akin to a production environment. This system should not impede rapid prototyping.

Specific Analytics and Analytic Workflow

Although MOSAIC is capable of supporting a wide variety of analytic subcomponents to be used in large-scale workflows, it is useful to provide particular illustrative examples of the types of analytics and analytic workflows we expect to handle and that make sense in a setting involving workflows going from Natural Language Processing to knowledge representation. In addition, a sample workflow that makes use of some of these subcomponents for a larger task is also described to further demonstrate the architectural capability.

The basic definition of an analytic is that of processing software that extracts or generates new data from the source files or the data in analytic artifact collections produced by other analytics. In this field, there are a wide variety of individual analytical tasks that can be performed on raw text, processed text, and previous natural language analytic results. A few of these are described here to give a representative picture of what are considered analytics.

Entity extraction from text is a prime example of such an analytic. Typically, given either a raw or zoned piece of text, these analytics recognize and identify the pieces of text that represent entities and classify them based on the set of entity types they are programmed to discover. Usually these analytics preserve the original text snippet where the entity appears, either identifying it inline or providing offsets to its location in a separate file. Often this includes the co-referencing of these specific mentions to indicate they specify a common entity.

Some analytics instead try to extract information about the document itself. Concept extractors examine the document in order to discover terms and keywords that describe or appear in the content. This can be a very broad class of terms, so typically concepts are identified by short keywords, and rather than being attached to specific spans of text, they are associated with the document as a whole.

Still other analytics are targeted to produce results based on the word and grammatical structure of the source text, and these include tasks such as chunking, sequence or sentence tagging, part-of-speech tagging, and syntactic parse. This is sometimes supporting information to future analytics that make use of this structural information.

Standing as examples of analytics that build on previous artifacts, relation extractors often require the extraction of entities first so there is a population among which to discover relations. Similarly for event extraction, participants in these events (again entities) are frequently prerequisites for recognizing and defining events found in the text. Some analytics include entity extraction as a part of the process leading to capturing relations and events, but there are others which expect it as input to be provided by a previously executed analytic.

Perhaps the most critical example of an analytic that relies on the input of previous analytics is one that resolves the results of two or more analytic components that endeavor to extract the same class of information. This is a need most commonly felt with the disambiguation of entities, as the same entity found by different extractors should be recognized as such instead of being considered two separate entities as would be the case without software to handle their resolution. The results of this resolution provides a list of entities believed to be distinct, while preserving the provenance in terms of which extractor produced the original inputs.

There are also various supporting analytics that provide a necessary document level analysis for the purposes of triage in advance of selection of specific workflow activities or perform essential pre-processing on the document, such as zoning the document into regions that distinguish content from metadata. These analytics are typically run in advance of the rest of the system, and while they do not produce information that will necessarily carry on past the life of a workflow instance, they do enable the content extracting analytics to function.

It is crucial that the distinction between an analytic and what is called an "adapter" is made clear. The analytic is concerned with the extraction or generation of artifacts from a given input, the emphasis here being production. The adapter is concerned with the transfer of one analytic format to another, the emphasis here being conversion. This is often a subtle distinction, as some analytic tasks have elements of conversion as a part of their production. Key to making a determination of whether something serves an analytic purpose as opposed to the role of an adapter is whether the content of the input undergoes more than a cursory examination to determine the output. In the case of an adapter, this examination should only be cursory, simply enough to follow preset and rigorous rules for making consistent format exchanges. Fundamental changes in the content, such as entity resolution or mapping between extracted text strings and hard data types (i.e. bridging the semantic gap) should be considered analytic tasks and not the province of adapters.

Analytic Pipeline

As for an example of a combination of analytics that MOSAIC can handle, a pipeline is presented consisting of analytics targeted to the task of converting raw text data into knowledge objects and relationships between them. This pipeline orchestrated and supported by MOSAIC is depicted conceptually in Figure 7.

The fundamental task here is to extract knowledge objects from textual sources (in this instance, email). Various analytics developed independently provide the different required functionality for achieving this. The executive of MOSAIC orchestrates the activation of these analytics as necessary, providing them with data that is adapted from a common interchange format (CIF) into the analytics' standard input. The output of these analytics is then adapted back into the CIF, and this material is stored in MOSAIC's data bus for the life of a given workflow instance until a final output is produced.

In this particular workflow of analytics, entities and events are extracted from the text as well as broad concepts that define the source document. Analytic functions depicted in this figure, such as entity extraction, can consist of multiple analytics that perform the same task but provide different results. Because entity objects can be produced by more than a single analytic, potentially redundant results are resolved into single objects before being converted into knowledge, the ultimate output. While the conceptual pipeline is arranged in a serial fashion in Figure 7, many of the depicted components can run simultaneously, data dependence permitting. For instance, there are event extraction analytics that require previously extracted entities to supplement their input.

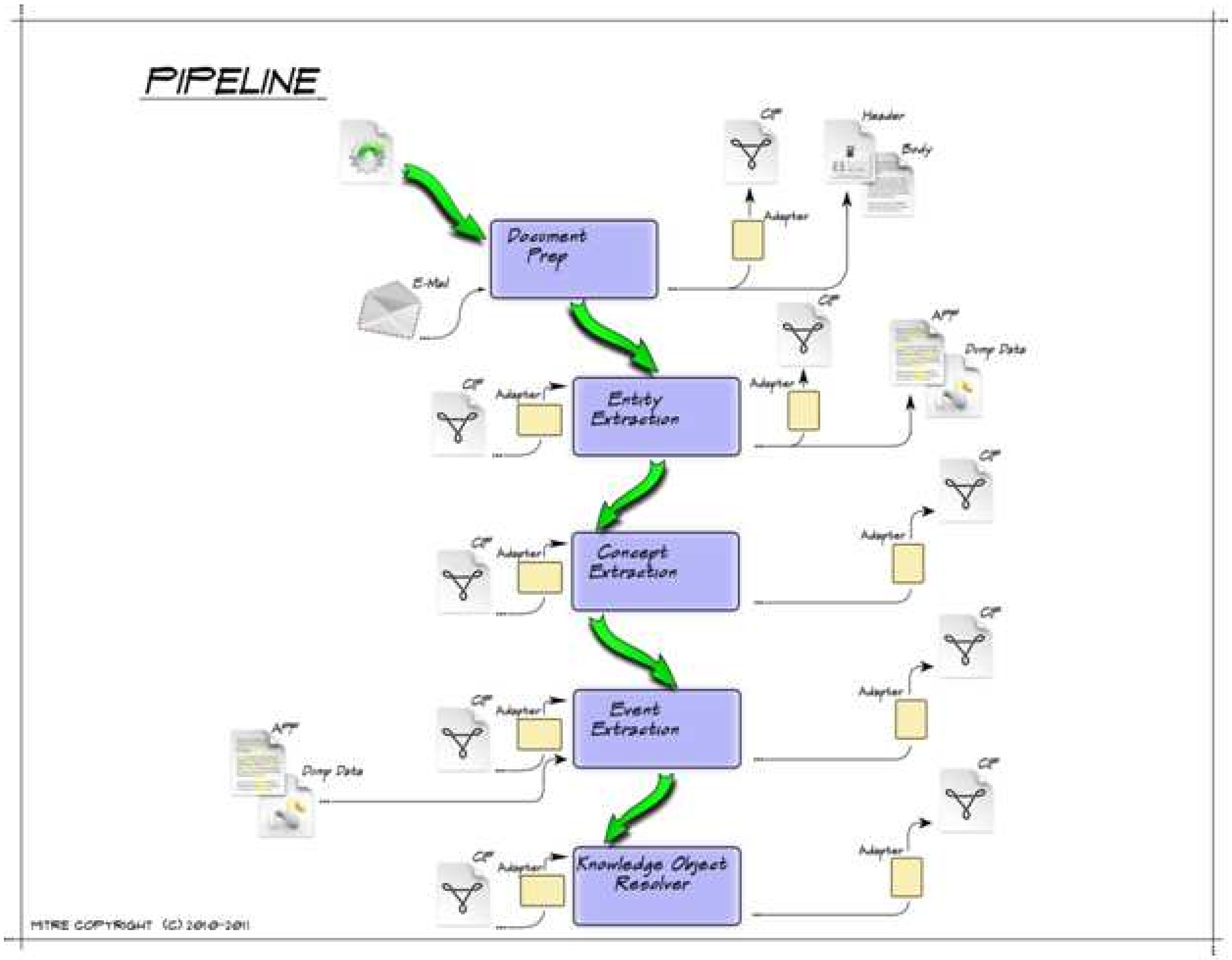

Figure 7. A current implementation use-case of the MOSAIC architecture featuring serialized analytics for extracting entities, concepts, and events as text and then resolving them into knowledge objects.

Discrete Process Architecture Technology Analysis: Adapters

In order to achieve success with a discrete process architecture that involves multiple legacy and newly developed analytics, it is crucial to provide some common model for the communication between them. It is unreasonable in the research environment to expect that all domain expert engineers will have composed and will continue to compose new prototypes that all output to and accept input from a specific model of representing the results. This diversity of input requirements and output formats among the analytics will require that there be some method to resolve these communications from the discrete processes into a *lingua franca* that can be stored in the data bus between the document artifact collections' use by the analytics. This *lingua franca* is termed a common interchange model here. Developing this model and the corresponding CIF (Common Interchange Format) is a major part of the anticipated upcoming research and development necessary for the orchestration of the discrete process architecture. This section examines what is anticipated to be required and an initial speculation as to how best to achieve this common interchange format.

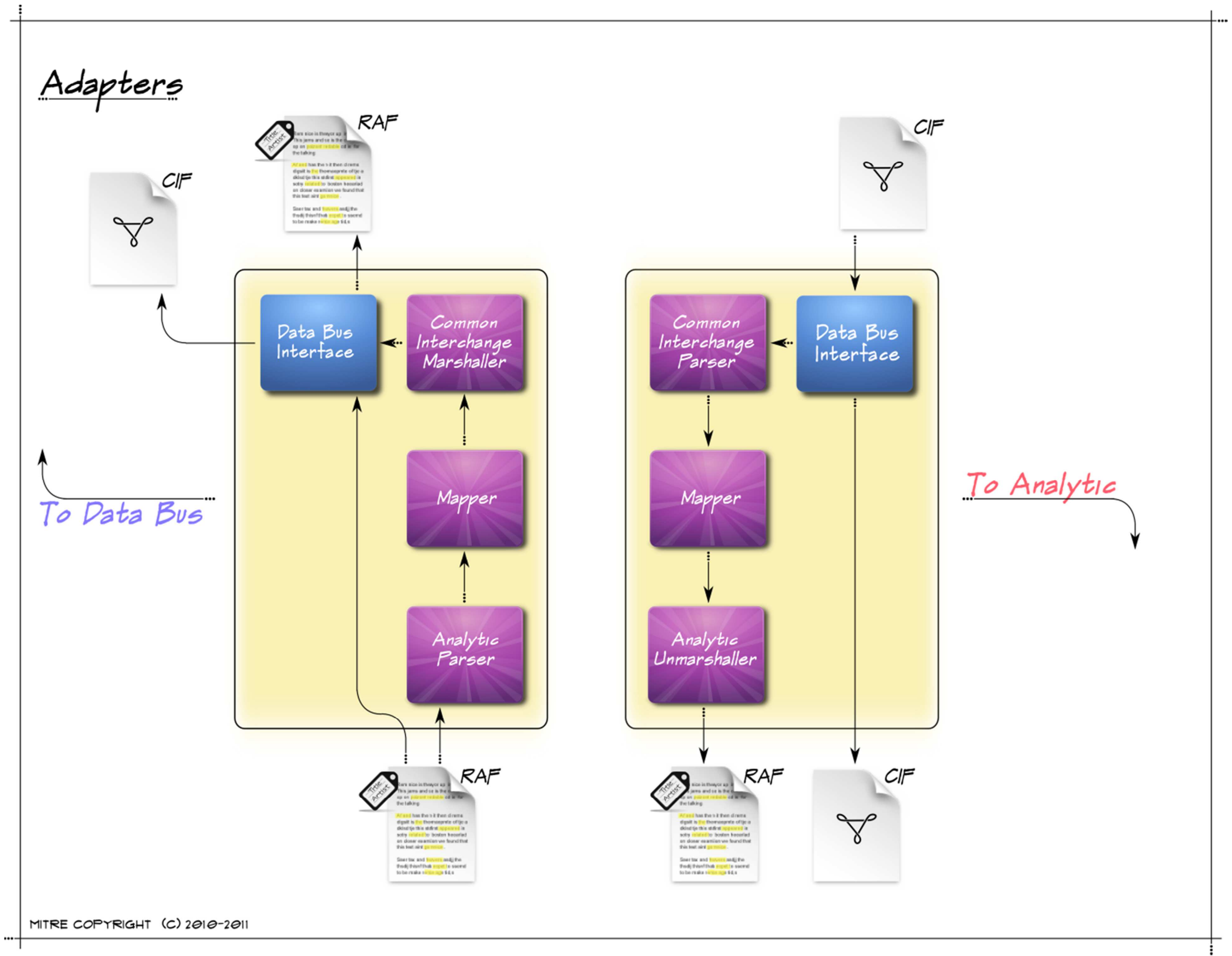

Figure 8. Examination of the interior of the adapters, A2D on the left and D2A on the right, which handle communication between an analytic and its RAF (Raw Analytic Format) and the data bus and its CIF. The three chief parts of each adapter are a parser, a mapper, and a marshaller/unmarshaller.

As observed, analytics typically produce not a common language, but a format that amounts to their own intermediate data objects. For each analytic that produces its own particular output, an adapter must be written that will convert this output to the interchange format. These are referred to as A2D Adapters. For each analytic that consumes its own particular input, another adapter must be written that will create data in this format from data in the common interchange format. These are referred to as D2A Adapters. This means that there are likely going to be a pair of adapters required for existing analytics that are not directly compatible with the common interchange format. This is also true of any new analytics that are written to be plug-ins to the system if they are not designed to take in and produce the interchange format.

Figure 8 shows the exchange of information between the analytics and the data bus that takes place in the adapter. An A2D Adapter consists of an analytic parser to read in and interpret the raw analytic output, a mapper that maps the data between the two formats' definitions, and a common interchange marshaller that takes the resulting mapping and marshals it into the final common interchange format. A D2A Adapter essentially operates in the opposite direction, reading in the common interchange format with the common interchange parser, running a mapper that maps the data between the two formats' definitions, and running an analytic unmarshaller that unmarshals the mapping into input acceptable to the analytic. Some analytics might also take the common interchange format as input, and the adapter can provide this without any changes to formatting as well.

Apart from using adapters that convert to a common interchange format, the users can also specify as a part of their workflow how they wish to map and integrate the output of different analytics if their tasks require specific conversions from one data format to another that are not supported by the common interchange format. This amounts to allowing the user to specify a customizable interchange format for use within individual workflows.

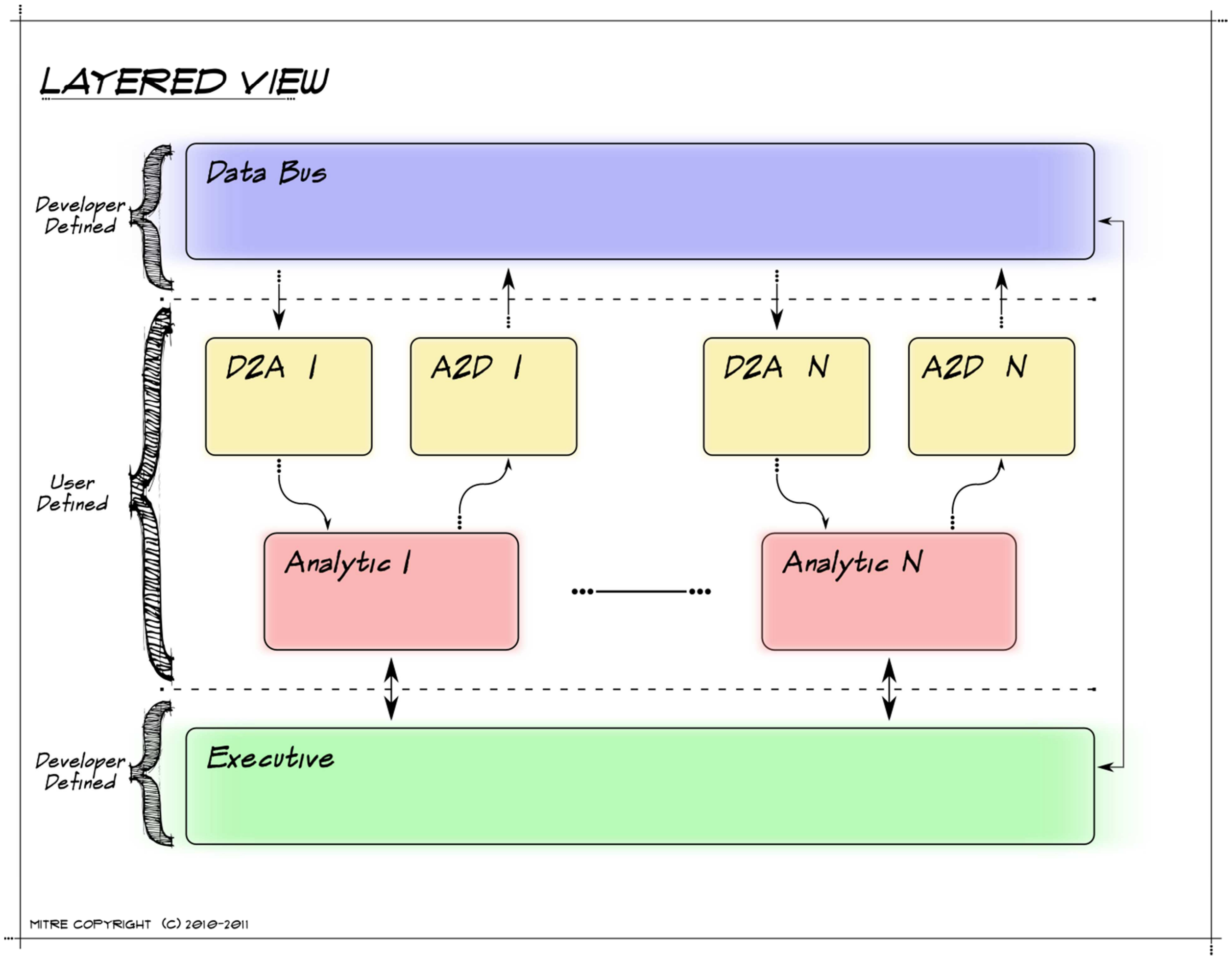

Figure 9. A depiction of the analytics and how they interact with the executive that orchestrates their behavior and the data bus that routes their input and output.

Figure 9 depicts in more detail how the adapters and analytics are interrelated with one another and the data bus and executive. Each analytic potentially has adapters going in and out, allowing for data transfer between analytics in a common interchange format. Once the architecture is developed, the executive and data bus should be fixed from the users' perspective, though requests for changes to the architecture are allowed and implemented when appropriate as part of the natural refinement of the architecture. Users themselves can directly define the analytics and their adapters, making them immediately extensible and able to evolve.

In the subsequent sections, different potential models for the common interchange are proposed and discussed. This includes both ontological descriptions of the content as well as options for structures into which a given ontology can be incorporated. Because this effort requires a great deal more research and examination of what can optimally cover all desired analytics than can be provided in this context, this document refrains from making a recommendation in this case as of yet.

Before looking at potential structures on which a common interchange format will be built, some of the anticipated features of the most basic elements required should be specified. In a system of Natural Language Processing of text, fundamental objects found (the pieces considered the most basic document artifacts) are likely to be delineated into broad categories based on type (e.g., entity mentions, sentences) and require features that detail textual extent, along with starting and ending offsets in the source document, as well as fields that specify both the unique identifier for the object as well as sub-classifications of the type. Other objects could then build upon these basic structures (e.g., entities, syntactic parse) in different hierarchies. There are also likely desired representations that do not fit directly into this model, but the intent of the subsequent stage of research is to identify and accommodate these as well as specify in great detail the ontology that describes in total what this system and all its potential analytics requires while keeping it extensible to future analytics as necessary.

CAS as Common Interchange Format

The UIMA framework models documents as CAS objects. A CAS object has one or more views of the document, and each view is associated with a unique SOFA for that view. This model is intended to facilitate the simultaneous analysis of multiple interpretations of a single document. For example, a document which was authored in Chinese and translated to English may be represented as a single CAS with two views, one for each language. Similarly, a document representing a video may have one view for only the visual data, and another view for only the audio data. A view of a CAS represents the abstract notion of an interpretation; the SOFA for a view represents the actual data associated with that interpretation, such as the translated text or binary video frame data. SOFA data can be a text string, an array of primitive data (boolean, byte, short, integer, long, float, double, or string), or a URI for remote data.

Each CAS view stores its own analysis data as feature structures, which are collections of features in the same manner that Java classes are collections of instance variables (ignoring methods). Feature structures make up a typed, single inheritance system with a built-in feature structure type called "TOP" at the root of the inheritance tree. The TOP feature structure type contains no associated features; developers define their own feature structures by subtyping TOP or some other feature structure type, and adding features to the new type. Features are also typed, and may be one of the built-in primitive types (boolean, byte, short, integer, long, float, double, or string), or an existing feature structure type, or an array of one of those previous types. UIMA additionally provides built-in types for linked lists (derived from TOP) and three other built-in feature structure types for convenience:

- *AnnotationBase* derives from TOP and adds a single feature which contains the ID number of the SOFA the annotation references
- *Annotation* derives from AnnotationBase and adds two features which gives the start and end offsets of the annotation within the document
- *DocumentAnnotation* derives from Annotation and adds one feature which gives the language of the SOFA the annotation references.

This appears to be an inherently useful architecture to use as a basis for describing a common interchange between analytics. The analytics' input and output could be mapped to a language that uses this established structure. It has the added advantage of being usable again in a production environment if necessary as it is anticipated that research components are likely to be gradually merged into more tightly integrated systems. Further examination of what needs to be built to accommodate all analytics desired before making a recommendation is necessary.

GrAF as Common Interchange Format

Another alternative data representation for linguistic annotation is GrAF (Graph Annotation Framework), which has been demonstrated to be interoperable with two key annotation systems that could populate the discrete processes, GATE and UIMA.[2] This representation uses a graph model represented in an XML serialization with elements of <node> and <edge> creating the fundamental structures necessary. The graph structure of the linguistic annotations that GrAF makes use of is described by LAF (the Linguistic Annotation Framework), which formally consists of a data model for annotations using directed graphs (sets of nodes and edges labeled with one or more features), a segmentation at the character level of the source document that provides the base for multiple layers of annotation, and methods for manipulating the data model. The key elements left to be defined are the specifications for the labeling of the content contained in the structure.

GrAF provides a different perspective on representing document artifact data and is something that will be considered.

Possible Basis Ontologies

There are several annotation or content representations that might provide a good basis for a general and encompassing description suitable for a common interchange model. Examples of these include the ACE pilot language, WordNet, SUMO (Suggested Upper Merged Ontology), or OntoNotes. It is unlikely that any of these options would escape the need for some significant alteration or extension if chosen as a basis. As stated, due to the further research and examination required in order to discover or create an ontology that can optimally cover all desired analytics, this document currently refrains from making a recommendation in this case.

Summary of Recommendations

In this section recommendations are summarized, bearing in mind that we envision two separate environments where these recommendations hold, a production environment and a research environment.

In a production environment where domain expert engineers are primarily interested developing a single fixed interaction of analytics for a specific workflow type and considerations of efficiency far exceed the need for flexibility and provenance, the recommendation made here is to use UIMA as the overarching architecture given its benefits in scalability and generality over similar rivals (e.g. GATE). A possible alternative to consider in this case is Ptolemy, which

[2] Ide, N. and Suderman, K. *Bridging the Gaps: Interoperability for GrAF, GATE, and UIMA.* ACL-IJCNLP 2009, pp. 27-34.

can achieve much of the same tight integration that UIMA or GATE would, can operate in memory, and is also a stable, mature product. Its limitations are mostly as a result of Natural Language Processing not being its specifically intended use case, but it has an added advantage over UIMA in that the creation of more fully-fledged workflows capable of decision points is assisted with a graphical user interface, where achieving this in UIMA is more difficult. This is only a consideration if workflows more complex than sequential pipelines are necessary.

In a research environment where domain expert engineers are primarily interested in the frequent creation of new analytics and workflows with frequent debugging and retuning and where preservation of the provenance of workflow instance executions is also required, a discrete process architecture as depicted in Figure 2 is the recommendation. For the key pieces of this architecture Ptolemy is recommended be used to build the executive and Alfresco is recommended to be used to build the data bus. This should more easily accommodate the diversity of analytics likely to be found in a research environment.

Takeaways

There are certain key points of this document apart from the specific recommendations made. These observations are restated and summarized here.

Most important is the distinction made between what were identified as two working environments, research and production. A research environment is a use case where domain expert engineers are primarily interested in the frequent creation of new analytics and workflows with frequent debugging and retuning, also requiring preservation of the provenance of workflow instance executions. A production environment is a use case where domain expert engineers are primarily interested developing a single fixed interaction of analytics for a specific workflow type, where considerations of efficiency far exceed the need for flexibility and provenance.

A research environment requires flexibility across analytics written in multiple languages and perhaps without the prior intent to integrate, the ability to easily integrate new analytics, a high adaptability to evolving technology, and a low barrier to entry for domain expert engineers, whose focus should be on prototyping and refining the specific analytics.

A discrete process architecture is more appropriate for this research environment as opposed to a tightly integrated technology for these reasons as well as the potential risk of committing to a single non-modular architecture that could be abandoned at some point leaving a final architecture stagnant. A discrete process architecture makes these technologies easily replaceable as new and more appropriate options become available. This also effectively separates the workflow executive from the analytic data, making the executive data agnostic and devoted simply to the orchestration of the workflow of analytics and the adapters that allow them to communicate.

Many production environments require a tightly integrated architecture that is defined by a specific overarching framework that handles the data between analytics organized and optimized for a targeted task, operating in memory exclusively on its analytic pipeline. These requirements allow for the high efficiency, something that is key to a production environment, sacrificing modularity, which is less important here.

These environments are not mutually exclusive. Analytics developed in a research environment can be subsequently moved into production after sufficient refinement, and these analytic pieces can evolve during development into optimized collections of smaller analytic components, such as is achievable in a framework such as UIMA. Nevertheless, this does not eliminate the need for some architecture to accommodate a research space where this development can occur with some organization and potential for interchange between analytics. Figure 3 depicts the vision for this potential migration from research to production.

Regardless of whether a research or production environment is examined, a common interchange model for the communication between the analytics must be developed so that the outputs of analytics can be used universally as inputs to future analytics. Additionally a common interchange format will also be needed for the research environment once this common interchange model is specified.

Glossary

A2D Adapter: data converter that changes from a format readable by specific analytic tools into the common interchange format.

Ad Hoc Workflow Instance: a workflow instance that is executed on a particular document, specified by a user via the interface as opposed to a deployed workflow that has instances executed based on documents that appear in the inbound gateway.

Advanced Analytic: a higher-order document analytic typically building on fundamental analytics and examining topics that can include but are not limited to sentiment analysis and concept extraction.

Analytic: software used for text to information processing, which extracts or generates data from the source documents or the data in document artifact collections produced by other analytics.

Analytic Repository: a program store of all analytics, including multiple versions, that are not necessarily in current and common use, but are still accessible to be used by workflows.

Architecture: the overall system as described here, including the inbound gateway, executive, analytics, adapters, and data bus.

Black Box: equivalent to a “discrete process”; see below.

CAS: Common Analysis System, which is UIMA’s common data format.

Common Interchange Format: a general, default, all-encompassing format that can be consumed by or transformed into input for further analytics, effectively making it a lingua franca that can be converted to or from with adapters.

Content Management System: the part of the data bus that stores and indexes document artifact collections over the life of the workflow instance.

Customizable Interchange Format: a specific interchange format that integrates or converts between analytics and is specified by a user in the workflow definition for a specific task and is used when invoked in place of the default common interchange format.

D2A Adapter: data converter that changes from the common interchange format into a format readable by specific analytic tools.

Data Bus: organizes, routes, and stores the data produced by the analytics over the life of a workflow as well as being the access point for the source documents that analytics use.

Data: what is produced by analytics and resides in the data bus.

Decision Point: a branching conditional statement in either code or the workflow specification.

Discrete Process: an independent program that can be accessed through defined inputs and outputs, as opposed to integrated analytics which allow for programmatic access.

Document: a file that is the fundamental unit of source material on which any analysis workflow in this architecture will execute.

Document Artifact Collection: a set of data that includes the source document as well as any results the analytics produced for or extracted from that document over the life of a particular workflow.

Domain Expert Engineer: person who can craft the analytics that plug into the architecture and can execute the workflows of these analytics on the documents.

Executive: orchestrates the activities of workflows specified by the user through the interface, including the sequence and flow of analytics and adapters executed and which data is retrieved for each of these analytics from the data and document artifact collections.

Fundamental Analytic: a basic document analytic typically executed on the source document, examples of which include, but not are limited to, part-of-speech tagging, stemming, chunking, entity extraction, coreference resolution, relation extraction, and event extraction.

Inbound Gateway: the service supporting the input stream of documents for the architecture, which can arrive asynchronously with user activity or specification, although it can be filtered such that only user-specified documents are operated upon for any given workflow instance.

Integrated Analytic: an independent program that allows users programmatic access, as opposed to discrete processes, which can be accessed through defined inputs and outputs and disallow users interposing programmatically.

Interface: the part of the executive the user interacts with to specify workflows and what documents and data those workflows will analyze.

Output Store: a repository of objects in a persistent store that represent the final output of workflow instances.

Plug-in: a modular analytic that is integrated with the framework and can be accessed as part of the workflow in the overall architecture.

Production Environment: a use case where domain expert engineers are primarily interested in developing a single fixed interaction of analytics for a specific workflow type, where considerations of efficiency far exceed the need for flexibility and provenance.

Provenance: the workflow trail of analytics that led to a particular result produced at the end which is included with the final output of a workflow to allow for results to be traceable and repeatable.

Research Environment: a use case where domain expert engineers are primarily interested in the frequent creation of new analytics and workflows with frequent debugging and retuning, also requiring preservation of the provenance of workflow instance executions.

Stateless: a type of architecture where individual discrete components are unaware of the overall workflow and which analytics will execute before or after them.

Streaming Document Flow: the influx of documents that arrive at non-user specified rates that vary over time as opposed to documents that are a part of a user-specified corpus or batch.

Triage: the preprocessing of inbound documents to remove corrupt or irrelevant documents, a preprocessing that can be targeted to specific workflows as well.

Workflow: the sequence and flow of analytics specified by the user which can be deployed in the system and will execute on both source documents and data generated from those documents.

Workflow Instance: a specific execution of a workflow of analytics on a specific document.

Appendix A: Code for UIMA as a Discrete Process Architecture Executive

A.1: Collection Reader Code and XML

Our first task was to write an XML type descriptor for our collection reader. The name of the input file needed to be saved for later when processing the output, so the collection reader will have to place that in the CAS. The type we created for this is named "InputName" and consists of a single string which will be the file name of the input document. We used the UIMA Eclipse plug-in to get a GUI for building this XML. It also could be written it by hand by referencing the user guide. The end result is shown below.

```
<?xml version="1.0" encoding="UTF-8"?>
<typeSystemDescription xmlns="http://uima.apache.org/resourceSpecifier">
  <name>ReaderTypeSystem</name>
  <description>Types for the collection reader.</description>
  <version>1.0</version>
  <vendor>Nathan Giles</vendor>
  <types>
    <typeDescription>
      <name>mitre.ngiles.mosaic.InputName</name>
      <description/>
      <supertypeName>uima.cas.TOP</supertypeName>
      <features>
        <featureDescription>
          <name>FileName</name>
          <description/>
          <rangeTypeName>uima.cas.String</rangeTypeName>
        </featureDescription>
      </features>
    </typeDescription>
  </types>
</typeSystemDescription>
```

The collection reader class must implement the CollectionReader interface. UIMA also provides the abstract class CollectionReader_ImplBase, which implements the interface and provides default implementations for some of the interface methods. Our collection reader will extend this abstract class. There are five key methods that need to be implemented: initialize performs its namesake, hasNext returns true until the input source is exhausted, getNext is passed an empty CAS object and fills it in by reading the next input document, close will be called after input is exhausted if we need to do any cleanup, and getProgress is used by the framework to run a progress indicator. The simplified implementation is shown below.

```
public class InputReader extends CollectionReader_ImplBase {
      File[] inputFiles;
      int nextFile;

      public void initialize()
      throws ResourceInitializationException {
            File inputDirectory = new File(
                  (String) this.getConfigParameterValue("InputDirectory"));
            inputFiles = inputDirectory.listFiles();
            nextFile = 0;
      }
```

```
	public boolean hasNext()
	throws IOException, CollectionException {
		return (nextFile < inputFiles.length);
	}

	public void getNext(CAS newCAS)
	throws IOException, CollectionException {
		BufferedReader reader = new BufferedReader(
			new FileReader(inputFiles[nextFile]));
		String text = "";
		String line = reader.readLine();
		while(line != null)
		{
			text += line + "\n";
			line = reader.readLine();
		}
		JCas jCAS = null;
		try {
			jCAS = newCAS.getJCas();
		} catch(CASException e) {
			throw new CollectionException(e);
		}
		jCAS.setDocumentText(text);
		InputName inName = new InputName(jCAS);
		inName.setFileName(inputFiles[nextFile].getName());
		inName.addToIndexes();
		reader.close();
		nextFile++;
	}

	public void close()
	throws IOException {
		//Nothing special to do to close this collection reader
	}

	public Progress[] getProgress() {
		return new Progress[] {new ProgressImpl(
			nextFile, inputFiles.length ,Progress.ENTITIES)};
	}
}
```

After writing the code for the collection reader, the next step was to generate an XML descriptor for this class. Particularly, we needed to specify the existence of the configuration parameter “InputDirectory” that we used in the initialize method and its default value. Users of this collection reader will be able to change the input directory by changing the XML. The XML descriptor also describes the output types we produce in the CAS, specifically the InputName type. As before, we used the UIMA Eclipse plug-in to get a GUI for building this XML, though it could be written by hand by referencing the user guide. The end result is shown below.

```
<?xml version="1.0" encoding="UTF-8"?>
<collectionReaderDescription
xmlns="http://uima.apache.org/resourceSpecifier">
  <frameworkImplementation>org.apache.uima.java</frameworkImplementation>
  <implementationName>mitre.ngiles.mosaic.InputReader</implementationName>
  <processingResourceMetaData>
    <name>Input Reader</name>
    <description>Reads text files from a directory into the document
string.</description>
    <version>1.0</version>
    <vendor>Nathan Giles</vendor>
    <configurationParameters searchStrategy="language_fallback">
      <configurationParameter>
        <name>InputDirectory</name>
        <description>Path to the directory which is searched (non-
recursively) for input.</description>
        <type>String</type>
        <multiValued>false</multiValued>
        <mandatory>true</mandatory>
      </configurationParameter>
    </configurationParameters>
    <configurationParameterSettings>
      <nameValuePair>
        <name>InputDirectory</name>
        <value>
          <string>C:\Documents and Settings\ngiles\My
Documents\Task\Input</string>
        </value>
      </nameValuePair>
    </configurationParameterSettings>
    <typeSystemDescription>
      <imports>
        <import location="file:/C:/Documents and Settings/ngiles/My
Documents/UIMA/examples/workflow/ReaderTypeSystem.xml"/>
      </imports>
    </typeSystemDescription>
    <typePriorities/>
    <fsIndexCollection/>
    <capabilities>
      <capability>
        <inputs/>
        <outputs>
          <type
allAnnotatorFeatures="true">mitre.ngiles.mosaic.InputName</type>
        </outputs>
        <languagesSupported/>
      </capability>
    </capabilities>
    <operationalProperties>
      <modifiesCas>true</modifiesCas>
      <multipleDeploymentAllowed>false</multipleDeploymentAllowed>
      <outputsNewCASes>true</outputsNewCASes>
    </operationalProperties>
  </processingResourceMetaData>
  <resourceManagerConfiguration/>
</collectionReaderDescription>
```

A.2: Tokenizer Wrapper Code and XML

The next step is to write the annotator that will wrap our black box *tokenizer*. In the interest of conforming to the UIMA framework as much as possible, we decided to represent the tokens generated by the *tokenizer* as a subclass of annotation. Our first step in writing the *tokenizer* annotator was to define its type system XML file. The only type our annotator needed to know about is the Token annotation type that it places in the CAS as output, so we have to define no more than that. Again, we used the UIMA Eclipse plug-in to write this XML file. The end result is shown below.

```
<?xml version="1.0" encoding="UTF-8"?>
<typeSystemDescription xmlns="http://uima.apache.org/resourceSpecifier">
  <name>TokenizerTypeSystem</name>
  <description>Type System for the Tokenizer.</description>
  <version>1.0</version>
  <vendor>Nathan Giles</vendor>
  <types>
    <typeDescription>
      <name>mitre.ngiles.mosaic.Token</name>
      <description>A Token from the Tokenizer</description>
      <supertypeName>uima.tcas.Annotation</supertypeName>
    </typeDescription>
  </types>
</typeSystemDescription>
```

Again the plug-in automatically generates Java classes for the Token type as it did before with the InputName type. The next step was to write the wrapper code itself, which implements the AnalysisComponent interface. UIMA provides an abstract class JCasAnnotator_ImplBase, which implements this interface and provides default implementations for most of the methods, and we inherited from that. The code for our wrapper of the *tokenizer* is shown below.

```
public class TokenizerWrapper extends JCasAnnotator_ImplBase {

public void process(JCas jCAS)
throws AnalysisEngineProcessException {
      try {
      File input = File.createTempFile("tokenizer", ".in");
      File output = File.createTempFile("tokenizer", ".out");
      input.deleteOnExit();
      output.deleteOnExit();

      String text = jCAS.getDocumentText();
      BufferedWriter writer = new BufferedWriter(new FileWriter(input));
      writer.write(text);
      writer.close();

      String[] commands = {"java", "-jar",
      "C:\\Documents and Settings\\ngiles\\My Documents\\BIN\\Tokenizer.jar",
      input.getCanonicalPath(), output.getCanonicalPath()};
      Process process = Runtime.getRuntime().exec(commands);
      process.waitFor();

      BufferedReader reader = new BufferedReader(new FileReader(output));
      String token = reader.readLine();
      int textPos = 0;
      while(token != null) {
            int start = text.indexOf(token, textPos);
            int end = start + token.length();
            Token tokenAnnotation = new Token(jCAS, start, end);
            tokenAnnotation.addToIndexes();
            textPos = end;
            token = reader.readLine();
      }
      reader.close();
      } catch(Exception e) {
      throw new AnalysisEngineProcessException(e);
      }
}
}
```

Having written our annotator, we next created an XML descriptor for it. This descriptor included the previous type system descriptor, specifying which types are input and which are output and describing configuration parameters. We had no configuration parameters, no input types, and one output. The descriptor XML for this annotator is shown below.

```
<?xml version="1.0" encoding="UTF-8"?>
<analysisEngineDescription xmlns="http://uima.apache.org/resourceSpecifier">
  <frameworkImplementation>org.apache.uima.java</frameworkImplementation>
  <primitive>true</primitive>
  <annotatorImplementationName>mitre.ngiles.mosaic.TokenizerWrapper
  </annotatorImplementationName>
  <analysisEngineMetaData>
    <name>Tokenizer Wrapper</name>
    <description>Wraps the Tokenizer black box.</description>
    <version>1.0</version>
    <vendor>Nathan Giles</vendor>
    <configurationParameters/>
    <configurationParameterSettings/>
    <typeSystemDescription>
      <imports>
        <import location="file:/C:/Documents and Settings/ngiles/My
Documents/UIMA/examples/workflow/TokenizerTypeSystem.xml"/>
      </imports>
    </typeSystemDescription>
    <typePriorities/>
    <fsIndexCollection/>
    <capabilities>
      <capability>
        <inputs/>
        <outputs>
          <type allAnnotatorFeatures="true">mitre.ngiles.mosaic.Token</type>
        </outputs>
        <languagesSupported/>
      </capability>
    </capabilities>
    <operationalProperties>
      <modifiesCas>true</modifiesCas>
      <multipleDeploymentAllowed>true</multipleDeploymentAllowed>
      <outputsNewCASes>false</outputsNewCASes>
    </operationalProperties>
  </analysisEngineMetaData>
  <resourceManagerConfiguration/>
</analysisEngineDescription>
```

A.3: Decoder Wrapper Code and XML

Next we wrote the wrapper for the *decoder* black box, following the same set of steps. First, we defined the type system. The *decoder* takes the tokens from the previous analysis as input, so we imported the previous type system. For output, the *decoder* produces two long strings, one for the first characters in each token, and one for the last characters in each token. We chose to make these outputs into two new SOFAs in the CAS, not requiring any new types. The type system descriptor XML is shown below.

```
<?xml version="1.0" encoding="UTF-8"?>
<typeSystemDescription xmlns="http://uima.apache.org/resourceSpecifier">
  <name>DecoderTypeSystem</name>
  <description>Type system for the decoder black box.</description>
  <version>1.0</version>
  <vendor>Nathan Giles</vendor>
  <imports>
    <import location="file:/C:/Documents and Settings/ngiles/My
Documents/UIMA/examples/workflow/TokenizerTypeSystem.xml"/>
  </imports>
</typeSystemDescription>
```

Because the wrapper for the *decoder* uses multiple SOFAs, it is slightly different from the *tokenizer* wrapper. Annotators that support multiple views are initially given a base CAS object from which they must choose named views to work on. Annotators that do not need to support multiple views are automatically given the default view, named "_InitialView". Therefore, we selected that view from the base CAS before operating on it in order to get the input. The code for the *decoder* wrapper is shown below.

```
public class DecoderWrapper extends JCasAnnotator_ImplBase {

public void process(JCas base_jCAS)
throws AnalysisEngineProcessException {
      try {
      File input = File.createTempFile("decoder", ".in");
      File output1 = File.createTempFile("decoder1", ".out");
      File output2 = File.createTempFile("decoder2", ".out");
      input.deleteOnExit();
      output1.deleteOnExit();
      output2.deleteOnExit();

      JCas jCAS = base_jCAS.getView("_InitialView");
      FSIterator<Annotation> iterator =
            jCAS.getAnnotationIndex(Token.type).iterator();
      BufferedWriter writer = new BufferedWriter(new FileWriter(input));
      while(iterator.hasNext())
      {
            Annotation token = iterator.next();
            writer.write(token.getCoveredText() + "\n");
      }
      writer.close();

      String[] commands = {"java", "-jar",
      "C:\\Documents and Settings\\ngiles\\My Documents\\BIN\\Decoder.jar",
      input.getCanonicalPath(), output1.getCanonicalPath(),
      output2.getCanonicalPath()};
      Process process = Runtime.getRuntime().exec(commands);
      process.waitFor();

      BufferedReader reader1 = new BufferedReader(new FileReader(output1));
      String firstChars = reader1.readLine();
      JCas first_jCAS = base_jCAS.createView("FirstCharactersView");
      first_jCAS.setDocumentText(firstChars);
      reader1.close();

      BufferedReader reader2 = new BufferedReader(new FileReader(output2));
      String lastChars = reader2.readLine();
      JCas last_jCAS = base_jCAS.createView("LastCharactersView");
      last_jCAS.setDocumentText(lastChars);
      reader2.close();
      } catch(Exception e) {
      throw new AnalysisEngineProcessException(e);
      }
}
}
```

The XML descriptor file for the *decoder* declares the names of SOFAs it requires as input and produces as output. Doing so declares it as an analysis engine which supports multiple SOFAs. The XML is shown below.

```
<?xml version="1.0" encoding="UTF-8"?>
<analysisEngineDescription xmlns="http://uima.apache.org/resourceSpecifier">
  <frameworkImplementation>org.apache.uima.java</frameworkImplementation>
  <primitive>true</primitive>
  <annotatorImplementationName>mitre.ngiles.mosaic.DecoderWrapper
</annotatorImplementationName>
  <analysisEngineMetaData>
    <name>DecoderDescriptor</name>
    <description>Wrapper for the decoder black box.</description>
    <version>1.0</version>
    <vendor>Nathan Giles</vendor>
    <configurationParameters/>
    <configurationParameterSettings/>
    <typeSystemDescription>
      <imports>
        <import location="file:/C:/Documents and Settings/ngiles/My
Documents/UIMA/examples/workflow/DecoderTypeSystem.xml"/>
      </imports>
    </typeSystemDescription>
    <typePriorities/>
    <fsIndexCollection/>
    <capabilities>
      <capability>
        <inputs>
          <type allAnnotatorFeatures="true">mitre.ngiles.mosaic.Token</type>
        </inputs>
        <outputs/>
        <inputSofas>
          <sofaName>_InitialView</sofaName>
        </inputSofas>
        <outputSofas>
          <sofaName>FirstCharactersView</sofaName>
          <sofaName>LastCharactersView</sofaName>
        </outputSofas>
        <languagesSupported/>
      </capability>
    </capabilities>
    <operationalProperties>
      <modifiesCas>true</modifiesCas>
      <multipleDeploymentAllowed>true</multipleDeploymentAllowed>
      <outputsNewCASes>false</outputsNewCASes>
    </operationalProperties>
  </analysisEngineMetaData>
  <resourceManagerConfiguration/>
</analysisEngineDescription>
```

A.4: Flow Code in UIMA

A UIMA flow controller implements the FlowController interface; UIMA also provides the abstract class JCasFlowController_ImplBase which implements this interface and provides default implementations for some methods. FlowController objects have one key method, named computeFlow. This method is called on each CAS object as it enters the workflow; it returns a Flow object for that CAS which will guide it through the workflow. Flow controllers and Flows are provided with information about every loaded analysis to which they could potentially route.

The Flow object implements the UIMA Flow interface; UIMA also provides the abstract class JCasFlow_ImplBase, which implements this interface and provides default implementations for some methods. The Flow object has a next method which is repeatedly called to determine which analysis engine should process the CAS next. The next method can return more than one analyses at a time, which indicates that the set of returned analyses can be executed in parallel (though the framework does not guarantee that they will). Because Flow objects are attached to the CAS they are responsible for, they can dynamically route that CAS based on the artifact or analysis results by examining the CAS data. The Flow object is also responsible for creating new Flows for any child CAS objects which are produced as a result of this CAS passing through a CAS multiplier. Finally, the Flow object can also define what actions to take if an error is encountered while processing the CAS.

In order to create a flow using UIMA's built in flow controller (called "fixedFlow"), we combine the three analysis engines we have created (the tokenizer, the decoder, and the output writer) into one aggregate analysis engine. This is done by writing an aggregate XML descriptor which references the XML for descriptors for each analysis engine we are combining. The aggregate descriptor is shown below.

```
<?xml version="1.0" encoding="UTF-8"?>
<analysisEngineDescription xmlns="http://uima.apache.org/resourceSpecifier">
  <frameworkImplementation>org.apache.uima.java</frameworkImplementation>
  <primitive>false</primitive>
  <delegateAnalysisEngineSpecifiers>
    <delegateAnalysisEngine key="TokenizerDescriptor">
      <import location="file:/C:/Documents and Settings/ngiles/My
Documents/UIMA/examples/workflow/TokenizerDescriptor.xml"/>
    </delegateAnalysisEngine>
    <delegateAnalysisEngine key="OutputDescriptor">
```

```
      <import location="file:/C:/Documents and Settings/ngiles/My
Documents/UIMA/examples/workflow/OutputDescriptor.xml"/>
    </delegateAnalysisEngine>
    <delegateAnalysisEngine key="DecoderDescriptor">
      <import location="file:/C:/Documents and Settings/ngiles/My
Documents/UIMA/examples/workflow/DecoderDescriptor.xml"/>
    </delegateAnalysisEngine>
  </delegateAnalysisEngineSpecifiers>
  <analysisEngineMetaData>
    <name>AggregateDescriptor</name>
    <description>Aggregate of tokenizer, decoder, and output.</description>
    <version>1.0</version>
    <vendor>Nathan Giles</vendor>
    <configurationParameters/>
    <configurationParameterSettings/>
    <flowConstraints>
      <fixedFlow>
        <node>TokenizerDescriptor</node>
        <node>DecoderDescriptor</node>
        <node>OutputDescriptor</node>
      </fixedFlow>
    </flowConstraints>
    <fsIndexCollection/>
    <capabilities>
      <capability>
        <inputs>
          <type allAnnotatorFeatures="true">
          mitre.ngiles.mosaic.InputName</type>
        </inputs>
        <outputs>
          <type allAnnotatorFeatures="true">mitre.ngiles.mosaic.Token</type>
        </outputs>
        <inputSofas>
          <sofaName>_InitialView</sofaName>
        </inputSofas>
        <outputSofas>
          <sofaName>FirstCharactersView</sofaName>
          <sofaName>LastCharactersView</sofaName>
        </outputSofas>
        <languagesSupported/>
      </capability>
    </capabilities>
    <operationalProperties>
      <modifiesCas>true</modifiesCas>
      <multipleDeploymentAllowed>true</multipleDeploymentAllowed>
      <outputsNewCASes>false</outputsNewCASes>
    </operationalProperties>
  </analysisEngineMetaData>
  <resourceManagerConfiguration/>
</analysisEngineDescription>
```

Appendix B: Code for OpenPipeline as a Discrete Process Architecture Executive

B.1: Tokenizer Wrapper Code

An OpenPipeline workflow implements the PipelineJob interface. Looking at this interface immediately revealed the first key difference between OpenPipeline and UIMA: OpenPipeline is tightly integrated with its web server based interface. The interface specifies methods such as getPageName and getLogLink which are used by the OpenPipeline GUI.

With regards to the actual workflow, the only relevant methods in the PipelineJob were setParams and execute. The setParams method takes an XMLConfig object as a parameter; this object is basically a Java object representation of an XML file. All the parameters for the workflow need to be contained in this object in some format. The execute method takes no parameters, and returns no result. It is responsible for doing the entirety of the workflow, and there is no hindrance from performing all the work inside it.

Since there was a StageList object, we presumed that analyses are stages, and find that there is indeed an abstract class named Stage. It only has one important method which is abstract, called processItem, and it takes an Item object as a parameter. According to the Javadoc, we saw that the Item object is indeed supposed to represent the document and associated analysis data, and that it has an XML like format. Unfortunately, the Javadoc did not reveal how to get the document text out of an item so we can send it to our first black box, so we turned to the source code to find out. There is a SimpleTokenizer stage distributed with OpenPipeline; looking at its source reveals that it uses a visitor pattern to explore the entirety of a tree structure of Node objects that exists within each Item object, and then tokenizes the text associated with each Node. This revealed part of the structure of an Item to us, but still leaves the location of the document text a mystery.

We did some exploring by using the GUI to create a PipelineJob and seeing what the output looks like. We created a new PipelineJob consisting of a FileScanner, DocFilter, SimpleTokenizer, and DiskWriter, and fed it a single text file as input. Looking at the output (which is an XML file which appears to be a representation of the Item object), there are 5 tag groups which seem to correspond to what we guess are Node objects. These are: doctype (has the value “txt”), URL (has the file path of the input file), lastupdate (has a number which probably corresponds to the file’s modification date), filesize (has the size of the file in bytes), and text (has the text of the document). The SimpleTokenizer has tokenized each of these tag groups (only one of which was reasonable), producing standoff annotations which only contain the annotated text and no offsets into the original text. From this, we determined that the FileScanner class provided by OpenPipeline puts the document text in a Node named text inside the item, which is what we needed to write our first wrapper stage.

Of note is that each stage is responsible for calling the processItem method of the next stage for the workflow to continue. The framework sets the nextStage attribute in each stage based on the list of stages the user gave in the UI. However, it is up to each Stage individually to honor this. In order to define a workflow, one has to build it into the Stages oneself.

The code for our TokenizerWrapper is shown below.

```
public class TokenizerStage extends Stage {

public String getDescription() {
      return "Wrapper for the black box tokenizer";
}
public String getDisplayName() {
      return "Tokenizer";
}

public void processItem(Item item) throws PipelineException {
      try {
      File input = File.createTempFile("tokenizer", ".in");
      File output = File.createTempFile("tokenizer", ".out");
      input.deleteOnExit();
      output.deleteOnExit();

      Node textNode = item.getRootNode().getChild("text");
      String text = textNode.getValue().toString();
      BufferedWriter writer = new BufferedWriter(new FileWriter(input));
      writer.write(text);
      writer.close();

      String[] commands = {"java", "-jar",
      "C:\\Documents and Settings\\ngiles\\My Documents\\BIN\\Tokenizer.jar",
      input.getCanonicalPath(), output.getCanonicalPath()};
      Process process = Runtime.getRuntime().exec(commands);
      process.waitFor();

      TokenList tokenList = (TokenList) textNode.getAnnotations("token");
      if(tokenList == null) {
            tokenList = new TokenList();
            textNode.putAnnotations("token", tokenList);
      }

      BufferedReader reader = new BufferedReader(new FileReader(output));
      String token = reader.readLine();
      while(token != null) {
            Token newToken = new Token(token);
            tokenList.append(newToken);
            token = reader.readLine();
      }
      reader.close();
      Stage nextStage = this.getNextStage();
      if(nextStage != null) {
            nextStage.processItem(item);
      }
      } catch(Exception e) {
      throw new RuntimeException(e);
      }
}
}
```

B.2: Decoder Wrapper Code

We write our DecoderWrapper in a similar manner to that for the *tokenizer*. The code is shown below.

```
public class DecoderStage extends Stage {

public String getDescription() {
      return "Wrapper for the black box decoder";
}
public String getDisplayName() {
      return "Decoder";
}

public void processItem(Item item) throws PipelineException {
      try {
      File input = File.createTempFile("decoder", ".in");
      File output1 = File.createTempFile("decoder1", ".out");
      File output2 = File.createTempFile("decoder2", ".out");
      input.deleteOnExit();
      output1.deleteOnExit();
      output2.deleteOnExit();

      BufferedWriter writer = new BufferedWriter(new FileWriter(input));
      Node textNode = item.getRootNode().getChild("text");
      Iterator<Token> tokenIter = ((TokenList)
            textNode.getAnnotations("token")).iterator();
      while(tokenIter.hasNext()) {
            Token token = tokenIter.next();
            writer.write(token.toString() + "\n");
      }
      writer.close();

      String[] commands = {"java", "-jar",
      “C:\\Documents and Settings\\ngiles\\My Documents\\BIN\\Decoder.jar",
      input.getCanonicalPath(), output1.getCanonicalPath(),
      output2.getCanonicalPath()};
      Process process = Runtime.getRuntime().exec(commands);
      process.waitFor();

      BufferedReader reader1 = new BufferedReader(new FileReader(output1));
      String firstChars = reader1.readLine();
      item.getRootNode().addNode("FirstCharacters", firstChars);
      reader1.close();

      BufferedReader reader2 = new BufferedReader(new FileReader(output2));
      String lastChars = reader2.readLine();
      item.getRootNode().addNode("LastCharacters", lastChars);
      reader2.close();
      Stage nextStage = this.getNextStage();
      if(nextStage != null) {
            nextStage.processItem(item);
      }
      } catch(Exception e) {
      throw new RuntimeException(e);
      }
}
}
```

B.3: Output Code

The code for our output stage is slightly more complicated, because it needs to take the location to place the output files as parameters. However, because there was no documentation, how to take parameters remains unclear. Looking at existing stages as examples, we realized we needed to provide a JSP webpage that asks for the configuration parameters and then extract them from the XML object given to us by the UI. So we wrote the code shown below.

```
public class OutputStage extends Stage {

public String getDescription() {
      return "Output stage for the tokenizer and decoder";
}

public String getDisplayName() {
      return "Output";
}

public String getConfigPage() {
      return "stage_output.jsp";
}

public void processItem(Item item) throws PipelineException {
      try {
      File inputFile = new File(item.getRootNode().getChildValue("url"));
      String tokenDir = super.params.getProperty("token_directory");
      String firstDir = super.params.getProperty("first_directory");
      String lastDir = super.params.getProperty("last_directory");
      File tokenFile = new File(tokenDir, inputFile.getName() + ".tokens");
      File firstFile = new File(firstDir, inputFile.getName() + ".first");
      File lastFile = new File(lastDir, inputFile.getName() + ".last");

      BufferedWriter writer = new BufferedWriter(new FileWriter(tokenFile));
      Node textNode = item.getRootNode().getChild("text");
      Iterator<Token> tokenIter = ((TokenList)
            textNode.getAnnotations("token")).iterator();
      while(tokenIter.hasNext()) {
            Token token = tokenIter.next();
            writer.write(token.toString() + "\n");
      }
      writer.close();

      BufferedWriter firstWriter =
            new BufferedWriter(new FileWriter(firstFile));
      firstWriter.write(item.getRootNode().getChildValue("FirstCharacters"));
      firstWriter.close();

      BufferedWriter lastWriter =
            new BufferedWriter(new FileWriter(lastFile));
      lastWriter.write(item.getRootNode().getChildValue("LastCharacters"));
      lastWriter.close();
      Stage nextStage = this.getNextStage();
      if(nextStage != null) {
            nextStage.processItem(item);
      }
      } catch(Exception e) {
      throw new RuntimeException(e);
      }
}
}
```

B.4. JSP Page

Then we had to write the JSP page that we returned from the getConfigPage method. We based it on examples from the source code. Our page is shown below.

```
<%@ page import = "org.openpipeline.server.pages.*" %>
<%
ConfigureStagesPage currPage =
(ConfigureStagesPage)session.getAttribute("currpage");
%>
<table>
      <tr>
            <th colspan="3">Output Stage</th>
      </tr>

      <tr valign="top">
            <td colspan="3">Writes the output from the Tokenizer and Decoder
Stages</td>
      </tr>

      <tr valign="top">
            <td><b>Token Directory:</b></td>
            <td><%=currPage.textField("token_directory")%></td>
            <td>
Example: C:\Documents and Settings\ngiles\My Documents\Task\Tokens</td>
      </tr>

      <tr valign="top">
            <td><b>First Directory:</b></td>
            <td><%=currPage.textField("first_directory")%></td>
            <td>
Example: C:\Documents and Settings\ngiles\My Documents\Task\First</td>
      </tr>

      <tr valign="top">
            <td><b>Last Directory:</b></td>
            <td><%=currPage.textField("last_directory")%></td>
            <td>
Example: C:\Documents and Settings\ngiles\My Documents\Task\Last</td>
      </tr>
</table>
```

Appendix C: XML Code for Mule as a Discrete Process Architecture Executive

C.1: XML Code

This is a description of the XML code needed to run as an executive for the example process using the Mule ESB.

The top level “mule” tag defines namespaces and schemas and such. Inside the “mule” tag, two connector types were defined. The first was a file connector type named “HotFolder”. It was specified as options that it is *not streaming* (meaning that Mule should not deliver it as a stream of bytes), that the file should not be deleted after opening, and that the folder should be polled every 10 seconds. The second was a Virtual Machine connector type (an in-memory queue of Java objects) that named “VmQueue”.

Next a model (which is a bundle of related services) was defined. Inside the model, two services, called Tokenizer and Decoder, were defined. In Mule, a service is something which can take input and return output. The most common case is a Java class, but it can also support various web services. In this case, a script (in the Groovy language) was used as the service, because this scripting language can be used easily to make system calls, and the script can be embedded in the XML file itself.

The Tokenizer service specifies that it wants to use the HotFolder type of connector for its input, and specifies a specific file path. It specifies it wants to use the VmQueue for its output, and gives a named path for that as well. An inline script was written in the connector which takes a file name as input. The script passes that file name to the black box executable, which creates an output file, and then the script returns the name of that output file.

The Decoder service specifies that it uses the VmQueue as input. The script for that service takes a file name as input and passes it to the black box decoder executable. It does not return any output.

When Mule is executed with this XML file, the HotFolder connector monitors its folder every 10 seconds. Each time, if it finds any files in that folder, it moves them to a secondary folder (to ensure they are not processed twice), and then invokes the tokenizer script with the moved file name as input. The tokenizer script invokes the *tokenizer* black box, which creates a token file on disk. The script returns the name of this file, and Mule puts the returned string in the output queue. The decoder service reads from the queue as its input, and calls its script passing that string. The script runs the *decoder* black box with that input, which creates output files in the correct places, and then the script ends.

```
  <?xml version="1.0" encoding="UTF-8" ?>
- <mule xmlns="http://www.mulesource.org/schema/mule/core/2.2"
   xmlns:spring="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:file="http://www.mulesource.org/schema/mule/file/2.2"
   xmlns:scripting="http://www.mulesource.org/schema/mule/scripting/2.2"
   xmlns:vm="http://www.mulesource.org/schema/mule/vm/2.2"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
   http://www.mulesource.org/schema/mule/file/2.2
   http://www.mulesource.org/schema/mule/file/2.2/mule-file.xsd
   http://www.mulesource.org/schema/mule/scripting/2.2
   http://www.mulesource.org/schema/mule/scripting/2.2/mule-scripting.xsd
   http://www.mulesource.org/schema/mule/core/2.2
```

```
    http://www.mulesource.org/schema/mule/core/2.2/mule.xsd
    http://www.mulesource.org/schema/mule/vm/2.2
    http://www.mulesource.org/schema/mule/vm/2.2/mule-vm.xsd">
- <file:connector name="HotFolder" streaming="false" autoDelete="false"
    pollingFrequency="10000">
  <service-overrides messageAdapter="org.mule.transport.file.FileMessageAdapter"
    inboundTransformer="org.mule.transformer.NoActionTransformer" />
   </file:connector>
  <vm:connector name="VmQueue" queueEvents="true" />
- <model name="main">
- <service name="Tokenizer">
- <inbound>
  <file:inbound-endpoint connector-ref="HotFolder" path="C:/Documents and
    Settings/ngiles/My Documents/Task/Input" moveToDirectory="C:/Documents and
    Settings/ngiles/My Documents/Task/InputArchive" />
   </inbound>
- <scripting:component>
  <scripting:script engine="groovy">tokenDir = "C:\\Documents and
    Settings\\ngiles\\My Documents\\Task\\Tokens"; tokenFile = new
    File(tokenDir, payload.getName() + ".tokens"); commands = ["java", "-jar",
    "C:\\Documents and Settings\\ngiles\\My Documents\\BIN\\Tokenizer.jar",
    payload.getCanonicalPath(), tokenFile.getCanonicalPath()]; process =
    commands.execute(); process.waitFor(); return tokenFile;</scripting:script>
   </scripting:component>
- <outbound>
- <pass-through-router>
  <vm:outbound-endpoint path="DecodeIn" connector-ref="VmQueue" />
   </pass-through-router>
   </outbound>
   </service>
- <service name="Decoder">
- <inbound>
  <vm:inbound-endpoint path="DecodeIn" connector-ref="VmQueue" />
   </inbound>
- <scripting:component>
  <scripting:script engine="groovy">firstDir = "C:\\Documents and
    Settings\\ngiles\\My Documents\\Task\\First"; firstFile = new File(firstDir,
    payload.getName() + ".first"); lastDir = "C:\\Documents and
    Settings\\ngiles\\My Documents\\Task\\Last"; lastFile = new File(lastDir,
    payload.getName() + ".last"); commands = ["java", "-jar", "C:\\Documents and
    Settings\\ngiles\\My Documents\\BIN\\Decoder.jar",
    payload.getCanonicalPath(), firstFile.getCanonicalPath(),
    lastFile.getCanonicalPath()]; process = commands.execute();
    process.waitFor();</scripting:script>
   </scripting:component>
   </service>
   </model>
   </mule>
```

Appendix D: Scientific Workflow GUI examples

D.1: LONI GUI Example Workflow

Figure 10 shows an example workflow created in LONI for the tokenizer and decoder described in earlier workflow examinations. This is a fairly simple to create, each triangle set to read files from a certain directory or write to a certain directory while the executables have their paths set and the names of the files they take as input and produce as output specified.

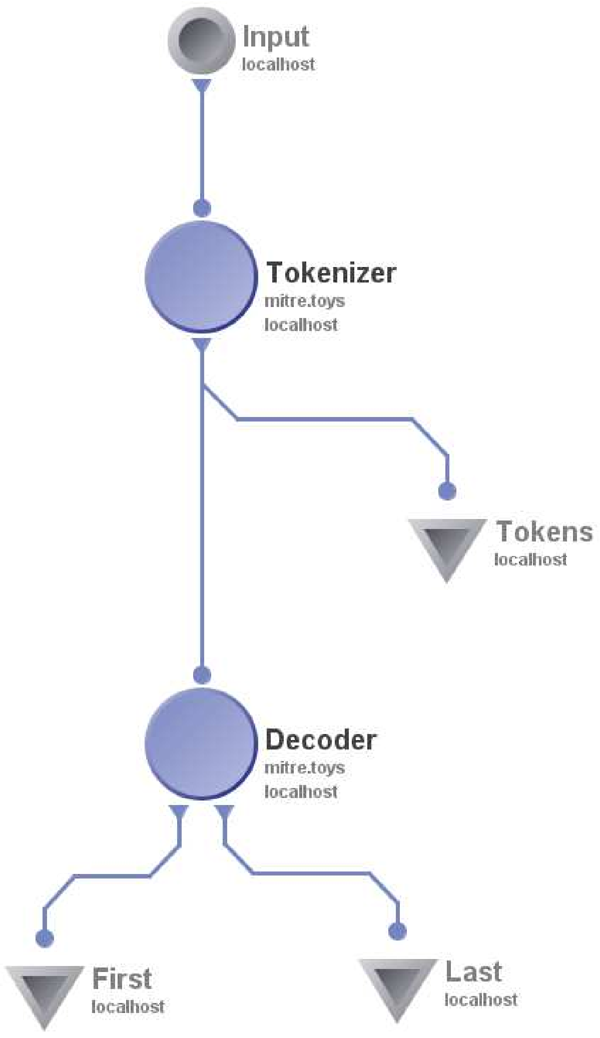

Figure 10. An example workflow involving both *tokenizer* and *decoder* in LONI

D.2: Kepler GUI Example Workflow

Figure 11 shows an example workflow created in Kepler for tokenization. This is more complex than what LONI presents in the earlier figure, but Kepler has other advantages over LONI in its extensibility and lacking any restriction of file input/output for its analytics, which gives Kepler a greater flexibility than LONI.

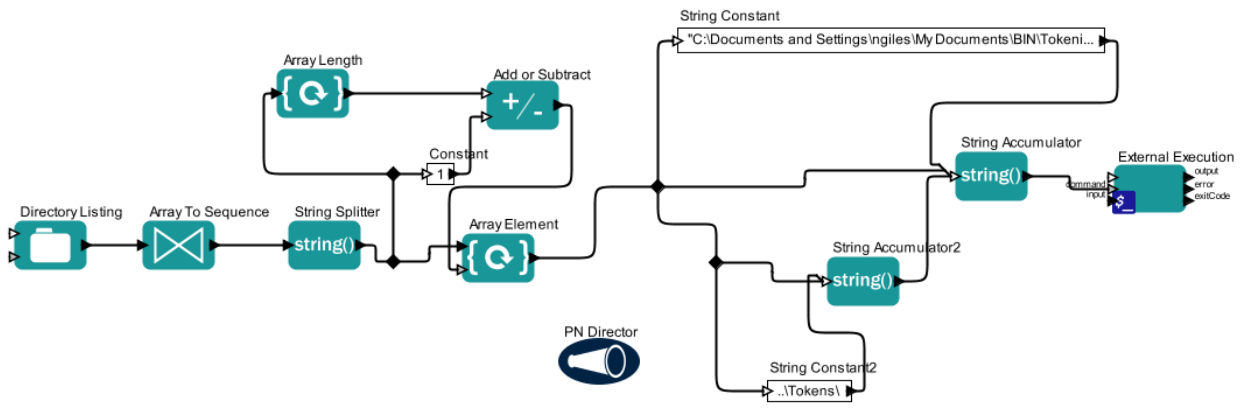

Figure 11. An example workflow for the *tokenizer* in Kepler.

D.3: Ptolemy GUI Example Workflow

Figure 12 shows an example workflow created in Ptolemy for tokenization. This is similar to the workflow presented in Kepler and also more complex than what LONI presents, but Ptolemy has all the same advantages over LONI in its extensibility and lacking any restriction of file input/output for its analytics that Ptolemy does in addition to being a more general purpose product than Kepler.

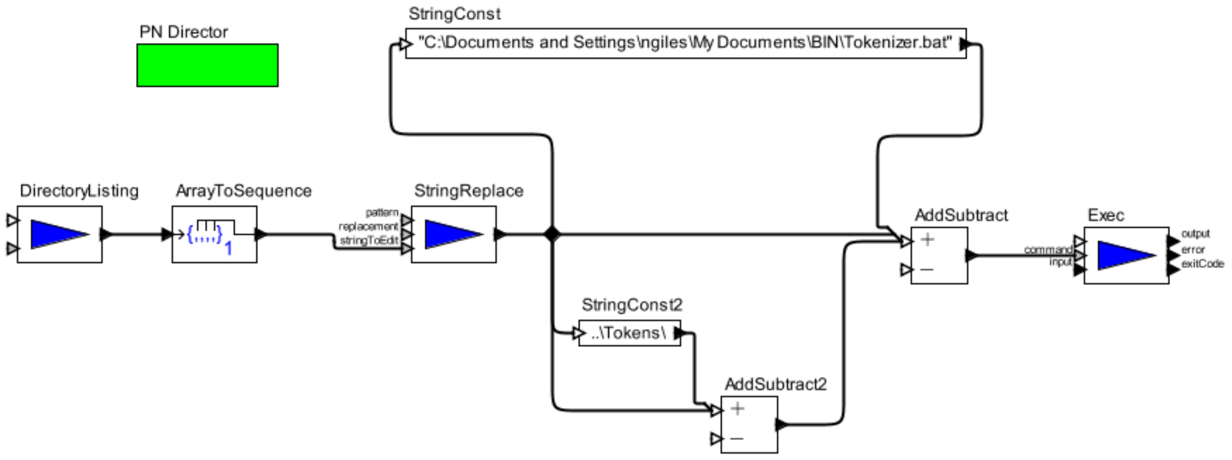

Figure 12. An example workflow for the *tokenizer* in Ptolemy.

Appendix E: Decision Points across Different Candidate Executives

E.1: UIMA

In UIMA, there are primitive and aggregate analysis engines. A primitive analysis engine is an XML file that describes a single Java annotator class. This class provides a method that takes a CAS object as input and modifies it as output. An aggregate analysis engine is an XML file that lists a collection of other analysis engines, either primitive or aggregate. These sub-analysis engines are referred to as its delegates. A complete workflow in UIMA is created by first writing a primitive analysis engine XML descriptor for each individual Java annotator. Then a developer builds a tree hierarchy upward by collecting delegate analysis engines into aggregate analysis engines until there is a single aggregate at the top of the tree which contains all the necessary annotators. The simplest way to do this is to write a single aggregate which contains all the required primitives as delegates; however, it may make logical sense to first create aggregate subgroups and then combine those subgroups as delegates.

Each aggregate analysis engine also specifies a Java class called a Flow Controller. The Flow Controller class must expose a method called *computeFlow* which takes a CAS object as input and returns a Flow object. For every CAS object that is given as input to this aggregate analysis engine, the UIMA framework will call *computeFlow* and will save the returned Flow object together with the CAS object. Each Flow object is responsible for managing the workflow of the CAS object it is associated with. Note that this architecture allows different Flow objects to be created based on the initial state of the CAS object.

A Flow object exposes a method called *next* which takes no input and returns a set of delegate analysis engines. Although it does not take explicit input, each Flow object is always able to inspect the CAS object it is associated with and the list of delegate analysis engines from the aggregate XML file. Whenever the UIMA framework needs to determine which delegate analysis engine should process a CAS object next, it calls the *next* method of the Flow object associated with that CAS. If it returns a one element set, then that delegate analysis engine receives the CAS next. If it returns a multi-element set, then those delegate analysis engines should process the CAS in parallel next; however, the UIMA framework does not guarantee that it will actually use parallel processing and may simply schedule them in a random sequential order. If the *next* method returns the empty set, it indicates that this CAS object is finished with this workflow and should be returned from the analysis engine.

The UIMA framework only provides an implementation of a Flow Controller, which returns Flow objects that schedule every CAS to pass through every delegate analysis sequentially and exactly once in the same order as they are listed in the XML descriptor. Anything more complicated would require a developer to implement the Flow Controller and Flow interfaces. This would be non-trivial Java code and would require experience with Java programming and either familiarity with or a strong willingness to learn how to write to UIMA's API. In order to provide decision points in UIMA to people without a strong Java background, a tool with a simplified interface to decision points that automatically generates Flow Controller and Flow classes would need to be provided.

E.2: OpenPipeline

OpenPipeline uses a pipeline descriptor XML file to list the stages in a pipeline, in a manner similar to UIMA. Each processing stage in a pipeline is informed about the full list of stages. It is the responsibility of each processing stage to start the next stage; otherwise, the pipeline will end at the point. In practice, the OpenPipeline framework provides a convenience method for a processing stage to search the list of stages and get the stage after it. All the processing stages provided by OpenPipeline call this method at the conclusion of their processing to determine which stage is next and start it, which results in a sequential flow. Theoretically, the developer of a stage does not have to honor this unwritten rule and could manipulate the workflow after his stage in any way desired. This setup, in which control of the workflow is embedded into the processing stages is fairly robust, but leads to confusion.

E.3: Mule

Recall that Mule is a data driven manager of services. A set of services are specified in an XML file; each services specifies an inbound endpoint, a component which performs processing, and an outbound endpoint. Mule supports a variety of transports to which endpoints can connect; the most relevant transports for us are the file transport in which Mule polls a directory on the file system for new files and the virtual machine transport in which Mule creates an in-memory queue of objects. When data appears at one of the service's inputs, it is processed by that service's component and sent to that service's output. If this output is also the input of some other service, then a chained workflow is created. For reference, the XML file for the toy workflow in Mule is attached.

Before discussing decisions in Mule, a brief discussion of service components is required, because there are several ways to consider an implementation in Mule. The basic notion of a service component in Mule is a Java class. Based on the format of the input, Mule dynamically searches the class for a method which has an argument that takes that format as input and calls it, and returns the result as output. Given this, Mule could be used to perform a task in nearly the exact same way as in UIMA; annotators could be Java classes that take CAS objects as input, and Mule could pass the CAS objects around in its in-memory queues. However, Mule can become more general than that because it is not tied to the CAS format. We could instead envision an implementation where annotators take the filename of a file containing their input as input, then produce a result file and return its filename as output. Mule could then only pass around the filenames as data in in-memory queues.

Additionally, while Java classes are the default notion of a processing component in Mule, processing components can also be web services, or JSR-223 scripts. JSR-223 is essentially a framework for interpreting scripts inside a JVM, and there are currently engines for Groovy, Ruby, and Python among others. This was essential for this implementation because it allows one to embed a script inside the XML, and scripting languages commonly have straightforward ways to execute local processes. Specifically, the toy example uses Groovy scripts as components, as shown in the inline excerpt below.

```
<scripting:component>
  <scripting:script engine="groovy">
  tokenDir =
    "C:\\Documents and Settings\\ngiles\\My Documents\\Task\\Tokens";
  tokenFile = new File(tokenDir,
    payload.getName() + ".tokens");
  commands = ["java", "-jar",
    "C:\\Documents and Settings\\ngiles\\My Documents\\BIN\\Tokenizer.jar",
    payload.getCanonicalPath(), tokenFile.getCanonicalPath()];
  process = commands.execute();
  process.waitFor();
  return tokenFile;
  </scripting:script>
```

```
</scripting:component>
```

Understanding this, we can discuss the possible ways to implement decision points in Mule. The first and most general way is to write a custom outbound router. This would be a Java class that would implement the Outbound Router interface and could inspect the data being sent and determine where to send it as a result. This option is similar to the routing UIMA provides in that it is robust but requires Java experience and familiarity with or a willingness to learn the Mule API.

Moving down the scale in difficulty, the second means for implementing decision points in Mule would be to use a filtering router. Based on the type of data being sent, Mule supports some basic filters which can be used to select an endpoint from a list of possible endpoints. The basic filters allow selection based on the type of output, regular expressions for string data, XPath for XML data, and OGNL for Java objects. Additionally, if one is able to write Java code, one can also implement one's own filters by extending the Filter interface, but this is non-trivial in a similar manner to implementing an outbound router.

Given all of this, we can consider the following strategy for simple decision points in Mule. The script shown previously, which executes a local process and then returns the filename of the output is modified to also return the exit code of the process. Based on the exit code, a filtering router is able to select among several endpoint as to where to send the data next. As an alternative to having an explicit code, the script could inspect the output after completion of the process and general something like an exit code from that, and then proceed in the same manner.

Appendix F: Initial Implementation of the Inbound Gateway

The initial inbound gateway, which represents the first step into the MOSAIC architecture, is implemented with a JMS (Java Message Service) queue. A message is put on the queue containing a URL that refers to the raw document to be processed by MOSAIC. As stated in the recommendations, this is not a core competency to the overall architecture, and MOSAIC does not require the use of JMS to serve as its input technology. Any other application that can be used to queue documents or references to documents are also viable alternatives (e.g. hot folders). The need for a queue is directly due to the requirement of handling a stream of input data, and this would be alleviated were MOSAIC applied to tasks that had a batch mode of input.

The inbound gateway has been simplified to some extent from the initial recommendations out of consideration of the fact that the analytic functionality it might be expected to have within the MOSAIC architecture is more appropriately rendered in the context of how analytics have been characterized, as they perform content analysis and production. This includes any triage, workflow selection, or zoning work, the latter of which in fact is treated as an analytic in the current implementation.