The Talent System: TEXTRACT Architecture and Data Model

Mary S. Neff
IBM Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598
MaryNeff@us.ibm.com

Roy J. Byrd
IBM Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598
byrd@watson.ibm.com

Branimir K. Boguraev
IBM Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598
bkb@watson.ibm.com

Abstract

We present the architecture and data model for TEXTRACT, a document analysis framework for text analysis components. The framework and components have been deployed in research and industrial environments for text analysis and text mining tasks.

1 Introduction

In response to a need for a common infrastructure and basic services for a number of different, but coordinated, text analysis activities with a common set of requirements, the Talent (Text Analysis and Language ENgineering Tools) project at IBM Research developed the first TEXTRACT system in 1993. It featured a common C API and a tripartite data model, consisting of linked list annotations and two hash table extensible vectors for a lexical cache and a document vocabulary. The experience of productizing this system as part of IBM's well-known commercial product Intelligent Miner for Text (IM4T[1]) in 1997, as well as new research requirements, motivated the migration of the analysis components to a C++ framework, a more modular architecture modeled upon IBM's Software Solutions (SWS) Text Analysis Framework (TAF).

The current version of TEXTRACT that we outline here is significantly different from the one in IM4T; however, it still retains the tripartite model of the central data store.

In this paper, we first give an overview of the TEXTRACT architecture. Section 3 outlines different operational environments in which the architecture can be deployed. In Section 4, we describe the tripartite data model. In Section 5, we illustrate some fundamentals of plugin design, by focusing on Talent's Finite State Transducer component and its interaction with the architecture and data model. Section 6 reviews related work. Finally, we conclude and chart future directions.

2 The TEXTRACT Architecture: Overview

TEXTRACT is a robust document analysis framework, whose design has been motivated by the requirements of an operational system capable of efficient processing of thousands of documents/gigabytes of data. It has been engineered for flexible configuration in implementing a broad range of document analysis and linguistic processing tasks. The common architecture features it shares with TAF include:

- interchangeable document parsers allow the 'ingestion' of source documents in more than one format (specifically, XML, HTML, ASCII, as well as a range of proprietary ones);
- a document model provides an abstraction layer between the character-based document stream and annotation-based document components, both structurally derived (such as paragraphs and sections) and linguistically discovered (such as named entities, terms, or phrases);
- linguistic analysis functionalities are provided via tightly coupled individual plugin components; these share the annotation repository, lexical cache, and vocabulary and communicate with each other by posting results to, and reading prior analyses from, them;
- plugins share a common interface, and are dispatched by a plugin manager according to declared dependencies among plugins; a resource manager controls shared resources such as lexicons, glossaries, or gazetteers; and at a higher

[1] http://www-3.ibm.com/software/data/iminer/fortext/

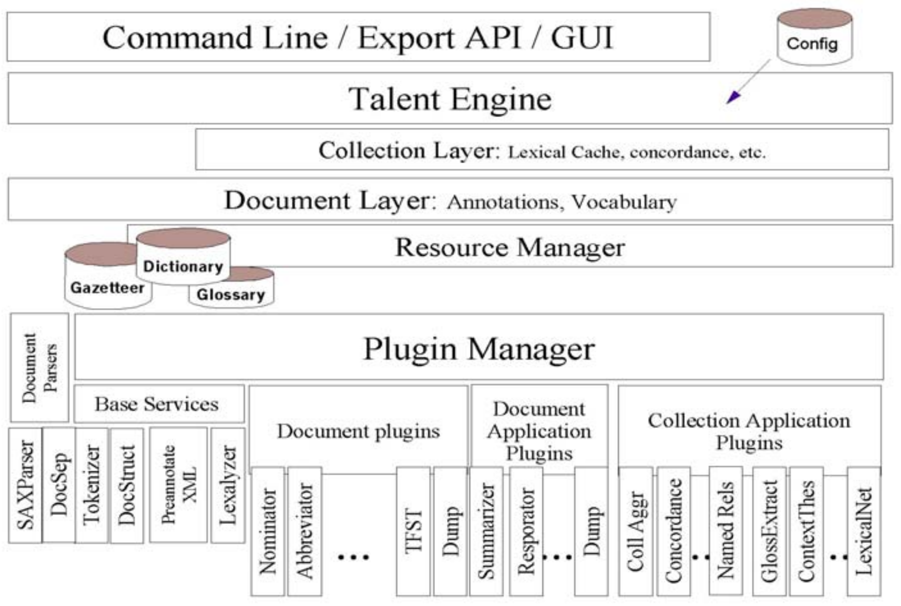

Figure 1: TEXTRACT Architecture

level of abstraction, an engine maintains the document processing cycle;

- the system and individual plugins are softly configurable, completely from the outside;
- the architecture allows for processing of a stream of documents; furthermore, by means of collection-level plugins and applications, cross-document analysis and statistics can be derived for entire document collections.

TEXTRACT is industrial strength (IBM, 1997), Unicode-ready, and language-independent (currently, analysis functionalities are implemented primarily for English). It is a cross-platform implementation, written in C++.

TEXTRACT is 'populated' by a number of plugins, providing functionalities for:

- tokenization;
- document structure analysis, from tags and white space;
- lexicon interface, complete with efficient look-up and full morphology;
- importation of lexical and vocabulary analyses from a non-TEXTRACT process via XML markup;
- analysis of out-of-vocabulary words (Park, 2002);
- abbreviation finding and expansion (Park and Byrd, 2001);
- named entity identification and classification (person names, organizations, places, and so forth) (Ravin and Wacholder, 1997);
- technical term identification, in technical prose (Justeson and Katz, 1995);
- vocabulary determination and glossary extraction, in specialized domains (Park et al., 2002);
- vocabulary aggregation, with reduction to canonical form, within and across documents;
- part-of-speech tagging (with different taggers) for determining syntactic categories in context;
- shallow syntactic parsing, for identifying phrasal and clausal constructs and semantic relations (Boguraev, 2000);
- salience calculations, both of inter- and intra-document salience;
- analysis of topic shifts within a document (Boguraev and Neff, 2000a);
- document clustering, cluster organization, and cluster labeling;
- single document summarization, configurable to deploy different algorithmic schemes (sentence extraction, topical highlights, lexical cohesion) (Boguraev and Neff, 2000a, 2000b);
- multi-document summarization, using iterative residual rescaling (Ando et al., 2000);
- pattern matching, deploying finite state technology specially designed to operate over document content abstractions (as opposed to a character stream alone).

The list above is not exhaustive, but indicative of the kinds of text mining TEXTRACT is being utilized for; we anticipate new technologies being continually added to the inventory of plugins. As will become clear later in the paper, the architecture of this system openly caters for third-party plugin writers.

Specific TEXTRACT configurations may deploy custom subsets of available plugin components, in order to effect certain processing; such configurations typically implement an application for a specific content analysis / text mining task. From an application's point of view, TEXTRACT plugins deposit analysis results in the shared repository; the application itself 'reads' these via a well defined interface. Document application examples to date include document summarization, a customer claims analysis system (Nasukawa and Nagano, 2001), and so forth.

Collection applications have a document analysis component, which may also write to the shared repository. These include named relation extraction (Byrd and Ravin, 1999), custom dictionary building (Park, et al., 2001), indexing for question answering (Prager et al., 2000), cross-document coreference (Ravin and Kazi, 1999), and statistical collection analysis for document summarization or lexical navigation (Cooper and Byrd, 1997).

3 Different Operational Environments

For the purposes of interactive (re-)configuration of TEXTRACT's processing chain, rapid application prototyping, and incremental plugin functionality development, the system's underlying infrastructure capabilities are available to a graphical interface. This allows control over individual plugins; in particular, it exploits the configuration object to dynamically reconfigure specified plugins on demand. By exposing access to the common analysis substrate and the document object, and by exploiting a mechanism for declaring, and interpreting, dependencies among individual plugins, the interface further offers functionality similar to that of GATE (Cunningham, 2002). Such functionality is facilitated by suitable annotation repository methods, including a provision for 'rolling back' the repository to an earlier state, without a complete system `reInit()`.

In addition, the GUI is configurable as a development environment for finite state (FS) grammar writing and debugging, offering native grammar editing and compilation, contextualized visualization of FS matching, and in-process inspection of the annotation repository at arbitrary level of granularity. Figure 2 is broadly indicative of some of the functional components exposed: in particular, it exemplifies a working context for a grammar writer, which includes an interface for setting operational parameters, a grammar editor/compiler, and multiple viewers for the results of the pattern match, mediated via the annotation repository, and making use of different presentation perspectives (e.g. a parse tree for structural analysis, concordance for pattern matching, and so forth.)

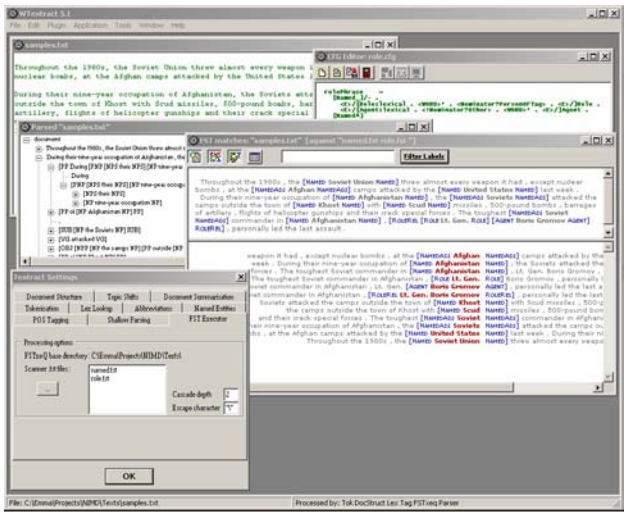

Figure 2: TEXTRACT's GUI

For packaging in applications, Textract has, in addition to native APIs, a C API layer for exporting the contents of the data store to external components in C++ or Java.

4 The TEXTRACT Data Model

The plugins and applications communicate via the annotations, vocabulary, and the lexical cache. The collection object owns the lexical cache; the document object contains the other two subsystems: the annotation repository, and the document vocabulary. Shared read-only resources are managed by the resource manager.

Annotations: Annotations contain, minimally, the character locations of the beginning and ending position of the annotated text within the base document, along with the type of the annotation. Types are organized into families: lexical, syntactic, document structure, discourse, and markup. The markup family provides access to the text buffer, generally only used by the tokenizer. The annotation repository owns the type system and pre-populates it at startup time. Annotation features vary according to the type; for example, position in a hierarchy of vocabulary categories (e.g. Person, Org) is a feature of lexical annotations. New types and features (but not new families) can be added dynamically by any system component. The annotation repository has a container of annotations ordered on start location (ascending), end location (descending), priority of type family (descending), priority within type family (descending), and type name (ascending). The general effect of the family and type priority order is to reflect nesting level in cases where there are multiple annotations at different levels with the same span. With this priority, an annotation iterator will always return an NP (noun phrase) annotation before a covered word annotation, no matter how many words are in the NP.

Iterators over annotations can move forward and backward with respect to this general order. Iterators can be filtered by set of annotation families, types or a specified text location. A particular type of filtered iterator is the subiterator, an iterator that covers the span of a given annotation (leaving out the given annotation). Iterators can be specified to be "ambiguous" or "unambiguous." Ambiguous scans return all the annotations encountered; unambiguous scans return only a single annotation covering each position in the document, the choice being made according to the sort order above. Unambiguous scans within family are most useful for retrieving just the highest order of analysis. All the different kinds of filters can be specified in any combination.

Lexical Cache: One of the features on a word annotation is a reference to an entry in the lexical cache. The cache contains one entry for each unique token in the text that contains at least one alphabetic character. Initially designed to improve performance of lexical lookup, the cache has become a central location for authority information about tokens, whatever the source: lexicon, stop word list, gazetteer, tagger model etc. The default lifetime of the lexical cache is the collection; however, performance can be traded for memory by a periodic cache refresh.

The lexical lookup (lexalyzer) plugin populates the lexical cache with tokens, their lemma forms, and morpho-syntactic features. Morpho-syntactic features are encoded in an interchange format which mediates among notations of different granularities (of syntactic feature distinctions or morphological ambiguity), used by dictionaries (we use the IBM LanguageWare dictionaries, available for over 30 languages), tag sets, and finite state grammar symbols. In principle, different plugins running together can use different tag sets by defining appropriate tagset mapping tables via a configuration file. Similarly, a different grammar morpho-syntactic symbol set can also be externally defined. As with annotations, an arbitrary number of additional features can be specified, on the fly, for tokens and/or lemma forms. For example, an indexer for domain terminology cross references different spellings, as well as misspellings, of the same thing. The API to the lexical cache also provides an automatic pass-through to the dictionary API, so that any plugin can look up a string that is not in the text and have it placed in the cache.

Vocabulary: Vocabulary annotations (names, domain terms, abbreviations) have a reference to an entry in the vocabulary. The canonical forms, variants, and categories in the vocabulary can be plugin-discovered (Nominator), or plugin-recovered (matched from an authority resource, such as a glossary). Collection salience statistics (e.g. tfxidf), needed, for example, by the summarizer application, are populated from a resource derived from an earlier collection run. As with the annotations and lexical entries, a plugin may define new features on the fly.

Resource Manager: The Resource Manager, implemented as a C++ singleton object so as to be available to any component anywhere, manages the files and API's of an eclectic collection of shared read-only resources: a names authority data base (gazetteer), prefix and suffix lists, stop word lists, the IBM LanguageWare dictionaries with their many functions (lemmatization, morphological lookup, synonyms, spelling verification, and spelling correction), and, for use in the research environment, WordNet (Fellbaum, 1998). The API wrappers for the resources are deliberately not uniform, to allow rapid absorption and reuse of components. For performance, the results of lookup in these resources are cached as features in the lexical cache or vocabulary.

5 TEXTRACT Plugins

TEXTRACT plugins and applications need only to conform to the API of the plugin manager, which cycles through the plugin vector with methods for: `construct()`, `initialize()`, `processDocument()`, and `endDocument()`. Collection applications and plugins look nearly the same to the plugin manager; they have, additionally, `startCollection()` and `endCollection()` methods. The complete API also includes the interfaces to the annotation repository, lexical cache, and vocabulary.

5.1 Plugin Example: TEXTRACT's Finite State Transducer

Numerous NLP applications today deploy finite state (FS) processing techniques—for, among other things, efficiency of processing, perspicuity of representation, rapid prototyping, and grammar reusability (see, for instance, Karttunen et al., 1996; Kornai, 1999). TEXTRACT's FS transducer plugin (henceforth TFST), encapsulates FS matching and transduction capabilities and makes these available for independent development of grammar-based linguistic filters and processors.

In a pipelined architecture, and in an environment designed to facilitate and promote reusability, there are some questions about the underlying data stream over which the FS machinery operates, as well as about the mechanisms for making the infrastructure components—in particular the annotation repository and shared resources—available to the grammar writer. Given that the document character buffer logically 'disappears' from a plugin's point of view, FS operations now have to be defined over annotations and their properties. This necessitates the design of a notation, in which grammars can be written with reference to TEXTRACT's underlying data model, and which still have access to the full complement of methods for manipulating annotations.

In the extreme, what is required is an environment for FS calculus over typed feature structures (see Becker et al., 2002), with pattern-action rules where patterns would be specified over type configurations, and actions would manipulate annotation types in the annotation repository. Manipulation of annotations from FS specifications is also done in other annotation-based text processing architectures (see, for instance, the JAPE system; Cunningham et al, 2000). However, this is typically achieved, as JAPE does, by allowing for code fragments on the right-hand side of the rules.

Both assumptions—that a grammar writer would be familiar with the complete type system employed by all 'upstream' (and possibly third party) plugins, and that a grammar writer would be knowledgeable enough to deploy raw API's to the annotation repository and resource manager—go against the grain of TEXTRACT's design philosophy.

Consequently, we make use of an abstraction layer between an annotation representation (as it is implemented) and a set of annotation property specifications which define individual plugin capabilities and granularity of analysis. We also have developed a notation for FS operations, which appeals to the system-wide set of annotation families, with their property attributes, as well as encapsulates operations over annotations—such as create new ones, remove existing ones, modify and/or add properties, and so forth—as primitive operations. Note that the abstraction hides from the grammar writer system-wide design decisions, which separate the annotation repository, the lexicon, and the vocabulary (see Section 3 above): thus, for instance, access to lexical resources with morpho-syntactic information, or, indeed, to external repositories like gazetteers or lexical databases, appears to the grammar writer as querying an annotation with morpho-syntactic properties and attribute values; similarly, a rule can post a new vocabulary item using notational devices identical to those for posting annotations.

The freedom to define, and post, new annotation types 'on the fly' places certain requirements on the FST subsystem. In particular, it is necessary to infer how new annotations and their attributes fit into an already instantiated data model. The FST plugin therefore incorporates logic in its `reInit()` method which scans an FST file (itself generated by an FST compiler typically running in the background), and determines—by deferring to a symbol compiler—what new annotation types and attribute features need to be dynamically configured and incrementally added to the model.

An annotation-based regime of FS matching needs a mechanism for picking a particular path through the input annotation lattice, over which a rule should be applied: thus, for instance, some grammars would inspect raw tokens, others would abstract over vocabulary items (some of which would cover multiple tokens), yet others might traffic in constituent phrasal units (with an additional constrain over phrase type) or/and document structure elements (such as section titles, sentences, and so forth).

For grammars which examine uniform annotation types, it is relatively straightforward to infer, and construct (for the run-time FS interpreter), an iterator over such a type (in this example, sentences). However, expressive and powerful FS grammars may be written which inspect, at different—or even the same—point of the analysis annotations of different types. In this case it is essential that the appropriate iterators get constructed, and composed, so that a felicitous annotation stream gets submitted to the run-time for inspection; TEXTRACT deploys a special dual-level iterator designed expressly for this purpose.

Additional features of the TFST subsystem allow for seamless integration of character-based regular expression matching, morpho-syntactic abstraction from the underlying lexicon representation and part-of-speech tagset, composition of complex attribute specification from simple feature tests, and the ability to constrain rule application within the boundaries of specified annotation types only. This allows for the easy specification, via the grammar rules, of a variety of matching regimes which can transparently query upstream annotators of which only the externally published capabilities are known.

A number of applications utilizing TFST include a shallow parser (Boguraev, 2000), a front end to a glossary identification tool (Park et al., 2002), a parser for temporal expressions, a named entity recognition device, and a tool for extracting hypernym relations.

6 Related Work

The Talent system, and TEXTRACT in particular, belongs to a family of language engineering systems which includes GATE (University of Sheffield), Alembic (MITRE Corporation), ATLAS (University of Pennsylvania), among others. Talent is perhaps closest in spirit to GATE. In Cunningham, et al. (1997), GATE is described as "a software infrastructure on top of which heterogeneous NLP processing modules may be evaluated and refined individually or may be combined into larger application systems." Thus, both Talent and GATE address the needs of researchers and developers, on the one hand, and of application builders, on the other.

The GATE system architecture comprises three components: The GATE Document Manager (GDM), The Collection of Reusable Objects for Language Engineering (CREOLE), and the GATE Graphical Interface (GGI). GDM, which corresponds to TEXTRACT's driver, engine, and plugin manager, is responsible for managing the storage and transmission (via APIs) of the annotations created and manipulated by the NLP processing modules in CREOLE. In TEXTRACT's terms, the GDM is responsible for the data model kept in the document and collection objects. Second, CREOLE is the GATE component model and corresponds to the set of TEXTRACT plugins. Cunningham, et al. (1997) emphasize that CREOLE modules, which can encapsulate both algorithmic and data resources, are mainly created by wrapping preexisting code to meet the GDM APIs. In contrast, TEXTRACT plugins are typically written expressly in order that they may directly manipulate the analyses in the TEXTRACT data model. According to Cunningham, et al. (2001), available CREOLE modules include: tokenizer, lemmatizer, gazetteer and name lookup, sentence splitter, POS tagger, and a grammar application module, called JAPE, which corresponds to TEXTRACT's TFST. Finally, GATE's third component, GGI, is the graphical tool which supports configuration and invocation of GDM and CREOLE for accomplishing analysis tasks. This component is closest to TEXTRACT's graphical user interface. As discussed earlier, the GUI is used primarily as a tool for grammar development and AR inspection during grammar writing. Most application uses of TEXTRACT are accomplished with the programming APIs and configuration tools, rather than with the graphical tool.

Most language engineering systems in the TEXTRACT family have been motivated by a particular set of applications: semi-automated, mixed-initiative annotation of linguistic material for corpus construction and interchange, and for NLP system creation and evaluation, particularly in machine-learning contexts. As a result, such systems generally highlight graphical user interfaces, for visualizing and manipulating annotations, and file formats, for exporting annotations to other systems. Alembic (MITRE, 1997) and ATLAS (Bird, et al., 2000) belong to this group. Alembic, built for participation in the MUC conferences and adhering to the TIPSTER API (Grishman, 1996), incorporates automated annotators ("plugins") for word/sentence tokenization, part-of-speech tagging, person/ organization/ location/ date recognition, and coreference analysis. It also provides a phrase rule interpreter similar to TFST. Alembic incorporates ATLAS's "annotation graphs" as its logical representation for annotations. Annotation graphs reside in "annotation sets," which are closest in spirit to TEXTRACT's annotation repository, although they don't apparently provide APIs for fine-grained manipulation of, and filtered iterations over, the stored annotations. Rather, ATLAS exports physical representations of annotation sets as XML files or relational data bases containing stand-off annotations, which may then be processed by external applications.

Other systems in this genre are Anvil (Vintar and Kipp (2001), LT-XML (Brew, et al., 2000), MATE (McKelvie, et al., 2000), and Transcriber (Barras, et al., (2001). Like ATLAS, some of these were originally built for processing speech corpora and have been extended for handling text. With the exception of GATE, all of these systems are devoted mainly to semi-automated corpus annotation and to evaluation of language technology, rather than to the construction of industrial NLP systems, which is TEXTRACT's focus. As a result, TEXTRACT uses a homogeneous implementation style for its annotation and application plugins, with a tight coupling to the underlying shared analysis data model. This is in contrast to the more loosely-coupled heterogeneous plugin and application model used by the other systems.

7 Conclusion

In this paper, we have described an industrial infrastructure for composing and deploying natural language processing components that has evolved in response to both research and product requirements. It has been widely used, in research projects and product-level applications.

A goal of the Talent project has been to create technology that is well-suited for building robust text analysis systems. With its simple plugin interface (see Section 5), its rich declarative data model, and the flexible APIs to it (Section 4), TEXTRACT has achieved that goal by providing a flexible framework for system builders. The system is *habitable* (external processes can be 'wrapped' as plugins, thus becoming available as stages in the processing pipeline), and *open* (completely new plugins can be written—by anyone—to a simple API, as long as their interfaces to the annotation repository, the lexical cache, and the vocabulary (Section 4), follow the published set of specifications.

Openness is further enhanced by encouraging the use of TFST, which directly supports the development, and subsequent deployment, of grammar-based plugins in a congenial style. Overall, TEXTRACT's design characteristics prompted the adoption of most of the architecture by a new framework for management and processing of unstructured information at IBM Research (see below).

Performance is not generally an inherent property of an architecture, but rather of implementations of that architecture. Also, the performance of different configurations of the system would be dependent on the number, type, and algorithmic design and implementation of plugins deployed for any given configuration. Thus it is hard to quantify TEXTRACT's performance. The most recent implementation of the architecture is in C++ and makes extensive use of algorithms, container classes and iterators from the C++ Standard Template Library for manipulating the data objects in the data model; its performance therefore benefits from state-of-the-art implementations of the STL. As an informal indication of achievable throughput, an earlier product implementation of the tokenization base services and annotation subsystem, in the context of an information retrieval indexer, was able to process documents at the rate of over 2 gigabytes-per-hour on a mid-range Unix workstation.

Allowing TEXTRACT's plugins to introduce — dynamically — new annotation types and properties is an important part of an open system. However, a limitation of the current design is the fixed organization of annotations into families (see Section 4). This makes it hard to accommodate new plugins which need to appeal to information which is either not naturally encodable in the family space TEXTRACT pre-defines, or requires a richer substrate of (possibly mutually dependent) feature sets.

In a move towards a fully declarative representation of linguistic information, where an annotation maximally shares an underlying set of linguistic properties, a rational re-design of TEXTRACT (Ferrucci and Lally, 2003) is adopting a hierarchical system of feature-based annotation types; it has been demonstrated that even systems supporting strict single inheritance only are powerful enough for a variety of linguistic processing applications (Shieber, 1986), largely through their well-understood mathematical properties (Carpenter, 1992).

Some of this migration is naturally supported by the initial TEXTRACT data model design. Other architectural components will require re-tooling; in particular, the FST subsystem will need further extensions for the definition of FS algebra over true typed feature structures (see, for instance, Brawer, 1998; Wunsch, 2003). We will return to this issue in a following paper.

8 Acknowledgements

We acknowledge the contributions of our colleagues, current and former, in the design and implementation of the Talent system and plugins: Rie Ando, Jim Cooper, Aaron Kershenbaum, Youngja Park, John Prager, Yael Ravin, Misook Choi, Herb Chong, and Zunaid Kazi.

References

Ando, Rie K., Branimir K. Boguraev, Roy J. Byrd and Mary S. Neff. 2000. Multi-document summarization by visualizing topical content. *Advanced Summarization Workshop*, NAACL/ANLP-2000, Seattle, WA, April 2000.

Barras, Claude, Edouard Geoffrois, Zhibiao Wu, and Mark Liberman. 2001. Transcriber: development and use of a tool for assisted speech corpora production. In *Speech Communication* (33):5-22.

Becker, Marcus, Witold Drożdżyński, Hans-Ulrich Krieger, Jakub Poskorski, Ulrich Schäfer, Feiyu Xu. 2002. SProUT—Shallow processing with unification and typed feature structures. *Proceedings of the International Conference on Natural Language Processing (ICON 2002)*, Mumbai, India.

Bird, Steven, David Day, John Garofolo, John Henderson, Christohe Laprun, and Mark Liberman. 2000. ATLAS: A Flexible and extensible architecture for linguistic annotation. In *Proceedings of the Second International Conference on Language Resources and Evaluation*: 1699-1706.

Brew, Chris, David McKelvie, Richard Tobin, Henry Thompson, and Andrei Mikheev. 2000. *The XML Library LT XML version 1.2 – User Documentation and Reference Guide,*" available at `http://www.ltg.ed.ac.uk/corpora/xmldoc/release/book1.htm`.

Boguraev, Branimir K. 2000. Towards finite-state analysis of lexical cohesion", In *Proceedings of the 3rd International Conference on Finite-State Methods for NLP*, INTEX-3, Liege, Belgium.

Boguraev, Branimir K. and Mary S. Neff. 2000a. Discourse segmentation in aid of document summarization. In *Proceedings of the 33rd Hawaii International Conference on System Sciences*, Maui, HI, January 2000.

Boguraev, Branimir K. and Mary S. Neff. 2000b. Lexical cohesion, discourse segmentation, and document summarization. In *RIAO-2000*, Paris, April 2000.

Brawer, Sascha. 1998. *Patti: Compiling Unification-Based Finite-State Automata into Machine Instructions for a Superscalar Pipelined RISC Processor,* MA Thesis, University of the Saarland, Saarbrücken, Germany.

Byrd, Roy and Yael Ravin. 1999. Identifying and extracting relations in text. Presented at the *NLDB'99 Conference*, Klagenfurt, Austria.

Carpenter, Robert. 1992. *The Logic of Typed Feature Structures*. Cambridge University Press, Cambridge, England.

Cooper, James and Roy J. Byrd. 1997. Lexical navigation – visually prompted query expansion and refinement. In *DIGILIB 97*.

Cunningham, Hamish, Diana Maynard and Valentin Tablan. 2000. *JAPE: A Java Annotation Patterns Engine.* Research memo CS – 00 – 10, Institute for Language, Speech and Hearing (ILASH), and Department of Computer Science, University of Sheffield, UK.

Cunningham, Hamish, Diana Maynard, Valentin Tablan, Cristian Ursu, and Kalina Bontcheva. 2001 *Developing Language Processing Components with GATE. GATE v2.0 User Guide*, University of Sheffield.

Cunningham, Hamish, Diana Maynard, Kalina Bontcheva, Valentin Tablan. 2002. GATE: A framework and graphical development environment for robust NLP tools and applications. *Proceedings of the 40th Anniversary Meeting of the Association for Computational Linguistics (ACL'02)*. Philadelphia.

Cunningham, Hamish, K. Humphreys, R. Gaizauskas, and Yorick Wilks. 1997. Software infrastructure for natural language processing," in *Proceedings of the Fifth Conference on Applied Natural Language Processing* (ANLP-97).

Grishman, Ralph. 1996. *TIPSTER Architecture Design Document Version 2.2 Technical Report*, DARPA.

Fellbaum, Christiane. 1998. *WordNet, An Electronic Lexical Database*. MIT Press.

Ferrucci, David and Adam Lally. 2003. Accelerating corporate research in the development, application, and deployment of human language technologies. NAACL Workshop on Software Engineering and Architecture of Language Technology Systems**,** Edmonton, Canada.

IBM Corp, *Intelligent Miner for Text Product Overview*, 1997. http://www3.ibm.com/software/data/iminer/fortext/.

Justeson, John S. and Slava Katz. 1995. Technical terminology: some linguistic properties and an algorithm for identification in text. *Natural Language Engineering*, 1(1):9-27.

Karttunen, Lauri, Jean-Pierre Chanod, Gregory Grefenstette and Anne Schiller. 1996. Regular expressions for language engineering. *Natural Language Engineering*, 4(1), pp.305-328.

Kornai, Andras. 1999. *Extended Finite State Models of Language*, Cambridge University Press, Cambridge, UK.

McKelvie, David, Amy Isard, Andreas Mengel, Morten Baun Møller, Michael Grosse, Marion Klein. 2000. The MATE Workbench - an annotation tool for XML coded speech corpora. In *Speech Communication*.

MITRE Corporation. 1997. *Alembic Workbench Users Guide.* available at http://www.mitre.org/technology/alembic-workbench/.

Nasukawa, Tetsuya and T. Nagano. 2001. Text analysis and knowledge mining system. In *IBM Systems Journal* (40:4): 967-984.

Park, Youngja. 2002. Identification of probable real words: an entropy-based approach. In *Proceedings of ACL Workshop on Unsupervised Lexical Acquisition*: pp 1-8.

Park, Youngja and Roy J. Byrd. 2001. Hybrid text mining for finding terms and their abbreviations, In *EMNLP-2001*.

Park, Youngja, Roy J. Byrd and Branimir K. Boguraev. 2002. Automatic glossary extraction: beyond terminology identification. In *Proceedings of the 19th International Conference on Computational Linguistics* (COLING): 772-778.

Prager, John, Eric Brown, Anni Coden, and Dragomir Radev. 2000. Question-answering by predictive annotation. In *Proceedings of SIGIR 2000*: 184-191, Athens, Greece.

Ravin, Yael and Zunaid Kazi. 1999. Is Hillary Rodham Clinton the president? Disambiguating names across documents. In *Proceedings of the ACL '99 Workshop on Coreference and its Applications*, June 1999.

Ravin, Yael and Nina Wacholder. 1997. Extracting names from natural-language text. IBM Research Report 20338.

Shieber, Stuart. 1986. *An Introduction to Unification-Based Approaches to Grammar*, CSLI Lecture Notes, Vol. 4, Stanford University, California.

Vintar, Spela and Michael Kipp. 2001. Multi-track annotation of terminology using Anvil.

Wunsch, Holger. 2003. *Annotation Grammars and Their Compilation into Annotation Transducers*. MA Thesis, University of Tübingen, Germany.