Integrating a Natural Language Message Pre-Processor with UIMA

Eric Nyberg, Eric Riebling, Richard C. Wang and Robert Frederking
Language Technologies Institute
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213 USA
E-mail: {ehn,er1k,rcwang,ref}@cs.cmu.edu

Abstract

This paper describes the use of the Unstructured Information Management Architecture (UIMA) to integrate a set of natural language processing (NLP) tools in the RADAR system. The challenge was to define a common data model and a set of component interfaces for these tools, so that they could be integrated into a single system. The integrated system is used to pre-process each email arriving in the RADAR user's IMAP store. We present a UIMA collection processing engine for RADAR, including a common type system for text analysis results and annotators for each of the NLP tools. The paper also includes an analysis of system performance and a discussion of the lessons learned through use the of UIMA for this integration task.

1. Introduction

This paper describes the use of the Unstructured Information Management Architecture (UIMA) to integrate a set of natural language processing (NLP) components in the RADAR system. The RADAR (Reflective Agent with Distributed Adaptive Reasoning) system is comprised of a set of intelligent agents that assist the user with routine tasks such as email and scheduling [1]. Its initial test domain is conference planning.

RADAR agents include a Calendar Agent, which notices requests for appointments and helps the user to fit them into his or her calendar, and a Briefing Assistant, which extracts important parts of documents such as meeting minutes to provide automatic briefings (Kumar et al. 2007). These two RADAR agents assume that email messages have been pre-processed with text analysis software to recognize important ranges of text (for example, a request for a meeting. or an action item). This pre-processing is accomplished by a large set of both pre-existing, and project-developed, NLP tools.

The architectural challenge was to define a common data model and a set of component interfaces for these tools, so that they could be integrated into a single system. The integrated system is used to pre-process each email arriving in the RADAR user's IMAP store; the output of the NLP tools is stored in the form of *standoff annotations* - data structures derived from text analysis which are stored separately from the text itself. The UIMA framework is used to define a common type system for text analysis results. An annotator wrapper was written for each NLP component in the pre-processor. Each annotator wrapper is responsible for providing input to an NLP component in its native format, and converting the output of the component back into standoff annotations. The annotator wrappers were integrated into a single collection processing engine (CPE). An object referred to as the common analysis structure (CAS) is created for each input message; this structure includes storage for the original text, as well as storage and an index for each annotation type. Components produce instances of annotation types, which are stored in the CAS as it is passed from component to component. The RADAR CPE also includes Collection Reader and CAS Consumer components, which are responsible for reading and writing email messages and their annotations to and from the persistent database storage. The full CPE is depicted in Figure 1.

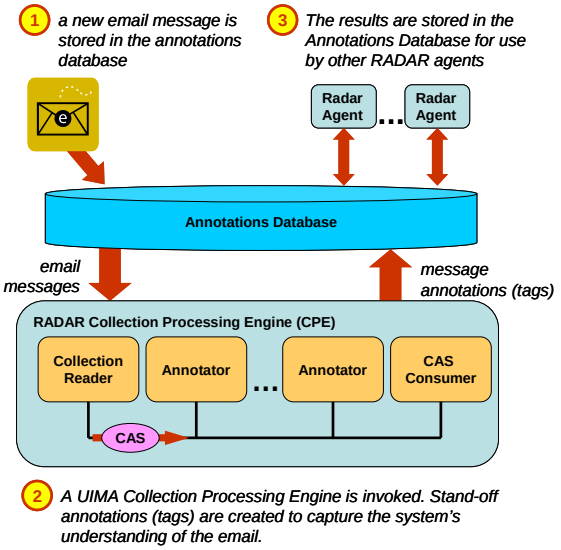

Figure 1: The RADAR Collection Processing Engine

The following sections provide more detail regarding the design and implementation of the RADAR CPE. Section 2 describes the NLP components that were integrated. Section 3 describes the process that was followed to integrate the different modules into the CPE. Section 4

[1] http://radar.cs.cmu.edu/

provides an analysis of system performance and discusses the lessons learned when using UIMA to integrate our suite of NLP components. Section 5 concludes with some suggestions for future work.

2. Annotators & Related Components

This section provides a list of the annotators in the RADAR CPE, including a brief description of each annotator's subtask, the component that it wraps, its input annotation types (if any) and its output annotation types. The annotators are described in the order that they are run for each input email.

2.1 Email Opening Annotator

The Email Opening Annotator identifies the span of text in the message that contains the opening or greeting, e.g. "Dear Blake,". This annotator is implemented by a hand-coded set of surface patterns written in the Mixup language provided by the MinorThird toolkit (Cohen, 2004). It is the first component to process the email text, and it requires no prior input annotations. It outputs instances of a single, simple annotation type with no attributes: EMAIL_HEADING.

2.2 Typo Annotator

The Typo Annotator identifies spans of text in the message that are likely to be misspelled words, and lists alternatives. This annotator is implemented via the open source Jazzy spell checker[2], and it requires no prior input annotations. It outputs instances of a single annotation type called TypoAnnotation, which has two attributes: Typo, the token string containing the spelling error, and Corrections, an array of strings containing proposed corrections, sorted in order of increasing string edit distance from the original token string.

2.3 Connexor Annotator

The Connexor Annotator uses the Connexor parser[3] to mark spans of text denoting sentences, tokens, parts of speech and lemmas for tokens, and functional dependency grammar parses for sentences. This annotator requires no prior input annotations, and outputs instances of three annotation types: ConnexorSentence, ConexorToken (including attributes POS and Lemma), and ConnexorParse (which includes an attribute, Value, which contains a string representation of the Connexor parse analysis).

2.4 Temporal Expression Annotator

The Temporal Expression Annotator identifies spans of text containing temporal expressions such as "next Thursday". This annotator is implemented using MinorThird compiled annotation rules. It outputs a single annotation type, TimeExpression, which includes an attribute, AnchoredValue, containing a canonical time expression for the precise date and time indicated by the surface text, in a format based on ISO8601 (Han et. Al, 2006). Note that a relative time expression like "next Thursday" can only be resolved to an AnchoredValue by calculating its calendar position relative to the time that the email was sent.

2.5 Functional Structure Annotator

The Functional Structure (FS) Annotator processes the information provided by the prior annotations to produce a grammatical functional structure or *f-structure* for each sentence. Each f-structure contains information about the grammatical functions (or roles) expressed in the sentence, such as subject, object, indirect object, etc. The FS Annotator is implemented… (need something from Eric R. here). This annotator requires input annotations EMAIL_HEADING, CXR_PARSE, and TEMPORAL_EXPRESSION, and produces a single output annotation, F_STRUCTURE.

2.6 General Frame (GFrame) Annotator

The General Frame (GFrame) Annotator processes the F_STRUCTURE annotations to produce a general (that is, not domain-specific) semantic frame representation. The GFrame Annotator uses the Mapper component from the KANTOO machine translation system (a general transformation engine for feature structures) (Nyberg et al., 2002), and a set of general (non-domain-specific) interpretation rules. The GFrame annotator outputs a single annotation type, GFRAME.

2.7 Domain Frame (DFrame) Annotator

The Domain Frame (DFrame) Annotator processes the F_STRUCTURE and Gframe annotations for each sentence to produce a domain-specific semantic frame representation, for the conference scheduling domain. The DFrame annotator is implemented using the KANTOO Mapper component and a set of domain-specific interpretation rules. The DFrame annotator outputs a single annotation type, DFRAME.

2.8 Person Name Annotator

The Person Name Annotator identifies possible person names using a Hidden Markov Model trained with the MinorThird toolkit. Outputs a single annotation type, PERSON_NAME_VPHMM.

2.9 RADAR Person Annotator

The RADAR Person Annotator identifies names of known individuals, e.g. "Blake Randal", using a Hidden Markov Model trained with the MinorThird toolkit[4]. Outputs a single annotation type, RADAR_PERSON.

2.10 SCONE Implicit Feature Annotator

The SCONE Implicit Feature Annotator looks up information from the SCONE Knowledge Base for terms in email. Outputs a single annotation type,

[2] http://jazzy.sourceforge.net/
[3] http://www.connexor.eu/
[4] http://minorthird.sourceforge.net/

IMPLICIT_FEATURE.

2.11 SCONE Semantic Annotator

The SCONE Semantic Annotator connects and communicates with a network-based SCONE and SconeGrammar server, retrieving semantic and/or element data relations that SCONE has for a given text. It outputs three annotations: DISCOURSE_STRUCTURE (attributes: SEMANTIC_VALUE, STRUCTURE_SPEC_TYPE), BRIEFING_CONCEPT (attribute: VALUE), and BRIEFING_HEURISTIC (attribute: VALUE).

2.12 Vendor XML Annotator

The Vendor XML Annotator produces a custom XML representation for conference vendor order confirmation and vendor quote e-mails (as for, e.g., food providers). This annotator outputs a single annotation type, VENDOR_XML (with attribute VALUE).

2.13 Space Request Annotator

The Space Request Annotator extracts information from e-mails requesting physical space (office/room/lab space, etc.) and produces a custom XML representation. This annotator outputs a single annotation type, SPACE_XML (with attribute SPACE_REQUEST_VALUE).

2.14 Task Annotator

The Task Annotator identifies overall tasks for the RADAR agents, where tasks are pre-defined in the conference scheduling domain. This annotator outputs a single annotation type, TASK (with attributes TASK_TEMPLATE and TASK_CATEGORY).

2.15 Briefing Annotator

The Briefing Annotator uses 6 sets of trained Minorthird models to guess the likelihood of each email being one of six types of briefing request. (These are used to generate a briefing email to the conference organizer's supervisor.) Returns a single annotation whose value is a string containing the subset of the 6 types deemed possible, as comma separated values: "attendance", "av", "food", "general", "reschedule", "room". This annotator produces a single annotation type, BRIEFING (with attribute BRIEFING_CATEGORIES).

3. RADAR CPE Integration

The components listed in Section 2 were integrated into a single Collection Processing Engine (CPE) for RADAR. In addition to the annotator listed above, two additional UIMA components were required: a) a Collection Reader to read incoming emails from the Annotations Database and convert them into run-time CAS objects, and b) a CAS Consumer to store the annotated CAS objects back into the Annotations Database (see Figure 1).

The annotators in the RADAR CPE include components which are written in Java, and which integrate directly into the UIMA run-time (which is also written in Java). The exceptions are legacy components (Connexor parser, KANTOO Mapper, and SCONE) which are deployed as network services; for these components, the annotator implementation consists of a UIMA wrapper which maintains a network connection to the appropriate network server and takes care of translation to/from the CAS representation when the remote service is used to process the email text.

4. Evaluation

The use of UIMA in deploying the RADAR NLP component architecture was evaluated along three dimensions: overall cost of adoption, measured in programmer effort; run-time performance of the completed system, measured in seconds; and robustness of the resulting implementation, which is discussed in terms of general observations about the system after several months of use.

4.1 Overall Cost of Adoption

The RADAR CPE was integrated by programmer who had already completed a UIMA tutorial and one prior UIMA deployment. The programmer was able to wrap and integrate the 15 annotators listed in Section 2 in about 6 weeks of full-time work. This work was greatly facilitated by the UIMA framework, which allowed the initial deployment to take place very quickly. Remaining concerns about use of UIMA in the longer term are related to robustness of the communication with remote services; see Section 4.4 for further discussion.

4.2 Run-Time Performance

The run-time speed of the annotators (measured over 250 sample messages) is shown in Table 1.

%	Time(ms)	s/doc	Annotator
65.27	5310311	21.24	DFrame
24.60	2001145	8.00	GFrame
2.99	243653	0.97	RADAR Person
2.65	215952	0.86	SCONE Sem.
1.50	122228	0.49	Temporal Expr.
1.03	83563	0.33	Person Name
0.71	57742	0.23	SCONE Impl.
0.54	44187	0.18	F-Structure
0.18	14889	0.06	Email Opening
0.17	13513	0.05	SpaceRequest
0.17	13445	0.05	Conexor
0.07	5835	0.02	Typo
0.06	4746	0.02	CAS Consumer
0.03	2725	0.01	Collection Reader
0.03	2415	0.01	Task
100.00	8136349	32.55	Entire Pipeline

Table 1: Annotator Processing Time, 250 messages

Most of the annotators required less than a second per document, on average. The most time-consuming

annotators are the DFrame and GFrame annotators, which evaluate two different sets of semantic interpretation rules at run time to transform the original functional structure into a final frame output.

4.3 Accuracy

We also evaluated the accuracy of some of the annotators in the RADAR CPE through human evaluation of the output. We randomly selected 50 messages and evaluated whether or not each annotation was correct. The precision (the percentage of annotations that were correct) are shown in Table 2. For comparison, the number of structures which were correct but is also shown.

Annotator	% Correct	% Partly Correct
Vendor Order Annotator	100%	--
Task Annotator	73%	77%
Person Name Annotator	76%	85%
Space Request Annotator	64%	79%

Table 2: Annotator Precision

4.4 Transparency and Robustness

Although UIMA provided excellent support for quickly integrating different NLP components, the most straightforward implementation of legacy components as networked services was not completely robust. The first implementation used direct TCP socket connections to remote server machines, and parsed the low-level string protocols provided by each service. There was no support for process logging or server restart in our implementation, so it became time-consuming to debug system failures when a networked component was involved. For example, if a new, buggy set of KANTOO Mapper rules was deployed for the KANTOO DFrame server, the DFrame Annotator might experience an error state when calling out to the server; the only means of debugging such a failure at present is to manually inspect the log messages on the KANTOO server, and to restart the service manually as required.

5. Conclusion and Future Work

Our adoption of UIMA for integrating the RADAR CPE was an overall success. In only six weeks a single programmer was able to integrate 15 different NLP components into a single pre-processor for incoming email messages. The system includes native Java components as well as remote services integrated via Java wrappers, and reads and writes annotated email messages from a persistent relational store.

In future work, we intend to address the robustness issues with better design for remote NLP services. It would be preferable to integrate such services via a common standard that is already well-supported in Java (for example, WSDL). This would simplify the integration of remote services in RADAR while placing responsibility for standards compliance on the service remote side, rather than the client side. In retrospect, it would have been a cleaner approach to write web service wrappers for each of the remote NLP components before integrating them into UIMA, but this would have taken additional programmer time before a working prototype would have been achieved.

Another possible approach is to deploy third-party annotators as brokered services. This would promote the migration of code for handling native service protocol messages from the UIMA client pipeline out to the remote service. Such a design would provide cleaner separation of responsibility between the service and client, since it does not require the UIMA client pipeline to incorporate low-level details of the third-party service protocol.

In order to improve the transparency and robustness of the system, better logging is required, especially with respect to the operations of remote services that are integrated into the pipeline. We are beginning to investigate the new UIMA-EE framework as a means to achieve better logging in the overall pipeline. Eventually, we hope to build a predictive model of remote service performance that will allow us to dynamically allocate back-end processing nodes for optimal pipeline throughput.

6. Acknowledgements

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. NBCHD030010. We also thank the anonymous reviewers for their helpful comments on an earlier draft of this paper.

7. References

Cohen, William W. (2004). Minorthird: Methods for Identifying Names and Ontological Relations in Text using Heuristics for Inducing Regularities from Data, http://minorthird.sourceforge.net.

Han, Benjamin, Donna Gates and Lori Levin (2006).Understanding temporal expressions in emails. *Proceedings of the Human Language Technology Conference*, Association for Computational Linguistics.

Kumar, M. et al. (2007). Summarizing Non-textual Events with a 'Briefing' Focus. *Proceedings of RIAO*, Centre De Hautes Etudes Internationales D'Informatique Documentaire.

Nyberg, E., T. Mitamura, K. Baker, D. Svoboda, B. Peterson and J. Williams (2002). "Deriving Semantic Knowledge from Descriptive Texts using an MT System", *Proceedings of AMTA 2002*.

Yang, Y. et al. (2005). Robustness of Adaptive Filtering Methods in a Cross-Benchmark Evaluation. *Proceedings of ACM SIGIR*, 98–105. ACM Press.