Aggregating Distributed STT, MT, and Information Extraction Engines:
The GALE Interoperability-Demo System

John F. Pitrelli¹, Burn L. Lewis¹, Edward A. Epstein¹, Martin Franz¹, Daniel Kieczka²,
Jerome L. Quinn¹, Ganesh Ramaswamy¹, Amit Srivastava², and Paola Virgá³

¹IBM T.J. Watson Research Center, Yorktown Heights, NY, U.S.A.
²BBN Technologies, Cambridge, MA, U.S.A.
³{pitrelli,burn,earl,franzm,jlquinn,ganeshr,pvirga}@us.ibm.com, {dkieczka,asrivast}@bbn.com

Abstract

Natural-language-processing engines are now attaining accuracy sufficient to begin combining them to perform more complex tasks. The GALE Interoperability Demo system consists of 12 engines including speech recognition, translation, and various information extraction engines, interoperated to make Arabic news video browsable as English text grouped and parallelized operation of these engines operating on their native computing platforms at their home sites.

Index Terms: interoperation, speech recognition, speech-to-text, entity detection, translation, story-boundary detection, summarization, headline generation, unstructured information, UIMA, NLP, STT, MT

1. Introduction

Recent research advances make the use of NLP systems attractive for new kinds of applications. These include systems that leverage combinations of individual NLP engines for uses that are tolerant of possibly-substantial error rates resulting from engine errors cascading and compounding.

This study describes the GALE® Interoperability Demonstration (IOD) system: a growing aggregate of engines which makes Arabic broadcast news browsable as English-text stories.

By aggregate we refer to a distributed set of engines interoperated via a single-point invocation. This work partly grew out of an IBM system for transcribing, translating, and indexing video and web news; that system exposed various issues such as real-time behavior, remote engine cooperation, error handling, data-format mismatches, and modeling mismatches. The focus here is not primarily on accuracy, but rather to explore challenges posed by the engine interactions in aggregate systems.

IOD currently interoperates the following 12 engines all running at their home sites:

- Speech-to-text (STT) (BBN [1], Carnegie-Mellon U. [2], and IBM [3]),
- Entity detection (ED) (Basis Tech [4], IBM [5]),
- Machine translation (MT) (RWTH Aachen [6], IBM [7]),
- Multi-engine MT (MEMT) (CMU [8]),
- Story-boundary detection (IBM [9]),
- Topic clustering of stories (U. Mass. Amherst [10]),
- Multi-story topic summarization (Columbia U. [11]) and
- Headline generation (IBM)².

While this task requires only one engine for each function, additional engines for the same function are helpful because different engines may be optimal for different inputs, and system-combination techniques can exploit them for improved accuracy [8] [12]. For this reason, IOD is typically run with two or three engines for each of STT, ED and MT. Then MEMT [8] mines the translations resulting from the various sequences of the preceding engines, and assembles a translation which tends to be better than that from any single STT-MT sequence.

The current IOD configuration is shown in Figure 1. IOD has been processing four hours of news daily from two Arabic news networks, Al-Arabiya and Al-Jazeera, for over a year. Results for a current four-day window are browsable at http://rosetta.watson.ibm.com:8888/iod/ which provides a menu of clickable topic headlines enabling drilling down to topic summaries, story headlines, entity mentions, and story translations aligned to video keyframes. A public demo allowing user configuration of aggregates of these engines, the CMU UIMA Component Container (UCC), is linked from http://uima.lti.cs.cmu.edu.

This paper addresses several challenges posed by engine interoperation:

- One engine’s output is typically not formatted in precisely the way that the next engine receives its input,
- Engines are implemented in varying programming languages and operating systems,
- Processing requirements and separately-located teams require engines to run on separate machines, and
- Failures of engines and their network connections, which will occur increasingly frequently as the aggregate grows, must be handled gracefully.

2. Enabling Interoperation: UIMA

IOD is implemented using Unstructured Information Management Architecture (UIMA) [13]. Originally created at IBM Research, Apache UIMA is designed to facilitate interoperation of sets of distributed engines employing heterogeneous computing environments, including Linux, Microsoft Windows, Java, C++, Perl, and Tcl. Communication with remotely-located engines is based on a recent extension, UIMA-AS, which uses the Apache ActiveMQ implementation of Java Message Service

²Details of the headline generation engine are not yet published. In brief, it uses a two-stage approach: (1) using a maximum-entropy model to select words from the input document to be included in the headline, and (2) applying rules based on those words² scores and entity boundaries to extract phrases containing those words from the document.
Create CASes
Create Arabic Text Views
Create English Text View
Create Arabic Video
Story Segmentation
Topic Summarization
Topic Clustering
Headline Generation
CAS Consumer
Browsable Transcript

Figure 1: IOD system diagram. Bold boxes indicate engines; other rectangles represent data-reorganization modules.

(JMS). Each engine is wrapped by a UIMA annotator and deployed as a UIMA-AS service. A provider of an engine service registers its input queue with a JMS broker, typically deployed at the same site as the engine, and provides its UIMA clients with the queue name and broker URL. The client configures an aggregate by specifying the queue and broker for each remote service, along with the necessary data-reorganization modules also implemented as UIMA annotators. UIMA-AS is now part of the Apache UIMA Incubator project [14].

2.1. Type system

UIMA requires a type system to facilitate sharing data among engines. For example, an aggregate including STT can use a data type called AudioToken to represent a word emitted by STT. This type could have a string attribute spelling to hold the word and numerical attributes begin and end for the begin and end timepoints for the word in the audio signal. A type system enables existing non-UIMA engines to be easily UIMA-adapted for addition to an aggregate. For example, an STT engine’s UIMA wrapper can simply create AudioToken objects from the engine’s native word-output format.

Note that the complete data-flow design of an aggregate consists of the type system plus conventions specifying which engines must/may create which object types/attributes.

2.2. Views and data reorganization

A key feature of UIMA is its provision of views, which allows each engine to focus on the data it needs. For example, an array of waveform samples comprises an audio view of the data for STT. Then, a data-reorganization module creates a text view by appending STT output strings, thereby converting the same content to a format suitable for text-processing engines. Such a module maintains cross-references aligning text in the new view with time spans in the audio view, to enable e.g. time-alignment of captions to audio. UIMA stores data for each segment of content in a Common Analysis Structure (CAS) which contains all of the views and annotations of the data. Each engine’s wrapper selects appropriate view(s) from the CAS before invoking the engine. In this way, the interface for text-processing engines is identical whether processing text documents or STT-generated text transcripts of audio segments.

2.3. Context-dependent processing

UIMA provides powerful CAS management features, including a CAS multiplier capability equipping a module to emit newly-created CASes, effectively splitting, merging, delaying, or arbitrarily re-segmenting the input CASes. CAS multipliers can process an input segment while using context beyond the length of the segment, by e.g. processing the second segment before emitting the first, or by creating new segments e.g. based on automatically-detected story boundaries.

2.4. Interfacing to applications

UIMA provides special modules to facilitate integrating an aggregate into an application. These modules are typically located at the head and at the tail of the aggregate. Specifically, a collection reader begins processing by creating a CAS from input data, typically starting with one view such as audio for each data segment. The aggregate ends with one or more CAS consumers, which extract data from the CAS into a format useful to the application.

2.5. Pipelining and parallelism

UIMA supports back-to-back processing of CASes in an aggregate, allowing multiple CASes to be processed at the same time by different engines, hence improving throughput and latency. In the case of a set of engines with no processing interdependencies, such as multiple STT engines, UIMA allows them to be invoked in parallel, reducing latency. A CAS is sent to all of them at once, and UIMA merges the results received from each.

2.6. Clients sharing services, and load balancing

UIMA’s adoption of JMS messaging enables each server to maintain a dedicated processing queue. This enables a service to serve multiple clients. JMS messaging enables multiple instances of a service, running on the same machine or different machines, to be deployed to serve the same queue. This feature facilitates load balancing: more-computationally-intensive and more-demanded services can be provided by multiple servers operating independently.

2.7. Fault handling and flow control

Inevitably a remote engine will become unavailable on occasion, whether due to over-demand, machine or algorithmic failure at the engine’s site, or network failure. UIMA allows clients to specify time-outs on processing, and a choice of actions to take. For a critical engine, a typical action is to terminate processing altogether. However, in the case of failure of one of
multiple engines serving the same function, the failed engine can be automatically removed from the aggregate and processing can continue without it. In this way, multiple engines of the same type provide fault tolerance. In the extreme case when all engines of one type fail, a customizable UIMA flow controller can terminate processing.

2.8. Engine-call overhead
Remote engine calls use an XML representation of CAS contents. This adds processing overhead: CAS serialization to XML and subsequent deserialization back to a CAS object. Data transmission adds further latency. However, these delays are small relative to the heavy-weight GALE analytics. Data re-organization modules, which are light-weight, execute in-process with the UIMA AS aggregate controller, sharing the in-memory CAS objects and incurring no serialization overhead.

3. Design Principles
Several principles guide the design of IOD:

1. Fault-tolerance: When practical, multiple engines of the same type yield robustness against single-engine failure.
2. General complementarity of engines: Different engines serving the same function typically vary in how well-suited they are to a variety of data, e.g. one STT engine trained on noisier audio than another. Interoperation increases the likelihood that the correct result will be available e.g. for MEMT to select.
3. Specific complementarity: Some engine combinations were originally designed tightly-coupled to work optimally together, e.g. an STT and an MT sharing coordinated statistical language models. Such mutual optimization should be exploited for accuracy when possible. However, tight integration at times conflicts with IOD’s goal of achieving accuracy and functionality through interoperation. For example, STT-MT integration straddling a data reorganization module may prevent interoperating those engines with other MT and STT engines. Thus, for IOD, specific complementarity is a secondary goal behind interoperability.
4. Interoperation of functions: Benefits enabled by interoperation of complementary functions should be exploited. Potential examples include MT and story-boundary detection exploiting ED outputs, and the triggering of any text-processing engines’ models specific to content originating as speech.

In short, the goal of IOD is to use UIMA to interoporate engines, to exploit the complementary performance and function of engines by imposing minimal constraints upon each engine. Interoperability requires each engine to conform to some input-output specifications which delineate functionality.

4. Realizing IOD Using UIMA

4.1. Views
IOD uses three kinds of views to provide each engine just the input data it requires: an audio view representing a segment as a sequence of waveform samples, Arabic-text views representing the segment as a sequence of Arabic characters, and an English-text view representing it as a sequence of English characters.

4.2. GALE Type System (GTS)
The GALE Type System (GTS) has been created for UIMA aggregates of engines such as IOD. GTS [15] defines types suitable for audio and text views to hold the inputs and outputs of a variety of speech- and text-processing engines.

4.3. Data-reorganization modules
IOD’s data-reorganization modules, shown in Figure 1, include:

- A collection reader taking the audio track of video segments, and initializing CASes with an audio view. Two-minute segments are used to provide sufficient context for audio processing while avoiding excessive latency.
- Another module assembles an Arabic-text view corresponding to each STT engine’s output. These views represent the content as a string concatenated from GTS AudioToken objects that were output from the STT engine. Such views serve as natural input for engines such as MT, which process Arabic text. MT annotates spans of Arabic text, e.g. sentences, with GTS TranslationResults, each of which contains an English text string representing the span’s translation along with alignment information between English and Arabic words or phrases.
- A similar module creates the English-text view by concatenating all TranslationResult strings output by a translation engine, typically MEMT, while maintaining cross-references aligning the English text back to the Arabic text. This allows subsequent monolingual-text functions like summarization and headline generation to focus on English text as their natural input.
- Another module re-segments data from two-minute CASes into story CASes using output of a story-boundary-detection engine. Since stories can span multiple CASes, story segmentation is implemented as a CAS multiplier. Re-segmentation allows subsequent engines, such as topic clustering, to operate directly on the appropriate data, i.e. a complete story.
- The aggregate terminates in a CAS consumer, which formats output data for the Web server mentioned above.

4.4. Context-dependent segment processing
In addition to story segmentation, IOD requires several instances of context-dependent processing of data segments:

- Story-boundary detection’s algorithm requires two minutes of context on each side of any span it processes. Given IOD’s two-minute segments, this means a window of three segments and a one-segment delay: story-boundary detection uses UIMA’s CAS multiplier capability to wait for segment \( N + 1 \) before processing segment \( N \). In addition, it may have to buffer any out-of-order segments resulting from services deployed with multiple instances earlier in the aggregate.
- Another form of context-dependent segment processing is the maintenance of history. The topic clusterer classifies each incoming story CAS according to the clusters it had built from previously-seen story CASes. The summarizer creates a summary of all stories in the same topic cluster as the current CAS. Both engines keep the previous-segment information they need, separated by topic and by a SessionID parameter the client puts in each CAS. SessionID by convention requires a user-ID component in order to maintain separate histories for different users. Currently a four-day window is implemented by adopting a new SessionID for each day’s run, and then re-running the preceding three days
of data starting with the first history-maintaining engine, thereby creating the intended history for the current day’s data. In the future, history-maintaining engines could be equipped with an automatic, parameterized “forgetting” mechanism, so that old topics would eventually be discarded.

4.5. Constraints on engines

Running an engine in the context of such an aggregate imposes constraints which do not apply to running in isolation. Adding a new engine to an aggregate often creates constraints on several other engines. Many constraints are low-level; discussion will be limited to illustrative examples. One pertains to empty strings input to MT, a degenerate case when running in isolation. However, preceding MT with STT means MT will get empty strings, e.g., when one STT finds words where another does not. MEMT requires a full matrix of inputs, so MT must output a string for each input string, even if empty.

Another example can apply to any engine. An engine’s standard output often includes special codes for exceptional situations, such as one MT’s “unk” for an unknown word. Isolated evaluations of engines account for such codes. However, in a growing aggregate, requiring engines to handle others’ codes becomes impractical. In this example, subsequent engines may mistreat all appearances of “unk” as occurrences of the same word, degrading accuracy, so MT needs another schema for handling unknown words, such as attempting a transliteration.

4.6. UIMAfaction case study: BBN STT

The BBN Audio Monitoring Component (AMC) product [1] produces in real-time a continuous rich transcript of foreign-language television broadcasts. Included in the transcript are speaker information, sentence boundaries, and named entities.

The complexity of integrating existing engines into IOD varies depending on a number of factors. For example, the integration of AMC into the IOD/UIMA framework was completed rapidly for several reasons: 1) AMC is a mature product producing XML-formatted transcripts which adhere to a strict schema contract, 2) GTS required only a subset of the AMC schema types and mapping these types was straightforward, and 3) UIMA provided the necessary infrastructure for quick integration, including simple framework installation, an API to generate annotations, and scripting language support. AMC for IOD is hosted at BBN Technologies. Incorporation of AMC’s named-entity detection remains work in progress, due to IOD’s categorization of entity detection as a text-annotating rather than a speech-annotating function. However, all other AMC functionality has been part of the entire history of IOD.

5. Performance Evaluation Issues

Quantitative evaluation of IOD accuracy is complicated by the number of interacting engines, the complexity in obtaining ground-truth data for comparison with results, and the difficulty of determining objective evaluation metrics for some of the text-processing engines. As a result, we have begun to define a formal framework for performance evaluation of engine aggregates such as IOD [16].

6. Conclusions

Natural-language-processing technologies have advanced to the point that large aggregates of NLP engines can provide useful output despite errors compounding across engines. UIMA enables remote interoperation of engines despite computing-platform variations. IOD has demonstrated the interoperability of speech recognition, named-entity detection, translation, story segmentation, topic clustering, summarization and headline generation, by processing four hours of Arabic news video daily and making it available as English text headlines hyper-linked to topic summaries, story headlines, and story translations. Future expansion will include the addition of engines for language identification, speaker identification, and text-to-speech synthesis.

7. Acknowledgements

Thanks to James Allan, Bob Armstrong, Shilpa Arora, Manuel Bardea, Oliver Bender, Eric Brown, Gopi Flaherty, Greg Haneman, Isaac Harris, Abe Ittycheriah, Nanda Kambhatla, Francis Kubala, Alon Lavie, Daben Liu, Benson Margulies, Evgeny Matusov, Kathy McKeown, Justin Merrill, Uma Murthy, Eric Nyberg, Leiming Qian, Eric Riebling, Salim Roukos, George Saon, Barry Schiffman, Sergey Sigelman, Olivier Siohan, David Svoboda, William Wong, and Jian-Ming Xu for their contributions to the IOD system. This work was supported in part by DARPA under contract HR0011-06-2-0001.

8. References