Natural Language Engineering **10** (3/4): 307–326. © 2004 Cambridge University Press
DOI: 10.1017/S1351324904003493 Printed in the United Kingdom

The Talent system: TEXTRACT *architecture and data model*

MARY S. NEFF, ROY J. BYRD
and BRANIMIR K. BOGURAEV
IBM T.J. Watson Research Center, 19 Skyline Drive, Hawthorne, NY 10532, USA
e-mail: {maryneff, roybyrd, bran}@us.ibm.com

(*Received 1 July 2003; revised 18 February 2004*)

Abstract

We present the architecture and data model for TEXTRACT, a robust, scalable and configurable document analysis framework. TEXTRACT has been engineered as a pipeline architecture, allowing for rapid prototyping and application development by freely mixing reusable, existing, language analysis plugins and custom, new, plugins with customizable functionality. We discuss design issues which arise from requirements of industrial strength efficiency and scalability, and which are further constrained by plugin interactions, both among themselves, and with a common data model comprising an annotation store, document vocabulary and a lexical cache. We exemplify some of these by focusing on a meta-plugin: an interpreter for annotation-based finite state transduction, through which many linguistic filters can be implemented as stand-alone plugins. The framework and component plugins have been extensively deployed in both research *and* industrial environments, for a broad range of text analysis and mining tasks.

1 Introduction

Language engineering for large-scale text processing presents many challenging problems, and a number of projects are looking at issues generally subsumed under the heading of architectures and infrastructures for robust, scalable, and configurable document analysis systems; for representative bibliography see, for instance, Cunningham (2002), selected publications in Ballim and Pallotta (2002), and most recently Patrick and Cunningham (2003). Common threads among such projects highlight notions like *componentized architecture*, promoting incremental development and reusability of function, *component interoperability*, allowing for overall system (re-)configurability, *annotation-based representation*, for recording results of component analysis and inter-component communication, and *transparent access to variegated resources*, for streamlining utilization of background knowledge sources.

The Talent (Text Analysis and Language ENgineering Tools) project at IBM Research evolved in response to a need for a common infrastructure and basic services for a number of different, but coordinated, text analysis activities. The first,

https://doi.org/10.1017/S1351324904003493 Published online by Cambridge University Press

prototype, TEXTRACT system grew out of a set of components initially designed for certain text analysis and Natural Language Processing (NLP) functions. Some of the architectural notions listed above found a realization in that early system: annotations over document spans were marked as data structures over string segments, with a doubly-linked list supporting typical scanning regimes; external resources were accessed via an authority file abstraction, or by means of published, external, lexicon API's; components could be swapped in and out of system configurations; and inter-component communication was constrained by means of a tri-partite data model, comprising annotations, lexical cache, and a vocabulary.

In retrospect, designing the system around a set of functions, rather than as an open-ended infrastructure for developing novel linguistic functions and configuring custom applications, resulted in certain deficiencies of the design. Thus, for instance, some knowledge of sequencing – what processes could be assumed to have run at any point – was still necessary for designing new components, largely because of non-declarative knowledge of which annotations were posted by which components. The simple model of annotations, through its very simplicity, made it hard to account adequately for ambiguities, or under-specification, in the analysis. Also, this simplicity placed severe limitations – at least because only straightforward modes of iteration over annotations could be supported – on representing multiple layers of analysis, freely mixing the results of processing at, for example, tokenization, lexical pre-processing, syntactic analysis, and document structure parsing. Dependencies among components were not known externally; thus placing a component at a certain position in the process sequence typically required "inside" knowledge to configure the requisite pre-processors in front.

In the abstract, these are not insurmountable problems. Indeed, tackling them in a principled way was a discovery process during which TEXTRACT evolved more of the design criteria listed in the beginning of this section. However, the characteristics of an industrial research environment not only amplify the requirements underlying such design criteria, but additionally impose further constraints. Among these are:

- the need for *efficiency*, as applications may need to be deployed over massive document collections, coupled with processing granularity capable of supporting *component function development*, outside of low-level code debugging;
- the process of *productization*, with all the implications of handing off research code to a product development group;
- leveraging the contributions from a *massively distributed research organization*;
- reconciling *different I/O behavior* from different, yet functionally identical, components;
- promoting *a common notion of 'basic services'* (e.g. document 'ingestion', tokenization, lexical look-up);
- the ability to layer *progressively more complex linguistic analysis* on top of simpler annotation mark-up, which may require a representational scheme compatible with, but extending beyond, the common stand-off annotation model;

https://doi.org/10.1017/S1351324904003493 Published online by Cambridge University Press

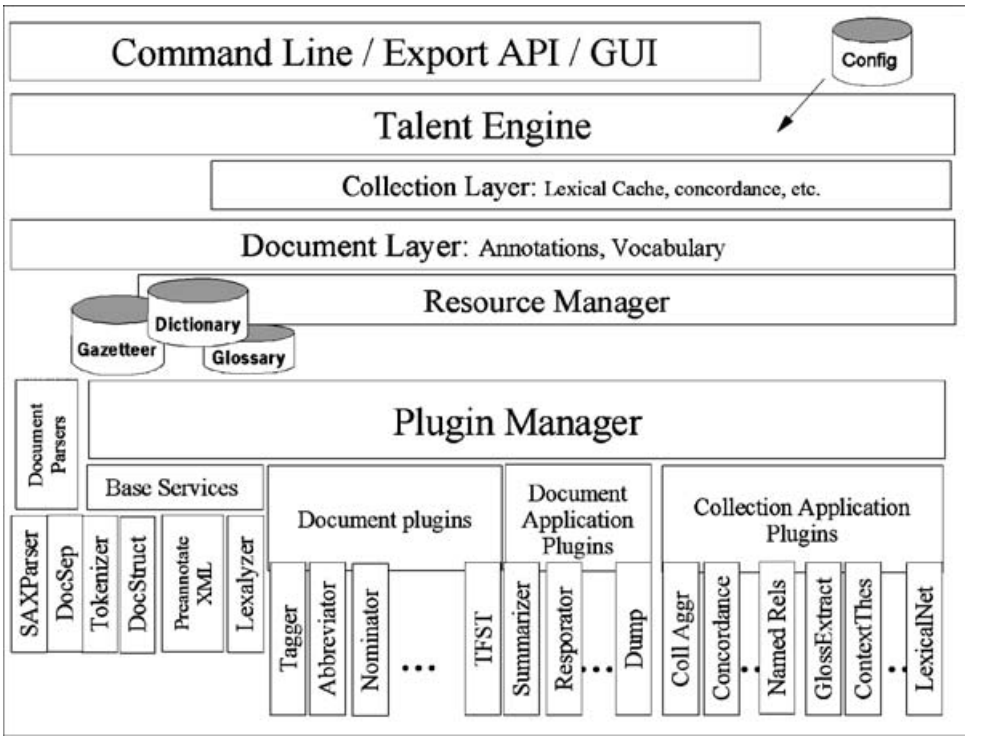


Fig. 1. TEXTRACT architecture.

- the ability to *configure novel applications*, with minimal modification to individual component functions;
- *internationalization*.

In response to such considerations, a rational re-design of TEXTRACT explicitly promoted the notion of centrality of a representationally rich data model, which mediates inter-component communication, access to external resources, caching of lexical and vocabulary analysis, and I/O behaviour, both at macro- and micro-levels.

Inspired by the interaction with the productization group, the re-design adopted a C++ framework, with Standard Template Library (STL) containers managing the components of the data model; this laid the foundation for efficient processing. In the new, highly modular, architecture the notion of a component is realized through the concept of a "plugin". Dedicated architectural elements, an "engine" and a "plugin manager", maintain strict separation among plugins and thus promote robustness, maintainability, interoperability, interchangeability, and incremental development of plugin function. Plugins, while appearing the same to the engine, can be nonetheless viewed broadly as falling into one of three categories (see Figure 1).

Commonly required, and thus underlying any document processing and analysis application, there are plugins which provide *basic services* (such as document ingestion, tokenization and lexical lookup). A large number of plugins are dedicated to performing *linguistic analysis functions* (for example, a part-of-speech tagging

https://doi.org/10.1017/S1351324904003493 Published online by Cambridge University Press

plugin, or a syntactic phrase finder plugin). At the end of processing, such plugins typically post the results of their analysis to a shared annotation repository (see section 2). Finally, there are plugins which provide the *application layer of overall functionality* – in effect, packaging the combined results of a particular set of plugin components. Application plugins largely consume linguistic analyses from the annotation repository.

In this paper, we focus on the design characteristics of TEXTRACT's architecture and data model, as they derive from the joint consideration of the best practices of NL engineering and the complex requirements of a hybrid research and development testbed system, also intended as a multi-application platform deployable outside of its initial laboratory setting. Following an overview of the current TEXTRACT architecture, section 3 presents details of the tri-partite data model. Section 4 outlines different operational environments in which the architecture can be deployed. In section 5, we illustrate some fundamentals of plugin design, by focusing on Talent's Finite State Transducer component and its interaction with the data model and other architectural elements. 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 is thus industrial strength, Unicode-ready, and language-independent (currently, analysis functionalities are implemented primarily for English). As a cross-platform implementation, written in C++, it has been engineered for flexible configuration in implementing a broad range of document analysis and linguistic processing tasks. TEXTRACT is realized as a pipeline architecture (see Figure 1), with plugins freely fitting in, or out of, the pipeline.

Plugin management and control is under the supervision of a *plugin manager*, itself invoked by an *engine* which mediates the environment in which any particular instantiation of TEXTRACT, or a TEXTRACT-based application, is activated. Plugins are dedicated to well-defined linguistic functions, which they perform in their pipeline sequence; they are configurable from the outside; and rather than communicating directly among themselves, they use the Talent data model as a sort of blackboard in which they examine their predecessors' decisions and to which they post their own results.

The data model is distributed between layered abstractions of an *individual document* (which gets refreshed between documents) and a *document collection*. An *annotation repository* (AR) is associated with the document; plugins read earlier analyses from the annotation repository, and post analysis results to it. Certain plugins, focusing on vocabulary items – such as named entities, technical terminology, domain-specific terms, glossary items – write information about such items to a *document vocabulary*. Unlike the annotation repository and the vocabulary, the *lexical cache* persists between documents.

https://doi.org/10.1017/S1351324904003493 Published online by Cambridge University Press

A *resource manager* (implemented as a C++ singleton object, available to any component anywhere) manages the files and API's of an eclectic collection of shared read-only resources. These include a named entity authority file, prefix and suffix lists, stop word lists, IBM's vast repository of dictionaries with their many functions (lemmatization, morphological look-up, synonyms, spelling verification, and spelling correction), and, for use in the research environment, WORDNET (Fellbaum 1999). The API wrappers for the resources are deliberately not uniform, to allow rapid absorption and reuse of components. For performance, the results of look-up in these resources are cached as features in the lexical cache or vocabulary.

In summary, the main architectural elements of TEXTRACT are:

- 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 a 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 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.

2.1 Plugin inventory

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

- tokenization;
- lexicon interface, complete with efficient look-up and full morphology;
- importation of lexical and vocabulary analyses from a non-TEXTRACT process via XML markup;
- document structure analysis, from tags and white space;
- 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);

https://doi.org/10.1017/S1351324904003493 Published online by Cambridge University Press

- technical term identification, in technical prose (Justeson and Katz 1995);
- vocabulary determination and glossary extraction, in specialized domains (Park, Byrd and Boguraev 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 a method of Iterative Residual Rescaling (Ando, Boguraev, Byrd and Neff 2000);
- pattern matching, deploying finite state technology specially designed to operate over document content abstractions (as opposed to a character stream alone) (see section 5.1).

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.

2.2 Applications and deployments

As outlined in section 1, 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 document-level plugins deposit analysis results in the shared repository; the application itself, often implemented using collection-level plugins, "reads" those analysis results via a well-defined interface.

TEXTRACT has been extensively deployed, both as base infrastructure for industry solutions, and in applications within IBM, for IBM customers, and by research partners. As a feature extractor, a TEXTRACT configuration typically comprises plugins for the base services, and any of name/term/abbreviation extraction, vocabulary aggregation, expression extraction, and relation extraction. Feature extraction has been used in tools for query refinement and lexical navigation in document search portals (Cooper and Byrd 1997; Mack, Ravin and Byrd 2001) for IBM consultants and for on-line help for customers of IBM products. In a marketing application, feature extraction operated on the output of the IBM ViaVoice speech recognizer, applied to recorded telephone conversations (Cooper, Viswanath and Kazi 2001), to spot names of people, companies, and products. IBM's entry in the ARDA

https://doi.org/10.1017/S1351324904003493 Published online by Cambridge University Press

ACQUAINT program used Talent feature extraction as part of a question-answering system (Prager, Coden and Radev 2000). Finally, a university research team with whom we collaborate has used Talent feature extraction to support multidocument summarization in an on-line news portal (Schiffman, Nenkova and McKeown 2002).

TEXTRACT-based single-document summarization is composed from plugins for the base services, topic shift analysis, salience calculations, and sentence extraction (Boguraev and Neff 2000a, 2000b). Such summarizer was used in an IBM-internal portal providing document search and retrieval in a data base of documents about IBM products and services (Mack, Ravin and Byrd 2001).

The TEXTRACT configuration for the English shallow parser invokes plugins for the base services, part-of-speech tagging, and finite state parsing with a cascade of grammars (Boguraev 2000). Initially developed for text mining of customer contact records in a customer relationship management system (Nasukawa and Nagano 2001), the shallow parser has been reused in other engagements. These include a system for text mining of biomedical documents and a system, deployed at a major manufacturing company, for analyzing customer contact records for information about product defects and related customer experiences. In ongoing research, the shallow parser is used for advanced feature extraction (e.g. of noun phrases or of grammatical and lexical relations) for various applications of machine learning to NLP.

When configured with plugins for the base services, shallow parser, abbreviation extraction, unknown word processing, and collection-level glossary item aggregation, TEXTRACT becomes a glossary extraction tool, GLOSSEX (Park, Byrd and Boguraev 2002). Applied to a domain corpus, GLOSSEX extracts domain-specific glossaries whose entries can be used as features and vocabulary items in applications already described. Specifically, glossary items have been used for lexical preprocessing in applications based on the shallow parser (Nasukawa and Nagano 2001).

3 The TEXTRACT data model

As discussed in the two preceding sections, TEXTRACT's data model encapsulates a number of representational abstractions designed to capture interrelated levels of linguistic analysis. The plugins and applications communicate via the lexical cache, annotations, and vocabulary. The lexical cache is a system object, accessible through the collection and document layers; it persists throughout collection processing. The annotation repository is a system object also, accessible through the document layer; the annotations are refreshed for each document. The document vocabulary is contained in the document object and is accessible through it (Figure 1). Shared read-only resources are managed by the resource manager (see section 2).

3.1 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: markup, lexical, syntactic, document

https://doi.org/10.1017/S1351324904003493 Published online by Cambridge University Press

structure, and discourse. The markup family provides access to the text buffer, generally only used by the tokenizer. The annotation repository owns the annotation type system and prepopulates 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 (however, see section 7 for current work on extending the data model) – 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, for example, 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. For individual Textract applications, priorities within families – and thus annotation nesting levels – may by configured through an API. In current work on extending the data model, reported in section 7, the priority system and the family/type system will become independent of one another.

Iterators over annotations can move forward and backward with respect to this general order. Iterators can be filtered by sets of annotation families, types or a specified text location. A particular type of filtered iterator is the sub-iterator, an iterator that covers the span of a given annotation (leaving out the given annotation and any of its "parents"[1]). 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.

The design of the annotation subsystem has been informed by considerations of general-purpose annotation formats, discussed, for instance, in Grishman (1996) and Bird and Liberman (2001). Given the specific constraints for TEXTRACT design and utilization (section 1), we have focused primarily on *searchability* and *browsability*, reflected in the notion of filtered iterators and sub-iterators over a number of annotation families. Additionally, our theory-neutral annotations allow for under-specification, ambiguity, sharing of a common type system, dynamic extension of the inventory of types, and streaming (in particular for output). Thus our design meets recently formulated guidelines for a linguistic annotation framework (Ide and Romary 2004), in particular those of *incrementality*, *semantic adequacy*, *extensibility*, and *human readability*.

[1] There are no explicit parent/child relationships in the annotation subsystem; we use the term here informally, to indicate co-terminous annotations of priority higher than that of the annotation spawning the sub-iterator. This behavior follows from the sub-iterator using, by design, the same priority ordering as the base iterator.

https://doi.org/10.1017/S1351324904003493 Published online by Cambridge University Press

3.2 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 look-up, 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 look-up plugin, LEXALYZER, populates the lexical cache with token strings, their lemma forms, and morpho-syntactic features (the token strings are derived from the original tokens, as delivered by the TOKENIZER plugin, but they are distinct from them, primarily for the purposes of maintaining the right data model abstractions). Morphosyntactic 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. Where required, different plugins running together can use different tag sets by defining appropriate tagset mapping tables via a configuration file. Similarly, a different morphosyntactic 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.

3.3 Vocabulary

Vocabulary annotations (proper names, domain terms, abbreviations) have a reference to an entry in the document vocabulary. The vocabulary mediates among canonical forms, variants, and categories of such supra-lexical items. Depending on the application, task, and environment, canonical forms and variants can be either plugin-recovered (e.g. by lookup against a gazetteer-style resource or glossary), or plugin-discovered (e.g. by named entity recognition and normalization components, such as TEXTRACT's dedicated names and abbreviations analysis functions, NOMINATOR and ABBREVIATOR, listed in section 2.1).

Collection salience statistics (e.g. $tf \times idf$), 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.

4 Different operational environments

Fundamentally, TEXTRACT's functionality is available through a set of *native APIs*. Configured packages, research prototypes, applications under development, and all

https://doi.org/10.1017/S1351324904003493 Published online by Cambridge University Press

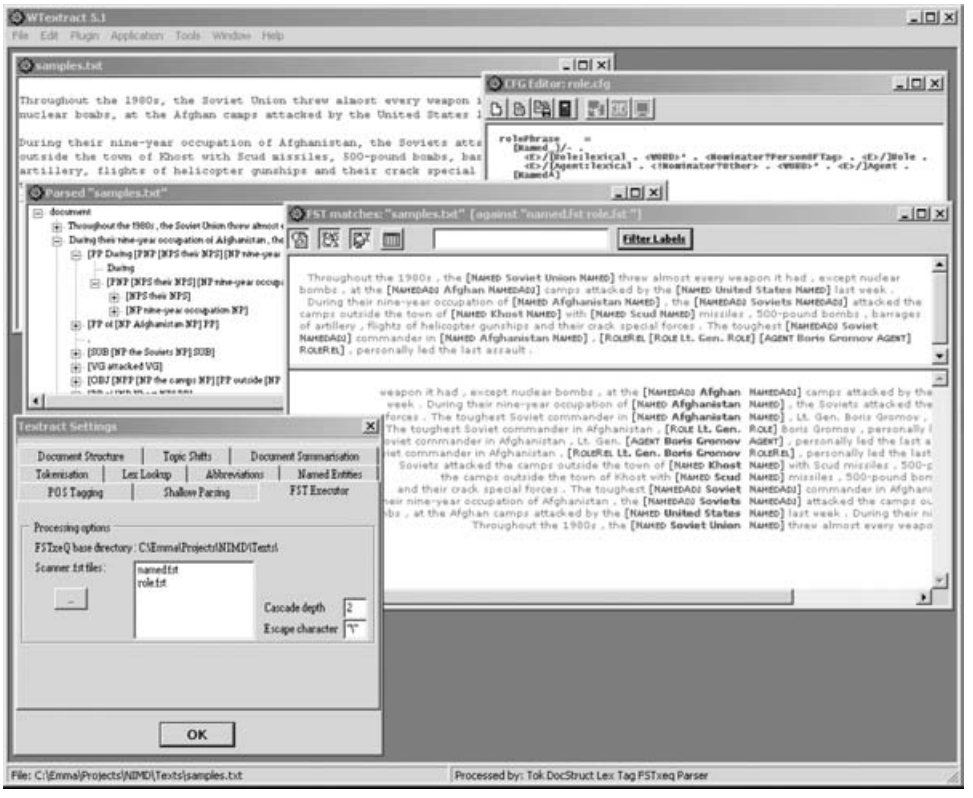


Fig. 2. wTextract's plugin development interface.

types of Textract configurations are built as wrappers appealing to a certain subset of basic services and document- (and collection-, if appropriate) level plugins (see section 1). Such configurations encapsulate raw function, and are typically invoked as *command-line executables*. A generic Dump plugin (strictly an annotation repository consumer only) implements a range of custom methods for displaying output in human-readable form. In addition to native API's, Textract offers an alternative way of being packaged in applications: it has a C API layer for *exporting the contents of the data store* to external components in C++ or Java. While convenient for subsequent deployment, or embedding in even larger infrastructures, this mode of access to functionality is not suitable for developing new functions/plugins.

For 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* (hereafter wTextract; see Figure 2). 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 *et al.* 2002). Such functionality is facilitated by suitable

https://doi.org/10.1017/S1351324904003493 Published online by Cambridge University Press

annotation repository methods, including a provision for 'rolling back' the repository to an earlier state, without a complete system `reInit()`.

There are also provisions for placing "hooks" inside of plugins, so that their *execution can be monitored* through the wTEXTRACT interface. The intent is to provide something akin to interactive stepping through a program, but instead of using native debugging facilities (which operate at the level of program instructions), the environment offers control over the granularity of the increment (which would be in terms appropriate to the algorithm/task at hand). This facilitates functionality development, both in terms of algorithmic details, and of interactions of any given plugin with the data model.

wTEXTRACT thus provides a uniform system of views, both 'inside' of plugin operations, as well as into the annotation repository as it gets modified by new analysis results; views are customized to best reflect individual plugin specifics.

With respect to one particular plugin, an annotation-based Finite State (FS) transducer (see section 5 below for description of Talent's FS subsystem), wTEXTRACT's system of process control, hooks, and views is additionally enriched, so that it can function as a *development environment for finite state 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 levels 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 (in this case, of a cascade of FSTs), a grammar editor/compiler, and multiple viewers for the results of the grammar application, 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.)

Yet another mode of using the graphical interface is that of *regression testing* and *evaluation*. In essence, custom document parsers are used to 'ingest' documents which are either appropriately marked-up outputs of previous system runs, or corpora annotated for certain linguistic analyses. The infrastructure allows for such parsers to replace, straightforwardly, the default document parsing services. The notion is to parse a set of annotations so that the mark-up is stripped from the document and represented in native TEXTRACT terms, yielding at the same time just a bare (un-annotated) token stream, compliant with the system's notion of a document ready for processing.

In a complementary mode, the document maintenance subsystem of wTEXTRACT can cache prior invocations of the engine, and compare these, side-by-side, with re-invocations under different operational settings – an operation critical, say, to the process of grammar development, where a grammar is being incrementally developed and tuned.

For strict evaluation purposes, generic evaluation methods are available to compute standard figures of precision, recall, and F-measure. In spirit, this is not dissimilar to a tool like GATE's AnnotationDiff (Maynard, Tablan, Cunningham, Ursu, Saggion, Bontcheva and Wilks 2002); still, we seek to expose some of the underlying function required by such a toolkit to other parts of the system (as exploited for

https://doi.org/10.1017/S1351324904003493 Published online by Cambridge University Press

FS grammar development using side-by-side comparison), while hiding programming and configuration details from the plugin developer/grammar writer (via the notion of externally declaring the format of an annotated corpus, and having a document parser – available as a basic services plugin – set up the two document streams required by the underlying evaluation methodology).

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, Chanod, Grefenstette and Schiller 1996). 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 (Becker, Drożdżyński, Krieger, Piskorski, Schäfer and Xu 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, Maynard and Tablan 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

https://doi.org/10.1017/S1351324904003493 Published online by Cambridge University Press

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, in the annotation repository) and a set of annotation property specifications which relate individual plugin capabilities to granularity of analysis. Matching against an annotation – within any family, and of any particular type – possibly further constrained by attributes specific to that type, becomes an atomic transition within a finite state device. We have developed a notation for FS operations, which appeals to the system-wide set of annotation families, with their property attributes. At any point in the annotation lattice, posted by plugins prior to the TFST plugin, the symbol for current match specifies the annotation type (with its full complement of attributes) to be picked from the lattice and considered by the match operator. Run-time behaviour of this operator is determined by a symbol compiler which uses the type system and the full complement of annotation iterators (as described in section 3.1) to construct, dynamically (see below), the sequence of annotations defined by the current grammar as a particular traversal of the annotation lattice, and to apply to each annotation in that sequence the appropriate (also dynamically defined) set of tests for the specified configuration of annotation attributes. Within the notation, it is also possible to express 'transduction' operations over annotations – such as create new ones, remove existing ones, modify and/or add properties, and so forth – as primitive operations. Full details concerning the specifics of the notation can be found in Boguraev and Neff (2003).

The uniform way of specifying annotation types on the transitions of an FST graph hides from the grammar writer system-wide design decisions, which separate the annotation repository, the lexicon, and the vocabulary (see section 3). 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 (section 3.3) 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 TFST 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 the 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 constraint over phrase type) or/and document structure elements (such as section titles, sentences, and so forth).

https://doi.org/10.1017/S1351324904003493 Published online by Cambridge University Press

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. However, expressive and powerful FS grammars may be written which inspect annotations of various types, at different – or even the same – point of the analysis. 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 the results of upstream annotators of which only the externally published capabilities are known.

In addition to the shallow parser applications mentioned in section 2, TFST provides the pattern-matching and parsing infrastructure for a number of ongoing Talent research efforts. Among these are a TimeML-compliant (Pustejovsky, Castaño, Ingria, Saurí, Gaizauskas, Setzer, Katz and Radev 2003) parser for temporal expressions, a named entity recognition device, and a tool for extracting hypernym/hyponym relations, based on patterns inspired by Hearst (1992).

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 (the MITRE Corporation) and 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, Humphreys, Gaizauskas and Wilks (1997) emphasize that

https://doi.org/10.1017/S1351324904003493 Published online by Cambridge University Press

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 look-up, sentence splitter, POS tagger, and a grammar application module, called JAPE, which is broadly analogous 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, as well as for incremental evaluation during (new) function development. Application uses of TEXTRACT are accomplished with the programming API's and configuration tools, rather than with the graphical tool.

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 Corporation 1997) and ATLAS (Bird, Day, Garofolo, Henderson, Laprun and Liberman 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 for pattern matching. 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 do not apparently provide API's 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, McKelvie, Tobin, Thompson and Mikheev 2000), MATE (McKelvie, Isard, Mengel, Miller, Grosse and Klein 2000), and Transcriber (Barras, Geoffrois, Wu and Liberman 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.

https://doi.org/10.1017/S1351324904003493 Published online by Cambridge University Press

7 Conclusion

In this paper, we have described an 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 engagements; indeed, a particular strength of this system is its adaptability to stand-alone, industrial-strength, document processing 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 3), 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 3), 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 influenced much of the architecture of 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 two gigabytes-per-hour on a mid-range Unix workstation.

Like performance, it is not clear how to assign a quantitative measure of 'how good' an architecture is. However, it is clear that for a framework for language engineering to be truly supportive of development of new functionalities and the configuration of new applications, it must at least encapsulate capabilities for assessment and evaluation of individual technologies. This is a point also reinforced in Cunningham, Maynard, Bontcheva and Tablan (2002) and MITRE Corp. (1997). TEXTRACT's development cycle caters for this, as already discussed in section 4.[2]

[2] In section 6, we pointed out that, in addition to their role in hosting NLP technologies for application construction (as in TEXTRACT) or for technology evaluation, systems such as Alembic and GATE also adopt the role of supporting semi-automated corpus annotation. For these systems, it is also reasonable to conduct evaluations of the productivity enhancement

https://doi.org/10.1017/S1351324904003493 Published online by Cambridge University Press

Additionally, it is also clear that the ability to effectively engineer a broad range of applications indirectly speaks for the quality of the underlying framework. The majority of technologies and applications developed within the TEXTRACT framework (as enumerated in section 2 and citations therein) have been independently evaluated.

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 3). 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 predefines, 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, the IBM Unstructured Information Management Architecture (UIMA) (Ferrucci and Lally 2004) has adopted 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).

As TEXTRACT plugins are migrating to the UIMA platform, they would need to revise their API's to the annotation store. Some of this migration, now appealing to typed feature structures, is naturally supported by the initial TEXTRACT data model design. The annotator API's remain nearly the same. TEXTRACT's system of annotation families and types and its hierarchy of vocabulary categories have a natural mapping to a hierarchy of types. The data model's API's used for extensibility of feature sets have become standardized and simplified. Because of the generality of the typed feature structure system, the TFST plugin instantly gained more power as a result of the migration, without needing any extra code. On the other hand, 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) and Wunsch (2003)). We shall return to this issue in a following paper.

Acknowledgements

We acknowledge the contributions of our colleagues, current and former, in the design and implementation of the TEXTRACT system and plugins: Rie Ando, Misook Choi, Herb Chong, Jim Cooper, Albert Eskenazi, Zunaid Kazi, Aaron Kershenbaum, Youngja Park, Son Bao Pham, John Prager, Yael Ravin, Max Silberztein, Nina Wacholder. This paper has benefited from two rounds of reviewing: a shorter version was presented at an HLT-NAACL workshop on *Software Engineering and Architecture of Language Technology Systems*, in May 2003; we also thank, in particular, the anonymous reviewers for this special issue.

the system affords the annotator, as was done for Alembic in Day, Aberdeen, Hirschman, Kozierok, Robinson and Vilain (1997). Since TEXTRACT has not (at least, to date) been incorporated in a corpus-annotation application, evaluation from that perspective is not relevant.

https://doi.org/10.1017/S1351324904003493 Published online by Cambridge University Press

References

Ando, R., Boguraev, B., Byrd, R. and Neff, M. (2000) Multi-document summarization by visualizing topical content. *Proceedings NAACL/ANLP Workshop on Advanced Text Summarization Methods*, Seattle, WA.

Ballim, A. and Pallotta, V., editors (2002) Robust methods in analysis of natural language data. *Natural Lang. Eng.* **8**(3).

Barras, C., Geoffrois, E., Wu, Z. and Liberman, M. (2001) Transcriber: development and use of a tool for assisted speech corpora production. *Speech Commun.* **33**: 5–22.

Becker, M., Drożdżyński, W., Krieger, H.-U., Piskorski, J., Schäfer, U. and Xu, F. (2002) SProUT-shallow processing with unification and typed feature structures. *Proceedings International Conference on Natural Language Processing (ICON 2002)*, Mumbai, India.

Bird, S., Day, D., Garofolo, J., Henderson, J., Laprun, C. and Liberman, M. (2000) ATLAS: A flexible and extensible architecture for linguistic annotation. *Proceedings Second International Conference on Language Resources and Evaluation*, pp. 1699–1706. Spain.

Bird, S. and Liberman, M. (2001) A formal framework for linguistic annotation. *Speech Commun.* **33**(1,2): 23–60.

Boguraev, B. (2000) Towards finite-state analysis of lexical cohesion. *Proceedings 3rd International Conference on Finite-State Methods for NLP*, INTEX-3, Liege, Belgium.

Boguraev, B. and Neff, M. (2000a) Discourse segmentation in aid of document summarization. *Proceedings Hawaii International IEEE Conference on System Sciences* (HICSS-33), Maui, Hawaii.

Boguraev, B. and Neff, M. (2000b) Lexical cohesion, discourse segmentation and document summarization. *Proceedings RIAO-2000, Content-Based Multimedia Information Access*, Paris, France.

Boguraev, B. and Neff, M. (2003) *The Talent 5.1 TFst system: user documentation and grammar writing manual*. IBM T. J. Watson Research Center, Yorktown Heights, New York. Technical Report RC22976.

Brawer, S. (1998) Patti: Compiling unification-based finite-state automata into machine instructions for a superscalar pipelined RISC processor. Master's thesis, University of the Saarland, Saarbrücken, Germany.

Brew, C., McKelvie, D., Tobin, R., Thompson, H. and Mikheev, A. (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`.

Carpenter, R. (1992) *The Logic of Typed Feature Structures*. Cambridge University Press, Cambridge, UK.

Cooper, J. and Byrd, R. (1997) Lexical navigation: visually-prompted query expansion and refinement. *Proceedings 2nd ACM Conference on Digital Libraries and Information Retrieval*, pp. 237–247. Philadelphia, PA.

Cooper, J., Viswanathan, M. and Kazi, Z. (2001) Samsa: A speech analysis, mining and summary application for outbound telephone calls. *Hawaii International Conference on System Sciences* (HICSS-34)*: Genres in Digital Documents*, volume 4. Maui, Hawaii.

Cunningham, H. (2002) GATE, a general architecture for language engineering. *Computers and the Humanities*, 36(2): 223–254.

Cunningham, H., Humphreys, K., Gaizauskas, R. and Wilks, Y. (1997) Software infrastructure for natural language processing. *Proceedings Fifth Conference on Applied Natural Language Processing*, Washington, DC.

Cunningham, H., Maynard, D., Bontcheva, K. and Tablan, V. (2002) GATE: An architecture for development of robust HLT applications. *Proceedings 40th Annual Meeting of the Association for Computational Linguistics (ACL)*, Philadelphia, PA.

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

https://doi.org/10.1017/S1351324904003493 Published online by Cambridge University Press

Cunningham, H., Maynard, D., Tablan, V., Ursu, C. and Bontcheva, K. (2001) *Developing Language Processing Components with* GATE. University of Sheffield, Sheffield, UK. GATE v2.0 User Guide.

Day, D., Aberdeen, J., Hirschman, L., Kozierok, R., Robinson, P. and Vilain, M. (1997) Mixed-initiative development of language processing systems. *Fifth ACL Conference on Applied Natural Language Processing*, Washington, DC.

Fellbaum, C., editor (1999) WORDNET*: An electronic lexical database and some of its applications*. MIT Press.

Ferrucci, D. and Lally, A. (2004) UIMA: an architectural approach to unstructured information processing in the corporate research environment. *Natural Lang. Eng.* (this issue). Special Issue on Software Architectures for Language Engineering.

Grishman, R. (1996) TIPSTER architecture design document. Technical report, DARPA. Version 2.2.

Hearst, M. A. (1992) Automatic acquisition of hyponyms from large text corpora. *Fourteenth International Conference on Computational Linguistics (*COLING-92*)*, Nantes, France.

Ide, N. and Romary, L. (2004) International standard for a linguistic annotation framework. *Natural Lang. Eng.* (this issue). Special Issue on Software Architectures for Language Engineering.

Justeson, J. and Katz, S. (1995) Technical terminology: some linguistic properties and an algorithm for identification in text. *Natural Lang. Eng.* **1**(1): 9–27.

Karttunen, L., Chanod, J.-P., Grefenstette, G. and Schiller, A. (1996) Regular expressions for language engineering. *Natural Lang. Eng.* **4**(1): 305–328.

Kornai, A., editor (1999) *Extended Finite State Models of Language*. Cambridge University Press.

Mack, R., Ravin, Y. and Byrd, R. (2001) Knowledge portals and the emerging digital workspace. *IBM Systems J.* **40**(4): 925–955.

Maynard, D., Tablan, V., Cunningham, H., Ursu, C., Saggion, H., Bontcheva, K. and Wilks, Y. (2002) Architectural elements of language engineering robustness. *Natural Lang. Eng.* **8**(2/3): 257–274. Special Issue, Robust Methods in Analysis of Natural Language Data.

McKelvie, D., Isard, A., Mengel, A., Miller, M. B., Grosse, M. and Klein, M. (2001) The MATE workbench – an annotation tool for XML coded speech corpora. *Speech Commun.* **33**(1–2): 97–112.

MITRE Corporation (1997) Alembic workbench users guide. Available at `http://www.mitre.org/tech/alembic-workbench/`.

Nasukawa, T. and Nagano, T. (2001) Text analysis and knowledge mining system. *IBM Systems J.* **40**(4): 967–984.

Park, Y. (2002) Identification of probable real words: an entropy-based approach. *Proceedings ACL Workshop on Unsupervised Lexical Acquisition*, pp. 1–8. Philadelphia, PA.

Park, Y. and Byrd, R. (2001) Hybrid text mining for finding abbreviations and their definitions. *Proceedings Conference on Empirical Methods in Natural Language Processing*, pp. 126–133. Pitsburgh, PA.

Park, Y., Byrd, R. and Boguraev, B. (2002) Automatic glossary extraction: beyond terminology identification. *Proceedings 19th International Conference on Computational Linguistics (COLING)*, pp. 772–778. Taiwan.

Patrick, J. and Cunningham, H., editors (2003) *Workshop on The Software Engineering and Architecture of Language Technology Systems*, Edmonton, Alberta, Canada. HLT-NAACL'03.

Prager, J., Coden, A. and Radev, D. (2000) Question answering by predictive annotation. *Proceedings 23rd Annual ACM/SIGIR Conference*, pp. 184–191. Athens, Greece.

Pustejovsky, J., Castaño, J., Ingria, R., Saurí, R., Gaizauskas, R., Setzer, A., Katz, G. and Radev, D. (2003) TimeML: Robust specification of event and temporal expressions in text. *AAAI Spring Symposium on New Directions in Question-Answering (Working Papers)*, pp. 28–34. Stanford, CA.

https://doi.org/10.1017/S1351324904003493 Published online by Cambridge University Press

Ravin, Y. and Wacholder, N. (1997) Extracting names from natural-language text. Technical Report RC-20338, IBM T. J. Watson Research Center, Yorktown Heights, NY.

Schiffman, B., Nenkova, A. and McKeown, K. (2002) Experiments in multidocument summarization. *Conference on Human Language Technology* (HLT-2002), San Diego, CA.

Shieber, S. (1986) *An Introduction to Unification-Based Approaches to Grammar*, volume 4 of *CSLI Lecture Notes*. University of Chicago Press.

Vintar, Š. and Kipp, M. (2001) Multi-track annotation of terminology using ANVIL. In: Ide, N., editor, *Workshop on Multi-layer Corpus-Based Analysis*, Eurolan-2001, pp. 1–13. Iasi, Romania.

Wunsch, H. (2003) Annotation grammars and their compilation into annotation transducers. Master's thesis, University of Tübingen, Germany.

https://doi.org/10.1017/S1351324904003493 Published online by Cambridge University Press