Developing Language Processing Components with GATE Version 5 (a User Guide)

For GATE version 5.0
(built May 28, 2009)

Hamish Cunningham
Diana Maynard
Kalina Bontcheva
Valentin Tablan
Cristian Ursu
Marin Dimitrov
Mike Dowman
Niraj Aswani
Ian Roberts
Yaoyong Li
Andrey Shafirin
Adam Funk

`http://gate.ac.uk/`

HTML version: `http://gate.ac.uk/userguide`

Work on GATE has been partly supported by EPSRC grants GR/K25267 (Large-Scale Information Extraction), GR/M31699 (GATE 2), RA007940 (EMILLE), GR/N15764/01 (AKT) and GR/R85150/01 (MIAKT), AHRB grant APN16396 (ETCSL/GATE), and several EU-funded projects (SEKT, TAO, NeOn, MediaCampaign, MUSING, KnowledgeWeb, PrestoSpace, h-TechSight, enIRaF).

Brief Contents

Contents

Chapter 1

Introduction

Software documentation is like sex: when it is good, it is very, very good; and when it is bad, it is better than nothing. (Anonymous.)

There are two ways of constructing a software design: one way is to make it so simple that there are obviously no deficiencies; the other way is to make it so complicated that there are no obvious deficiencies. (C.A.R. Hoare)

A computer language is not just a way of getting a computer to perform operations but rather that it is a novel formal medium for expressing ideas about methodology. Thus, programs must be written for people to read, and only incidentally for machines to execute. (The Structure and Interpretation of Computer Programs, H. Abelson, G. Sussman and J. Sussman, 1985.)

If you try to make something beautiful, it is often ugly. If you try to make something useful, it is often beautiful. (Oscar Wilde)[1]

GATE is an infrastructure for developing and deploying software components that process human language. GATE helps scientists and developers in three ways:

1. by specifiying an **architecture**, or organisational structure, for language processing software;
2. by providing a **framework**, or class library, that implements the architecture and can be used to embed language processing capabilities in diverse applications;
3. by providing a **development environment** built on top of the framework made up of convenient graphical tools for developing components.

The architecture exploits component-based software development, object orientation and mobile code. The framework and development environment are written in Java and available

[1] These were, at least, our ideals; of course we didn't completely live up to them. . .

as open-source free software under the GNU library (or lesser) licence[2]. GATE uses Unicode throughout [Unicode Consortium 96, Tablan *et al.* 02], and has been tested on a variety of Slavic, Germanic, Romance, and Indic languages [Maynard *et al.* 01, Gambäck & Olsson 00, McEnery *et al.* 00].

From a scientific point-of-view, GATE's contribution is to quantitative measurement of accuracy and repeatability of results for verification purposes.

GATE has been in development at the University of Sheffield since 1995 and has been used in a wide variety of research and development projects [Maynard *et al.* 00]. Version 1 of GATE was released in 1996, was licensed by several hundred organisations, and used in a wide range of language analysis contexts including Information Extraction ([Cunningham 99b, Appelt 99, Gaizauskas & Wilks 98, Cowie & Lehnert 96]) in English, Greek, Spanish, Swedish, German, Italian, French, Bulgarian, Russian, and a number of other languages. Version 4 of the system is available from http://gate.ac.uk/download/.

This book describes how to use GATE to develop language processing components, test their performance and deploy them as parts of other applications. In the rest of this chapter:

- section 1.1 describes the best way to use this book;
- section 1.2 briefly notes that the context of GATE is applied language processing, or *Language Engineering*;
- section 1.3 gives an overview of developing using GATE;
- section 1.4 describes the structure of the rest of the book;
- section 1.5 lists other publications about GATE.

Note: if you don't see the component you need in this document, or if we mention a component that you can't see in the software, contact `gate-users@lists.sourceforge.net`[3] – various components are developed by our collaborators, who we will be happy to put you in contact with. (Often the process of getting a new component is as simple as typing the URL into GATE; the system will do the rest.)

1.1 How to Use This Text

It is a good idea to read all of this introduction (you can skip sections 1.2 and 1.5 if pressed); then you can either continue wading through the whole thing or just use chapter 3 as a

[2]This is a restricted form of the main GNU licence, which means that GATE can be embedded in commercial products if required.

[3]Follow the 'support' link from the GATE web server to subscribe to the mailing list.

reference and dip into other chapters for more detail as necessary. Chapter 3 gives instructions for completing common tasks with GATE, organised in a FAQ style: details, and the reasoning behind the various aspects of the system, are omitted in this chapter, so where more information is needed refer to later chapters.

The structure of the book as a whole is detailed in section 1.4 below.

1.2 Context

GATE can be thought of as a **Software Architecture for Language Engineering** [Cunningham 00].

'Software Architecture' is used rather loosely here to mean computer infrastructure for software development, including development environments and frameworks, as well as the more usual use of the term to denote a macro-level organisational structure for software systems [Shaw & Garlan 96].

Language Engineering (LE) may be defined as:

> . . . the discipline or act of engineering software systems that perform tasks involving processing human language. Both the construction process and its outputs are measurable and predictable. The literature of the field relates to both application of relevant scientific results and a body of practice. [Cunningham 99a]

The relevant scientific results in this case are the outputs of Computational Linguistics, Natural Language Processing and Artificial Intelligence in general. Unlike these other disciplines, LE, as an engineering discipline, entails *predictability*, both of the process of constructing LE-based software and of the performance of that software after its completion and deployment in applications.

Some working definitions:

1. **Computational Linguistics (CL):** science of language that uses computation as an investigative tool.
2. **Natural Language Processing (NLP):** science of computation whose subject matter is data structures and algorithms for computer processing of human language.
3. **Language Engineering (LE):** building NLP systems whose cost and outputs are measurable and predictable.
4. **Software Architecture:** macro-level organisational principles for families of systems. In this context is also used as **infrastructure**.

5. **Software Architecture for Language Engineering (SALE):** software infrastructure, architecture and development tools for applied CL, NLP and LE.

(Of course the practice of these fields is broader and more complex than these definitions.)

In the scientific endeavours of NLP and CL, GATE's role is to support experimentation. In this context GATE's significant features include support for automated measurement (see section 13), providing a 'level playing field' where results can easily be repeated across different sites and environments, and reducing research overheads in various ways.

1.3 Overview

1.3.1 Developing and Deploying Language Processing Facilities

GATE as an architecture suggests that the elements of software systems that process natural language can usefully be broken down into various types of component, known as resources[4]. Components are reusable software chunks with well-defined interfaces, and are a popular architectural form, used in Sun's Java Beans and Microsoft's .Net, for example. GATE components are specialised types of Java Bean, and come in three flavours:

- LanguageResources (LRs) represent entities such as lexicons, corpora or ontologies;
- ProcessingResources (PRs) represent entities that are primarily algorithmic, such as parsers, generators or ngram modellers;
- VisualResources (VRs) represent visualisation and editing components that participate in GUIs.

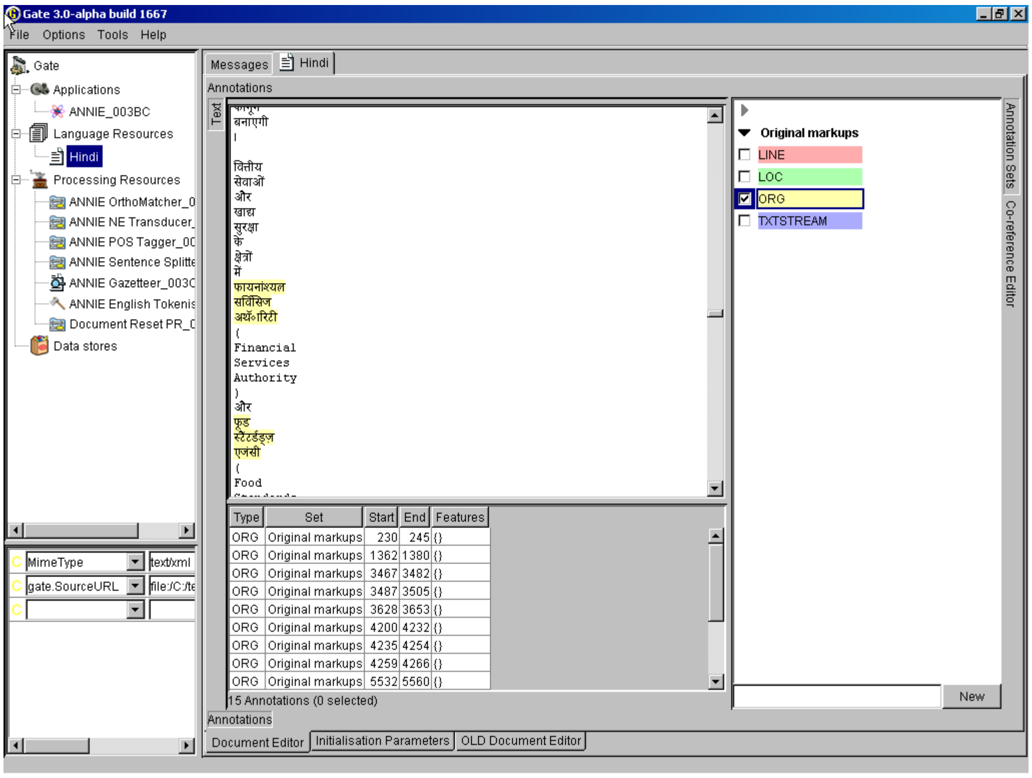
These definitions can be blurred in practice as necessary.

Collectively, the set of resources integrated with GATE is known as **CREOLE**: a Collection of REusable Objects for Language Engineering. All the resources are packaged as Java Archive (or 'JAR') files, plus some XML configuration data. The JAR and XML files are made available to GATE by putting them on a web server, or simply placing them in the local file space. Section 1.3.2 introduces GATE's built-in resource set.

When using GATE to develop language processing functionality for an application, the developer uses the development environment and the framework to construct resources of the three types. This may involve programming, or the development of Language Resources

[4]The terms 'resource' and 'component' are synonymous in this context. 'Resource' is used instead of just 'component' because it is a common term in the literature of the field: cf. the Language Resources and Evaluation conference series [LREC-1 98, LREC-2 00].

such as grammars that are used by existing Processing Resources, or a mixture of both. The development environment is used for visualisation of the data structures produced and consumed during processing, and for debugging, performance measurement and so on. For example, figure 1.1 is a screenshot of one of the visualisation tools (displaying named-entity



MimeType	text/xml
gate.SourceURL	file:/C:/te

```
बनाएगी
।

वित्तीय
सेवाओं
और
खाद्य
सुरक्षा
के
क्षेत्रों
में
फायनांश्यल
सर्विसिज
अथॉरिटी
(
Financial
Services
Authority
)
और
फूड
स्टैंटर्डड्ज़
एजंसी
(
Food
```

Type	Set	Start	End	Features
ORG	Original markups	230	245	{}
ORG	Original markups	1362	1380	{}
ORG	Original markups	3467	3482	{}
ORG	Original markups	3487	3505	{}
ORG	Original markups	3628	3653	{}
ORG	Original markups	4200	4232	{}
ORG	Original markups	4235	4254	{}
ORG	Original markups	4259	4266	{}
ORG	Original markups	5532	5560	{}

15 Annotations (0 selected)

Figure 1.1: One of GATE's visual resources

extraction results for a Hindi sentence).

The GATE development environment is analogous to systems like Mathematica for Mathematicians, or JBuilder for Java programmers: it provides a convenient graphical environment for research and development of language processing software.

When an appropriate set of resources have been developed, they can then be embedded in the target client application using the GATE framework. The framework is supplied as two JAR files.[5] To embed GATE-based language processing facilities in an application, these JAR files are all that is needed, along with JAR files and XML configuration files for the various resources that make up the new facilities.

[5]The main JAR file (**gate.jar**) supplies the framework, built-in resources and various 3rd-party libraries; the second file (**guk.jar**, the GATE Unicode Kit) contains Unicode support (e.g. additional input methods for languages not currently supported by the JDK). They are separate because the latter has to be a Java extension with a privileged security profile.

1.3.2 Built-in Components

GATE includes resources for common LE data structures and algorithms, including documents, corpora and various annotation types, a set of language analysis components for Information Extraction and a range of data visualisation and editing components.

GATE supports documents in a variety of formats including XML, RTF, email, HTML, SGML and plain text. In all cases the format is analysed and converted into a single unified model of *annotation*. The annotation format is a modified form the TIPSTER format [Grishman 97] which has been made largely compatible with the Atlas format [Bird & Liberman 99], and uses the now standard mechanism of 'stand-off markup'. GATE documents, corpora and annotations are stored in databases of various sorts, visualised via the development environment, and accessed at code level via the framework. See chapter 6 for more details of corpora etc.

A family of Processing Resources for language analysis is included in the shape of ANNIE, A Nearly-New Information Extraction system. These components use finite state techniques to implement various tasks from tokenisation to semantic tagging or verb phrase chunking. All ANNIE components communicate exclusively via GATE's document and annotation resources. See chapter 8 for more details. See chapter 5 for visual resources. See chapter 9 for other miscellaneous CREOLE resources.

1.3.3 Additional Facilities

Three other facilities in GATE deserve special mention:

- JAPE, a Java Annotation Patterns Engine, provides regular-expression based pattern/action rules over annotations – see chapter 7.
- The 'annotation diff' tool in the development environment implements performance metrics such as precision and recall for comparing annotations. Typically a language analysis component developer will mark up some documents by hand and then use these along with the diff tool to automatically measure the performance of the components. See section 13.
- GUK, the GATE Unicode Kit, fills in some of the gaps in the JDK's[6] support for Unicode, e.g. by adding input methods for various languages from Urdu to Chinese. See section 3.37 for more details.

And by version 4 it will make a mean cup of tea.

[6]JDK: Java Development Kit, Sun Microsystem's Java implementation. Unicode support is being actively improved by Sun, but at the time of writing many languages are still unsupported. In fact, Unicode itself doesn't support all languages, e.g. Sylheti; hopefully this will change in time.

1.3.4 An Example

This section gives a very brief example of a typical use of GATE to develop and deploy language processing capabilities in an application, and to generate quantitative results for scientific publication.

Let's imagine that a developer called Fatima is building an email client[7] for Cyberdyne Systems' large corporate Intranet. In this application she would like to have a language processing system that automatically spots the names of people in the corporation and transforms them into `mailto` hyperlinks.

A little investigation shows that GATE's existing components can be tailored to this purpose. Fatima starts up the development environment, and creates a new document containing some example emails. She then loads some processing resources that will do named-entity recognition (a tokeniser, gazetteer and semantic tagger), and creates an application to run these components on the document in sequence. Having processed the emails, she can see the results in one of several viewers for annotations.

The GATE components are a decent start, but they need to be altered to deal specially with people from Cyberdyne's personnel database. Therefore Fatima creates new "cyber-" vesions of the gazetteer and semantic tagger resources, using the "bootstrap" tool. This tool creates a directory structure on disk that has some Java stub code, a Makefile and an XML configuration file. After several hours struggling with badly written documentation, Fatima manages to compile the stubs and create a JAR file containing the new resources. She tells GATE the URL of these files[8], and the system then allows her to load them in the same way that she loaded the built-in resources earlier on.

Fatima then creates a second copy of the email document, and uses the annotation editing facilities to mark up the results that she would like to see her system producing. She saves this and the version that she ran GATE on into her Oracle datastore (set up for her by the Herculean efforts of the Cyberdyne technical support team, who like GATE because it enables them to claim lots of overtime). From now on she can follow this routine:

1. Run her application on the email test corpus.
2. Check the performance of the system by running the 'annotation diff' tool to compare her manual results with the system's results. This gives her both percentage accuracy figures and a graphical display of the differences between the machine and human outputs.
3. Make edits to the code, pattern grammars or gazetteer lists in her resources, and recompile where necessary.

[7]Perhaps because Outlook Express trashed her mail folder again, or because she got tired of Microsoft-specific viruses and hadn't heard of Netscape or Emacs.

[8]While developing, she uses a `file:/...` URL; for deployment she can put them on a web server.

4. Tell GATE to re-initialise the resources.
5. Go to 1.

To make the alterations that she requires, Fatima re-implements the ANNIE gazetteer so that it regenerates itself from the local personnel data. She then alters the pattern grammar in the semantic tagger to prioritise recognition of names from that source. This latter job involves learning the JAPE language (see chapter 7), but as this is based on regular expressions it isn't too difficult.

Eventually the system is running nicely, and her accuracy is 93% (there are still some problem cases, e.g. when people use nicknames, but the performance is good enough for production use). Now Fatima stops using the GATE development environment and works instead on embedding the new components in her email application. This application is written in Java, so embedding is very easy[9]: the two GATE JAR files are added to the project CLASSPATH, the new components are placed on a web server, and with a little code to do initialisation, loading of components and so on, the job is finished in half a day – the code to talk to GATE takes up only around 150 lines of the eventual application, most of which is just copied from the example in the `sheffield.examples.StandAloneAnnie` class.

Because Fatima is worried about Cyberdyne's unethical policy of developing Skynet to help the large corporates of the West strengthen their strangle-hold over the World, she wants to get a job as an academic instead (so that her conscience will only have to cope with the torture of students, as opposed to humanity). She takes the accuracy measures that she has attained for her system and writes a paper for the Journal of Nasturtium Logarithm Encitement describing the approach used and the results obtained. Because she used GATE for development, she can cite the repeatability of her experiments and offer access to example binary versions of her software by putting them on an external web server.

And everybody lived happily ever after.

1.4 Structure of the Book

The material presented in this book ranges from the conceptual (e.g. 'what is software architecture?') to practical instructions for programmers (e.g. how to deal with GATE exceptions) and linguists (e.g. how to write a pattern grammar). This diversity is something of an organisational challenge. Our (no doubt imperfect) solution is to collect specific instructions for 'how to do X' in a separate chapter (3). Other chapters give a more discursive presentation. In order to understand the whole system you must, unfortunately, read much of the book; in order to get help with a particular task, however, look first in chapter 3 and refer to other material as necessary.

[9]Languages other than Java require an additional interface layer, such as JNI, the Java Native Interface, which is in C.

The other chapters:

Chapter 4 describes the GATE architecture's component-based model of language processing, describes the lifecycle of GATE components, and how they can be grouped into applications and stored in databases and files.

Chapter 5 describes the set of Visual Resources that are bundled with GATE.

Chapter 6 describes GATE's model of document formats, annotated documents, annotation types, and corpora (sets of documents). It also covers GATE's facilities for reading and writing in the XML data interchange language.

Chapter 7 describes JAPE, a pattern/action rule language based on regular expressions over annotations on documents. JAPE grammars compile into cascaded finite state transducers.

Chapter 8 describes ANNIE, a pipelined Information Extraction system which is supplied with GATE.

Chapter 9 describes CREOLE resources bundled with the system that don't fit into the previous categories.

Chapter 10 describes processing resources and language resources for working with ontologies.

Chapter 11 describes a machine learning layer specifically targetted at NLP tasks including text classification, chunk learning (e.g. for named entity recognition) and relation learning.

Chapter 13 describes how to measure the performance of language analysis components.

Chapter 14 describes the data store security model.

Appendix A discusses the design of the system.

Appendix B describes the implementation details and formal definitions of the JAPE annotation patterns language.

Appendix D describes in some detail the JAPE pattern grammars that are used in ANNIE for named-entity recognition.

1.5 Further Reading

Lots of documentation lives on the GATE web server, including:

- the concise application developer's guide (with emphasis on using the GATE API);
- movies of the system in operation;

- the main system documentation tree;
- JavaDoc API documentation;
- HTML of the source code;
- parts of the requirements analysis that version 3 is based on.

For more details about Sheffield University's work in human language processing see the NLP group pages or *A Definition and Short History of Language Engineering* ([Cunningham 99a]). For more details about Information Extraction see *IE, a User Guide* or the GATE IE pages.

A list of publications on GATE and projects that use it (some of which are available on-line):

[**Cunningham 05**] is an overview of the field of Information Extraction for the 2nd Edition of the Encyclopaedia of Language and Linguistics.

[**Cunningham & Bontcheva 05**] is an overview of the field of Software Architecture for Language Engineering for the 2nd Edition of the Encyclopaedia of Language and Linguistics.

[**Li** *et al.* **04**] (Machine Learning Workshop 2004) describes an SVM based learning algortihm for IE using GATE.

[**Wood** *et al.* **04**] (NLDB 2004) looks at ontology-based IE from parallel texts.

[**Cunningham & Scott 04b**] (JNLE) is a collection of papers covering many important areas of Software Architecture for Language Engineering.

[**Cunningham & Scott 04a**] (JNLE) is the introduction to the above collection.

[**Bontcheva 04**] (LREC 2004) describes lexical and ontological resources in GATE used for Natural Language Generation.

[**Bontcheva** *et al.* **04**] (JNLE) discusses developments in GATE in the early naughties.

[**Maynard** *et al.* **04a**] (LREC 2004) presents algorithms for the automatic induction of gazetteer lists from multi-language data.

[**Maynard** *et al.* **04c**] (AIMSA 2004) presents automatic creation and monitoring of semantic metadata in a dynamic knowledge portal.

[**Maynard** *et al.* **04b**] (ESWS 2004) discusses ontology-based IE in the hTechSight project.

[**Dimitrov** *et al.* **04**] (Anaphora Processing) gives a lightweight method for named entity coreference resolution.

[**Kiryakov 03**] (Technical Report) discusses semantic web technology in the context of multimedia indexing and search.

[**Tablan** *et al.* **03**] (HLT-NAACL 2003) presents the OLLIE on-line learning for IE system.

[**Wood** *et al.* **03**] (Recent Advances in Natural Language Processing 2003) discusses using parallel texts to improve IE recall.

[**Maynard** *et al.* **03a**] (Recent Advances in Natural Language Processing 2003) looks at semantics and named-entity extraction.

[**Maynard** *et al.* **03b**] (ACL Workshop 2003) describes NE extraction without training data on a language you don't speak (!).

[**Maynard** *et al.*] (EACL 2003) looks at the distinction between information and content extraction.

[**Manov** *et al.* **03**] (HLT-NAACL 2003) describes experiments with geographic knowledge for IE.

[**Saggion** *et al.* **03a**] (EACL 2003) discusses robust, generic and query-based summarisation.

[**Saggion** *et al.* **03c**] (EACL 2003) discusses event co-reference in the MUMIS project.

[**Saggion** *et al.* **03b**] (Data and Knowledge Engineering) discusses multimedia indexing and search from multisource multilingual data.

[**Cunningham** *et al.* **03**] (Corpus Linguistics 2003) describes GATE as a tool for collaborative corpus annotation.

[**Bontcheva** *et al.* **03**] (NLPXML-2003) looks at GATE for the semantic web.

[**Dimitrov 02a, Dimitrov** *et al.* **02**] (DAARC 2002, MSc thesis) discuss lightweight coreference methods.

[**Lal 02**] (Master Thesis) looks at text summarisation using GATE.

[**Lal & Ruger 02**] (ACL 2002) looks at text summarisation using GATE.

[**Cunningham** *et al.* **02**] (ACL 2002) describes the GATE framework and graphical development environment as a tool for robust NLP applications.

[**Bontcheva** *et al.* **02b**] (NLIS 2002) discusses how GATE can be used to create HLT modules for use in information systems.

[**Tablan** *et al.* **02**] (LREC 2002) describes GATE's enhanced Unicode support.

[**Maynard** *et al.* **02a**] (ACL 2002 Summarisation Workshop) describes using GATE to build a portable IE-based summarisation system in the domain of health and safety.

[**Maynard** *et al.* **02c**] (Nordic Language Technology) describes various Named Entity recognition projects developed at Sheffield using GATE.

[**Maynard** *et al.* **02b**] (AIMSA 2002) describes the adaptation of the core ANNIE modules within GATE to the ACE (Automatic Content Extraction) tasks.

[**Maynard** *et al.* **02d**] (JNLE) describes robustness and predictability in LE systems, and presents GATE as an example of a system which contributes to robustness and to low overhead systems development.

[**Bontcheva** *et al.* **02c**], [**Dimitrov 02a**] **and** [**Dimitrov 02b**] (TALN 2002, DAARC 2002, MSc thesis) describe the shallow named entity coreference modules in GATE: the orthomatcher which resolves pronominal coreference, and the pronoun resolution module.

[**Bontcheva** *et al.* **02a**] (ACl 2002 Workshop) describes how GATE can be used as an environment for teaching NLP, with examples of and ideas for future student projects developed within GATE.

[**Pastra** *et al.* **02**] (LREC 2002) discusses the feasibility of grammar reuse in applications using ANNIE modules.

[**Baker** *et al.* **02**] (LREC 2002) report results from the EMILLE Indic languages corpus collection and processing project.

[**Saggion** *et al.* **02b**] **and** [**Saggion** *et al.* **02a**] (LREC 2002, SPLPT 2002) describes how ANNIE modules have been adapted to extract information for indexing multimedia material.

[**Maynard** *et al.* **01**] (RANLP 2001) discusses a project using ANNIE for named-entity recognition across wide varieties of text type and genre.

[**Cunningham 00**] (PhD thesis) defines the field of Software Architecture for Language Engineering, reviews previous work in the area, presents a requirements analysis for such systems (which was used as the basis for designing GATE versions 2 and 3), and evaluates the strengths and weaknesses of GATE version 1.

[**Cunningham 02**] (Computers and the Humanities) describes the philosophy and motivation behind the system, describes GATE version 1 and how well it lived up to its design brief.

[**McEnery** *et al.* **00**] (Vivek) presents the EMILLE project in the context of which GATE's Unicode support for Indic languages has been developed.

[**Cunningham** *et al.* **00d**] **and** [**Cunningham 99c**] (technical reports) document early versions of JAPE (superceded by the present document).

[**Cunningham** *et al.* **00a**], [**Cunningham** *et al.* **98a**] **and** [**Peters** *et al.* **98**] (OntoLex 2000, LREC 1998) presents GATE's model of Language Resources, their access and distribution.

[**Maynard** *et al.* **00**] (technical report) surveys users of GATE up to mid-2000.

[**Cunningham** *et al.* **00c**] **and** [**Cunningham** *et al.* **99**] (COLING 2000, AISB 1999) summarise experiences with GATE version 1.

[**Cunningham** *et al.* **00b**] (LREC 2000) taxonomises Language Engineering components and discusses the requirements analysis for GATE version 2.

[**Bontcheva** *et al.* **00**] **and** [**Brugman** *et al.* **99**] (COLING 2000, technical report) describe a prototype of GATE version 2 that integrated with the EUDICO multimedia markup tool from the Max Planck Institute.

[**Gambäck & Olsson 00**] (LREC 2000) discusses experiences in the Svensk project, which used GATE version 1 to develop a reusable toolbox of Swedish language processing components.

[**Cunningham 99a**] (JNLE) reviewed and synthesised definitions of Language Engineering.

[**Stevenson** *et al.* **98**] **and** [**Cunningham** *et al.* **98b**] (ECAI 1998, NeMLaP 1998) report work on implementing a word sense tagger in GATE version 1.

[**Cunningham** *et al.* **97b**] (ANLP 1997) presents motivation for GATE and GATE-like infrastructural systems for Language Engineering.

[**Gaizauskas** *et al.* **96b, Cunningham** *et al.* **97a, Cunningham** *et al.* **96e**] (ICTAI 1996, TITPSTER 1997, NeMLaP 1996) report work on GATE version 1.

[**Cunningham** *et al.* **96c, Cunningham** *et al.* **96d, Cunningham** *et al.* **95**] (COLING 1996, AISB Workshop 1996, technical report) report early work on GATE version 1.

[**Cunningham** *et al.* **96b**] (TIPSTER) discusses a selection of projects in Sheffield using GATE version 1 and the TIPSTER architecture it implemented.

[**Cunningham** *et al.* **96a**] (manual) was the guide to developing CREOLE components for GATE version 1.

[**Gaizauskas** *et al.* **96a**] (manual) was the user guide for GATE version 1.

[**Humphreys** *et al.* **96**] (manual) desribes the language processing components distributed with GATE version 1.

[**Cunningham 94, Cunningham** *et al.* **94**] (NeMLaP 1994, technical report) argue that software engineering issues such as reuse, and framework construction, are important for language processing R&D.

[**Dowman** *et al.* **05b**] (World Wide Web Conference Paper) The Web is used to assist the annotation and indexing of broadcast news.

[**Dowman** *et al.* **05a**] (Euro Interactive Television Conference Paper) A system which can use material from the Internet to augment television news broadcasts.

[**Dowman** *et al.* **05c**] (Second European Semantic Web Conference Paper) A system that semantically annotates television news broadcasts using news websites as a resource to aid in the annotation process.

[**Li** *et al.* **05a**] (Proceedings of Sheffield Machine Learning Workshop) describe an SVM based IE system which uses the SVM with uneven margins as learning component and the GATE as NLP processing module.

[**Li** *et al.* **05b**] (Proceedings of Ninth Conference on Computational Natural Language Learning (CoNLL-2005)) uses the uneven margins versions of two popular learning algorithms SVM and Perceptron for IE to deal with the imbalanced classification problems derived from IE.

[**Li** *et al.* **05c**] (Proceedings of Fourth SIGHAN Workshop on Chinese Language processing (Sighan-05)) used Perceptron learning, a simple, fast and effective learning algorithm, for Chinese word segmentation.

[**Aswani** *et al.* **05**] (Proceedings of Fifth International Conference on Recent Advances in Natural Language Processing (RANLP2005)) It is a full-featured annotation indexing and search engine, developed as a part of the GATE. It is powered with Apache Lucene technology and indexes a variety of documents supported by the GATE.

[**Li** *et al.* **05c**] (Proceedings of Fourth SIGHAN Workshop on Chinese Language processing (Sighan-05)) a system for Chinese word segmentation based on Perceptron learning, a simple, fast and effective learning algorithm.

[**Wang** *et al.* **05**] (Proceedings of the 2005 IEEE/WIC/ACM International Conference on Web Intelligence (WI 2005)) Extracting a Domain Ontology from Linguistic Resource Based on Relatedness Measurements.

[**Ursu** *et al.* **05**] (Proceedings of the 2nd European Workshop on the Integration of Knowledge, Semantic and Digital Media Technologies (EWIMT 2005))Digital Media Preservation and Access through Semantically Enhanced Web-Annotation.

[Polajnar *et al.* 05] (University of Sheffield-Research Memorandum CS-05-10) User-Friendly Ontology Authoring Using a Controlled Language.

[Aswani *et al.* 06] (Proceedings of the 5th International Semantic Web Conference (ISWC2006)) In this paper the problem of disambiguating author instances in ontology is addressed. We describe a web-based approach that uses various features such as publication titles, abstract, initials and co-authorship information.

Chapter 2

Change Log

This chapter lists major changes to GATE in roughly chronological order by release. Changes in the documentation are also referenced here.

2.1 Version 5.0 (May 2009)

> **Note:** *existing users – if you delete your user configuration file for any reason you will find that GATE no longer loads the ANNIE plugin by default. You will need to manually select "load always" in the plugin manager to get the old behaviour.*

2.1.1 Major new features

JAPE language improvements

Several new extensions to the JAPE language to support more flexible pattern matching. Full details are in chapter 7 but briefly:

- Negative constraints, that prevent a rule from matching if certain other annotations are present (section 7.4).
- Additional matching operators for feature values, so you can now look for `{Token.length < 5}`, `{Lookup.minorType != "ignore"}`, etc. as well as simple equality (section 7.1).
- "Meta-property" accessors, to permit access to the string covered by an annotation, the length of the annotation, etc., e.g. `{Lookup@length > 4}`.

- Contextual operators, allowing you to search for one annotation contained within (or containing) another, e.g. `{Sentence contains {Lookup.majorType == "location"}}` (see section 7.1.4).
- Additional Kleene operator for ranges, e.g. `({Token})[2,5]` matches between 2 and 5 consecutive tokens.
- Additional operators can be added via runtime configuration.

Some of these extensions are similar to, but not the same as, those provided by the Montreal Transducer plugin. If you are already familiar with the Montreal Transducer, you should first look at section 7.11 which summarises the differences.

Resource configuration via Java 5 Annotations

Introduced an alternative style for supplying resource configuration information via Java 5 annotations rather than in `creole.xml`. The previous approach is still fully supported as well, and the two styles can be freely mixed. See section 4.9 for full details.

Ontology-based Gazetteer

Added a new plugin Ontology_Based_Gazetteer, which contains OntoRoot Gazetteer – a dynamically created gazetteer which is, in combination with few other generic GATE resources, capable of producing ontology-aware annotations over the given content with regards to the given ontology. For more details see section 9.31.

Inter-annotator agreement and merging

New plugins to support tasks involving several annotators working on the same annotation task on the same documents. The "iaaPlugin" (section 13.6) computes inter-annotator agreement scores between the annotators, the "copyAS2AnoDoc" plugin (section 9.33) copies annotations from several parallel documents into a single master document, and the "annotationMerging" plugin (section 9.30) merges annotations from multiple annotators into a single "consensus" annotation set.

Packaging self-contained applications for GATE Teamware

Added a mechanism to assemble a saved GATE application along with all the resource files it uses into a single self-contained package to run on another machine (e.g. as a service in GATE Teamware). This is available as a menu option (section 3.24) which will work for most common cases, but for complex cases you can use the underlying Ant task described in section C.2.

GUI improvements

- A new schema-driven tool to streamline manual annotation tasks (see section 3.19.1).
- Context-sensitive help on elements in the resource tree and when pressing F1 key. Search in mailing list from the Help menu. Help is displayed in your browser or in a Java browser if you don't have one.
- Improved search function inside documents with a regular expression builder. Search and replace annotation function in all annotation editors.
- Remember for each resource type the last path used when loading/saving a resource.
- Remember the last annotations selected in the annotation set view when you shift click on the annotation set view button.
- Improved context menu and when possible added drag and drop in: resource tree, annotation set view, annotation list view, corpus view, controller view. Context menu key can be now used if you have Java 1.6.
- New dialog box for error messages with user oriented messages, optional display of the configuration and proposing some useful actions. This will progressively replace the old stack trace dump into the message panel which is still here for the moment but should be hide by default in the future.
- Add read-only document mode that can be enable from the Options menu.
- Add a selection filter in the status bar of the annotations list table to easily select rows based on the text you enter.
- Add the last five applications loaded/saved in the context menu of the language resources in the resources tree.
- Display more informations on what going's on in the waiting dialog box when running an application. The goal is to improve it to get a global progress bar and estimated time.

2.1.2 Other new features and improvements

- New parser plugins: A new plugin for the Stanford Parser (see section 9.13) and a rewritten plugin for the RASP NLP tools (section 9.11).
- A new sentence splitter, based on regular expressions, has been added to the ANNIE plugin. More details in section 8.4.

- “Real-time” corpus controller (section 4.6), which terminates processing of a document if it takes longer than a configurable timeout..
- Major update to Annie OrthoMatcher coreference engine. Now correctly matches the sequence ”David Jones ... David ... David Smith ... David” as referring to two people. Also handles nicknames (David = Dave) via a new nickname list. Added optional parameter ”highPrecisionOrgs”, which if set to true turns off riskier org matching rules. Many misc. bug fixes.
- Improved alignment editor (chapter 12) with several advanced features and an API for adding your own actions to the editor.
- A new plugin for Chinese word segmentation, which is based on our work using machine learning algorithms for the Sighan-05 Chinese word segmentation task. It can learn a model from manually segmented text, and apply a learned model to segment Chinese text. In addition several learned models are available with the plugin, which can be used to segment text. For details about the plugin and those learned models see Section 9.32.
- New features in the ML API to produce an n-gram based language model from a corpus and a so-called “document-term matrix” (see section 9.19). Also introduced features to support active learning, a new learning algorithm (PAUM) and various optimisations including the ability to use an external executable for SVM training. Full details in chapter 11.
- A new plugin to compute BDM scores for an ontology. The BDM score can be used to evaluate ontology based information extraction and classification. For details about the plugin see Section 13.7.
- Added new “getCovering” method to AnnotationSet. This method returns annotations that completely span the provided range. An optional annotation type parameter can be provided to further limit the returned set.
- Complete redesign of ANNIC GUI. More details in section 9.29.

2.1.3 Specific bug fixes

- HTML document format parser: several bugs fixed, including a null pointer exception if the document contained certain characters illegal in HTML (#1754749). Also, the HTML parser now respects the “Add space on markup unpack” configuration option – previously it would always add space, even if the option was set to false.
- Fixed a severe performance bug in the Annie Pronominal Coreferencer resulting in a 50X speed improvement.
- JAPE did not always correctly handle the case when the input and output annotation sets for a transducer were different. This has now been fixed.

- 'Save preserving document format' was not correctly escaping ampersands and less than signs when two HTML entities are close together. Only the first one was replaced: A & B & C was output as A &amp; B & C instead of A &amp; B &amp; C. This has now been fixed, and the fix is also valid for the flexible exporter but only if the standoff annotations parameter is set to false.

Plus many more minor bug fixes

2.2 Version 4.0 (July 2007)

2.2.1 Major new features

ANNIC

ANNotations In Context: a full-featured annotation indexing and retrieval system designed to support corpus querying and JAPE rule authoring. It is provided as part of an extention of the Serial Datastores, called Searchable Serial Datastore (SSD). See section 9.29 for more details.

New machine learning API

A brand new machine learning layer specifically targetted at NLP tasks including text classification, chunk learning (e.g. for named entity recognition) and relation learning. See chapter 11 for more details.

Ontology API

A new ontology API, based on OWL In Memory (OWLIM), which offers a better API, revised ontology event model and an improved ontology editor to name but few. See chapter 10 for more details.

OCAT

Ontology-based Corpus Annotation Tool to help annotators to manually annotate documents using ontologies. For more details please see section 10.9.

Alignment Tools

A new set of components (e.g. CompoundDocument, AlignmentEditor etc.) that help in building alignment tools and in carrying out cross-document processing. See chapter 12 for more details.

New HTML Parser

A new HTML document format parser, based on Andy Clark's NekoHTML. This parser is much better than the old one at handling modern HTML and XHTML constructs, JavaScript blocks, etc., though the old parser is still available for existing applications that depend on its behaviour.

Java 5.0 support

GATE now requires Java 5.0 or later to compile and run. This brings a number of benefits:

- Java 5.0 syntax is now available on the right hand side of JAPE rules with the default Eclipse compiler. See section B.8 for details.
- `enum` types are now supported for resource parameters. see section 3.13 for details on defining the parameters of a resource.
- `AnnotationSet` and the `CreoleRegister` take advantage of generic types. The `AnnotationSet` interface is now an extension of `Set<Annotation>` rather than just `Set`, which should make for cleaner and more type-safe code when programming to the API, and the `CreoleRegister` now uses parameterized types, which are backwards-compatible but provide better type-safety for new code.

2.2.2 Other new features and improvements

- Hiding the view for a particular resource (by right clicking on its tab and selecting "Hide this view") will now completely close the associated viewers and dispose them. Re-selecting the same resource at a later time will lead to re-creating the necessary viewers and displaying them. This has two advantages: firstly it offers a mechanism for disposing views that are not needed any more without actually closing the resource and secondly it provides a way to refresh the view of a resource in the situations where it becomes corrupted.
- The DataStore viewer now allows multiple selections. This lets users load or delete an arbitrarily large number of resources in one operation.

- The Corpus editor has been completely overhauled. It now allows re-ordering of documents as well as sorting the document list by either index or document name.
- Support has been added for resource parameters of type `gate.FeatureMap`, and it is also possible to specify a default value for parameters whose type is `Collection`, `List` or `Set`. See section 3.13 for details.
- (Feature Request #1446642) After several requests, a mechanism has been added to allow overriding of GATE's document format detection routine. A new creation-time parameter `mimeType` has been added to the standard document implementation, which forces a document to be interpreted as a specific MIME type and prevents the usual detection based on file name extension and other information. See section 6.5.1 for details.
- A capability has been added to specify arbitrary sets of additional features on individual gazetteer entries. These features are passed forward into the Lookup annotations generated by the gazetteer. See section 8.2 for details.
- As an alternative to the Google plugin, a new plugin called *yahoo* has been added to GATE to allow users to submit their query to the Yahoo search engine and to load the found pages as GATE documents. See section 9.22 for more details.
- It is now easier to run a corpus pipeline over a single document in the GATE GUI – documents now provide a right-click menu item to create a singleton corpus containing just this document. See section 3.12.1 for details.
- A new interface has been added that lets PRs receive notification at the start and end of execution of their containing controller. This is useful for PRs that need to do cleanup or other processing after a whole corpus has been processed. See section 4.6 for details.
- The GATE GUI does not call System.exit() any more when it is closed. Instead an effort is made to stop all active GATE threads and to release all GUI resources, which leads to the JVM exiting gracefully. This is particularly useful when GATE is embedded in other systems as closing the main GATE window will not kill the JVM process any more.
- The set of AnnotationSchemas that used to be included in the core gate.jar and laoded as builtins have now been moved to the ANNIE plugin. When the plugin is loaded, the default annotation schemas are instantiated automatically and are available when doing manual annotation.
- There is now support in creole.xml files for automatically creating instances of a resource that are hidden (i.e. do not show in the GUI). One example of this can be seen in the creole.xml file of the ANNIE plugin where the default annotation schemas are defined.

- A couple of helper classes have been added to assist in using GATE within a Spring application. Section 3.29 explains the details.
- Improvements have been made to the thread-safety of some internal GATE components, which mean that it is now safe to create resources in multiple threads (though it is not safe to use the same resource instance in more than one thread). This is a big advantage when using GATE in a multithreaded environment, such as a web application. See section 3.31 for details.
- Plugins can now provide custom icons for their PRs and LRs in the plugin JAR file. See section 3.13 for details.
- It is now possible to override the default location for the saved session file using a system property. See section 3.3 for details.
- The TreeTagger plugin supports a system property to specify the location of the shell interpreter used for the tagger shell script. In combination with Cygwin this makes it much easier to use the tagger on Windows. See section 9.7 for details.
- The `Buchart` plugin has been removed. It is superseded by SUPPLE, and instructions on how to upgrade your applications from Buchart to SUPPLE are given in section 9.12. The probability finder plugin has also been removed, as it is no longer maintained.
- The bootstrap wizard now creates a basic plugin that builds with Ant. Since a Unix-style make command is no longer required this means that the generated plugin will build on Windows without needing Cygwin or MinGW.
- The GATE source code has moved from CVS into Subversion. See section 3.2.3 for details of how to check out the code from the new repository.
- An optional parameter, keepOriginalMarkupsAS, has been added to the DocumentReset PR which allows users to decide whether to keep the Original Markups AS or not while reseting the document. See section 9.1 for more details.

2.2.3 Bug fixes and optimizations

- The Morphological Analyser has been optimized. A new FSM based, although with minor alteration to the basic FSM algorithm, has been implemented to optimize the GATE Morphological Analyser. The previous profiling figures show that the morpher when integrated with ANNIE application used to take upto 60% of the overall processing time. The optimized version only takes 7.6% of the total processing time. See section 9.9 for more details on the morpher.
- The ANNIE Sentence Splitter was optimised. The new version is about twice as fast as the previous one. The actual speed increase varies widely depending on the nature of the document.

- The imlementation of the *OrthoMatcher* component has been improved. This resources takes significantly less time on large documents.
- The implementation of `AnnotationSets` has been improved. GATE now requires up to 40% less memory to run and is also 20% faster on average. The `get` methods of `AnnotationSet` return instances of `ImmutableAnnotationSet`. Any attempt at modifying the content of these objects will trigger an `Exception`. An empty `ImmutableAnnotationSet` is returned instead of `null`.
- The Chemistry tagger (section 9.16) has been updated with a number of bugfixes and improvements.
- The Document user interface has been optimised to deal better with large bursts of events which tend to occur when the document that is currently displayed gets modified. The main advantages brought by this new implementation are:
  - The document UI refreshes faster than before.
  - The presence of the GUI for a document induces a smaller performance penalty than it used to. Due to a better threading implementation, machines benefiting from multiple CPUs (e.g. dual CPU, dual core or hyperthreading machines) should only see a negligible increase in processing time when a document is displayed compared to the situations where the document view is not shown. In the previous version, displaying a document while it was processed used to increase execution time by an order of magnitude.
  - The GUI is more responsive now when a large number of annotations are displayed, hidden or deleted.
  - The strange exceptions that used to occur occasionally while working with the document GUI should not happen any more.

And as always there are many smaller bugfixes too numerous to list here...

2.3 Version 3.1 (April 2006)

2.3.1 Major new features

Support for UIMA

UIMA (http://www.research.ibm.com/UIMA/) is a language processing framework developed by IBM. UIMA and GATE share some functionality but are complementary in most respects. GATE now provides an interoperability layer to allow UIMA applications to include GATE components in their processing and vice-versa. For full information, see chapter 16.

New Ontology API

The ontology layer has been rewritten in order to provide an abstraction layer between the model representation and the tools used for input and output of the various representation formats. An implementation that uses Jena 2 (http://jena.sourceforge.net/ontology) for reading and writing OWL and RDF(S) is provided.

Ontotext Japec Compiler

Japec is a compiler for JAPE grammars developed by Ontotext Lab. It has some limitations compared to the standard JAPE transducer implementation, but can run JAPE grammars up to five times as fast. By default, GATE still uses the stable JAPE implementation, but if you want to experiment with Japec, see section 9.28.

2.3.2 Other new features and improvements

- Addition of a new JAPE matching style "all". This is similar to Brill, but once all rules from a given start point have matched, the matching will continue from the next offset to the current one, rather than from the position in the document where the longest match finishes. More details can be found in Section 7.3.
- Limited support for loading PDF and Microsoft Word document formats. Only the text is extracted from the documents, no formatting information is preserved.
- The Buchart parser has been deprecated and replaced by a new plugin called SUPPLE - the Sheffield University Prolog Parser for Language Engineering. Full details, including information on how to move your application from Buchart to SUPPLE, is in section 9.12.
- The Hepple POS Tagger is now open-source. The source code has been included in the GATE distribution, under src/hepple/postag. More information about the POS Tagger can be found in Section 8.5.
- Minipar is now supported on Windows. *minipar-windows.exe*, a modified version of *pdemo.cpp* is added under the gate/plugins/minipar directory to allow users to run Minipar on windows platform. While using Minipar on Windows, this binary should be provided as a value for *miniparBinary* parameter. For full information on Minipar in GATE, see section 9.10.
- The XmlGateFormat writer(Save As Xml from GATE GUI, gate.Document.toXml() from GATE API) and reader have been modified to write and read GATE annotation IDs. For backward compatibility reasons the old reader has been kept. This change fixes a bug which manifested in the following situation: If a GATE document had annotations carrying features of which values were numbers representing other GATE

annotation IDs, after a save and a reload of the document to and from XML, the former values of the features could have become invalid by pointing to other annotations. By saving and restoring the GATE annotation ID, the former consistency of the GATE document is maintained. For more information, see Section 6.5.2.

- The NP chunker and chemistry tagger plugins have been updated. Mark Greenwood has relicenced them under the LGPL, so their source code has been moved into the GATE distribution. See sections 9.3 and 9.16 for details.
- The Tree Tagger wrapper has been updated with an option to be less strict when characters that cannot be represented in the tagger's encoding are encountered in the document. Details are in section 9.7.
- JAPE Transducers can be serialized into binary files. The option to load serialized version of JAPE Transducer (an init-time parameter *binaryGrammarURL*) is also implemented which can be used as an alternative to the parameter *grammarURL*. More information can be found in Section 7.9.
- On Mac OS, GATE now behaves more 'naturally'. The application menu items and keyboard shortcuts for *About* and *Preferences* now do what you would expect, and exiting GATE with command-Q or the *Quit* menu item properly saves your options and current session.
- Updated versions of Weka(3.4.6) and Maxent(2.4.0).
- Optimisation in *gate.creole.ml*: the conversion of AnnotationSet into ML examples is now faster.
- It is now possible to create your own implementation of `Annotation`, and have GATE use this instead of the default implementation. See `AnnotationFactory` and `AnnotationSetImpl` in the `gate.annotation` package for details.

2.3.3 Bug fixes

- The Tree Tagger wrapper has been updated in order to run under Windows. See 9.7.
- The SUPPLE parser has been made more user-friendly. It now produces more helpful error messages if things go wrong. Note that you will need to update any saved applications that include SUPPLE to work with this version - see section 9.12 for details.
- Miscellaneous fixes in the Ontotext JapeC compiler.
- Optimization : the creation of a Document is much faster.
- Google plugin: The optional pagesToExclude parameter was causing a NullPointerException when left empty at run time. Full details about the plugin functionality can be found in section 9.21.

- Minipar, SUPPLE, TreeTagger: These plugins that call external processes have been fixed to cope better with path names that contain spaces. Note that some of the external tools themselves still have problems handling spaces in file names, but these are beyond our control to fix. If you want to use any of these plugins, be sure to read the documentation to see if they have any such restrictions.
- When using a non-default location for GATE configuration files, the configuration data is saved back to the correct location when GATE exits. Previously the default locations were always used.
- Jape Debugger: ConcurrentModificationException in JAPE debugger. The JAPE debugger was generating a ConcurrentModificationException during an attempt to run ANNIE. There is no exception when running without the debugger enabled. As result of fixing one unnesesary and incorrect callback to debugger was removed from SinglePhaseTransducer class.
- Plus many other small bugfixes...

2.4 January 2005

Release of version 3.

New plugins for processing in various languages (see 9.15). These are not full IE systems but are designed as starting points for further development (French, German, Spanish, etc.), or as sample or toy applications (Cebuano, Hindi, etc.).

Other new plugins:

- Chemistry Tagger 9.16
- Montreal Transducer 9.14
- RASP Parser 9.11
- MiniPar 9.10
- Buchart Parser 9.12
- MinorThird 9.25
- NP Chunker 9.3
- Stemmer 9.8
- TreeTagger 9.7
- Probability Finder

- Crawler 9.20
- Google PR 9.21

Support for SVM Light, a support vector machine implementation, has been added to the machine learning plugin (see section 9.24.7).

2.5 December 2004

GATE no longer depends on the Sun Java compiler to run, which means it will now work on any Java runtime environment of at least version 1.4. JAPE grammars are now compiled using the Eclipse JDT Java compiler by default.

A welcome side-effect of this change is that it is now much easier to integrate GATE-based processing into web applications in Tomcat. See section 3.30 for details.

2.6 September 2004

GATE applications are now saved in XML format using the XStream library, rather than by using native java serialization. On loading an application, GATE will automatically detect whether it is in the old or the new format, and so applications in both formats can be loaded. However, older versions of GATE will be unable to load applications saved in the XML format. (A `java.io.StreamCorruptedException: invalid stream header exception` will occcur.) It is possible to get new versions of GATE to use the old format by setting a flag in the source code. (See the Gate.java file for details.) This change has been made because it allows the details of an application to be viewed and edited in a text editor, which is sometimes easier than loading the application into GATE.

2.7 Version 3 Beta 1 (August 2004)

Version 3 incorporates a lot of new functionality and some reorganisation of existing components.

Note that Beta 1 is feature-complete but needs further debugging (please send us bug reports!).

Highlights include: completely rewritten document viewer/editor; extensive ontology support; a new plugin management system; separate .jar files and a Tomcat classloading fix; lots more CREOLE components (and some more to come soon).

Almost all the changes are backwards-compatible; some recent classes have been renamed (particularly the ontologies support classes) and a few events added (see below); datastores created by version 3 will probably not read properly in version 2. If you have problems use the mailing list and we'll help you fix your code!

The gorey details:

- Anonymous CVS is now available. See section 3.2.3 for details.
- CREOLE repositories and the components they contain are now managed as plugins. You can select the plugins the system knows about (and add new ones) by going to "Manage CREOLE Plugins" on the file menu.
- The `gate.jar` file no longer contains all the subsidiary libraries and CREOLE component resources. This makes it easier to replace library versions and/or not load them when not required (libraries used by CREOLE builtins will now not be loaded unless you ask for them from the plugins manager console).
- ANNIE and other bundled components now have their resource files (e.g. pattern files, gazetteer lists) in a separate directory in the distribution – `gate/plugins`.
- Some testing with Sun's JDK 1.5 pre-releases has been done and no problems reported.
- The `gate://` URL system used to load CREOLE and ANNIE resources in past releases is no longer needed. This means that loading in systems like Tomcat is now much easier.
- MAC OS X is now properly supported by the installed and the runtime.
- An Ontology-based Corpus Annotation Tool (OCAT) has been implemented as a GATE plugin. Documentation of its functionality is in Section 10.9.
- The NLG Lexical tools from the MIAKT project have now been released. See documentation in Section 9.26.
- The Features viewer/editor has been completely updated – see Sections 3.16 and 3.19 for details.
- The Document editor has been completely rewritten – see Section 3.6 for more information.
- The datastore viewer is now a full-size VR – see Section 3.22 for more information.

2.8 July 2004

GATE Documents now fire events when the document content is edited. This was added in order to support the new facility of editing documents from the GUI. This change will break

backwards compatibility by requiring all DocumentListener implementations to implement a new method:
`public void contentEdited(DocumentEvent e);`

2.9 June 2004

A new algorithm has been implemented for the AnnotationDiff function. A new, more usable, GUI is included, and an ”Export to HTML” option added. More details about the AnnotationDiff tool are in Section 3.25.

A new build process, based on ANT (http://ant.apache.org/) is now available for GATE. The old build process, based on make, is now unsupported. See Section 3.8 for details of the new build process.

A Jape Debugger from Ontos AG has been integrated in GATE. You can turn integration ON with command line option ”-j”. If you run the GATE GUI with this option, the new menu item for Jape Debugger GUI will appear in the Tools menu. The default value of integration is OFF. We are currently awaiting documentation for this.

NOTE! Keep in mind there is ClassCastExceprion if you try to debug ConditionalCorpusPipeline. Jape Debugger is designed for Corpus Pipeline only. The Ontos code needs to be changed to allow debugging of ConditionalCorpusPipeline.

2.10 April 2004

GATE now has two alternative strategies for ontology-aware grammar transduction:

- using the [ontology] feature both in grammars and annotations; with the default Transducer.
- using the ontology aware transducer – passing an ontology LR to a new subsume method in the SimpleFeatureMapImpl. the latter strategy does not check for ontology features (this will make the writing of grammars easier – no need to specify ontology).

The changes are in:

- SinglePhaseTransducer (always call subsume with ontology – if null then the ordinary subsumption takes place)
- SimpleFeatureMapImpl (new subsume method using an ontology LR)

More information about the ontology-aware transducer can be found in Section 10.6.

A morphological analyser PR has been added to GATE. This finds the root and affix values of a token and adds them as features to that token.

A flexible gazetteer PR has been added to GATE. This performs lookup over a document based on the values of an arbitrary feature of an arbitrary annotation type, by using an externally provided gazetteer. See 9.5 for details.

2.11 March 2004

Support was added for the MAXENT machine learning library. (See 9.24.6 for details.)

2.12 Version 2.2 – August 2003

Note that GATE 2.2 works with JDK 1.4.0 or above. Version 1.4.2 is recommended, and is the one included with the latest installers.

GATE has been adapted to work with Postgres 7.3. The compatibility with PostgreSQL 7.2 has been preserved. See 3.38 for more details.

New library version – Lucene 1.3 (rc1)

A bug in gate.util.Javac has been fixed in order to account for situations when String literals require an encoding different from the platform default.

Temporary .java files used to compile JAPE RHS actions are now saved using UTF-8 and the "-encoding UTF-8" option is passed to the javac compiler.

A custom tools.jar is no longer necessary

Minor changes have been made to the look and feel of GATE to improve its appearance with JDK 1.4.2

Some bug fixes (087, 088, 089, 090, 091, 092, 093, 095, 096 – see http://gate.ac.uk/gate/doc/bugs.html for more details).

2.13 Version 2.1 – February 2003

Integration of Machine Learning PR and WEKA wrapper (see Section 9.24).

Addition of DAML+OIL exporter.

Integration of WordNet in GATE (see Section 9.23).

The syntax tree viewer has been updated to fix some bugs.

2.14 June 2002

Conditional versions of the controllers are now available (see Section 3.15). These allow processing resources to be run conditionally on document features.

PostgreSQL Data Stores are now supported (see Section 4.7). These store data into a PostgreSQL RDBMS.

Addition of OntoGazetteer (see Section 5.2), an interface which makes ontologies visible within GATE, and supports basic methods for hierarchy management and traversal.

Integration of Protégé, so that people with developed Protégé ontologies can use them within GATE.

Addition of IR facilities in GATE (see Section 9.19).

Modification of the corpus benchmark tool (see Section 3.26), which now takes an application as a parameter.

See also for details of other recent bug fixes.

Chapter 3

How To...

"The law of evolution is that the strongest survives!"

"Yes; and the strongest, in the existence of any social species, are those who are most social. In human terms, most ethical. ... There is no strength to be gained from hurting one another. Only weakness."

The Dispossessed [p.183], Ursula K. le Guin, 1974.

This chapter describes how to complete common tasks using GATE.

Read first the sections with an asterisk (*).

Sections that relate to the Development Environment are flagged [**D**]; those that relate to the framework are flagged [**F**]; sections relating to both are flagged [**D,F**].

There are two other primary sources for this type of information:

1. for the development environment, see the visual tutorials available on our movies page;
2. for the framework, see the example code at http://gate.ac.uk/gate-examples/doc/.

3.1 Download GATE*

To download GATE point your web browser at http://gate.ac.uk/download/.

You should next read the section 3.2 to install and run GATE.

3.2 Install and Run GATE*

GATE 3.1 will run anywhere that supports Java version 1.4.2 or later, including Solaris, Linux and Windoze platforms. GATE 4 and 5 require Java 5.0. We don't run tests on other platforms, but have had reports of successful installs elsewhere. We are also testing released installers on MacOS X.

You should next read the section 3.6 to know about the GUI.

3.2.1 The Easy Way

The easy way to install is to use one of the platform-specific installers (created using the excellent IzPack). Download a 'platform-specific installer' and follow the instructions it gives you. Once the installation is complete, you can start GATE using `gate.exe` (Windows) or `GATE.app` (Mac) in the top-level installation directory, or `gate.sh` in the bin directory (other platforms).

3.2.2 The Hard Way (1)

Download the Java-only release package or the binary build snapshot, and follow the instructions below.

Prerequisites:

- A conforming Java 2 environment,
  - version 1.4.2 or above for GATE 3.1
  - version 5.0 for GATE 4.0 beta 1 or later.

  available free from Sun Microsystems or from your UNIX supplier. (We test on various Sun JDKs on Solaris, Linux and Windows XP.)
- Binaries from the GATE distribution you downloaded: `gate.jar`, `lib/ext/guk.jar` (Unicode editing support) and a suitable script to start Ant, e.g. `ant.sh` or `ant.bat`. These are held in a directory called `bin` like this:

```
.../bin/
    gate.jar
    ant.sh
    ant.bat
```

You will also need the `lib` directory, containing various libraries that GATE depends on.

- An open mind and a sense of humour.

Using the binary distribution:

- Unpack the distribution, creating a directory containing jar files and scripts.
- To run the development environment: on Windows, start a Command Prompt window, change to the directory where you unpacked the GATE distribution and run `‘‘bin/ant.bat run’’`; on UNIX run `‘‘bin/ant run’’`.
- To embed GATE as a library, put `gate.jar` and all the libraries in the lib directory in your `CLASSPATH` and tell Java that guk.jar is an extension (`-Djava.ext.dirs=path-to-guk.jar`).

The Ant scripts that start GATE (`ant.bat` or `ant`) requires you to set the `JAVA_HOME` environment variable to point to the top level directory of your JAVA installation. The value of `GATE_CONFIG` is passed to the system by the scripts using either a `-i` command-line option, or the Java property `gate.config`.

3.2.3 The Hard Way (2): Subversion

The GATE code is maintained in a Subversion repository. You can use a Subversion client to check out the source code – the most up-to-date version of GATE is the trunk:

```
svn checkout https://gate.svn.sourceforge.net/svnroot/gate/gate/trunk gate
```

Once you have checked out the code you can build GATE using Ant (see section 3.8)

You can browse the complete Subversion repository online at http://gate.svn.sourceforge.net/gate.

3.3 [D,F] Use System Properties with GATE

During initialisation, GATE reads several Java system properties in order to decide where to find its configuration files.

Here is a list of the properties used, their default values and their meanings:

gate.home sets the location of the GATE install directory. This should point to the top level directory of your GATE installation. This is the only property that is required. If this is not set, the system will display an error message and them it will attempt to guess the correct value.

gate.plugins.home points to the location of the directory containing installed GATE plug-ins (a.k.a. CREOLE directories). If this is not set then the default value of {`gate.home`}`/plugins` is used.

gate.site.config points to the location of the configuration file containing the site-wide options. If not set this will default to {`gate.home`}`/gate.xml`. The site configuration file must exist!

gate.user.config points to the file containing the user's options. If not specified, or if the specified file does not exist at startup time, the default value of gate.xml (.gate.xml on Unix platforms) in the user's home directory is used.

gate.user.session points to the file containing the user's saved session. If not specified, the default value of gate.session (.gate.session on Unix) in the user's home directory is used. When starting up the GUI the session is reloaded from this file if it exists, and when exiting the GUI the session is saved to this file (unless the user has disabled "save session on exit" in the configuration dialog). The session is not used when using GATE as a library.

load.plugin.path is a path-like structure, i.e. a list of URLs separated by ';'. All directories listed here will be loaded as CREOLE plugins during initialisation. This has similar functionality with the the `-d` command line option.

gate.builtin.creole.dir is a URL pointing to the location of GATE's built-in CREOLE directory. This is the location of the `creole.xml` file that defines the fundamental GATE resource types, such as documents, document format handlers, controllers and the basic visual resources that make up the GATE GUI. The default points to a location inside `gate.jar` and should not generally need to be overridden.

When using GATE as a library, you can set the values for these properties before you call `Gate.init()`. Alternatively, you can set the values programmatically using the static methods `setGateHome()`, `setPluginsHome()`, `setSiteConfigFile()`, etc. before calling `Gate.init()`. See the Javadoc documentation for details. If you want to set these values from the command line you can use the following syntax for setting `gate.home` for example:

```
java -Dgate.home=/my/new/gate/home/directory -cp...  gate.Main
```

When running the GUI, you can set the properties by creating a file `build.properties` in the top level GATE directory. In this file, any system properties which are prefixed with "`run.`" will be passed to GATE. For example, to set an alternative user config file, put the following line in `build.properties`[1]:

```
run.gate.user.config=${user.home}/alternative-gate.xml
```

[1]In this specific case, the alternative config file must already exist when GATE starts up, so you should copy your standard `gate.xml` file to the new location.

This facility is not limited to the GATE-specific properties listed above, for example the following line changes the default temporary directory for GATE (note the use of forward slashes, even on Windows platforms):

```
run.java.io.tmpdir=d:/bigtmp
```

3.4 [D,F] Use (CREOLE) Plug-ins

The definitions of CREOLE resources (see Chapter 4) are stored in CREOLE directories (directories containing an XML file describing the resources, the Java archive with the compiled executable code and whatever libraries are required by the resources).

Starting with version 3, CREOLE directories are called "CREOLE Plugins" or simply "Plugins". In previous versions, the CREOLE resources distributed with GATE used be included in the monolithic `gate.jar` archive. Version 3 includes them as separate directories under the `plugins` directory of the distribution. This allows easy access to the linguistic resources used without the requirement to unpack the `gate.jar` file.

Plugins can have one or more of the following states in relation with GATE:

known plugins are those plugins that the system knows about. These include all the plugins in the `plugins` directory of the GATE installation (the so–called *installed* plugins) as well all the plugins that were manually loaded from the user interface.

loaded plugins are the plugins currently loaded in the system. All CREOLE resource types from the loaded plugins are available for use. All known plugins can easily be loaded and unloaded using the user interface.

auto-loadable plugins are the list of plugins that the system loads automatically during initialisation.

The default location for installed plugins can be modified using the `gate.plugins.home` system property while the list of auto-loadable plugins can be set using the `load.plugin.path` property, see Section 3.3 above.

The CREOLE plugins can be managed through the graphical user interface which can be activated by selecting "Manage CREOLE plugins" from the "File" menu. This will bring up a window listing all the known plugins. For each plugin there are two check-boxes – one labelled "Load now", which will load the plugin, and the other labelled "Load always" which will add the plugin to the list of auto-loadable plugins. A "Delete" button is also provided – which will remove the plugin from the list of known plugins. Note the the installed plugins will return to the list of known plugins next time when GATE is started. They can only be removed by physically removing (or moving) the actual directory on disk outside the GATE plugins directory.

When using GATE as a library the following API calls are relevant to working with plugins:

Class gate.Gate

`public static void addKnownPlugin(URL pluginURL)` adds the plugin to the list of known plugins.

`public static void removeKnownPlugin(URL pluginURL)` tells the system to "forget" about one previously known directory. If the specified directory was loaded, it will be unloaded as well - i.e. all the metadata relating to resources defined by this directory will be removed from memory.

`public static void addAutoloadPlugin(URL pluginUrl)` adds a new directory to the list of plugins that are loaded automatically at start-up.

`public static void removeAutoloadPlugin(URL pluginURL)` tells the system to remove a plugin URL from the list of plugins that are loaded automatically at system start-up. This will be reflected in the user's configuration data file.

Class gate.CreoleRegister

`public void registerDirectories(URL directoryUrl)` loads a new CREOLE directory. The new plugin is added to the list of known plugins if not already there.

`public void removeDirectory(URL directory)` unloads a loaded CREOLE plugin.

3.5 Troubleshooting

On Windoze 95 and 98, you may need to increase the amount of **environment space** available for the `gate.bat` script. Right click on the script, hit the memory tab and increase the 'initial environment' value to maximum.

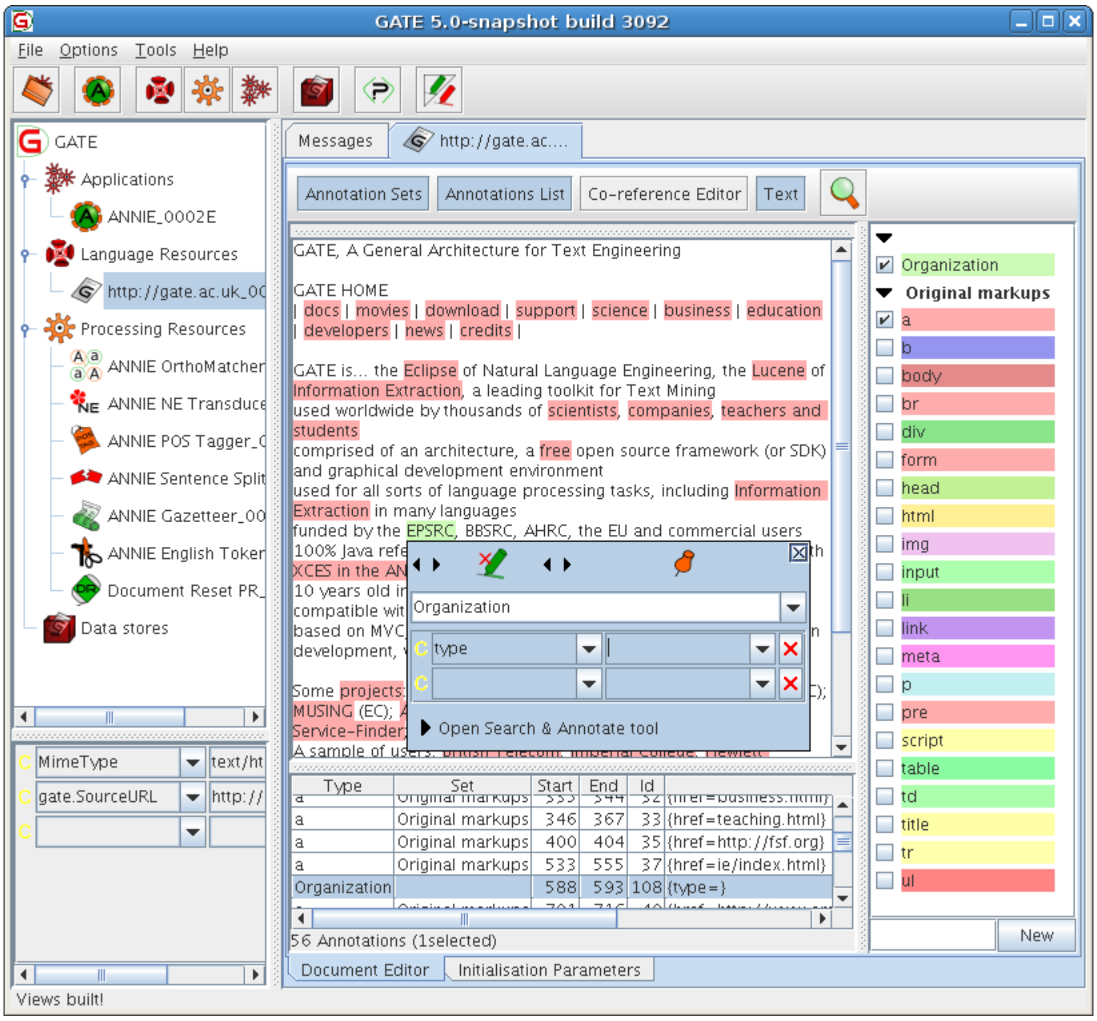
Note that the `gate.bat` script uses `javaw.exe` to run GATE which means that you will see no console for the java process. If you have problems starting GATE and you would like to be able to see the console to check for messages then you should edit the `gate.bat` script and replace `javaw.exe` with `java.exe` in the definition of the `JAVA` environment variable.

When our FTP server is overloaded you may get a **blank download link** in the email sent to you after you register. Please try again later.

3.6 [D] Get Started with the GUI*

Probably the best way to learn how to use the GATE graphical development environment is to look at the demonstrations and tutorials movies. There is specific links to them in this chapter.

This section gives a short description of what is where in the main window of the system.



MimeType	text/ht
gate.SourceURL	http://

GATE, A General Architecture for Text Engineering

GATE HOME
| docs | movies | download | support | science | business | education
| developers | news | credits |

GATE is... the Eclipse of Natural Language Engineering, the Lucene of Information Extraction, a leading toolkit for Text Mining
used worldwide by thousands of scientists, companies, teachers and students
comprised of an architecture, a free open source framework (or SDK) and graphical development environment
used for all sorts of language processing tasks, including Information Extraction in many languages
funded by the EPSRC, BBSRC, AHRC, the EU and commercial users
100% Java refe
XCES in the AN
10 years old in
compatible wit
based on MVC
development,

Some projects:
MUSING (EC);
Service-Finder;
A sample of users:

Organization

type

Type	Set	Start	End	Id	
a	Original markups	346	367	33	{href=teaching.html}
a	Original markups	400	404	35	{href=http://fsf.org}
a	Original markups	533	555	37	{href=ie/index.html}
Organization		588	593	108	{type=}

56 Annotations (1selected)

Figure 3.1: Main Window

Figure 3.1 shows the main window of the application, with a single document loaded. There are five main areas of the window:

1. the *menus bar* along the top, with 'File' etc.;
2. in the top left of the main area, a tree starting from 'GATE' and containing 'Applications', 'Language Resources' etc. – this is the *resources tree*;

3. in the bottom left of the main area, a rectangle, which is the *small resource viewer*;
4. on the right of the main area, containing tabs with 'Messages' or the name of a resource from the resource tree, the *main resource viewer*;
5. the *messages bar* along the bottom (where it says 'Views built!').

The menu and the messages bar do the usual things. Longer messages are displayed in the messages tab in the main resource viewer area.

The resource tree and resource viewer areas work together to allow the system to display diverse resources in various ways. Visual Resources integrated with GATE can have a small view or a large view. For example, data stores have a small view; documents have a large view.

All the resources, applications and datastores currently loaded in the system appear in the resources tree; double clicking on a resource will load a viewer for the resource in one of the resource view areas.

You should next read the section 3.12 to load creole resources.

3.7 [D,F] Configure GATE

When the GATE development environment is started, or when `Gate.init()` is called from the API, GATE loads various sorts of configuration data stored as XML in files generally called something like `gate.xml` or `.gate.xml`. This data holds information such as:

- whether to save settings on exit;
- what fonts the GUI should use;
- where the local Oracle database lives.

All of this type of data is stored at two levels (in order from general to specific):

- the site-wide level, which by default is located the `gate.xml` file in top level directory of the GATE installation (i.e. the `GATE home`. This location can be overridden by the Java system property `gate.site.config`;
- the user level, which lives in the user's HOME directory on UNIX or their profile directory on Windoze (note that parts of this file are overwritten by GATE when saving user settings). The default location for this file can be overridden by the Java system property `gate.user.config`.

Where configuration data appears on several different levels, the more specific ones overwrite the more general. This means that you can set defaults for all GATE users on your system, for example, and allow individual users to override those defaults without interfering with others.

Configuration data can be set from the GUI via the ‘Options’ menu, ‘Configuration’ choice. The user can change the appearance of the GUI (via the Appearance submenu), which includes the options of font and the “look and feel”. The “Advanced” submenu enables the user to include annotation features when saving the document and preserving its format, to save the selected Options automatically on exit, and to save the session automatically on exit. The Input Methods menu (available via the Options menu) enables the user to change the default language for input. These options are all stored in the user’s .gate.xml file.

When using GATE from the framework, you can also set the site config location using `Gate.setSiteConfigFile(File)` prior to calling `Gate.init()`.

3.7.1 [F] Save Config Data to gate.xml

Arbitrary feature/value data items can be saved to the user’s `gate.xml` file via the following API calls:

To get the config data: `Map configData = Gate.getUserConfig()`.

To add config data simply put pairs into the map: `configData.put("my new config key", "value");`.

To write the config data back to the XML file: `Gate.writeUserConfig();`.

Note that new config data will simply override old values, where the keys are the same. In this way defaults can be set up by putting their values in the main gate.xml file, or the site gate.xml file; they can then be overridden by the user’s gate.xml file.

3.8 Build GATE

Note that you don’t need to build GATE unless you’re doing development on the system itself.

Prerequisites:

- A conforming Java environment as above.

- A copy of the GATE sources and the build scripts – either the SRC distribution package from the nightly snapshots or a copy of the code obtained through Subversion (see Section 3.2.3).
- An appreciation of natural beauty.

GATE now includes a copy of the ANT build tool which can be accessed through the scripts included in the `bin` directory (use `ant.bat` for Windows 98 or ME, `ant.cmd` for Windows NT, 2000 or XP, and `ant.sh` for Unix platforms).

To build gate, cd to gate and:

1. Type:
   `bin/ant`
2. [optional] To test the system:
   `bin/ant test`
   (Note that DB tests may fail unless you can connect to Sheffield's Oracle server.)
3. [optional] To make the Javadoc documentation:
   `bin/ant doc`
4. You can also run GATE using Ant, by typing:
   `bin/ant run`
5. To see a full list of options type: `bin/ant help`

(The details of the build process are all specified by the build.xml file in the gate directory.)

You can also use a development environment like Borland JBuilder (click on the `gate.jpx` file), but note that it's still advisable to use ant to generate documentation, the jar file and so on. Also note that the run configurations have the location of a `gate.xml` site configuration file hard-coded into them, so you may need to change these for your site.

3.9 [D] Use GATE with Maven or JPF

This section is based on contributions by Georg ttl and William Oberman.

To use GATE with Maven you need a definition of the dependencies in POM format. There's an example POM here.

To use GATE with JPF (a Java plugin framework) you need a plugin definition like this one.

3.10 [D,F] Create a New (CREOLE) Resource

CREOLE resources are Java Beans (see chapter 4). They come in three types: Language Resource, Processing Resource and Visual Resource (see chapter 1 section 1.3.1). To create a new resource you need to:

- write a Java class that implements GATE's beans model;
- compile the class, and any others that it uses, into a Java Archive (JAR) file;
- write some XML configuration data for the new resource;
- tell GATE the URL of the new JAR and XML files.

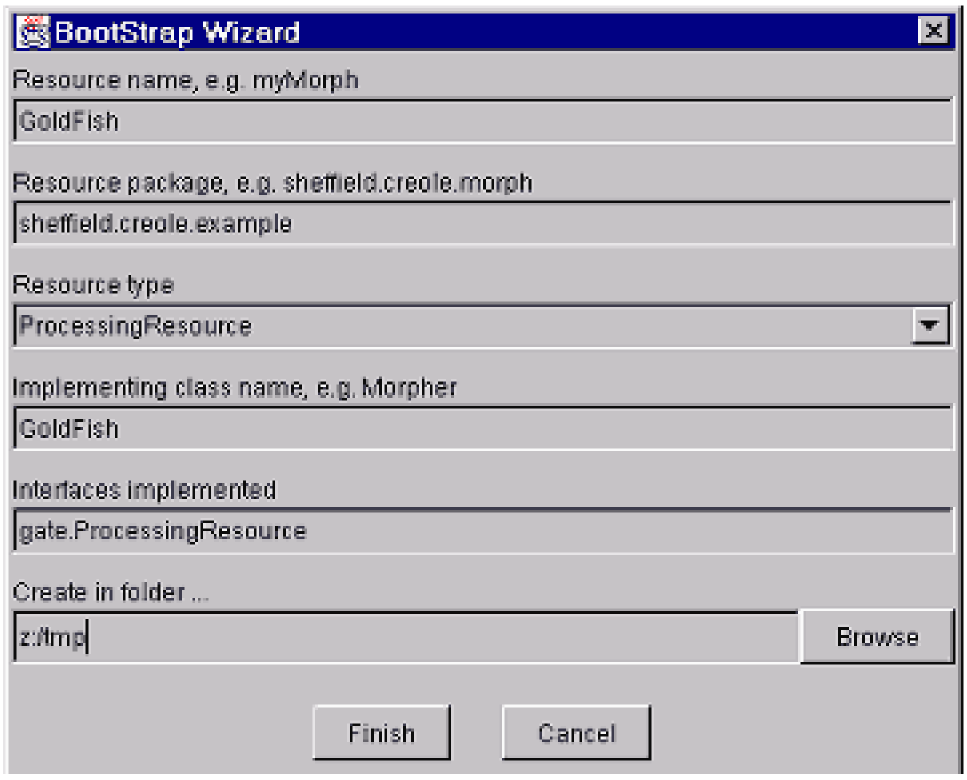
The GATE development environment helps you with this process by creating a set of directories and files that implement a basic resource, including a Java code file and a Makefile. This process is called 'bootstrapping'.

For example, let's create a new component called GoldFish, which will be a Processing Resource that looks for all instances of the word 'fish' in a document and adds an annotation of type 'GoldFish'.

First start the GATE development environment (see section 3.2). From the 'Tools' menu select 'BootStrap Wizard', which will pop up the dialogue in figure 3.2. The meaning of the data entry fields:

- The 'resource name' will be displayed when GATE loads the resource, and will be the name of the directory the resource lives in. For our example: `GoldFish`.
- 'Resource package' is the Java package that the class representing the resource will be created in. For our example: `sheffield.creole.example`.
- 'Resource type' must be one of Language, Processing or Visual Resource. In this case we're going to process documents (and add annotations to them), so we select `ProcessingResource`.
- 'Implementing class name' is the name of the Java class that represents the resource. For our example: `GoldFish`.
- The 'interfaces implemented' field allows you to add other interfaces (e.g. `gate.creole.ControllerAwarePR`[2]) that you would like your new resource to im-

[2]See section 4.6.


BootStrap Wizard

Resource name, e.g. myMorph

GoldFish

Resource package, e.g. sheffield.creole.morph

sheffield.creole.example

Resource type

ProcessingResource

Implementing class name, e.g. Morpher

GoldFish

Interfaces implemented

gate.ProcessingResource

Create in folder ...

z:/tmp

Figure 3.2: BootStrap Wizard Dialogue

plement. In this case we just leave the default (which is to implement the `gate.ProcessingResource` interface).

- The last field selects the directory that you want the new resource created in. For our example: `z:/tmp`.

Now we need to compile the class and package it into a JAR file. The bootstrap wizard creates an Ant build file that makes this very easy – so long as you have Ant set up properly, you can simply run

```
ant jar
```

This will compile the Java source code and package the resulting classes into `GoldFish.jar`. If you don't have your own copy of Ant, you can use the one bundled with GATE - suppose your GATE is installed at `/opt/gate-5.0-snapshot`, then you can use `/opt/gate-5.0-snapshot/bin/ant jar` to build.

You can now load this resource into GATE; see

- section 3.11 for how to instantiate the resource from the framework;

- section 3.12 for how to load the resource in the development environment;
- section 3.13 for how to configure and further develop your resource (which will, by default, do nothing!).

The default Java code that was created for our GoldFish resource looks like this:

```
/*
 *  GoldFish.java
 *
 *  You should probably put a copyright notice here. Why not use the
 *  GNU licence? (See http://www.gnu.org/.)
 *
 *  hamish, 26/9/2001
 *
 *  $Id: howto.tex,v 1.130 2006/10/23 12:56:37 ian Exp $
 */

package sheffield.creole.example;

import java.util.*;
import gate.*;
import gate.creole.*;
import gate.util.*;

/**
 * This class is the implementation of the resource GOLDFISH.
 */
@CreoleResource(name = "GoldFish",
        comment = "Add a descriptive comment about this resource")
public class GoldFish extends AbstractProcessingResource
  implements ProcessingResource {


} // class GoldFish
```

The default XML configuration for GoldFish looks like this:

```
<!-- creole.xml GoldFish -->
<!--  hamish, 26/9/2001 -->
<!-- $Id: howto.tex,v 1.130 2006/10/23 12:56:37 ian Exp $ -->

<CREOLE-DIRECTORY>
  <JAR SCAN="true">GoldFish.jar</JAR>
</CREOLE-DIRECTORY>
```

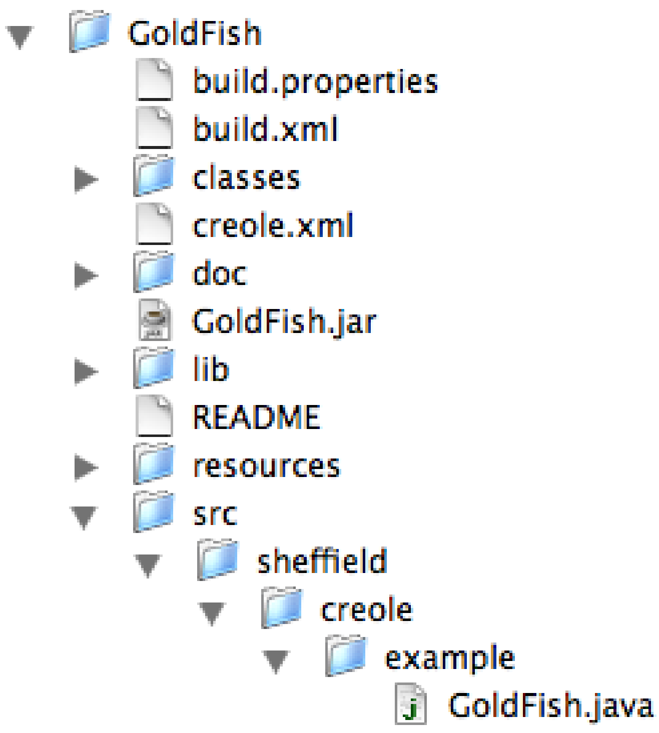


Figure 3.3: BootStrap directory tree

The directory structure containing these files is shown in figure 3.3. `GoldFish.java` lives in the `src/sheffield/creole/example` directory. `creole.xml` and `build.xml` are in the top `GoldFish` directory. The `lib` directory is for libraries; the `classes` directory is where Java class files are placed; the `doc` directory is for documentation. These last two, plus `GoldFish.jar` are created by Ant.

This process has the advantage that it creates a complete source tree and build structure for the component, and the disadvantage that it creates a complete source tree and build structure for the component. If you already have a source tree, you will need to chop out the bits you need from the new tree (in this case `GoldFish.java` and `creole.xml`) and copy it into your existing one.

See the example code at http://gate.ac.uk/gate-examples/doc/.

3.11 [F] Instantiate (CREOLE) Resources

This section describes how to create CREOLE resources as objects in a running Java virtual machine. This process involves using GATE's `Factory` class, and, in the case of LRs, may also involve using a `DataStore`.

CREOLE resources are Java Beans; creation of a resource object involves using a default constructor, then setting parameters on the bean, then calling an `init()` method[3]. The Factory takes care of all this, makes sure that the GUI is told about what is happening (when GUI components exist at runtime), and also takes care of restoring LRs from DataStores. So a programmer using GATE should **never call the constructor** of a resource: always use the Factory.

The valid parameters for a resource are described in the resource's section of its `creole.xml` file or in Java annotations on the resource class – see section 4.9.

Creating a resource via the Factory involves passing values for any create-time parameters that require setting to the Factory's `createResource` method. If no parameters are passed, the defaults are used. So, for example, the following code creates a default ANNIE part-of-speech tagger:

```
Gate.getCreoleRegister().registerDirectories(new File(
  Gate.getPluginsHome(), ANNIEConstants.PLUGIN_DIR).toURI().toURL());
FeatureMap params = Factory.newFeatureMap(); // empty map: default parameters
ProcessingResource tagger = (ProcessingResource)
  Factory.createResource("gate.creole.POSTagger", params);
```

Note that if the resource created here had any parameters that were both mandatory and had no default value, the `createResource` call would throw an exception. In this case, all the information needed to create a tagger is available in default values given in the tagger's XML definition (in `plugins/ANNIE/creole.xml`):

```
<RESOURCE>
  <NAME>ANNIE POS Tagger</NAME>
  <COMMENT>Mark Hepple's Brill-style POS tagger</COMMENT>
  <CLASS>gate.creole.POSTagger</CLASS>
  <PARAMETER NAME="document"
    COMMENT="The document to be processed"
    RUNTIME="true">gate.Document</PARAMETER>
....
  <PARAMETER NAME="rulesURL" DEFAULT="resources/heptag/ruleset"
    COMMENT="The URL for the ruleset file"
    OPTIONAL="true">java.net.URL</PARAMETER>
</RESOURCE>
```

Here the two parameters shown are either 'runtime' parameters, which are set before a PR is executed, or have a default value (in this case the default rules file is distributed with GATE itself).

[3]This method is not part of the beans spec.

When creating a Document, however, the URL of the source for the document must be provided[4]. For example:

```
URL u = new URL("http://gate.ac.uk/hamish/");
FeatureMap params = Factory.newFeatureMap();
params.put("sourceUrl", u);
Document doc = (Document)
  Factory.createResource("gate.corpora.DocumentImpl", params);
```

The document created here is transient: when you quit the JVM the document will no longer exist. If you want the document to be persistent, you need to store it in a `DataStore`. Assuming that you have a `DataStore` already open called `myDataStore`, this code will ask the data store to take over persistence of your document, and to synchronise the memory representation of the document with the disk storage:

```
Document persistentDoc = myDataStore.adopt(doc, mySecurity);
myDataStore.sync(persistentDoc);
```

Security:
User access to the LRs is provided by a security mechanism of users and groups, similar to those on an operating system. When users create/save LRs into Oracle, they specify reading and writing access rights for users from their group and other users. For example, LRs created by one user/group can be made read-only to others, so they can use the data, but not modify it. The access modes are:

- others: read/none;
- group: modify/read/none;
- owner: modify/read.

If needed, ownership can be transferred from one user to another. Users, groups and LR permissions are administered in a special administration tool, by a privileged user. For more details see chapter 14.

When you want to restore a document (or other LR) from a data store, you make the same `createResource` call to the Factory as for the creation of a transient resource, but this time you tell it the data store the resource came from, and the ID of the resource in that datastore:

```
  URL u = ....; // URL of a serial data store directory
  SerialDataStore sds = new SerialDataStore(u.toString());
```

[4]Alternatively a string giving the document source may be provided.

```
sds.open();

// getLrIds returns a list of LR Ids, so we get the first one
Object lrId = sds.getLrIds("gate.corpora.DocumentImpl").get(0);

// we need to tell the factory about the LR's ID in the data
// store, and about which data store it is in - we do this
// via a feature map:
FeatureMap features = Factory.newFeatureMap();
features.put(DataStore.LR_ID_FEATURE_NAME, lrId);
features.put(DataStore.DATASTORE_FEATURE_NAME, sds);

// read the document back
Document doc = (Document)
  Factory.createResource("gate.corpora.DocumentImpl", features);
```

See the example code at http://gate.ac.uk/gate-examples/doc/.

3.12 [D] Load Resources: document, tokenizer...*

3.12.1 Loading Language Resources: document, corpora...

Load a language resource by right clicking on "Language Resources" in the resource tree and selecting a language resource type (document, corpus or annotation schema) or by going to the 'File' menu and choosing 'New Language Resource'. Choose optionally a name for the resource, and choose any parameters as necessary.

For a document, a file or URL should be entered as the value of "sourceUrl" (double clicking in the "values" box brings up a tree structure to enable selection of documents). The easiest for a start is to enter an URL like 'http://gate.ac.uk'. Other parameters can be selected or changed as necessary, such as the encoding of the document, and whether it should be markup aware.

See also the movie for creating documents.

There are three ways of adding documents to a corpus:

1. When creating the corpus, clicking on the icon under Value brings up a popup window with a list of the documents already loaded into GATE. This enables the user to add any documents to the corpus.
2. Alternatively, the corpus can be loaded first, and documents added later by double clicking on the corpus and using the + and - icons to add or remove documents to the

corpus. Note that the documents must have been loaded into GATE before they can be added to the corpus.

3. Once loaded, the corpus can be populated by right clicking on the corpus and selecting "Populate". With this method, documents do not have to have been previously loaded into GATE, as they will be loaded during the population process. Select the directory containing the relevant files, choose the encoding, and check or uncheck the "recurse directories" box as appropriate. The initial value for the encoding is the platform default.

Additionally, right-clicking on a loaded document in the tree and selecting the "New corpus with this document" option creates a new transient corpus named `Corpus for` ***`document name`*** containing just this document. To add a new annotation schema, simply choose the name and the path or Url. For more information about schema, see 6.4.1.

See also the movie for creating and populating corpora.

You should next read the section 3.16 to view annotations.

3.12.2 Loading Processing Resources: tokenizer, gazetteer...

This section describes how to load and run CREOLE resources not present in ANNIE. To load ANNIE, see Section 3.17. For technical descriptions of these resources, see Chapter 9. First ensure that the necessary plugins have been loaded (see Section 3.4). If the resource you require does not appear in the list of Processing Resources, then you probably do not have the necessary plugin loaded. Processing resources are loaded by selecting them from the set of Processing Resources (right click on Processing Resources or select "New Processing Resource" from the File menu), adding them to the application and selecting the necessary parameters (e.g. input and output Annotation Sets).

See also the movie for loading processing resources.

You should next read the section 3.14 to create and run an application.

3.12.3 Loading and Processing Large Corpora

When trying to process a larger corpus (i.e. one that would not fit in memory at one time) the use of a datastore and persistent corpora is required.

Open or create a datastore (see section 3.22) and then create a corpus. Save the so far empty corpus to the datastore – this will convert it to a persistent corpus.

When populating or processing the persistent corpus, the documents contained will only be loaded one by one thus reducing the amount of memory required to only that necessary for loading the largest document in the collection.

3.13 [D,F] Configure (CREOLE) Resources

For full details on how to supply configuration data for resources can be found in section 4.9.

- To collect PRs into an application and run them, see section 3.14.
- GATE's internal creole.xml file (note that there are no JAR entries there, as the file is bundled with GATE itself).

3.14 [D] Create and Run an Application*

Once all the resources have been loaded, an application can be created and run. Right click on "Applications" and select "New" and then either "Corpus Pipeline" or "Pipeline". A pipeline application can only be run over a single document, while a corpus pipeline can be run over a whole corpus.

To build the pipeline, double click on it, and select the resources needed to run the application (you may not necessarily wish to use all those which have been loaded). Transfer the necessary components from the set of "loaded components" displayed on the left hand side of the main window to the set of "selected components" on the right, by selecting each component and clicking on the left and right arrows, or by double-clicking on each component. Ensure that the components selected are listed in the correct order for processing (starting from the top). If not, select a component and move it up or down the list using the up/down arrows at the left side of the pane. Ensure that any parameters necessary are set for each processing resource (by clicking on the resource from the list of selected resources and checking the relevant parameters from the pane below). For example, if you wish to use annotation sets other than the Default one, these must be defined for each processing resource. Note that if a corpus pipeline is used, the corpus needs only to be set once, using the drop-down menu beside the "corpus" box. If a pipeline is used, the document must be selected for each processing resource used. Finally, right-click on "Run" to run the application on the document or corpus.

See also the movie for loading and running processing resources.

For how to use the *conditional* versions of the pipelines see section 3.15 and for saving/restoring the configuration of an application see section 3.23.

You should next read the section 3.17 to do information extraction.

3.15 [D] Run PRs Conditionally on Document Features

The “Conditional Pipeline” and “Conditional Corpus Pipeline” application types are conditional versions of the pipelines mentioned in section 3.14 and allow processing resources to be run or not according to the value of a feature on the document. In terms of graphical interface, the only addition brought by the conditional versions of the applications is a box situated underneath the lists of available and selected resources which allows the user to choose whether the currently selected processing resource will run always, never or only on the documents that have a particular value for a named feature.

If the *Yes* option is selected then the corresponding resource will be run on all the documents processed by the application as in the case of non- conditional applications. If the *No* option is selected then the corresponding resource will never be run; the application will simply ignore its presence. This option can be used to temporarily and quickly disable an application component, for debugging purposes for example.

The *If value of feature* option permits running specific application components conditionally on document features. When selected, this option enables two text input fields that are used to enter the name of a feature and the value of that feature for which the corresponding processing resource will be run. When a conditional application is run over a document, for each component that has an associated condition, the value of the named feature is checked on the document and the component will only be used if the value entered by the user matches the one contained in the document features.

3.16 [D] View Annotations*

If you have no document already loaded in the resource tree, see first section 3.12.

To view a document, double click on the filename in the resource tree (left hand pane). Note that it may take a few seconds for the text to be displayed if it is a big document and/or with a lot of annotations.

To view the annotation sets, click on the 'Annotation Sets' button at the top of the document view or use F3 key. If you keep Shift key pressed when doing so the annotations will be selected as the last document viewed otherwise no annotation will be selected. This will bring up the annotation sets viewer, which displays the annotation sets available and their corresponding annotation types. Note that the default annotation set has no name. If no application has been run, the only annotations to be displayed will be those corresponding to the document format analysis performed automatically by GATE on loading the document (e.g. HTML or XML tags). If an application has been run, other annotation types and/or annotation sets may also be present. The fonts and colours of the annotations can be edited by double clicking on the annotation name or pressing Enter key.

Select the annotation types to be viewed by clicking on the appropriate checkbox(es) or pressing Space key. The text segments corresponding to these annotations will be highlighted in the main text window.

To view the annotations and their features, click on the 'Annotations list' button at the top or bottom of the main window or use F4 key. The annotation list viewer will appear above or below the main text, respectively. It will only contain the annotations selected from the annotation sets. These lists can be sorted in ascending and descending order by any column, by clicking on the corresponding column heading. Moreover you can hide a column by using the context menu with right-click. Clicking on an entry in the table will also highlight the respective matching text portion.

Hovering over some part of the text in the main window will bring up a popup box containing a list of the annotations associated with it (assuming that the relevant annotation types have been selected from the annotation set viewer).

Annotations relating to coreference (if relevant) are displayed separately in the coreference viewer. This operates in the same way as the annotation sets viewer.

At any time, the main viewer can also be used to display other information, such as Messages, by clicking on the header at the top of the main window. If an error occurs in processing, the messages tab will flash red, and an additional popup error message may also occur.

Text in a loaded document can be edited in the document viewer. The usual platform specific cut, copy and paste keyboard shortcuts should also work, depending on your operating system (e.g. CTRL-C, CTRL-V for Windows). The last icon, a magnifying glass, at the top of the document editor is for searching in the document. To prevent the new annotation windows popping up when a piece of text is selected, hide the AnnotationSets view (the tree on the right) first to make it inactive. The highlighted portions of the text will still remain visible.

See also the movie for inspecting the processing results.

You should next read the section 3.19 to create and edit annotations.

3.17 [D] Do Information Extraction with ANNIE*

This section describes how to load and run ANNIE (see Chapter 8) from the development environment. To embed ANNIE in other software, see section 3.28.

From the File menu, select “Load ANNIE system”. To run it in its default state, choose

"With Defaults". This will automatically load all the ANNIE resources, and create a corpus pipeline called ANNIE with the correct resources selected in the right order, and the default input and output annotation sets.

If "Without Defaults" is selected, the same processing resources will be loaded, but a popup window will appear for each resource, which enables the user to specify a name and location for the resource. This is exactly the same procedure as for loading a processing resource individually, the difference being that the system automatically selects those resources contained within ANNIE. When the resources have been loaded, a corpus pipeline called ANNIE will be created as before.

The next step is to add a corpus (see Section 3.12.1), and select this corpus from the drop-down Corpus menu in the Serial Application editor. Finally click on Run (from the Serial Application editor, or by right clicking on the application name and selecting "Run"). To view the results, double click on the filename in the left hand pane. No Annotation Sets nor Annotations will be shown until annotations are selected in the Annotation Sets; the Default set is indicated only with an unlabelled right-arrowhead which must be selected in order to make visible the available annotations.

See also the movie for loading and running ANNIE.

3.18 [D] Modify ANNIE

You will find the ANNIE resources in gate/plugins/ANNIE/resources. Simply locate the existing resources you want to modify, make a copy with a new name, edit them, and load the new resources into GATE as new Processing Resources (see Section 3.12.2).

3.19 [D] Create and Edit Annotations*

Since many NLP algorithms require annotated corpora for training, GATE's development environment provides easy-to-use and extendable facilities for text annotation. The annotation can be done manually by the user or semi-automatically by running some processing resources over the corpus and then correcting/adding new annotations manually. Depending on the information that needs to be annotated, some ANNIE modules can be used or adapted to bootstrap the corpus annotation task.

To create annotations manually:

- Select the text you want to annotate and hover the mouse on the selection.

Then, to edit annotations:

- Hover the mouse on an annotation.
- The most recent annotation type to have been used will be displayed at the top in a drop down box that you can edit to create a new annotation type. If it is correct, you need to do nothing further. You can add or change features and their values in the table below.
- To delete an annotation, click on the red X icon at the top of the popup window.
- To change the span of the annotation, use the small arrow icon at the top of the popup window.
- You can move the popup window by dragging it and it will be set in pinned mode which means it won't move when selecting another annotation. The pin icon at the top set the pinned mode too.

The popup menu only contains annotation types present in the Annotation Schema and those already listed in the relevant Annotation Set. To create a new Annotation Schema, see Section 3.21. The popup menu can be edited to add a new annotation type, however.

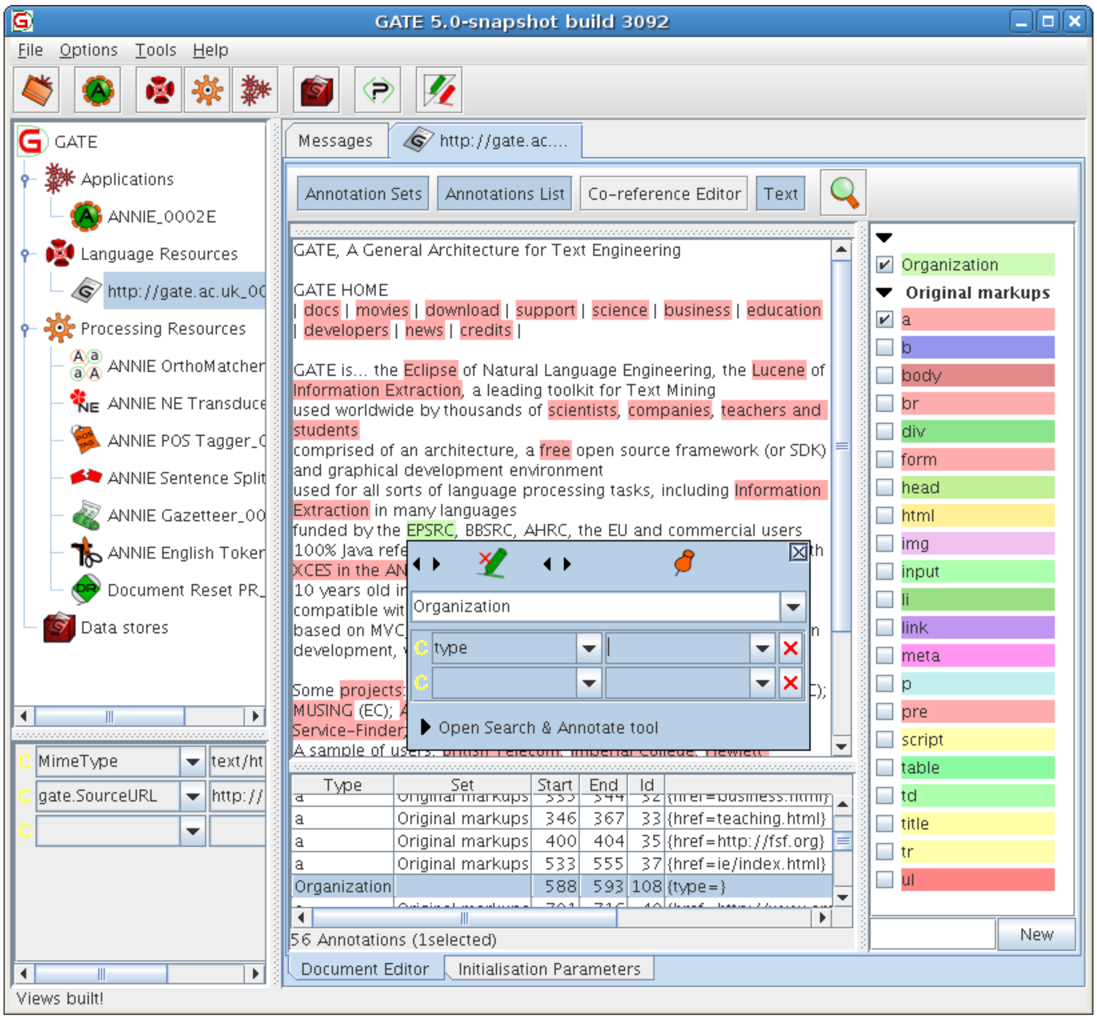
The new annotation created will automatically be placed in the annotation set that has been selected (highlighted) by the user. To create a new annotation set, type the name of the new set to be created in the box below the list of annotation sets, and click on "New".

Figure 3.4 demonstrates adding a 'Organization' annotation for the string "EPSRC" (highlighted in green) to the default annotation set (blank name in the annotation set view on the right) and a feature name 'type' with a value about to be added.

To add a second annotation to a selected piece of text, or to add an overlapping annotation to an existing one, press the CTRL key to avoid the existing annotation popup appearing, and then select the text and create the new annotation. Again by default the last annotation type to have been used will be displayed; change this to the new annotation type. When a piece of text has more than one annotation associated with it, on mouseover all the annotations will be displayed. Selecting one of them will bring up the relevant annotation popup.

To search and annotate automatically the document use the search and annotate function like shown on figure 3.5:

- Create and/or select an annotation to be used as a model to annotate.
- Open the panel at the bottom of the annotation editor window.
- Change the expression to search if necessary.
- Use the [First] button or Enter key to select the first expression to annotate.



MimeType	text/ht
gate.SourceURL	http://

GATE, A General Architecture for Text Engineering

GATE HOME
| docs | movies | download | support | science | business | education
| developers | news | credits |

GATE is... the Eclipse of Natural Language Engineering, the Lucene of
Information Extraction, a leading toolkit for Text Mining
used worldwide by thousands of scientists, companies, teachers and
students
comprised of an architecture, a free open source framework (or SDK)
and graphical development environment
used for all sorts of language processing tasks, including Information
Extraction in many languages
funded by the EPSRC, BBSRC, AHRC, the EU and commercial users
100% Java refe
XCES in the AN
10 years old i
compatible wit
based on MVC
development,

Some projects:
MUSING (EC);
Service-Finder
A sample of users: British Telecom, Imperial College, Hewlett

Organization
type

Type	Set	Start	End	Id	
a	Original markups	346	367	33	{href=teaching.html}
a	Original markups	400	404	35	{href=http://fsf.org}
a	Original markups	533	555	37	{href=ie/index.html}
Organization		588	593	108	{type=}

56 Annotations (1selected)

Figure 3.4: Adding an Organization annotation to the Default Annotation Set

- Use the [Annotate] button if the selection is correct otherwise the [Next] button. After a few cycles of [Annotate] and [Next], Use the [Ann. all next] button and if you made an error use the [Undo] button.

Note that after using the [First] button you can move the caret in the document and use the [Next] button to avoid continuing the search from the beginning of the document. The [?] button at the end of the search text field will help you to build powerful regular expressions to search.

You should next read the section 3.20 to save annotations.

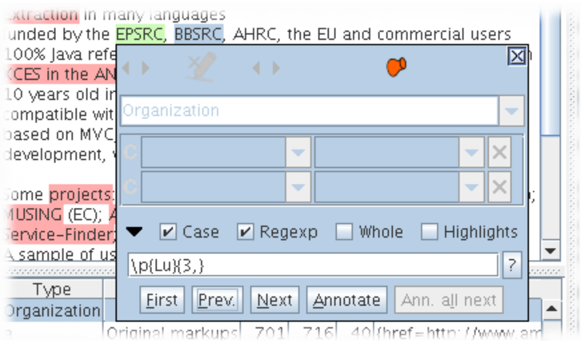


Figure 3.5: Search and annotate function of the annotation editor.

3.19.1 Schema-driven editing

An alternative annotation editor component is available which constrains the available annotation types and features much more tightly, based on the annotation schemas that are currently loaded. This is particularly useful when annotating large quantities of data or for use by less skilled users.

Annotation schemas provide a means to define types of annotations in GATE - basically this means that GATE "knows about" annotations defined in a schema.

The default annotation schema contains common named entities such as Person, Organisation, Location, etc. You can modify the existing schema or create a new one, in order to tell GATE about other kinds of annotations you frequently use. You can still create annotations in GATE without having specified them in an annotation schema, but you may then need to tell GATE about the properties of that annotation type each time you create an annotation for it.

To use this, you must load the `Schema_Annotation_Editor` plugin. With this plugin loaded, the annotation editor will *only* offer the annotation types permitted by the currently loaded set of schemas, and when you select an annotation type only the features permitted by the schema are available to edit[5]. Where a feature is declared as having an enumerated type the available enumeration values are presented as an array of buttons, making it easy to select the required value quickly.

To load an annotation schema use the resource tree and right-click on Language Resources or use the File menu then New language resource item.

See Section 3.21 for creating new annotation schemas.

[5]existing features outwith the schema, e.g. those created by previously-run processing resources, are not editable but not modified or removed by the editor.

3.20 [D] Saving annotations*

The data can either be dumped out as a file (see Section 3.33) or saved in a data store (see Section 3.22).

3.21 [D,F] Create a New Annotation Schema

GUI

An annotation schema file can be loaded or unloaded in GATE just like any other language resource. Once loaded into the system, the Schema Annotation Editor will use this definition when creating or editing annotations.

API

Another way to bring an annotation schema inside GATE is through creole.xml file. By using the AUTOINSTANCE element, one can create instances of resources defined in creole.xml. The gate.creole.AnnotationSchema (which is the Java representation of an annotation schema file) initializes with some predefined annotation definitions (annotation schemas) as specified by the GATE team.

Example from GATE's internal creole.xml (in `src/gate/resources/creole`*):*

```
<!-- Annotation schema -->
<RESOURCE>
  <NAME>Annotation schema</NAME>
  <CLASS>gate.creole.AnnotationSchema</CLASS>
  <COMMENT>An annotation type and its features</COMMENT>
  <PARAMETER NAME="xmlFileUrl" COMMENT="The url to the definition file"
    SUFFIXES="xml;xsd">java.net.URL</PARAMETER>
  <AUTOINSTANCE>
    <PARAM NAME ="xmlFileUrl" VALUE="schema/AddressSchema.xml" />
  </AUTOINSTANCE>
  <AUTOINSTANCE>
    <PARAM NAME ="xmlFileUrl" VALUE="schema/DateSchema.xml" />
  </AUTOINSTANCE>
  <AUTOINSTANCE>
    <PARAM NAME ="xmlFileUrl" VALUE="schema/FacilitySchema.xml" />
  </AUTOINSTANCE>
  <!-- etc. -->
</RESOURCE>
```

In order to create a gate.creole.AnnotationSchema object from a schema annotation file, one must use the gate.Factory class.

Eg:
```
FeatureMap params = new FeatureMap();
param.put("xmlFileUrl",annotSchemaFile.toURL());
AnnotationSchema annotSchema =
Factory.createResurce("gate.creole.AnnotationSchema", params);
```

Note: All the elements and their values must be written in lower case, as XML is defined as case sensitive and the parser used for XML Schema inside GATE searches is case sensitive.

In order to be able to write XML Schema definitions, the ones defined in GATE (resources/creole/schema) can be used as a model, or the user can have a look at *http://www.w3.org/2000/10/XMLSchema* for a proper description of the semantics of the elements used.

Some examples of annotation schemas are given in Section 6.4.1.

3.22 [D] Save and Restore LRs in Data Stores

To save a text in a data store, a new data store must first be created if one does not already exist. Create a data store by right clicking on Data Store in the left hand pane, and select the option "Create Data Store". Select the data store type you wish to use. Create a directory to be used as the data store (note that the data store is a directory and not a file).

You can either save a whole corpus to the datastore (in which case the structure of the corpus will be preserved) or you can save individual documents. The recommended method is to save the whole corpus. To save a corpus, right click on the corpus name and select the "Save to..." option (giving the name of the datastore created earlier). To save individual documents to the data store, right clicking on each document name and follow the same procedure.

To load a document from a data store, do not try to load it as a language resource. Instead, open the data store by right clicking on Data Store in the left hand pane, select "Open Data Store" and choose the data store to open. The data store tree will appear in the main window. Double click on a corpus or document in this tree to open it. To save a corpus and document back to the same datastore, simply select the "Save" option.

See also the movie for creating a data store and the movie for loading corpus and documents from a data store.

3.23 [D] Save Resource Parameter State to File

Resources, and applications that are made up of them, are created based on the settings of their parameters (see section 3.12). It is possible to save the data used to create an application to a file and re-load it later. To save the application to a file, right click on it in the resources tree and select “Save application state”, which will give you a file creation dialogue.

To restore the application later, select “Restore application from file” from the “File” menu.

Note that the data that is saved represents how to *recreate* an application – not the resources that make up the application itself. So, for example, if your application has a resource that initialises itself from some file (e.g. a grammar, a document) then that file must still exist when you restore the application.

In case you don’t want to save the corpus configuration associated with the application then you must select ’<none>’ in the corpus list of the application before to save the application.

The file resulted from saving the application state contains the values of the initialisation parameters for all the processing resources contained by the stored application. For the parameters of type URL (which are typically used to select external resources such as grammars or rules files) a transformation is applied so that all the paths are relative to the location of the file used to store the state. This means that the resource files used by an application do not need to be in the same location as when the application was initially created but rather in the same *location relative to the location of the application file.* This allows the creation and deployment of portable applications by keeping the application file and the resource files used by the application together.

If you want to save your application along with all the resources it requires you can use the “Export for Teamware” option (see section 3.24).

See also the movie for saving and restoring applications.

3.24 [D] Save an application with its resources (e.g. GATE Teamware)

When you save an application using the “Save application state” option (see section 3.23), the saved file contains references to the plugins that were loaded when the application was saved, and to any resource files required by the application. To be able to reload the file, these plugins and other dependencies must exist at the same locations (relative to the saved state file). While this is fine for saving and loading applications on a single machine it means that if you want to package your application to run it elsewhere (e.g. deploy it to a GATE Teamware installation) then you need to be careful to include all the resource files and plugins at the right locations in your package. The "Export for Teamware" option on the right-click menu for an application helps to automate this process.

When you export an application in this way, GATE produces a ZIP file containing the saved application state (in the same format as "Save application state"). Any plugins and resource files that the application refers to are also included in the zip file, and the relative paths in the saved state are rewritten to point to the correct locations within the package. The resulting package is therefore self-contained and can be copied to another machine and unpacked there, or passed to your Teamware Administrator for deployment.

As well as selecting the location where you want to save the package, the "Export for Teamware" option will also prompt you to select the annotation sets that your application uses for input and output. For example, if your application makes use of the unpacked XML markup in source documents and creates annotations in the default set then you would select "Original markups" as an input set and the "*<Default annotation set>*" as an output set. GATE will try to make an educated guess at the correct sets but you should check and amend the lists as necessary.

There are a few important points to note about the export process:

- The complete contents of all the plugin directories that are loaded when you perform the export will be included in the resulting package. Use the plugin manager to unload any plugins your application is not using before you export it.
- If your application refers to a resource file in a directory that is not under one of the loaded plugins, the entire contents of this directory will be recursively included in the package. If you have a number of unrelated resources in a single directory (e.g. many sets of large gazetteer lists) you may want to separate them into separate directories so that only the relevant ones are included in the package.
- The packager only knows about resources that your application refers to directly in its parameters. For example, if your application includes a multi-phase JAPE grammar the packager will only consider the main grammar file, not any of its sub-phases. If the sub-phases are not contained in the same directory as the main grammar you may find they are not included. If indirect references of this kind are all to files under the same directory as the "master" file it will work OK.

If you require more flexibility than this option provides you should read section C.2, which describes the underlying Ant task that the exporter uses.

3.25 [D,F] Perform Evaluation with the Annotation-Diff tool

Section 13 describes the theory behind this tool.

The annotation tool is activated by selecting it from the Tools menu at the top of the window. It will appear in a new window. Select the key and response documents to be used (note that both must have been previously loaded into the system), the annotation sets to be used for each, and the annotation type to be evaluated.

Note that the tool automatically intersects all the annotation types from the selected key annotation set with all types from the response set.

On a separate note, you can perform a diff on the same document, between two different annotation sets. One annotation set could contain the key type and another could contain the response one.

After the type has been selected, the user is required to decide how the features will be compared. It is important to know that the tool compares them by analyzing if features from the key set are contained in the response set. It checks for both the feature name and feature value to be the same.

There are three basic options to select:

- To take all the features from the key set into consideration
- To take only the user selected ones
- To ignore all the features from the key set.

If false positives are to be measured, select the annotation type (and relevant annotation set) to be used as the denominator (normally, Token or Sentence). The weight for the F-Measure can also be changed - by default it is set to 0.5 (i.e. to give precision and recall equal weight). Finally, click on "Evaluate" to display the results. Note that the window may need to be resized manually, by dragging the window edges or internal bars as appropriate).

In the main window, the key and response annotations will be displayed. They can be sorted by any category by clicking on the relevant column header. The key and response annotations will be aligned if their indices are identical, and are color coded according to the legend displayed.

Precision, recall, F-measure and false positives are also displayed below the annotation tables, each according to 3 criteria - strict, lenient and average. See sections 13.1 and 13.4 for more details about the evaluation metrics.

The results can be saves to an HTML file by pressing the "Export to HTML" button. This creates an HTML snapshot of what the AnnotationDiff interface shows at that moment.The columns and rows in the table will be shown in the same order, and the hidden columns will not appear in the HTML file. The colours will also be the same.

3.26 [D] Use the Corpus Benchmark Evaluation tool

The Corpus Benchmark tool can be run in two ways: standalone and GUI mode. Section 13.3 describes the theory behind this tool.

3.26.1 GUI mode

To use the tool in GUI mode, first make sure the properties of the tool have been set correctly (see section 3.26.2 for how to do this). Then select "Corpus Benchmark Tool" from the Options menu. There are 3 ways in which it can be run:

- **Default mode** compares the stored processed set with the current processed set and the human-annotated set. This will give information about how well the system is doing compared with a previous version.
- **Human marked against stored processing results** compares the stored processed set with the human-annotated set.
- **Human marked against current processing results** compares the current processed set with the human-annotated set.

Once the mode has been selected, choose the directory where the corpus is to be found. The corpus must have a directory structure consisting of "clean" and "marked" subdirectories (note that these names are case sensitive). The clean directory should contain the raw texts; the marked directory should contain the human-annotated texts. Finally, select the application to be run on the corpus (for "default" and "human v current" modes).

If the tool is to be used in Default or Current mode, the corpus must first be processed with the current set of resources. This is done by selecting "Store corpus for future evaluation" from the Corpus Benchmark Tool. Select the corpus to be processed (from the top of the subdirectory structure, i.e. the directory containing the marked and stored subdirectories). If a "processed" subdirectory exists, the results will be placed there; if not, one will be created.

Once the corpus has been processed, the tool can be run in Default or Current mode. The resulting HTML file will be output in the main GATE messages window. This can then be pasted into a text editor and viewed in an Internet browser for easier viewing.

The tool can be used either in verbose or non-verbose mode, by selecting the verbose option from the menu. In verbose mode, any score below the user's pre-defined threshold (stored in corpus_tool.properties file) will show the relevant annotations for that entity type, thereby enabling the user to see where problems are occurring.

3.26.2 How to define the properties of the benchmark tool

The properties of the benchmark tool are defined in the file corpus_tool.properties, which should be located in the directory from which GATE is run (usually gate/build or gate/bin).

The following properties should be set:

- the threshold for the verbose mode (by default this is set to 0.5);
- the name of the annotation set containing the human-marked annotations (annotSetName);
- the name of the annotation set containing the system-generated annotations (outputSetName);
- the annotation types to be considered (annotTypes);
- the feature values to be considered, if any (annotFeatures).

The default Annotation Set has to be represented by an empty String. Note also that outputSetName and annotSetName must be different. If they are the same, then use the Annotation Set Transfer PR to change one of them.

An example file is shown below:

```
threshold=0.7
annotSetName=Key
outputSetName=ANNIE
annotTypes=Person;Organization;Location;Date;Address;Money
annotFeatures=type;gender
```

3.27 [D] Write JAPE Grammars

JAPE is a language for writing regular expressions over annotations, and for using patterns matched in this way as the basis for creating more annotations. JAPE rules compile into finite state machines. GATE's built-in Information Extraction tools use JAPE (amongst other things). For information on JAPE see:

- chapter 7 describes how to write JAPE rules;
- chapter 8 describes the built-in IE components;
- appendix B describes how JAPE is implemented and formally defines the language's grammar;
- appendix D describes the default Named Entity rules distributed with GATE.

3.28 [F] Embed NLE in other Applications

Embedding GATE-based language processing in other applications is straightforward:

- add `gate.jar` and the JAR files in `gate/lib` to the `CLASSPATH`,
- tell Java that the GATE Unicode Kit is an extension (`-Djava.ext.dirs=/home/hamish/gate/bi` for example);
- initialise GATE with `gate.Gate.init()`;
- program to the framework API.

For example, this code will create the ANNIE extraction system:

```
// initialise the GATE library
Gate.init();

// load ANNIE as an application from a gapp file
SerialAnalyserController controller = (SerialAnalyserController)
  PersistenceManager.loadObjectFromFile(new File(new File(
    Gate.getPluginsHome(), ANNIEConstants.PLUGIN_DIR),
      ANNIEConstants.DEFAULT_FILE));
```

If you want to use resources from any plugins, you need to load the plugins before calling `createResource`:

```
Gate.init();

// need Tools plugin for the Morphological analyser
Gate.getCreoleRegister().registerDirectories(
  new File(Gate.getPluginsHome(), "Tools").toURL()
);

...

ProcessingResource morpher = (ProcessingResource)
  Factory.createResource("gate.creole.morph.Morph");
```

Instead of creating your processing resources individually using the `Factory`, you can create your application in the GUI, save it using the “save application state” option (see section 3.23), and then load the saved state from your code. This will automatically reload any plugins that were loaded when the state was saved, you do not need to load them manually.

```
  Gate.init();

  CorpusController controller = (CorpusController)
    PersistenceManager.loadObjectFromFile(new File("savedState.xgapp"));

  // loadObjectFromUrl is also available
```

There are longer examples available at http://gate.ac.uk/gate-examples/doc/.

3.29 [F] Use GATE within a Spring application

GATE provides helper classes to allow GATE resources to be created and managed by the Spring framework. For Spring 2.0 or later, GATE provides a custom namespace handler that makes them extremely easy to use. To use this namespace, put the following declarations in your bean definition file:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:gate="http://gate.ac.uk/ns/spring"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
         http://www.springframework.org/schema/beans
         http://www.springframework.org/schema/beans/spring-beans.xsd
         http://gate.ac.uk/ns/spring
         http://gate.ac.uk/ns/spring.xsd">
```

You can have Spring initialise GATE:

```
  <gate:init gate-home="WEB-INF" user-config-file="WEB-INF/user.xml">
```

```
    <gate:preload-plugins>
      <value>WEB-INF/ANNIE</value>
      <value>http://example.org/gate-plugin</value>
    </gate:preload-plugins>
  </gate:init>
```

To create a GATE resource, use the `<gate:resource>` element.

```
  <gate:resource id="sharedOntology" scope="singleton"
          resource-class="gate.creole.ontology.owlim.OWLIMOntologyLR">
    <gate:parameters>
      <entry key="rdfXmlURL">
        <value type="org.springframework.core.io.Resource"
          >WEB-INF/ontology.rdf</value>
      </entry>
    </gate:parameters>
  </gate:resource>
```

If you are familiar with Spring you will see that `<gate:parameters>` uses the same format as the standard `<map>` element, but values whose type is a Spring `Resource` will be converted to URLs before being passed to the GATE resource.

You can load a GATE saved application with

```
  <gate:saved-application location="WEB-INF/application.gapp" scope="prototype">
    <gate:customisers>
      <gate:set-parameter pr-name="custom transducer" name="ontology"
                          ref="sharedOntology" />
    </gate:customisers>
  </gate:saved-application>
```

"Customisers" are used to customise the application after it is loaded. In the example above, we load a singleton copy of an ontology which is then shared between all the separate instances of the (prototype) application. The `<gate:set-parameter>` customiser accepts all the same ways to provide a value as the standard Spring `<property>` element (a "value" or "ref" attribute, or a sub-element - `<value>`, `<list>`, `<bean>`, `<gate:resource>` . . .).

The `<gate:add-pr>` customiser provides support for the case where most of the application is in a saved state, but we want to create one or two extra PRs with Spring (maybe to inject other Spring beans as init parameters) and add them to the pipeline.

```
  <gate:saved-application ...>
    <gate:customisers>
      <gate:add-pr add-before="OrthoMatcher" ref="myPr" />
    </gate:customisers>
  </gate:saved-application>
```

By default, the `<gate:add-pr>` customiser adds the target PR at the end of the pipeline, but an `add-before` or `add-after` attribute can be used to specify the name of a PR before (or after) which this PR should be placed. Alternatively, an `index` attribute places the PR at a specific (0-based) index into the pipeline. The PR to add can be specified either as a "ref" attribute, or with a nested `<bean>` or `<gate:resource>` element.

These custom elements all define various factory beans. For full details, see the JavaDocs for `gate.util.spring` (the factory beans) and `gate.util.spring.xml` (the `gate:` namespace handler).

Note: the former approach using factory methods of the `gate.util.spring.SpringFactory` class will still work, but should be considered deprecated in favour of the new factory beans.

3.30 [F] Use GATE within a Tomcat Web Application

Embedding GATE in a Tomcat web application involves several steps.

1. Put the necessary JAR files (gate.jar and all or most of the jars in `gate/lib`) in your `webapp/WEB-INF/lib`.
2. Put the plugins that your application depends on in a suitable location (e.g. `webapp/WEB-INF/plugins`).
3. Create suitable gate.xml configuration files for your environment.
4. Set the appropriate paths in your application before calling `Gate.init()`.

This process is detailed in the following sections.

3.30.1 Recommended Directory Structure

You will need to create a number of other files in your web application to allow GATE to work:

- Site and user gate.xml config files - we highly recommend defining these specifically for the web application, rather than relying on the default files on your application server.
- The plugins your application requires.

In this guide, we assume the following layout:

```
webapp/
  WEB-INF/
    gate.xml
    user-gate.xml
    plugins/
      ANNIE/
      etc.
```

3.30.2 Configuration files

Your `gate.xml` (the "site-wide configuration file") should be as simple as possible:

```
<?xml version="1.0" encoding="UTF-8" ?>
<GATE>
  <GATECONFIG Save_options_on_exit="false"
              Save_session_on_exit="false" />
</GATE>
```

Similarly, keep the `user-gate.xml` (the "user config file") simple:

```
<?xml version="1.0" encoding="UTF-8" ?>
<GATE>
  <GATECONFIG Known_plugin_path=";"
              Load_plugin_path=";" />
</GATE>
```

This way, you can control exactly which plugins are loaded in your webapp code.

3.30.3 Initialization code

Given the directory structure shown above, you can initialize GATE in your web application like this:

```
// imports
...
public class MyServlet extends HttpServlet {
  private static boolean gateInited = false;
  public void init() throws ServletException {
    if(!gateInited) {
      try {
        ServletContext ctx = getServletContext();

        // use /path/to/your/webapp/WEB-INF as gate.home
        File gateHome = new File(ctx.getRealPath("/WEB-INF"));

        Gate.setGateHome(gateHome);
        // thus webapp/WEB-INF/plugins is the plugins directory, and
        // webapp/WEB-INF/gate.xml is the site config file.

        // Use webapp/WEB-INF/user-gate.xml as the user config file, to avoid
        // confusion with your own user config.
        Gate.setUserConfigFile(new File(gateHome, "user-gate.xml"));

        Gate.init();
        // load plugins, for example...
        Gate.getCreoleRegister().registerDirectories(
          ctx.getResource("/WEB-INF/plugins/ANNIE"));

        gateInited = true;
      }
      catch(Exception ex) {
        throw new ServletException("Exception initialising GATE",
                                   ex);
      }
    }
  }
}
```

Once initialized, you can create GATE resources using the Factory in the usual way (for example, see section 3.28 for an example of how to create an ANNIE application). You should also read section 3.31 for important notes on using GATE in a multithreaded application.

Instead of an initialization servlet you could also consider doing your initialization in a `ServletContextListener`, or using Spring (see section 3.29).

3.31 [F] Use GATE in a Multithreaded Environment

GATE can be used in multithreaded applications, so long as you observe a few restrictions. First, you must initialise GATE by calling `Gate.init()` *exactly once* in your application, typically in the application startup phase before any concurrent processing threads are started.

Secondly, you must not make calls that affect the global state of GATE (e.g. loading or unloading plugins) in more than one thread at a time. Again, you would typically load all the plugins your application requires at initialisation time. It is safe to create *instances* of resources in multiple threads concurrently.

Thirdly, it is important to note that individual GATE processing resources, language resources and controllers are by design *not* thread safe – it is not possible to use a single instance of a controller/PR/LR in multiple threads at the same time – but for a well written resource it should be possible to use several different instances of the same resource at once, each in a different thread. When writing your own resource classes you should bear the following in mind, to ensure that your resource will be useable in this way.

- Avoid static data. Where possible, you should avoid using static fields in your class, and you should try and take all configuration data via the CREOLE parameters you declare in your creole.xml file. System properties may be appropriate for truly static configuration, such as the location of an external executable, but even then it is generally better to stick to CREOLE parameters – a user may wish to use two different instances of your PR, each talking to a different executable.
- Read parameters at the correct time. Init-time parameters should be read in the `init()` (and `reInit()`) method, and for processing resources runtime parameters should be read at each `execute()`.
- Use temporary files correctly. If your resource makes use of external temporary files you should create them using `File.createTempFile()` at `init` or `execute` time, as appropriate. Do not use hardcoded file names for temporary files.
- If there are objects that can be shared between different instances of your resource, make sure these objects are accessed either read-only, or in a thread-safe way. In particular you must be very careful if your resource can take other resource instances as init or runtime parameters (e.g. the Flexible Gazetteer, section 9.5).

Of course, if you are writing a PR that is simply a wrapper around an external library that imposes these kinds of limitations there is only so much you can do. If your resource cannot be made safe you should *document this fact clearly.*

All the standard ANNIE PRs are safe when independent instances are used in different threads concurrently, as are the standard transient document, transient corpus and controller classes. A typical pattern of development for a multithreaded GATE-based application is:

- Develop your GATE processing pipeline in the GATE GUI.
- Save your pipeline as a `.gapp` file.

- In your application's initialisation phase, load *n* copies of the pipeline using `PersistenceManager.loadObjectFromFile()` (see the Javadoc documentation for details) and either give one to each thread or store them in a pool (e.g. a LinkedList).
- When you need to process a text, get one copy of the pipeline from the pool, and return it to the pool when you have finished processing.

3.32 [D,F] Add support for a new document format

In order to add a new document format, one needs to extend the `gate.DocumentFormat` class and to implement an abstract method called:

```
public void unpackMarkup(Document doc) throws
DocumentFormatException
```

This method is supposed to implement the functionality of each format reader and to create annotation on the document. Finally the document's old content will be replaced with a new one containing only the text between markups (see the GATE API documentation for more details on this method functionality).

If one needs to add a new textual reader will extend the gate.corpora. TextualDocument-Format and override the `unpackMarkup(doc)` method.

This class needs to be implemented under the Java bean specifications because it will be instantiated by GATE using `Factory.createResource()` method.

The `init()` method that one needs to add and implement is very important because in here the reader defines its means to be selected successfully by GATE. What one need to do is to add some specific information into certain static maps defined in `DocumentFormat` class, that will be used at reader detection time.

After that, a definition of the reader will be placed into the one's creole.xml file and the reader will be available to GATE.

We present for the rest of the section a complete three steps example of adding such a reader. The reader we describe in here is an XML reader.

Step 1

Create a new class called `XmlDocumentFormat` that extends `gate.corpora.TextualDocumentFormat`.

Step 2

Implement the `unpackMarkup(Document doc)` which performs the required functionality for the reader. Add XML detection means in init() method:

```
public Resource init() throws ResourceInstantiationException{
  // Register XML mime type
  MimeType mime = new MimeType("text","xml");
  // Register the class handler for this mime type
  mimeString2ClassHandlerMap.put(mime.getType()+ "/" + mime.getSubtype(),
                                                                    this);
  // Register the mime type with mine string
  mimeString2mimeTypeMap.put(mime.getType() + "/" + mime.getSubtype(), mime);
  // Register file sufixes for this mime type
  suffixes2mimeTypeMap.put("xml",mime);
  suffixes2mimeTypeMap.put("xhtm",mime);
  suffixes2mimeTypeMap.put("xhtml",mime);
  // Register magic numbers for this mime type
  magic2mimeTypeMap.put("<?xml",mime);
  // Set the mimeType for this language resource
  setMimeType(mime);
  return this;
}// init()
```

More details about the information from those maps can be found in Section 6.5.1

Step 3

Add the following creole definition in the creole.xml document.

```
    <RESOURCE>
      <NAME>My XML Document Format</NAME>
      <CLASS>mypackage.XmlDocumentFormat</CLASS>
      <AUTOINSTANCE/>
      <PRIVATE/>
    </RESOURCE>
```

More information on the operation of GATE's document format analysers may be found in section 6.5.

3.33 [D] Dump Results to File

There are three main ways to dump out the results of, for example, some language analysis or Information Extraction process running over documents:

1. preserving the original document format, with optional added annotations;
2. in GATE's own XML serialisation format (including all the annotations on the document);

3. by writing your own dump algorithm as a ProcessingResource.

This section describes how to use the first two options.

Both types of data export are available in the popup menu triggered by right-clicking on a document in the resources tree (see section 3.6): type 1 is called ‘save preserving format’ and type 2 is called ‘save as XML’.

Selecting the save as XML option leads to a file open dialogue; give the name of the file you want to create, and the whole document and all its data will be exported to that file. If you later create a document from that file, the state will be restored. (**Note:** because GATE’s annotation model is richer than that of XML, and because our XML dump implementation sometimes cuts corners[6], the state may not be identical after restoration. If your intention is to store the state for later use, use a DataStore instead.)

The save preserving format option also leads to a file dialogue; give a name and the data you require will be dumped into the file. The difference is that the file will preserve the original format of the source document. You can add annotations to the dump file: if there is a document viewer open in the main resource viewer area (see section 3.6), then any annotations that are selected (i.e. are visible in the table at the bottom of the viewer) will be included in the output dump. This is the best way to use the system to add markup based on some analysis process: select those annotations in the document viewer, save preserving format and you will have a file identical to the original source document with just the annotations you selected added. By default, the added annotations will contain no feature data; if you want the process to also dump features, set the ‘Include annotation features...’ option in the advanced options dialogue (see section 3.7). Note that GATE’s model of annotation allows graph structures, which are difficult to represent in XML (XML is a tree-structured representation format). During the dump process, annotations that cross each other in ways that can’t be represented straightforwardly in XML will be discarded, and a warning message printed.

3.34 [D] Stop GUI ‘Freezing’ on Linux

There is a problem with some versions of Linux that causes the GUI to appear to freeze. The problem occurs when you take some action, like loading a resource or browsing for a file, that pops up a dialogue box. This box sometimes fails to appear in a visible area of the screen, at which point the rest of the GUI waits for you to do something intelligent with the dialogue box, while you wait for the GUI to do something. This is an excellent feature for those without tight deadlines to meet, and the best solution is to stop work and go home for a long while. Alternatively, you can play ‘hunt the dialogue box’.

[6]Gorey details: features of annotations and documents in GATE may be any virtually any Java object; serialising arbitrary binary data to XML is not simple; instead we serialise them as strings, and therefore they will be re-loaded as strings.

This feature is available totally free of charge.

3.35 [D] Stop GUI Crashing on Linux

On some configurations of Red Hat 7.0 the GUI crashes on startup. The solution is to limit the initial stack size prior to launch: `ulimit -s 2048`.

3.36 [D] Stop GATE Restoring GUI Sessions/Options

GATE will remember GUI options and the state of the resource tree when it exits. The options are saved by default; the session state is not saved by default. This default behaviour can be changed from the “Advanced” tab of the “Configuration” choice on the “Options” menu.

If a problem occurs and the saved data prevents GATE from starting, you can fix it by deleting the configuration and session data files. These are stored in your home directory, and are called `gate.xml` and `gate.sesssion` or `.gate.xml` and `.gate.sesssion` depending on platform. On Windoze your home is:

95, 98, NT: Windows Directory/profiles/username

2000, XP: Windows Drive/Documents and Settings/username

3.37 Work with Unicode

GATE provides various facilities for working with Unicode beyond those that come as default with Java[7]:

1. a Unicode editor with input methods for many languages;
2. use of the input methods in all places where text is edited in the GUI;
3. a development kit for implementing input methods;
4. ability to read diverse character encodings.

[7]Implemented by Valentin Tablan, Mark Leisher and Markus Kramer. Initial version developed by Mark Leisher.

1 using the editor:
In the GUI, select 'Unicode editor' from the 'Tools' menu. This will display an editor window, and, when a language with a custom input method is selected for input (see next section), a virtual keyboard window with the characters of the language assigned to the keys on the keyboard. You can enter data either by typing as normal, or with mouse clicks on the virtual keyboard.

2 configuring input methods:
In the editor and in GATE's main window, the 'Options' menu has an 'Input methods' choice. All supported input languages (a superset of the JDK languages) are available here. Note that you need to use a font capable of displaying the language you select. By default GATE will choose a Unicode font if it can find one on the platform you're running on. Otherwise, select a font manually from the 'Options' menu 'Configuration' choice.

3 using the development kit:
GUK, the GATE Unicode Kit, is documented at http://gate.ac.uk/gate/doc/javadoc/guk/package-summary.html.

4 reading different character encodings:
When you create a document from a URL pointing to textual data in GATE, you have to tell the system what character encoding the text is stored in. By default, GATE will set this parameter to be the empty string. This tells Java to use the default encoding for whatever platform it is running on at the time – e.g. on Western versions of Windoze this will be ISO-8859-1, and Eastern ones ISO-8859-9. A popular way to store Unicode documents is in UTF-8, which is a superset of ASCII (but can still store all Unicode data); if you get an error message about document I/O during reading, try setting the encoding to UTF-8, or some other locally popular encoding. (To see a list of available encodings, try opening a document in GATE's unicode editor – you will be prompted to select an encoding.)

3.38 Work with Oracle and PostgreSQL

GATE's Oracle layer is documented separately in http://gate.ac.uk/gate/doc/persistence.pdf. Note that running an Oracle installation is not for the faint-hearted!

GATE version 2.2 has been adapted to work with Postgres 7.3. The compatibility with PostgreSQL 7.2 has been preserved. Since version 7.3 the Postgres server doesn't downcast from int4 to int2 automatically. However, the JDBC drivers seem to have a bug and send the SMALLINT (aka INT2) parameters as INT (aka INT4). This causes some stored procedures (i.e. all that have input parameters of type INT2) not be recognised. We have fixed this problem by modifying the stored procedures to expose the parameters as INT4 and to manually downcast them inside the stored procedure body.

Please note also the following:

PostgreSQL 7.3 refuses to index values larger than 8Kb/3 (2730 bits). The previous versions probably did the same but without raising an exception.

The only case when such a situation can occur in GATE is when a feature has a TEXTUAL value larger than 2730b. This will be signalled by an exception being raised about the value being too large for the index.

To "solve" this, one can remove the index on the ft_character_value field of the t_feature table. This will have the usual effects caused by removing an index (incapacity of performing efficient searches).

See the example code at http://gate.ac.uk/gate-examples/doc/.

3.39 Annotate using ontologies

This section deals especially with ontologies: how to create/edit them, make Annotation Schemas out of them, and annotate texts automatically with respect to these schemas.

You can load the ontology tools via the plugin manager from the File menu and then Manage CREOLE plugins. Select the Ontology Tools plugin and these tools will be available to you as Processing Resources in the usual way.

Ontology Editor If you double-click on a loaded ontology in the left resources tree in GATE, it will be loaded in the main window. For more information, see section 10.4.

OntoGazetteer The OntoGazetteer is a Processing Resource in which lists of instances of ontology concepts can be loaded. For the OntoGazetteer to function properly, it needs one or more .lst files, and two .def files, usually called mappings.def, and lists.def.

The .lst file is simply a file with an instance on every line. For instance persons.lst will have, on every line, the name of a person.

The mappings.def file describes the relations between the .lst files and the ontology concepts. The format is .lst file:ontology file:ontology concept.

The lists.def file states the relations between the .lst files and the annotation feature the OntoGazetteer should generate. The format is: .lst file:feature

The OntoGazetteer, when run, generates annotations for every instance mentioned in the .lst files. All these instances will be annotated with the same annotation type, called 'Lookup' (in the 'default' Annotation Set). Every Lookup annotation will have a feature, which is its majorType, which will differ per .lst file.

This is not very useful, because every different concept from the ontology will have the same annotation type (namely 'Lookup'). However, since the 'majorType' feature differs, we can process them further. And to do that, we need a Jape Transducer.

Jape Transducer A Jape transducer is a Processing Resource for manipulating annotations. For instance, the annotations generated by the OntoGazetteer can be transcribed to distinct annotations. For this, the Jape Transducer needs a Jape grammar, usually stored in a .jape file. A Jape grammar describes which annotations should be changed, and how. See Chapter 7 for further reference on Jape rules.

Creating a pipeline Once a Processing Resource is loaded, it can be included in a pipeline. To do so, right click on 'Applications' in the left resources tree, and choose one of the appropriate options. A particularly useful one is the 'Corpus Pipeline' that lets you run an application on an entire corpus. A pipeline is created by dragging the Processing Resources into it. They will be run one after another.

NOTE that in order to run the OntoGazetteer, only the OntoGazetteer needs to be selected, and not the Hash Gazetteer.

How to generate annotations automatically Now that all the building blocks have been discussed, let us describe how to actually construct something useful out of it.

Our aim is to have an Annotation Set that contains the concepts out of the ontology. First, we make an OntoGazetteer. This we build from our ontology, and lists.def and mappings.def file. Suppose we run the OntoGazetteer on our corpus. This would produce, as described above, 'Lookup' annotations with a feature majorType set to 'Department'.

Now we want to have a Jape rule that turns this annotation into an annotation of type 'Department'. So it should select every annotation with 'Department' as its majorType, and convert it. This is the rule that does it:

```
Rule: departmentsRule
(
{Lookup.majorType == Department}
):departmentslabel
-->
:departmentslabel.Department = {rule = "departmentsRule"}
```

This then is all we need. We build a corpus pipeline with first the OntoGazetteer, and then the Jape Transducer, and if it is run, the corpus will get annotated automatically.

Chapter 4

CREOLE: the GATE Component Model

. . . Noam Chomsky's answer in *Secrets, Lies and Democracy* (David Barsamian 1994; Odonian) to "What do you think about the Internet?"

"I think that there are good things about it, but there are also aspects of it that concern and worry me. This is an intuitive response – I can't prove it – but my feeling is that, since people aren't Martians or robots, direct face-to-face contact is an extremely important part of human life. It helps develop self-understanding and the growth of a healthy personality.

"You just have a different relationship to somebody when you're looking at them than you do when you're punching away at a keyboard and some symbols come back. I suspect that extending that form of abstract and remote relationship, instead of direct, personal contact, is going to have unpleasant effects on what people are like. It will diminish their humanity, I think."

Chomsky, quoted at http://photo.net/wtr/dead-trees/53015.htm.

The GATE architecture is based on components: reusable chunks of software with well-defined interfaces that may be deployed in a variety of contexts. The design of GATE is based on an analysis of previous work on infrastructure for LE, and of the typical types of software entities found in the fields of NLP and CL (see in particular chapters 4–6 of [Cunningham 00]). Our research suggested that a profitable way to support LE software development was an architecture that breaks down such programs into components of various types. Because LE practice varies very widely (it is, after all, predominantly a research field), the architecture must avoid restricting the sorts of components that developers can plug into the infrastructure. The GATE framework accomplishes this via an adapted version of the *Java Beans* component framework from Sun. Section 4.2 describes Java's component model, Java Beans; section 4.3 describes GATE's extended Beans model.

GATE components may be implemented by a variety of programming languages and databases, but in each case they are represented to the system as a Java class. This class may do nothing other than call the underlying program, or provide an access layer to a database; on the other hand it may implement the whole component.

GATE components are one of three types:

- LanguageResources (LRs) represent entities such as lexicons, corpora or ontologies;
- ProcessingResources (PRs) represent entities that are primarily algorithmic, such as parsers, generators or ngram modellers;
- VisualResources (VRs) represent visualisation and editing components that participate in GUIs.

Section 4.4 discusses the disctinction between Language Resources and Processing Resources. Collectively, the set of resources integrated with GATE is known as **CREOLE**: a Collection of REusable Objects for Language Engineering.

In the rest of this chapter:

- section 4.5 describes the lifecycle of GATE components;
- section 4.6 describes how Processing Resources can be grouped into applications;
- section 4.7 describes the relationship between Language Resources and their data stores;
- section 4.8 summarises GATE's set of built-in components;
- section 4.9 describes how configuration data for Resource types is supplied to GATE.

4.1 The Web and CREOLE

GATE allows resource implementations and Language Resource persistent data to be distributed over the Web, and uses Java annotations and XML for configuration of resources (and GATE itself).

Resource implementations are grouped together as "plugins", stored at a URL (when the resources are in the local file system this can be a `file:/` URL). When a plugin is loaded GATE it looks for a configuration file called `creole.xml` relative to the plugin URL and uses the contents of this file to determine what resources this plugin declares and where to find the classes that implement the resource types (typically these classes are stored in a JAR file in the plugin directory). Configuration data for the resources may be stored directly in the creole.xml file, or it may be stored as Java annotations on the resource classes themselves; in either case GATE retrieves this configuration information and adds the resource definitions to the CREOLE register. When a user requests an instantiation of a resource, GATE creates an instance of the resource class in the virtual machine.

Language resource data can be stored in binary serialised form in the local file system, or in an RDBMS like Oracle. In the latter case, communication with the database is over JDBC[1], allowing the data to be located anywhere on the network (or anywhere you can get Oracle running, that is!).

4.2 Java Beans: a Simple Component Architecture

All GATE resources are *Java Beans*, the Java platform's model of software components. Beans are simply Java classes that obey certain interface conventions. These conventions allow development tools such as GATE, or Borland JBuilder, to manipulate software components without knowing very much about them. The advantage of this is that users of such systems can extend them in diverse ways without having to touch the underlying core of the development tools.

The key parts of the Java Beans specification as used in GATE are:

- accessor and mutator methods for data members are named after those members plus `get` and `set` (meaning that the tool can figure out how to use a member, or *property*, of a bean, from information provided by Java reflection);
- beans must have no-argument constructors (so that tools can construct instances of beans without knowing about complex initialisation parameters).

The rest of this section says a little more about the Beans specification; skip to the next if you're only interested in how it works in GATE.

Quoting from [Campione *et al.* 98] at Sun's Java website:

> The JavaBeans API makes it possible to write component software in the Java programming language. Components are self-contained, reusable software units that can be visually composed into composite components, applets, applications, and servlets using visual application builder tools. JavaBean components are known as *Beans*.

In this context we may think of the GATE development environment as a 'builder tool'. While the emphasis in the quoted text is on visual representation of components, note that

[1]The Java DataBase Connectivity layer.

GATE (and other) beans can also be plugged together 'invisibly'; this is what the framework does and how GATE beans are typically deployed into other applications.

> Components expose their features (for example, public methods and events) to builder tools for visual manipulation. A Bean's features are exposed because feature names adhere to specific *design patterns*. A JavaBeans-enabled builder tool can then examine the Bean's patterns, discern its features, and expose those features for visual manipulation. A builder tool maintains Beans in a palette or toolbox. You can select a Bean from the toolbox, drop it into a form, modify it's appearance and behavior, define its interaction with other Beans, and compose it and other Beans into an applet, application, or new Bean. All this can be done without writing a line of code.

In GATE you develop sets of beans that do language processing tasks and then the framework wires them together without any code from you.

> - Builder tools discover a Bean's features (that is, its properties, methods, and events) by a process known as *introspection*. Beans support introspection in two ways:
>   - By adhering to specific rules, known as *design patterns*, when naming Bean features. The `Introspector` class examines Beans for these design patterns to discover Bean features. The `Introspector` class relies on the core reflection API. ...

The next section describes GATE's extended beans model.

4.3 The GATE Framework

We can think of the GATE framework as a backplane into which plug beans-based CREOLE components. The user gives the system a list of URLs to search when it starts up, and components at those locations are loaded by the system.

The backplane performs these functions:

- component discovery, bootstrapping, loading and reloading;
- management and visualisation of native data structures for common information types;
- generalised data storage and process execution.

A set of components plus the framework is a deployment unit which can be embedded in another application.

The key task of the development environment is to facilitate constructing components, and viewing and measuring their results.

4.4 Language Resources and Processing Resources

This section describes in more detail the *Language Resource* and *Processing Resource* terminology introduced earlier. If you're happy with these terms you can safely skip this section.

Like other software, LE programs consist of data and algorithms. The current orthodoxy in software development is to model both data and algorithms together, as *objects*[2]. Systems that adopt the new approach are referred to as Object-Oriented (OO), and there are good reasons to believe that OO software is easier to build and maintain than other varieties [Booch 94, Yourdon 96].

In the domain of human language processing R&D, however, the terminology is a little more complex. Language data, in various forms, is of such significance in the field that it is frequently worked on independently of the algorithms that process it. For example: a treebank[3] can be developed independently of the parsers that may later be trained from it; a thesaurus can be developed independently of the query expansion or sense tagging mechanisms that may later come to use it. This type of data has come to have its own term, *Language Resources* (LRs) [LREC-1 98], covering many data sources, from lexicons to corpora.

In recognition of this distinction, we will adopt the following terminology:

Language Resource (LR): refers to data-only resources such as lexicons, corpora, thesauri or ontologies. Some LRs come with software (e.g. Wordnet has both a user query interface and C and Prolog APIs), but where this is only a means of accessing the underlying data we will still define such resources as LRs.

Processing Resource (PR): refers to resources whose character is principally programmatic or algorithmic, such as lemmatisers, generators, translators, parsers or speech recognisers. For example, a part-of-speech tagger is best characterised by reference to the process it performs on text. PRs typically *include* LRs, e.g. a tagger often has a lexicon; a word sense disambiguator uses a dictionary or thesaurus.

Additional terminology worthy of note in this context: *language data* refers to LRs which are at their core examples of language in practice, or 'performance data', e.g. corpora of texts or

[2]Older development methods like Jackson Structured Design [Jackson 75] or Structured Analysis [Yourdon 89] kept them largely separate.

[3]A corpus of texts annotated with syntactic analyses.

speech recordings (possibly including added descriptive information as markup); *data about language* refers to LRs which are purely descriptive, such as a grammar or lexicon.

PRs can be viewed as algorithms that map between different types of LR, and which typically use LRs in the mapping process. An MT engine, for example, maps a monolingual corpus into a multilingual aligned corpus using lexicons, grammars, etc.[4]

Further support for the PR/LR terminology may be gleaned from the argument in favour of declarative data structures for grammars, knowledge bases, etc. This argument was current in the late 1980s and early 1990s [Gazdar & Mellish 89], partly as a response to what has been seen as the overly procedural nature of previous techniques such as augmented transition networks. Declarative structures represent a separation between data about language and the algorithms that use the data to perform language processing tasks; a similar separation to that used in GATE.

Adopting the PR/LR distinction is a matter of conforming to established domain practice and terminology. It does not imply that we cannot model the domain (or build software to support it) in an Object-Oriented manner; indeed the models in GATE are themselves Object-Oriented.

4.5 The Lifecycle of a CREOLE Resource

CREOLE resources exhibit a variety of forms depending on the perspective they are viewed from. Their implementation is as a Java class plus an XML metadata file living at the same URL. When using the development environment, resources can be loaded and viewed via the resources tree (left pane) and the ”create resource” mechanism. When programming with the framework, they are Java objects that are obtained by making calls to GATE's `Factory` class. These various incarnations are the phases of a CREOLE resource's 'lifecycle'. Depending on what sort of task you are using GATE for, you may use resources in any or all of these phases. For example, you may only be interested in getting a graphical view of what GATE's ANNIE Information Extraction system (see chapter 8) does; in this case you will use the GUI to load the ANNIE resources, and load a document, and create an ANNIE application and run it on the document. If, on the other hand, you want to create your own resources, or modify the Java code of an existing resource (as opposed to just modifying its grammar, for example), you will need to deal with all the lifecylce phases.

The various phases may be summarised as:

Creating a new resource from scratch (bootstrapping). To create the binary image of a resource (a Java class in a JAR file), and the XML file that describes the resource to GATE, you need to create the appropriate `.java` file(s), compile them and package them as a `.jar`. The GATE development environment provides a bootstrap tool to

[4]This point is due to Wim Peters.

start this process – see section 3.10. Alternatively you can simply copy code from an existing resource.

Instantiating a resource in the framework. To create a resource in your own Java code, use GATE's `Factory` class (this takes care of parameterising the resource, restoring it from a database where appropriate, etc. etc.). Section 3.11 describes how to do this.

Loading a resource in the development environment. To load a resource in the development environment, use the various "New ... resource" options from the `File` menu and elsewhere. See section 3.12.

Resource configuration and implementation. GATE's bootstrap tool will create an empty resource that does nothing. In order to achieve the behaviour you require, you'll need to change the configuration of the resource (by editing the `creole.xml` file) and/or change the Java code that implements the resource. See section 4.9.

More details of the specifics of tasks related to these phases are available in chapter 3.

4.6 Processing Resources and Applications

PRs can be combined into *applications*. Applications model a control strategy for the execution of PRs. In the framework applications are called 'controllers' accordingly.

Currently only sequential, or pipeline, execution is supported. There are two main types of pipeline:

Simple pipelines simply group a set of PRs together in order and execute them in turn. The implementing class is called `SerialController`.

Corpus pipelines are specific for LanguageAnalysers – PRs that are applied to documents and corpora. A corpus pipeline opens each document in the corpus in turn, sets that document as a runtime parameter on each PR, runs all the PRs on the corpus, then closes the document. The implementing class is called `SerialAnalyserController`.

Conditional versions of these controllers are also available. These allow processing resources to be run conditionally on document features. See Section 3.15 for how to use these.

There is also a real-time version of the corpus pipeline. When creating such a controller, a `timeout` parameter needs to be set which determines the maximum amount of time (in milliseconds) allowed for the processing of a document. Documents that take longer to process, are simply ignored and the execution moves to the next document after the timeout interval has lapsed.

All controllers have special handling for processing resources that implement the interface `gate.creole.ControllerAwarePR`. This interface provides methods that are called by the controller at the start and end of the whole application's execution – for a corpus pipeline, this means before any document has been processed and after all documents in the corpus have been processed, which is useful for PRs that need to share data structures across the whole corpus, build aggregate statistics, etc. For full details, see the JavaDoc documentation for `ControllerAwarePR`.

4.7 Language Resources and Datastores

Language Resources can be stored in Data Stores. Data Stores are an abstract model of disk-based persistence, which can be implemented by various types of storage mechanism. Here are the types implemented:

Serial Data Stores are based on Java's serialisation system, and store data directly into files and directories.

Lucene Data Stores is a full-featured annotation indexing and retrieval system. It is provided as part of an extension of the Serial Data Stores. See section 9.29 for more details.

Oracle Data Stores store data into an Oracle RDBMS. For details of how to set up an Oracle DB for GATE, see http://gate.ac.uk/gate/doc/persistence.pdf.

PostgreSQL Data Stores store data into a PostgreSQL RDBMS. For details of how to set up a PostgreSQL DB for GATE, see http://gate.ac.uk/gate/doc/persistence.pdf.

4.8 Built-in CREOLE Resources

GATE comes with various built-in components:

- Language Resources modelling Documents and Corpora, and various types of Annotation Schema – see chapter 6.
- Processing Resources that are part of the ANNIE system – see chapter 8.

- Visual Resources for viewing and editing corpora, annotations, etc.
- Other miscellaneous resources – see chapter 9.

4.9 CREOLE Resource Configuration

This section describes how to supply GATE with the configuration data it needs about a resource, such as what its parameters are, how to display it if it has a visualisation, etc. Several GATE resources can be grouped into a single *plugin*, which is a directory containing an XML configuration file called `creole.xml`. Configuration data for the plugin's resources can be given in the `creole.xml` file or directly in the Java source file using Java 5 annotations.

A `creole.xml` file has a root element `<CREOLE-DIRECTORY>`, but the further contents of this element depend on the configuration style. The following three sections discuss the different styles – all-XML, all-annotations and a mixture of the two.

4.9.1 Configuration with XML

To configure your resources in the `creole.xml` file, the `<CREOLE-DIRECTORY>` element should contain one `<RESOURCE>` element for each resource type in the plugin. The `<RESOURCE>` elements may optionally be contained within a `<CREOLE>` element (to allow a single `creole.xml` file to be built up by concatenating multiple separate files). For example:

```
<CREOLE-DIRECTORY>

<CREOLE>
  <RESOURCE>
    <NAME>Minipar Wrapper</NAME>
    <JAR>MiniparWrapper.jar</JAR>
    <CLASS>minipar.Minipar</CLASS>
    <COMMENT>MiniPar is a shallow parser. It determines the
    dependency relationships between the words of a sentence.</COMMENT>
    <HELPURL>http://gate.ac.uk/cgi-bin/userguide/sec:misc-creole:minipar</HELPURL>
    <PARAMETER NAME="document"
  RUNTIME="true"
  COMMENT="document to process">gate.Document</PARAMETER>
    <PARAMETER NAME="miniparDataDir"
          RUNTIME="true"
          COMMENT="location of the Minipar data directory">
          java.net.URL
    </PARAMETER>
    <PARAMETER NAME="miniparBinary"
          RUNTIME="true"
          COMMENT="Name of the Minipar command file">
          java.net.URL
    </PARAMETER>
    <PARAMETER NAME="annotationInputSetName"
          RUNTIME="true"
          OPTIONAL="true"
          COMMENT="Name of the input Source">
          java.lang.String
    </PARAMETER>
    <PARAMETER NAME="annotationOutputSetName"
          RUNTIME="true"
          OPTIONAL="true"
          COMMENT="Name of the output AnnotationSetName">
          java.lang.String
    </PARAMETER>
    <PARAMETER NAME="annotationTypeName"
          RUNTIME="false"
          DEFAULT="DepTreeNode"
          COMMENT="Annotations to store with this type">
          java.lang.String
    </PARAMETER>
  </RESOURCE>
</CREOLE>
</CREOLE-DIRECTORY>
```

Basic resource-level data

Each resource must give a name, a Java class and the JAR file that it can be loaded from. The above example is taken from the `Minipar` plugin in GATE, and defines a single resource with a number of parameters.

The full list of valid elements under `<RESOURCE>` is as follows:

NAME the name of the resource, as it will appear in the “New” menu in the GATE GUI. If omitted, defaults to the bare name of the resource class (without a package name).

CLASS the fully qualified name of the Java class that implements this resource.

JAR names JAR files required by this resource (paths are relative to the location of `creole.xml`). Typically this will be the JAR file containing the class named by the `<CLASS>` element, but additional `<JAR>` elements can be used to name third-party JAR files that the resource depends on.

COMMENT a descriptive comment about the resource, which will appear as the tooltip when hovering over an instance of this resource in the resources tree in the GUI. If omitted, no comment is used.

HELPURL a URL to a help document on the web for this resource. It is used in the help browser inside GATE.

INTERFACE the interface type implemented by this resource, for example new types of document would specify `<INTERFACE>gate.Document</INTERFACE>`.

ICON the icon used to represent this resource in the GATE GUI. This is a path inside the plugin's JAR file, for example `<ICON>/some/package/icon.png</ICON>`. If the path specified does not start with a forward slash, it is assumed to name an icon from the GATE default set, which is located in gate.jar at gate/resources/img. If no icon is specified, a generic language resource or processing resource icon (as appropriate) is used.

PRIVATE if present, this resource type is hidden in the GUI, i.e. it is not shown in the "New" menus. This is useful for resource types that are intended to be created internally by other resources, or for resources that have parameters of a type that cannot be set in the GUI. `<PRIVATE/>` resources can still be created in Java code using the `Factory`.

AUTOINSTANCE (and HIDDEN-AUTOINSTANCE) tells GATE to automatically create instances of this resource when the plugin is loaded. Any number of auto instances may be defined, GATE will create them all. Each `<AUTOINSTANCE>` element may optionally contain `<PARAM NAME="..." VALUE="..." />` elements giving parameter values to use when creating the instance. Any parameters not specified explicitly will take their default values. Use `<HIDDEN-AUTOINSTANCE>` if you want the auto instances not to show up in the GATE GUI – this is useful for things like document formats where there should only ever be a single instance in GATE and that instance should not be deleted.

For visual resources, a `<GUI>` element should also be provided. This takes a `TYPE` attribute, which can have the value `LARGE` or `SMALL`. LARGE means that the visual resource is a large viewer and should appear in the main part of the GATE window on the right hand side, SMALL means the VR is a small viewer which appears in the space below the resources tree in the bottom left. The `<GUI>` element supports the following sub-elements:

RESOURCE_DISPLAYED the type of GATE resource this VR can display. Any resource whose type is assignable to this type will be displayed with this viewer, so for example a VR that can display all types of document would specify `gate.Document`, whereas a VR that can only display the default GATE document implementation would specify `gate.corpora.DocumentImpl`.

MAIN_VIEWER if present, GATE will consider this VR to be the "most important" viewer for the given resource type, and will ensure that if several different viewers are all applicable to this resource, this viewer will be the one that is initially visible.

For annotation viewers, you should specify an `<ANNOTATION_TYPE_DISPLAYED>` element giving the annotation type that the viewer can display (e.g. `Sentence`).

Resource parameters

Resources may also have parameters of various types. These resources, from the GATE distribution, illustrate the various types of parameters:

```
<RESOURCE>
  <NAME>GATE document</NAME>
  <CLASS>gate.corpora.DocumentImpl</CLASS>
  <INTERFACE>gate.Document</INTERFACE>
  <COMMENT>GATE transient document</COMMENT>
  <OR>
    <PARAMETER NAME="sourceUrl"
      SUFFIXES="txt;text;xml;xhtm;xhtml;html;htm;sgml;sgm;mail;email;eml;rtf"
      COMMENT="Source URL">java.net.URL</PARAMETER>
    <PARAMETER NAME="stringContent"
      COMMENT="The content of the document">java.lang.String</PARAMETER>
  </OR>
  <PARAMETER
    COMMENT="Should the document read the original markup"
    NAME="markupAware" DEFAULT="true">java.lang.Boolean</PARAMETER>
  <PARAMETER NAME="encoding" OPTIONAL="true"
    COMMENT="Encoding" DEFAULT="">java.lang.String</PARAMETER>
  <PARAMETER NAME="sourceUrlStartOffset"
    COMMENT="Start offset for documents based on ranges"
    OPTIONAL="true">java.lang.Long</PARAMETER>
  <PARAMETER NAME="sourceUrlEndOffset"
    COMMENT="End offset for documents based on ranges"
    OPTIONAL="true">java.lang.Long</PARAMETER>
  <PARAMETER NAME="preserveOriginalContent"
    COMMENT="Should the document preserve the original content"
    DEFAULT="false">java.lang.Boolean</PARAMETER>
  <PARAMETER NAME="collectRepositioningInfo"
    COMMENT="Should the document collect repositioning information"
    DEFAULT="false">java.lang.Boolean</PARAMETER>
  <ICON>lr.gif</ICON>
</RESOURCE>
```

```
<RESOURCE>
  <NAME>Document Reset PR</NAME>
  <CLASS>gate.creole.annotdelete.AnnotationDeletePR</CLASS>
  <COMMENT>Document cleaner</COMMENT>
  <PARAMETER NAME="document" RUNTIME="true">gate.Document</PARAMETER>
  <PARAMETER NAME="annotationTypes" RUNTIME="true"
    OPTIONAL="true">java.util.ArrayList</PARAMETER>
</RESOURCE>
```

Parameters may be optional, and may have default values (and may have comments to describe their purpose, which is displayed by the GUI during interactive parameter setting).

Some PR parameters are execution time (`RUNTIME`), some are initialisation time. E.g. at execution time a doc is supplied to a language analyser; at initilisation time a grammar may be supplied to a language analyser.

The `<PARAMETER>` tag takes the following attributes:

NAME: name of the JavaBean property that the parameter refers to, i.e. for a parameter named "someParam" the class must have `setSomeParam` and `getSomeParam` methods.[5]

DEFAULT: default value (see below).

RUNTIME: doesn't need setting at initialisation time, but must be set before calling `execute()`. Only meaningfull for PRs

OPTIONAL: not required

COMMENT: for display purposes

ITEM_CLASS_NAME: (only applies to parameters whose type is `java.util.Collection` or a type that implements or extends this) this specifies the type of elements the collection contains, so the GUI can use the right type when parameters are set. If omitted, the GUI will pass in the elements as Strings.

SUFFIXES: (only applies to parameters of type `java.net.URL`) a semicolon-separated list of file suffixes that this parameter typically accepts, used as a filter in the file chooser provided by the GUI to select a local file as the parameter value.

It is possible for two or more parameters to be mutually exclusive (i.e. a user must specify one or the other but not both). In this case the `<PARAMETER>` elements should be grouped together under an `<OR>` element.

[5]The JavaBeans spec allows `is` instead of `get` for properties of the primitive type `boolean`, but GATE does not support parameters with primitive types. Parameters of type `java.lang.Boolean` (the wrapper class) are permitted, but these have `get` accessors anyway.

The type of the parameter is specified as the text of the `<PARAMETER>` element, and the type supplied must match the return type of the parameter's `get` method. Any reference type (class, interface or enum) may be used as the parameter type, including other resource types – in this case the GUI will offer a list of the loaded instances of that resource as options for the parameter value. Primitive types (char, boolean, ...) are not supported, instead you should use the corresponding wrapper type (`java.lang.Character`, `java.lang.Boolean`, ...). If the getter returns a parameterized type (e.g. `List<Integer>`) you should just specify the raw type (`java.util.List`) here[6].

The DEFAULT string is converted to the appropriate type for the parameter - `java.lang.String` parameters use the value directly, primitive wrapper types e.g. `java.lang.Integer` use their respective `valueOf` methods, and other built-in Java types can have defaults specified provided they have a constructor taking a `String`.

The type `java.net.URL` is treated specially: if the default string is not an absolute URL (e.g. http://gate.ac.uk/) then it is treated as a path relative to the location of the `creole.xml` file. Thus a DEFAULT of `"resources/main.jape"` in the file `file:/opt/MyPlugin/creole.xml` is treated as the absolute URL `file:/opt/MyPlugin/resources/main.jape`.

For `Collection`-valued parameters multiple values may be specified, separated by semi-colons, e.g. `"foo;bar;baz"`; if the parameter's type is an interface – `Collection` or one of its sub-interfaces (e.g. `List`) – a suitable concrete class (e.g. `ArrayList`, `HashSet`) will be chosen automatically for the default value.

For parameters of type `gate.FeatureMap` multiple `name=value` pairs can be specified, e.g. `"kind=word;orth=upperInitial"`. For `enum`-valued parameters the default string is taken as the name of the enum constant to use. Finally, if no DEFAULT attribute is specified, the default value is `null`.

4.9.2 Configuring resources using annotations

Annotation-driven configuration is only available in snapshot builds of GATE, build 2988 and later (subversion revision 9845)

As an alternative to the XML configuration style, GATE provides Java 5 annotation types to embed the configuration data directly in the Java source code. `@CreoleResource` is used to mark a class as a GATE resource, and parameter information is provided through annotations on the JavaBean `set` methods. At runtime these annotations are read and mapped into the equivalent entries in `creole.xml` before parsing. The metadata annotation types are all marked `@Documented` so the CREOLE configuration data will be visible in the generated JavaDoc documentation.

[6]In this particular case, as the type is a collection, you would specify `java.lang.Integer` as the `ITEM_CLASS_NAME`.

For more detailed information, see the JavaDoc documentation for `gate.creole.metadata`.

To use annotation-driven configuration a `creole.xml` file is still required but it need only contain the following:

```
<CREOLE-DIRECTORY>
  <JAR SCAN="true">myPlugin.jar</JAR>
  <JAR>lib/thirdPartyLib.jar</JAR>
</CREOLE-DIRECTORY>
```

This tells GATE to load `myPlugin.jar` and scan its contents looking for resource classes annotated with `@CreoleResource`. Other JAR files required by the plugin can be specified using other `<JAR>` elements without `SCAN="true"`.

Basic resource-level data

To mark a class as a CREOLE resource, simply use the `@CreoleResource` annotation (in the `gate.creole.metadata` package), for example:

```
import gate.creole.AbstractLanguageAnalyser;
import gate.creole.metadata.*;

@CreoleResource(name = "GATE Tokeniser",
                comment = "Splits text into tokens and spaces")
public class Tokeniser extends AbstractLanguageAnalyser {
  ...
```

The `@CreoleResource` annotation provides slots for all the values that can be specified under `<RESOURCE>` in `creole.xml`, except `<CLASS>` (inferred from the name of the annotated class) and `<JAR>` (taken to be the JAR containing the class):

name (String) the name of the resource, as it will appear in the “New” menu in the GATE GUI. If omitted, defaults to the bare name of the resource class (without a package name). (XML equivalent `<NAME>`)

comment (String) a descriptive comment about the resource, which will appear as the tooltip when hovering over an instance of this resource in the resources tree in the GUI. If omitted, no comment is used. (XML equivalent `<COMMENT>`)

helpURL (String) a URL to a help document on the web for this resource. It is used in the help browser inside GATE. (XML equivalent `<HELPURL>`)

isPrivate (boolean) should this resource type be hidden from the GUI, so it does not appear in the "New" menus? If omitted, defaults to false (i.e. not hidden). (XML equivalent `<PRIVATE/>`)

icon (String) the icon to use to represent the resource in the GUI. If omitted, a generic language resource or processing resource icon is used. (XML equivalent `<ICON>`, see the description above for details)

interfaceName (String) the interface type implemented by this resource, for example a new type of document would specify `"gate.Document"` here. (XML equivalent `<INTERFACE>`)

autoInstances (array of `@AutoInstance` annotations) definitions for any instances of this resource that should be created automatically when the plugin is loaded. If omitted, no auto-instances are created by default. (XML equivalent, one or more `<AUTOINSTANCE>` and/or `<HIDDEN-AUTOINSTANCE>` elements, see the description above for details)

For visual resources only, the following elements are also available:

guiType (GuiType enum) the type of GUI this resource defines. (XML equivalent `<GUI TYPE="LARGE|SMALL">`)

resourceDisplayed (String) the class name of the resource type that this VR displays, e.g. `"gate.Corpus"`. (XML equivalent `<RESOURCE_DISPLAYED>`)

mainViewer (boolean) is this VR the "most important" viewer for its displayed resource type? (XML equivalent `<MAIN_VIEWER/>`, see above for details)

For annotation viewers, you should specify an `annotationTypeDisplayed` element giving the annotation type that the viewer can display (e.g. `Sentence`).

Resource parameters

Parameters are declared by placing annotations on their JavaBean `set` methods. To mark a setter method as a parameter, use the `@CreoleParameter` annotation, for example:

```
@CreoleParameter(comment = "The location of the list of abbreviations")
public void setAbbrListUrl(URL listUrl) {
  ...
```

GATE will infer the parameter's name from the name of the JavaBean property in the usual way (i.e. strip off the leading `set` and convert the following character to lower case, so in this example the name is `abbrListUrl`). The parameter name is *not* taken from the name of the method parameter. The parameter's type is inferred from the type of the method parameter (`java.net.URL` in this case).

The annotation elements of `@CreoleParameter` correspond to the attributes of the `<PARAMETER>` tag in the XML configuration style:

comment (String) an optional descriptive comment about the parameter. (XML equivalent `COMMENT`)

defaultValue (String) the optional default value for this parameter. The value is specified as a string but is converted to the relevant type by GATE according to the conversions described in the previous section. Note that relative path default values for URL-valued parameters are still relative to the location of the `creole.xml` file, not the annotated class. (XML equivalent `DEFAULT`)

suffixes (String) for URL-valued parameters, a semicolon-separated list of default file suffixes that this parameter accepts. (XML equivalent `SUFFIXES`)

collectionElementType (Class) for `Collection`-valued parameters, the type of the elements in the collection. This can usually be inferred from the generic type information, for example `public void setIndices(List<Integer> indices)`, but must be specified if the `set` method's parameter has a raw (non-parameterized) type. (XML equivalent `ITEM_CLASS_NAME`)

Mutually-exclusive parameters (such as would be grouped in an `<OR>` in `creole.xml`) are handled by adding a `disjunction="`*label*`"` to the `@CreoleParameter` annotation – all parameters that share the same label are grouped in the same disjunction.

Optional and runtime parameters are marked using extra annotations, for example:

```
@Optional
@RunTime
@CreoleParameter
public void setAnnotationSetName(String asName) {
  ...
```

Inheritance

Unlike with pure XML configuration, when using annotations a resource will inherit any configuration data that was not explicitly specified from annotations on its parent class and on any interfaces it implements. Specifically, if you do not specify a comment, interfaceName, icon, annotationTypeDisplayed or the GUI-related elements (guiType and resourceDisplayed) on your `@CreoleResource` annotation then GATE will look up the class tree for other `@CreoleResource` annotations, first on the superclass, its superclass, etc., then at any implemented interfaces, and use the first value it finds. This is useful if you are defining a family of related resources that inherit from a common base class.

The resource name and the `isPrivate` and `mainViewer` flags are *not* inherited.

Parameter definitions are inherited in a similar way. This is one of the big advantages of annotation configuration over pure XML – if one resource class extends another then with pure XML configuration all the parent class's parameter definitions must be duplicated in the subclass's `creole.xml` definition. With annotations, parameters are inherited from the parent class (and its parent, etc.) as well as from any interfaces implemented. For example, the `gate.LanguageAnalyser` interface provides two parameter definitions via annotated `set` methods, for the `corpus` and `document` parameters. Any `@CreoleResource` annotated class that implements `LanguageAnalyser`, directly or indirectly, will get these parameters automatically.

Of course, there are some cases where this behaviour is not desirable, for example if a subclass calculates a value for a superclass parameter rather than having the user set it directly. In this case you can hide the parameter by overriding the `set` method in the subclass and using a marker annotation:

```
@HiddenCreoleParameter
public void setSomeParam(String someParam) {
  super.setSomeParam(someParam);
}
```

The overriding method will typically just call the superclass one, as its only purpose is to provide a place to put the `@HiddenCreoleParameter` annotation.

Alternatively, you may want to override some of the configuration for a parameter but inherit the rest from the superclass. Again, this is handled by trivially overriding the `set` method and re-annotating it:

```
// superclass
@CreoleParameter(comment = "Location of the grammar file",
                 suffixes = "jape")
public void setGrammarUrl(URL grammarLocation) {
  ...
}

@Optional
@RunTime
@CreoleParameter(comment = "Feature to set on success")
public void setSuccessFeature(String name) {
  ...
}
```

```
  //-----------------------------------
  // subclass

  // override the default value, inherit everything else
  @CreoleParameter(defaultValue = "resources/defaultGrammar.jape")
  public void setGrammarUrl(URL url) {
    super.setGrammarUrl(url);
  }

  // we want the parameter to be required in the subclass
  @Optional(false)
  @CreoleParameter
  public void setSuccessFeature(String name) {
    super.setSuccessFeature(name);
  }
```

Note that for backwards compatibility, data is only inherited from superclass annotations if the subclass is itself annotated with `@CreoleResource`. If the subclass is not annotated then GATE assumes that *all* its configuration is contained in `creole.xml` in the usual way.

4.9.3 Mixing the configuration styles

It is possible and often useful to mix and match the XML and annotation-driven configuration styles. The rule is always that *anything specified in the XML takes priority over the annotations*. The following examples show what this allows.

Overriding configuration for a third-party resource

Suppose you have a plugin from some third party that uses annotation-driven configuration. You don't have the source code but you would like to override the default value for one of the parameters of one of the plugin's resources. You can do this in the `creole.xml`:

```
<CREOLE-DIRECTORY>
  <JAR SCAN="true">acmePlugin-1.0.jar</JAR>

  <!-- Add the following to override the annotations -->
  <RESOURCE>
    <CLASS>com.acme.plugin.UsefulPR</CLASS>
    <PARAMETER NAME="listUrl"
      DEFAULT="resources/myList.txt">java.net.URL</PARAMETER>
  </RESOURCE>
</CREOLE-DIRECTORY>
```

The default value for the `listUrl` parameter in the annotated class will be replaced by your value.

External AUTOINSTANCEs

For resources like document formats, where there should always and only be one instance in GATE at any time, it makes sense to put the auto-instance definitions in the `@CreoleResource` annotation. But if the automatically created instances are a convenience rather than a neccessity it may be better to define them in XML so other users can disable them without re-compiling the class:

```
<CREOLE-DIRECTORY>
  <JAR SCAN="true">myPlugin.jar</JAR>

  <RESOURCE>
    <CLASS>com.acme.AutoPR</CLASS>
    <AUTOINSTANCE>
      <PARAM NAME="type" VALUE="Sentence" />
    </AUTOINSTANCE>
    <AUTOINSTANCE>
      <PARAM NAME="type" VALUE="Paragraph" />
    </AUTOINSTANCE>
  </RESOURCE>
</CREOLE-DIRECTORY>
```

Inheriting parameters

If you would prefer to use XML configuration for your own resources, but would like to benefit from the parameter inheritance features of the annotation-driven approach, you can write a normal `creole.xml` file with all your configuration and just add a blank `@CreoleResource` annotation to your class. For example:

```
package com.acme;
import gate.*;
import gate.creole.metadata.CreoleResource;

@CreoleResource
public class MyPR implements LanguageAnalyser {
  ...
}

<!-- creole.xml -->
```

```
<CREOLE-DIRECTORY>
  <CREOLE>
    <RESOURCE>
      <NAME>My Processing Resource</NAME>
      <CLASS>com.acme.MyPR</CLASS>
      <COMMENT>...</COMMENT>
      <PARAMETER NAME="annotationSetName"
        RUNTIME="true" OPTIONAL="true">java.lang.String</PARAMETER>
      <!--
      don't need to declare document and corpus parameters, they
      are inherited from LanguageAnalyser
      -->
    </RESOURCE>
  </CREOLE>
</CREOLE-DIRECTORY>
```

N.B. Without the `@CreoleResource` the parameters would not be inherited.

Chapter 5

Visual CREOLE

...neurobiologists still go on openly studying reflexes and looking under the hood, not huddling passively in the trenches. Many of them still keep wondering: how does the inner life arise? Ever puzzled, they oscillate between two major fictions: (1) The brain can be understood; (2) We will never come close. Meanwhile they keep pursuing brain mechanisms, partly from habit, partly out of faith. Their premise: The brain is the organ of the mind. Clearly, this three-pound lump of tissue is the source of our "insight information" about our very being. Somewhere in it there might be a few hidden guidelines for better ways to lead our lives.

Zen and the Brain, James H. Austin, 1998 (p. 6).

This chapter details the other visual resources that can be used in GATE. While these tools were not included as part of earlier releases of GATE, as of GATE version 3.0, they are included as part of the standard release, and are now open source. GAZE, Ontogazetteer and the Protégé VR for GATE were all developed by Ontotext, who should be contacted for further information about these components.

5.1 Gazetteer Visual Resource - GAZE

Gaze is a tool for editing the gazetteer lists, definitions and mapping to ontology. It is suitable for use both for Plain/Linear Gazetteers (Default and Hash Gazetteers) and Ontology-enabled Gazetteers (OntoGazetteer). The Gazetteer PR associated with the viewer is reinitialised every time a save operation is performed. Note that GAZE does not scale up to very large lists (we suggest not using it to view over 40,000 entries and not to copy inside more than 10, 000 entries).

5.1.1 Running Modes

The running mode depends on the type of gazetteer loaded in the VR. The mode in which Linear/Plain Gazetteers are loaded is called Linear/Plain Mode. In this mode, the Linear Definition is displayed in the left pane, and the Gazetteer List is displayed in the right pane. The Extended/Ontology/Mapping mode is on when the displayed gazetteer is ontology-aware, which means that there exists a mapping between classes in the ontology and lists of phrases. Two more panes are displayed when in this mode. On the top in the left-most pane there is a tree view of the ontology hierarchy, and at the bottom the mapping definition is displayed.

5.1.2 Loading a Gazetteer

To load a gazetteer into the viewer it is necessary to associate the Gaze VR with the gazetteers. Afterwards whenever a gazetteer PR is loaded, Gaze will appear on double-click over the gazetteer in the Processing Resources branch of the Resources Tree.

5.1.3 Linear Definition Pane

This pane displays the nodes of the linear definition, and allows manipulation of the whole definition as a file, as well as the single nodes. Whenever a gazetteer list is modified, its node in the linear definition is coloured in red.

5.1.4 Linear Definition Toolbar

All the functionality explained in this section (New, Load, Save, Save As) is accessible also via File — Linear Definition in the menu bar of Gaze.

New – Pressing New invokes a file dialog where the location of the new definition is specified.

Load – Pressing Load invokes a file dialog, and after locating the new definition it is loaded by pressing Open.

Save – Pressing Save saves the definition to the location from which it has been read.

Save As – Pressing Save As allows another location to be chosen, and the definition saved there.

5.1.5 Operations on Linear Definition Nodes

Double-click node – Double-clicking on a definition node forces the displaying of the gazetteer list of the node in the right-most pane of the viewer.

Insert – On right-click over a node and choosing Insert, a dialog is displayed, requesting List, Major Type, Minor Type and Languages. The mandatory fields are List and Major Type. After pressing OK, a new linear node is added to the definition.

Remove – On right-click over a node and choosing Remove, the selected linear node is removed from the definition.

Edit – On right-click over a node and choosing Edit a dialog is displayed allowing changes of the fields List, Major Type, Minor Type and Languages.

5.1.6 Gazetteer List Pane

The gazetteer list pane has a toolbar with similar to the linear definition's buttons (New, Load, Save, Save As). They work as predicted by their names and as explained in the Linear Definition Pane section, and are also accessible from File / Gazetteer List in the menu bar of Gaze. The only addition is Save All which saves all modified gazetteer lists. The editing of the gazetteer list is as simple as editing a text file. One could use Ctrl+A to select the whole list, Ctrl+C to copy the selected, Ctrl+V to paste it, Del to delete the selected text or a single character, etc.

5.1.7 Mapping Definition Pane

The mapping definition is displayed one mapping node per row. It consists of a gazetteer list, ontology URL, and class id. The content of the gazetteer list in the node is accessible through double-clicking. It is displayed in the Gazetteer List Pane. The toolbar allows the creation of a new definition (New), the loading of an existing one (Load), saving to the same or new location (Save/Save As). The functionality of the toolbar buttons is also available via File.

5.2 Ontogazetteer

The Ontogazetteer, or Hierarchical Gazetteer, is an interface which makes ontologies "visible" in GATE, supporting basic methods for hierarchy management and traversal. In GATE, an ontology is represented at the same level as a document, and has nodes called classes (for consistency with RDFs ad DAML+OIL, though they are really just types). The OntoGazetteer assigns classes rather than major or minor types, and is aware of mappings between lists and class IDs. There are two Visual Resources, one for editing the standard gazetteer lists (including the definition files and the mappings to the ontology), and one for editing the ontology itself.

5.2.1 Gazetteer Lists Editor and Mapper

This is a VR for editing the gazetteer lists, and mapping them to classes in an ontology. It provides load/store/edit for the lists, load/store/edit for the mapping information, loading of ontologies, load/store/edit for the linear definition file, and mapping of the lists file to the major type, minor type and language.

Left pane: A single ontology is visualized in the left pane of the VR. The mapping between a list and a class is displayed by showing the list as a subclass with a different icon. The mapping is specified by drag and drop from the linear definition pane (in the middle) and/or by right click menu.

Middle pane: The middle pane displays the nodes/lines in the linear definition file. By double clicking on a node the corresponding list is opened. Editing of the line/node is done by right clicking and choosing edit: a dialogue appears (lower part of the scheme) allowing the modification of the members of the node.

Right pane: In the right pane a single gazetteer list is displayed. It can be edited and parts of it can be cut/copied/pasted.

5.2.2 Ontogazetteer Editor

This is a VR for editing the class hierarchy of an ontology. it provides storing to and loading from RDF/RDFS, and provides load/edit/store of the class hierarchy of an ontology.

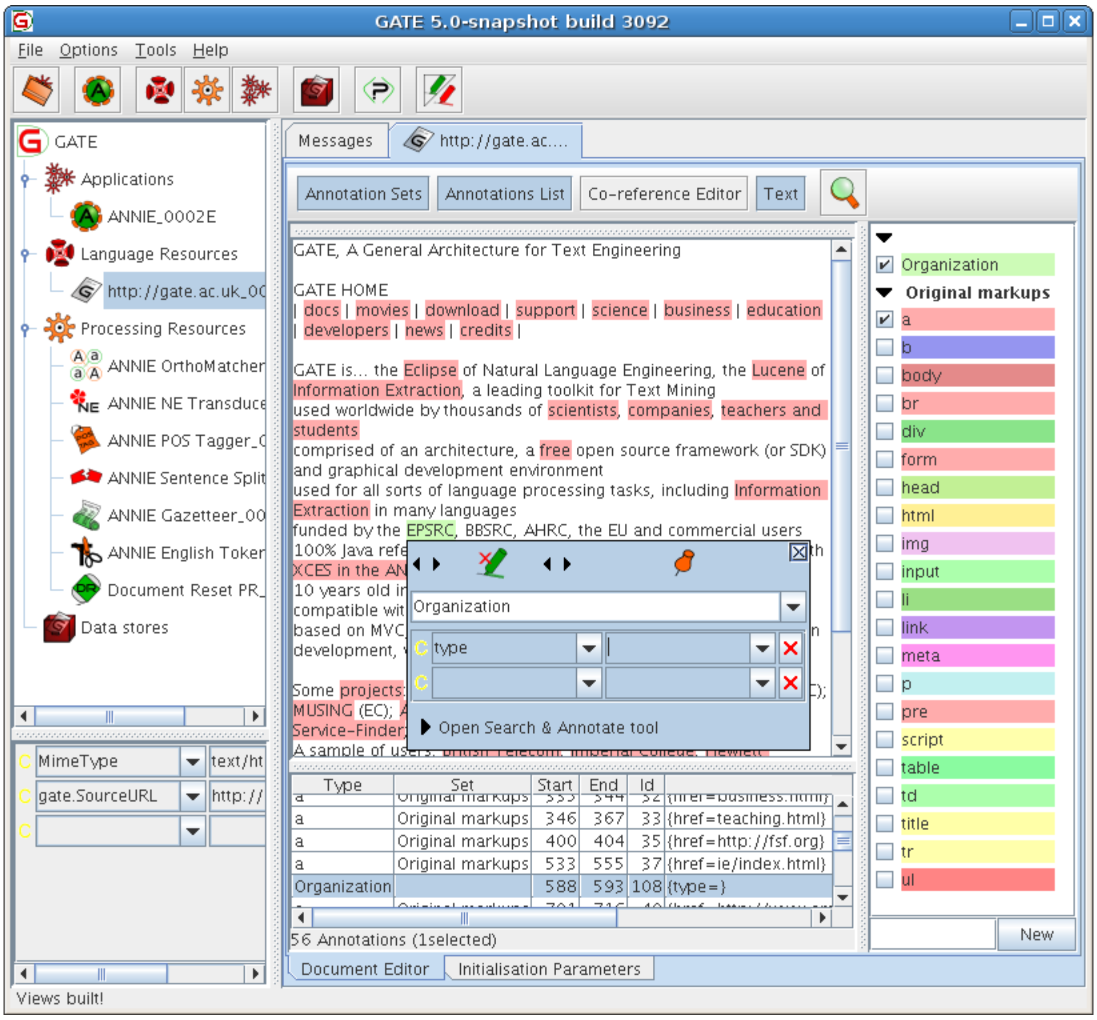
Left pane: The various ontologies loaded are listed here. On double click or right click and edit from the menu the ontology is visualized in the Right pane.

Right pane: Besides the visualization of the class hierarchy of the ontology the following operations are allowed:

- expanding/collapsing parts of the ontology
- adding a class in the hierarchy: by right clicking on the intended parent of the new class and choosing add sub class.
- removing a class: via right clicking on the class and choosing remove.

As a result of this VR, the ontology definition file is affected/altered.

5.3 The Document Editor



MimeType	text/ht
gate.SourceURL	http://

GATE, A General Architecture for Text Engineering

GATE HOME
| docs | movies | download | support | science | business | education
| developers | news | credits |

GATE is... the Eclipse of Natural Language Engineering, the Lucene of
Information Extraction, a leading toolkit for Text Mining
used worldwide by thousands of scientists, companies, teachers and
students
comprised of an architecture, a free open source framework (or SDK)
and graphical development environment
used for all sorts of language processing tasks, including Information
Extraction in many languages
funded by the EPSRC, BBSRC, AHRC, the EU and commercial users
100% Java refe
XCES in the AN
10 years old in
compatible wit
based on MVC
development,

Some projects:
MUSING (EC);
Service-Finder
A sample of users: British Telecom, Imperial College, Hewlett

Organization

| type | |

Type	Set	Start	End	Id	
a	Original markups	333	344	32	{href=business.html}
a	Original markups	346	367	33	{href=teaching.html}
a	Original markups	400	404	35	{href=http://fsf.org}
a	Original markups	533	555	37	{href=ie/index.html}
Organization		588	593	108	{type=}

56 Annotations (1selected)

Figure 5.1: Main window with a document editor showing the location http://gate.ac.uk. You can see a popup window under the word 'EPSRC' for creating/editing an annotation, the table of annotations highlighted at the bottom and the list of existing annotation types on the left.

The document editor is contained in the central tabbed pane as seen on figure 5.1. It consist of a top panel with buttons and icons that control the display of different views and the search box.

The central part is the text view, then at the bottom there is the annotations list view and at the left the annotation sets view which can be replaced with the co-reference editor.

These views are describe in the next subsections so we will now focus only on the annotation editor popup window that you can see in the middle of the document editor.

The annotation editor consist of different action icons at the top then a drop down box for the annotation type, a table of features names and values and finally a disclosure panel for the search and annotate function.

To grow/shrink the span of the annotation at its start use the two arrow icons on the left or Right and Left keys. Use the two arrow icons next on the right to change the annotation end or Alt+Right and Alt+Left keys. Add Shift and Control+Shift keys to make the span increment bigger. The red X icon is for removing the annotation and the pin icon is to pin the window so it doesn't move when you select another annotation.

All the views are updated each time an annotation change. There is more than one way to create or edit annotations so try to find the best for your task. For example, if you want to delete all the annotations of one type that are at the beginning of a document you can use the annotations list view, then sort it by start offset, select the rows to delete and right-click for the context menu to delete the selection. It will be much faster than selecting each annotation in the document editor and delete it.

For more information on how to create and edit annotations or search and annotate the document see section 3.19.

See also section 12.2.2 for the compound document editor.

5.3.1 The Annotation Sets View

The annotation sets view is displayed on the left part of the document editor. It's a tree-like view with a root for each annotation set. The first annotation set being the default one without name.

To display all the annotation of one type tick its checkbox or use Space key. To delete an annotation type use Delete key. To change the color use Enter key. There is a context menu for all these actions that you can display by right-clicking on one annotation type, a selection or an annotation set.

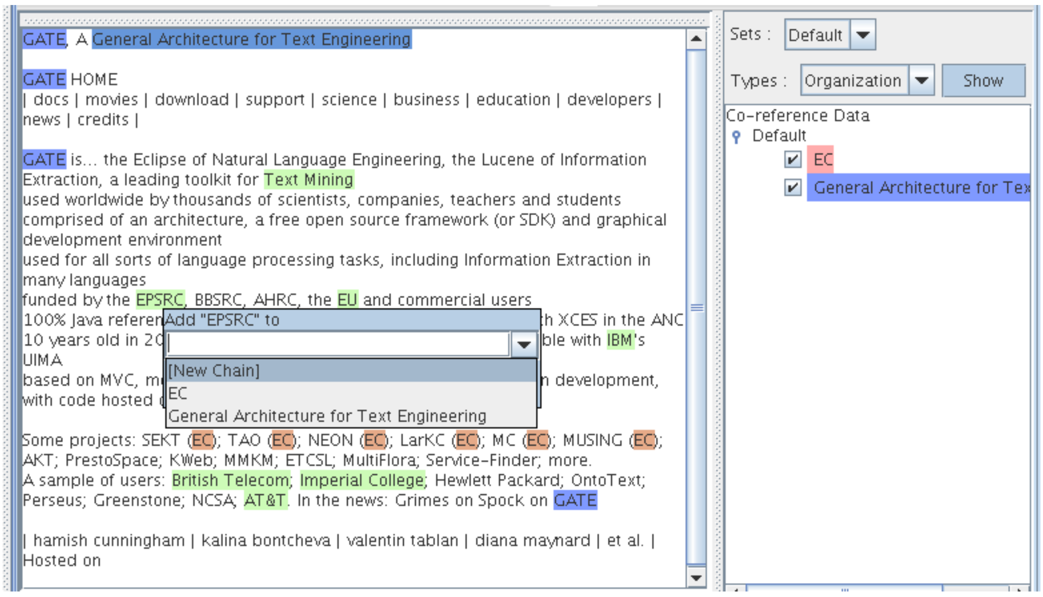
To create a new annotation set use the text field at the bottom and the 'New' button.

5.3.2 The Annotations List View

The annotations list view is displayed at the bottom of the document editor. It's a table of all the highlighted annotations in the document. You can sort the table by clicking on the headers and hide some column by right-clicking on the headers.

A context menu is available on the selected rows and allow to delete annotations or display one or more annotation editor.

5.3.3 The Co-reference Editor


GATE, A General Architecture for Text Engineering

GATE HOME
| docs | movies | download | support | science | business | education | developers |
news | credits |

GATE is... the Eclipse of Natural Language Engineering, the Lucene of Information
Extraction, a leading toolkit for Text Mining
used worldwide by thousands of scientists, companies, teachers and students
comprised of an architecture, a free open source framework (or SDK) and graphical
development environment
used for all sorts of language processing tasks, including Information Extraction in
many languages
funded by the EPSRC, BBSRC, AHRC, the EU and commercial users
100% Java referen h XCES in the ANC
10 years old in 20 ble with IBM's
UIMA
based on MVC, m n development,
with code hosted (

Add "EPSRC" to
[New Chain]
EC
General Architecture for Text Engineering

Some projects: SEKT (EC); TAO (EC); NEON (EC); LarKC (EC); MC (EC); MUSING (EC);
AKT; PrestoSpace; KWeb; MMKM; ETCSL; MultiFlora; Service-Finder; more.
A sample of users: British Telecom; Imperial College; Hewlett Packard; OntoText;
Perseus; Greenstone; NCSA; AT&T. In the news: Grimes on Spock on GATE

| hamish cunningham | kalina bontcheva | valentin tablan | diana maynard | et al. |
Hosted on

Sets : Default
Types : Organization
Co-reference Data
Default
✓ EC
✓ General Architecture for Tex

Figure 5.2: Co-reference editor inside a document editor. The popup window in the document under the word 'EPSRC' is used to add highlighted annotations to a co-reference chain. Here the annotation type 'Organization' of the annotation set 'Default' is highlighted and also the co-references 'EC' and 'GATE'.

The co-reference editor allows co-reference chains (see section 8.8) to be displayed and edited in the GATE GUI. To display the co-reference editor, first open a document in GATE, and then click on the `Co-reference Editor` button in the document viewer.

The combo box at the top of the co-reference editor allows you to choose which annotation set to display co-references for. If an annotation set contains no co-reference data, then the tree below the combo box will just show 'Coreference Data' and the name of the annotation set. However, when co-reference data does exist, a list of all the co-reference chains that are based on annotations in the currently selected set is displayed. The name of each co-reference chain in this list is the same as the text of whichever element in the chain is the longest. It is possible to highlight all the member annotations of any chain by selecting it in the list.

When a co-reference chain is selected, if the mouse is placed over one of its member annotations, then a pop-up box appears, giving the user the option of deleting the item from the chain. If the only item in a chain is deleted, then the chain itself will cease to exist, and it will be removed from the list of chains. If the name of the chain was derived from the item that was deleted, then the chain will be given a new name based on the next longest item in the chain.

A combo box near the top of the co-reference editor allows the user to select an annotation type from the current set. When the `Show` button is selected all the annotations of the selected type will be highlighted. Now when the mouse pointer is placed over one of those annotations, a pop-up box will appear giving the user the option of adding the annotation to a co-reference chain. The annotation can be added to an existing chain by typing the name of the chain (as shown in the list on the right) in the pop-up box. Alternatively, if the user presses the down cursor key, a list of all the existing annotations appears, together with the option `[New Chain]`. Selecting the `[New Chain]` option will cause a new chain to be created containing the selected annotation as its only element.

Each annotation can only be added to a single chain, but annotations of different types can be added to the same chain, and the same text can appear in more than one chain if it is referenced by two or more annotations.

Chapter 6

Language Resources: Corpora, Documents and Annotations

Sometimes in life you've got to dance like nobody's watching.

...

I think they should introduce 'sleeping' to the Olympics. It would be an excellent field event, in which the 'athletes' (for want of a better word) all lay down in beds, just beyond where the javelins land, and the first one to fall asleep and not wake up for three hours would win gold. I, for one, would be interested in seeing what kind of personality would be suited to sleeping in a competitive environment.

...

Life is a mystery to be lived, not a problem to be solved.

Round Ireland with a Fridge, Tony Hawks, 1998 (pp. 119, 147, 179).

This chapter documents GATE's model of corpora, documents and annotations on documents. Section 6.1 describes the simple attribute/value data model that corpora, documents and annotations all share. Section 6.2, section 6.3 and section 6.4 describe corpora, documents and annotations on documents respectively. Section 6.5 describes GATE's support for diverse document formats, and section 6.6 describes facilities for XML input/output.

6.1 Features: Simple Attribute/Value Data

GATE has a single model for information that describes documents, collections of documents (corpora), and annotations on documents, based on attribute/value pairs. Attribute names are strings; values can be any Java object. The API for accessing this feature data is Java's `Map` interface (part of the Collections API).

6.2 Corpora: Sets of Documents plus Features

A Corpus in GATE is a Java Set whose members are Documents. Both Corpora and Documents are types of LanguageResource (LR); all LRs have a FeatureMap (a Java Map) associated with them that stored attribute/value information about the resource. FeatureMaps are also used to associate arbitrary information with ranges of documents (e.g. pieces of text) via the annotation model (see below).

Documents have a DocumentContent which is a text at present (future versions may add support for audiovisual content) and one or more AnnotationSets which are Java Sets.

6.3 Documents: Content plus Annotations plus Features

Documents are modelled as content plus annotations (see section 6.4) plus features (see section 6.1). The content of a document can be any subclass of `DocumentContent`.

6.4 Annotations: Directed Acyclic Graphs

Annotations are organised in graphs, which are modelled as Java sets of Annotation. Annotations may be considered as the arcs in the graph; they have a start Node and an end Node, an ID, a type and a FeatureMap. Nodes have pointers into the sources document, e.g. character offsets.

6.4.1 Annotation Schemas

Annotation schemas provide a means to define types of annotations in GATE. GATE uses the XML Schema language supported by W3C for these definitions. When using the development environment to create/edit annotations, a component is available (`gate.gui.SchemaAnnotationEditor`) which is driven by an annotation schema file. This component will constrain the data entry process to ensure that only annotations that correspond to a particular schema are created. (Another component allows unrestricted annotations to be created.)

Schemas are resources just like other GATE components. Below we give some examples of such schemas. Section 3.21 describes how to create new schemas.

Date schema

```
<?xml version="1.0"?>
<schema
xmlns="http://www.w3.org/2000/10/XMLSchema">
 <!-- XSchema deffinition for Date-->
  <element name="Date">
    <complexType>
      <attribute name="kind"  use="optional">
        <simpleType>
          <restriction base="string">
            <enumeration value="date"/>
            <enumeration value="time"/>
            <enumeration value="dateTime"/>
          </restriction>
        </simpleType>
    </attribute>
  </complexType>
 </element>
</schema>
```

Person schema

```
<?xml version="1.0"?>
<schema
xmlns="http://www.w3.org/2000/10/XMLSchema">
    <!-- XSchema definition for Person-->
    <element name="Person" />
</schema>
```

Address schema

```
<?xml version="1.0"?> <schema
xmlns="http://www.w3.org/2000/10/XMLSchema">
    <!-- XSchema deffinition for Address-->
    <element name="Address">
      <complexType>
        <attribute name="kind"  use="optional">
          <simpleType>
            <restriction base="string">
              <enumeration value="email"/>
              <enumeration value="url"/>
              <enumeration value="phone"/>
              <enumeration value="ip"/>
              <enumeration value="street"/>
              <enumeration value="postcode"/>
              <enumeration value="country"/>
              <enumeration value="complete"/>
            </restriction>
        </simpleType>
    </attribute>
  </complexType>
</element>
</schema>
```

6.4.2 Examples of Annotated Documents

This section shows some simple examples of annotated documents.

This material is adapted from [Grishman 97], the TIPSTER Architecture Design document upon which GATE version 1 was based. Version 2 has a similar model, although annotations are now graphs, and instead of multiple spans per annotation each annotation now has a single start/end node pair. The current model is largely compatible with [Bird & Liberman 99], and roughly isomorphic with "stand-off markup" as latterly adopted by the SGML/XML community.

Each example is shown in the form of a table. At the top of the table is the document being annotated; immediately below the line with the document is a ruler showing the position (byte offset) of each character (see TIPSTER Architecture Design Document).

Underneath this appear the annotations, one annotation per line. For each annotation is shown its Id, Type, Span (start/end offsets derived from the start/end nodes), and Features. Integers are used as the annotation Ids. The features are shown in the form name = value.

The first example shows a single sentence and the result of three annotation procedures: tokenization with part-of-speech assignment, name recognition, and sentence boundary recognition. Each token has a single feature, its part of speech (pos), using the tag set from the University of Pennsylvania Tree Bank; each name also has a single feature, indicating the type of name: person, company, etc.

Annotations will typically be organized to describe a hierarchical decomposition of a text. A simple illustration would be the decomposition of a sentence into tokens. A more complex case would be a full syntactic analysis, in which a sentence is decomposed into a noun phrase and a verb phrase, a verb phrase into a verb and its complement, etc. down to the level of individual tokens. Such decompositions can be represented by annotations on nested sets of spans. Both of these are illustrated in the second example, which is an elaboration of our first example to include parse information. Each non-terminal node in the parse tree is represented by an annotation of type parse.

Text				
`Cyndi savored the soup.`				
`^0...^5...^10..^15..^20`				
Annotations				
Id	Type	SpanStart	Span End	Features
1	token	0	5	pos=NP
2	token	6	13	pos=VBD
3	token	14	17	pos=DT
4	token	18	22	pos=NN
5	token	22	23	
6	name	0	5	name_type=person
7	sentence	0	23	

Table 6.1: Result of annotation on a single sentence

Text				
`Cyndi savored the soup.`				
`^0...^5...^10..^15..^20`				
Annotations				
Id	Type	SpanStart	Span End	Features
1	token	0	5	pos=NP
2	token	6	13	pos=VBD
3	token	14	17	pos=DT
4	token	18	22	pos=NN
5	token	22	23	
6	name	0	5	name_type=person
7	sentence	0	23	constituents=[1],[2],[3].[4],[5]

Table 6.2: Result of annotations including parse information

Text				
`To:  All Barnyard Animals`				
`^0...^5...^10..^15..^20.`				
`From:  Chicken Little`				
`^25..^30..^35..^40..`				
`Date:  November 10,1194`				
`...^50..^55..^60..^65.`				
`Subject:  Descending Firmament`				
`.^70..^75..^80..^85..^90..^95`				
`Priority:  Urgent`				
`.^100.^105.^110.`				
`The sky is falling.  The sky is falling.`				
`....^120.^125.^130.^135.^140.^145.^150.`				
Annotations				
Id	Type	SpanStart	Span End	Features
1	Addressee	4	24	
2	Source	31	45	
3	Date	53	69	ddmmyy=101194
4	Subject	78	98	
5	Priority	109	115	
6	Body	116	155	
7	Sentence	116	135	
8	Sentence	136	155	

Table 6.3: Annotation showing overall document structure

In most cases, the hierarchical structure could be recovered from the spans. However, it may be desirable to record this structure directly through a constituents feature whose value is a sequence of annotations representing the immediate constituents of the initial annotation. For the annotations of type parse, the constituents are either non-terminals (other annotations in the parse group) or tokens. For the sentence annotation, the constituents feature points to the constituent tokens. A reference to another annotation is represented in the table as "[Annotation Id]"; for example, "[3]" represents a reference to annotation 3. Where the value of an feature is a sequence ofitems, these items are separated by commas. No special operations are provided in the current architecture for manipulating constituents. At a less esoteric level, annotations can be used to record the overall structure of documents, including in particular documents which have structured headers, as is shown in the third example (Table 6.3).

If the Addressee, Source, ... annotations are recorded when the document is indexed for retrieval, it will be possible to perform retrieval selectively on information in particular fields. Our final example (Table 6.4) involves an annotation which effectively modifies the document. The current architecture does not make any specific provision for the modification

Text				
`Topster tackles 2 terrorbytes.`				
`^0...^5...^10..^15..^20..^25..`				
Annotations				
Id	Type	SpanStart	Span End	Features
1	token	0	7	pos=NP correction=TIPSTER
2	token	8	15	pos=VBZ
3	token	16	17	pos=CD
4	token	18	29	pos=NNS correction=terabytes
5	token	29	30	

Table 6.4: Annotation modifying the document

of the original text. However, some allowance must be made for processes such as spelling correction. This information will be recorded as a correction feature on token annotations and possibly on name annotations:

6.4.3 Creating, Viewing and Editing Diverse Annotation Types

Note that annotation types should consist of a single word with no spaces. Otherwise they may not be recognised by other components such as JAPE transducers, and may create problems when annotations are saved as inline (save preserving format).

To view and edit annotation types, see Section 3.16. To add annotations of a new type, see Section 3.19. To add a new annotation schema, see Section 3.21.

6.5 Document Formats

The following document formats are supported by GATE:

- Plain Text
- HTML
- SGML
- XML
- RTF
- Email

- PDF (some documents)
- Microsoft Word (some documents)

By default GATE will try and identify the type of the document, then strip and convert any markup into GATE's annotation format. To disable this process, set the `markupAware` parameter on the document to `false`.

When reading a document of one of these types, GATE extracts the text between tags (where such exist) and create a GATE annotation filled as follows:

The name of the tag will constitute the annotation's type, all the tags attributes will materialize in the annotation's features and the annotation will span over the text covered by the tag. A few exceptions of this rule apply for the RTF, Email and Plain Text formats, which will be described later in the input section of these formats.

The text between tags is extracted and appended to the GATE document's content and all annotations created from tags will be placed into a GATE annotation set named "Original markups".

Example:

If the markup is like this:

```
<aTagName attrib1="value1" attrib2="value2" attrib3="value3"> A
piece of text</aTagName>
```

then the annotation created by GATE will look like:

```
annotation.type = "aTagName";
annotation.fm={attrib1=value1;atrtrib2=value2;attrib3=value3};
annotation.start=startNode;
annotation.end = endNode;
```

The startNode and endNode are created from offsets refereing the beginning and the end of "A piece of text" in the document's content.

The documents supported by GATE have to be in one of the encodings accepted by Java. The most popular is the *"UTF-8"* encoding which is also the most storage efficient one for UNICODE. If, when loading a document in GATE the *encoding* parameter is set to ""(the empty string), then the default encoding of the platform will be used.

6.5.1 Detecting the right reader

In order to successfully apply the document creation algorithm described above, GATE needs to detect the proper reader to use for each document format. If the user knows in advance what kind of document they are loading then they can specify the MIME type (e.g. *text/html*) using the init parameter `mimeType`, and GATE will respect this. If an explicit type is not given, GATE attempts to determine the type by other means, taking into consideration (where possible) the information provided by three sources:

- Document's extension
- The web server's content type
- Magic numbers detection

The first represents the extension of a file like (*xml,htm,html,txt,sgm,rtf, etc*), the second represents the HTTP information sent by a web server regarding the content type of the document being send by it (*text/html; text/xml, etc*), and the third one represents certain sequences of chars which are ultimately number sequences. GATE is capable to support multimedia documents, if the right reader is added to the framework. Sometimes, multimedia documents are identified by a signature consisting in a sequence of numbers. Inside GATE they are called magic numbers. For textual documents, certain char sequences form such magic numbers. Examples of magic numbers sequences will be provided in the Input section of each format supported by GATE.

All those tests are applied to each document read, and after that, a voting mechanism decides what is the best reader to associate with the document. There is a degree of priority for all those tests. The document's extension test has the highest priority. If the system is in doubt which reader to choose, then the one associated with document's extension will be selected. The next higher priority is given to the web server's content type and the third one is given to the magic numbers detection. However, any two tests that identify the same mime type, will have the highest priority in deciding the reader that will be used. The web server test is not always successful as there might be documents that are loaded from a local file system, and the magic number detection test is not always applicable. In the next paragraphs we will se how those tests are performed and what is the general mechanism behind reader detection.

The method that detects the proper reader is a static one, and it belongs to the `gate.DocumentFormat` class. It uses the information stored in the maps filled by the init() method of each reader. This method comes with three signatures:

```
static public DocumentFormat getDocumentFormat( gate.Document
aGateDocument, URL url)
```

```
static public DocumentFormat getDocumentFormat(gate.Document
aGateDocument, String fileSuffix)

static public DocumentFormat getDocumentFormat(gate.Document
aGateDocument, MimeType mimeType)
```

The first two methods try to detect the right MimeType for the GATE document, and after that, they call the third one to return the reader associate with a MimeType. Of course, if an explicit `mimeType` parameter was specified, GATE calls the third form of the method directly, passing the specified type. GATE uses the implementation from "http://jigsaw.w3.org" for mime types.

The magic numbers test is performed using the information form magic2mimeTypeMap map. Each key from this map, is searched in the first bufferSize (the default value is 2048) chars of text. The method that does this is called `runMagicNumbers(InputStreamReader aReader)` and it belongs to DocumentFormat class. More details about it can be found in the GATE API documentation.

In order to activate a reader to perform the unpacking, the creole definition of a GATE document defines a parameter called "markupAware" initialized with a default value of **true**. This parameter, forces GATE to detect a proper reader for the document being read. If no reader is found, the document's content is load and presented to the user, just like any other text editor (this for textual documents).

The next subsections investigates particularities for each format and will describe the file extensions registered with each document format.

6.5.2 XML

Input

GATE permits the processing of any XML document and offers support for XML namespaces. It benefits the power of Apache's Xerces parser and also makes use of Sun's JAXP layer. Changing the XML parser in GATE can be achieved by simply replacing the value of a Java system property ("javax.xml.parsers.SAXParserFactory").

GATE will accept any well formed XML document as input. Although it has the possibility to validate XML documents against DTDs it does not do so because the validating procedure is time consuming and in many cases it issues messages that are annoying for the user.

There is an open problem with the general approach of reading XML, HTML and SGML documents in GATE. As we previously said, the text covered by tags/elements is appended to the GATE document content and a GATE annotation refers to this particular span of text.

When appending, in cases such as "`end.</P><P>Start`" it might happen to concatenate the ending word of the previous annotation with the beginning phrase of the annotation currently being created, resulting in a garbage input for GATE processing resources that operate at the text surface.

Let's take another example in order to better understand the problem :

```
<title>This is a title</title><p>This is a paragraph</p><a
href="#link">Here is an useful link</a>
```

When the markup is transformed to annotations, it is likely that the text from the document's content will be as follows:

```
This is a titleThis is a paragraphHere is an useful link
```

The annotations created will refer the right parts of the texts but for the GATE's processing resources like (tokenizer, gazetter, etc) which work on this text, this will be a major diaster. Therefore, in order to prevent this problem from happening, GATE checks if it's likely to join words and if this happens then it inserts a space between those words. So, the text will look like this after loaded in GATE:

```
This is a title This is a paragraph Here is an useful link
```

There are cases when these words are meant to be joined, but they are just a few. This is why it's an open problem.

The extensions associate with the XML reader are:

- xml
- xhtm
- xhtml

The web server content type associate with xml documents is: *text/xml.*

The magic numbers test searches inside the document for the XML(`<?xml version="1.0"`) signature. It is also able to detect if the XML document uses the semantic described in the GATE document format DTD (see section 6.5.2) or uses other semantics.

Output

GATE is capable to assure persistence for its resources. These layers of persistence are various and they span until database persistence. However, for some purposes, a light and simple level of persistence would be highly appreciated. The types of persistent storage used for Language Resources are:

- Databases (like Oracle);
- Java serialization;
- XML serialization.

We describe the latter case in here.

XML persistence doesn't necessarily preserve all the objects belonging to the annotations, documents or corpora. Their features can be of all kinds of objects, with various layers of nesting. For example, *lists containing lists containing maps, etc.* Serializing these arbitrary data types in XML is not a simple task; GATE does the best it can, and supports native Java types such as Integers and Booleans, but where complex data types are used, information may be lost(the types will be converted into Strings). GATE provides a full serialization of certain types of features such as collections, strings and numbers. It is possible to serialize only those collections containing strings or numbers. The rest of other features are serialized using their string representation and when read back, they will be all strings instead of being the original objects. Consequences of this might be observed when performing evaluations(see the evaluation section).

When GATE outputs an XML document it may do so in one of two ways:

- When the original document that was imported into GATE was an XML document, GATE can dump that document back into XML (possibly with additional markup added);
- For all document formats, GATE can dump its internal representation of the document into XML.

In the former case, the XML output will be close to the original document. In the latter case, the format is a GATE-specific one which can be read back by the system to recreate all the information that GATE held internally for the document.

In order to understand why there are two types of XML serialization, one needs to understand the structure of a GATE document. GATE allows a graph of annotations that refer to parts of the text. Those annotations are grouped under annotation sets. Because of this structure, sometimes it is impossible to save a document as XML using tags that surround the text referred by the annotation, because tags crossover situations could appear (XML is essentially a tree-based model of information, whereas GATE uses graphs). Therefore, in order to preserve all annotations in a GATE document, a custom type of XML document was developed.

The problem of crossover tags appears with GATE's second option (the preserve format one), which is implemented at the cost of loosing certain annotations. The way it is applied in GATE is that it tries to restore the original markup and where it is possible, to add in the same manner annotations produced by GATE.

How to access and make use of the two ways of XML serialization

Save As XML option

This option is available in GATE's GUI in the pop up menu associate with each language resource (document or corpus). Saving a corpus as XML is done by calling save as XML on each document of the corpus. This option saves all the annotations of a document together their features(applying the restrictions previously discussed), using the GateDocument.dtd :

```
<!ELEMENT GateDocument (GateDocumentFeatures,
          TextWithNodes, (AnnotationSet+))>
<!ELEMENT GateDocumentFeatures (Feature+)>
<!ELEMENT Feature (Name, Value)>
<!ELEMENT Name (\#PCDATA)>
<!ELEMENT Value (\#PCDATA)>
<!ELEMENT TextWithNodes (\#PCDATA | Node)*>
<!ELEMENT AnnotationSet (Annotation*)>
<!ATTLIST AnnotationSet  Name CDATA \#IMPLIED>
<!ELEMENT Annotation (Feature*)>
<!ATTLIST Annotation  Type      CDATA \#REQUIRED
                      StartNode CDATA \#REQUIRED
                      EndNode   CDATA \#REQUIRED>
<!ELEMENT Node EMPTY>
<!ATTLIST Node id CDATA \#REQUIRED>
```

The document is saved under a name chosen by the user and it may have any extension. However, the recommended extension would be "xml".

Using GATE's API, this option is available by calling `gate.Document's toXml()` method. This method returns a string which is the XML representation of the document on which the method was called.

Note: It is recommended that the string representation to be saved on the file system using the UTF-8 encoding, as the first line of the string is : `<?xml version="1.0" encoding="UTF-8"?>`

Example of such a GATE format document:

```
<?xml version="1.0" encoding="UTF-8" ?>
<GateDocument>

<!-- The =document's features-->

<GateDocumentFeatures>
<Feature>
  <Name className="java.lang.String">MimeType</Name>
  <Value className="java.lang.String">text/plain</Value>
</Feature>
<Feature>
  <Name className="java.lang.String">gate.SourceURL</Name>
  <Value className="java.lang.String">file:/G:/tmp/example.txt</Value>
</Feature>
</GateDocumentFeatures>

<!-- The document content area with serialized nodes -->

<TextWithNodes>
<Node id="0"/>A TEENAGER <Node
id="11"/>yesterday<Node id="20"/> accused his parents of cruelty
by feeding him a daily diet of chips which sent his weight
ballooning to 22st at the age of 12<Node id="146"/>.<Node
id="147"/>
</TextWithNodes>

<!-- The default annotation set -->

<AnnotationSet>
<Annotation Type="Date" StartNode="11"
EndNode="20">
<Feature>
  <Name className="java.lang.String">rule2</Name>
  <Value className="java.lang.String">DateOnlyFinal</Value>
</Feature> <Feature>
  <Name className="java.lang.String">rule1</Name>
  <Value className="java.lang.String">GazDateWords</Value>
</Feature> <Feature>
  <Name className="java.lang.String">kind</Name>
  <Value className="java.lang.String">date</Value>
</Feature> </Annotation> <Annotation Type="Sentence" StartNode="0"
EndNode="147"> </Annotation> <Annotation Type="Split"
StartNode="146" EndNode="147"> <Feature>
  <Name className="java.lang.String">kind</Name>
  <Value className="java.lang.String">internal</Value>
</Feature> </Annotation> <Annotation Type="Lookup" StartNode="11"
EndNode="20"> <Feature>
  <Name className="java.lang.String">majorType</Name>
  <Value className="java.lang.String">date_key</Value>
</Feature> </Annotation>
</AnnotationSet>

<!-- Named annotation set -->

<AnnotationSet Name="Original markups" >
```

```
 <Annotation
Type="paragraph" StartNode="0" EndNode="147"> </Annotation>
</AnnotationSet>
</GateDocument>
```

Note: One must know that all features that are not collections containing numbers or strings or that are not numbers or strings are discarded. With this option, GATE does not preserve those features it cannot restore back.

The preserve format option

This option is available in the GATE GUI from the popup menu of the annotations table. If no annotation in this table is selected, then the option will restore the document's original markup. If certain annotations are selected, then the option will attempt to restore the original markup and insert all the selected ones. When an annotation violates the crossed over condition, that annotation is discarded and a message is issued by GATE.

This option makes possible to generate an XML document with tags surrounding the annotation's refereed text and feature saved as attributes. All features which are collections, strings or numbers are saved, and the others are discarded. However, when read back, only the attributes under the GATE namespace (see bellow) are reconstructed back different than the others. That is because GATE does not store in the XML document the information about the features class and for collections the class of the items. So, when read back all features will become strings, except those under the GATE namespace.

One will notice that all generated tags have an attribute called "gateId" under the namespace "http://www.gate.ac.uk". The attribute is used when the document is read back in GATE, in order to restore the annotation's old ID. This feature is needed because it works in close cooperation with another attribute under the same namespace, called "matches". This attribute indicates annotations/tags that refer the same entity[1]. They are under this namespace because GATE is sensitive to them and treats them differently then all other elements with their attributes which falls under the general reading algorithm described at the beginning of this section.

The "gateId" under GATE namespace is used to create an annotation which have as ID, the value indicated by this attribute. The "matches" attribute is used to create an ArrayList in which the items will be Integers, representing the ID of annotations that the current one matches.

Example:

If the text being processed is as follows:

```
<Person gate:gateId="23">John</Person> and <Person
```

[1]It's not an XML entity but a information extraction named entity

```
gate:gateId="25" gate:matches="23;25;30">John Major</Person> are
the same person.
```

What GATE does when parses this text, is to create two annotations:

```
a1.type = "Person"
a1.ID=Integer(23)
a1.start=<the start offset of
John>
a1.end = <the end offset of John>
a1.featureMap = {}

a2.type="Person"
a2.ID = Integer(25)
a2.start= <the start offset
of John Major>
a2.end = <the end offset of John Major>
a2.featureMap ={matches=[Integer(23); Integer(25); Integer(30)]}
```

Under GATE's API, this option is available by calling `gate.Document's toXml(Set aSetContainingAnnotations)` method. This method returns a string which is the XML representation of the document on which the method was called. If called with **null** as a parameter, then the method will attempt to restore only the original markup. If the parameter is a set that contains annotations, then each annotation is tested against the crossover restriction, and for those found to violate it, a warning will be issued and they will be discarded.

In the next subsections we will show how this options applies to the other formats supported by GATE.

6.5.3 HTML

Input

HTML documents are parsed by GATE using the NekoHTML parser. The documents are read and created in GATE the same way as the XML documents.

The extensions associate with the HTML reader are:

- htm
- html

The web server content type associate with html documents is: *text/html*.

The magic numbers test searches inside the document for the HTML(`<html`) signature.There are certain HTML documents that do not contain the HTML tag, so the magical numbers test might not hold.

There is a certain degree of customization for HTML documents in that GATE introduces new lines into the document's text content in order to obtain a readable form. The annotations will refer the pieces of text as described in the original document but there will be a few extra new line characters inserted.

After reading H1,H2,H3,H4,H5,H6,TR,CENTER,LI,BR and DIV tags, GATE will introduce a new line(NL) char into the text. After a TITLE tag it will introduce two NLs. With P tags, GATE will introduce one NL at the beginning of the paragraph and one at the end of the paragraph. All newly added NLs are not considered to be part of the text contained by the tag.

Output

The Save as XML option works exactly the same for all GATE's documents so there is no particular observation to be made for the HTML formats.

When attempting to preserve the original markup formatting, GATE will generate the document in xhtml. The html document will look the same with any browser after processed by GATE but it will be in another syntax.

6.5.4 SGML

Input

The SGML support in GATE is fairly light as there is no freely available Java SGML parser. GATE uses a light converter attempting to transform the input SGML file into a well formed XML. Because it does not make use of a DTD, the conversion might not be always good. It is advisable to perform a SGML2XML conversion outside the system(using some other specialized tools) before using the SGML document inside GATE.

The extensions associate with the SGML reader are:

- sgm
- sgml

The web server content type associate with xml documents is : *text/sgml*.

There is no magic numbers test for SGML.

Output

When attempting to preserve the original markup formatting, GATE will generate the document as XML because the real input of a SGML document inside GATE is an XML one.

6.5.5 Plain text

Input

When reading a plain text document, GATE attempts to detect its paragraphs and add "paragraph" annotations to the document's "Original markups" annotation set. It does that by detecting two consecutive NLs. The procedure works for both UNIX like or DOS like text files.

Example:

If the plain text read is as follows:

```
Paragraph 1. This text belongs to the first paragraph.

Paragraph 2. This text belongs to the second paragraph
```

then two "paragraph" type annotation will be created in the "Original markups" annotation set (refereing the first and second paragraphs) with an empty feature map.

The extensions associate with the plain text reader are:

- txt
- text

The web server content type associate with plain text documents is: *text/plain.*

There is no magic numbers test for plain text.

Output

When attempting to preserve the original markup formatting, GATE will dump XML markup that surrounds the text refereed.

The procedure described above applies both for plain text and RTF documents.

6.5.6 RTF

Input

Accessing RTF documents is performed by using the Java's RTF editor kit. It only extracts the document's text content from the RTF document.

The extension associate with the RTF reader is *"rtf"*.

The web server content type associate with xml documents is : *text/rtf*.

The magic numbers test searches for {\\`rtf1`.

Output

Same as the plain tex output.

6.5.7 Email

Input

GATE is able to read email messages packed in one document (UNIX mailbox format). It detects multiple messages inside such documents and for each message it creates annotations for all the fields composing an e-mail, like date, from, to, subject, etc. The message's body is analyzed and a paragraph detection is performed (just like in the plain text case) . All annotation created have as type the name of the e-mail's fields and they are placed in the Original markup annotation set.

Example:

```
From someone@zzz.zzz.zzz Wed Sep  6 10:35:50 2000

Date: Wed, 6 Sep2000 10:35:49 +0100 (BST)

From: forename1 surname2 <someone1@yyy.yyy.xxx>

To: forename2 surname2 <someone2@ddd.dddd.dd.dd>

Subject: A subject
```

```
Message-ID: <Pine.SOL.3.91.1000906103251.26010A-100000@servername>
MIME-Version: 1.0
Content-Type: TEXT/PLAIN; charset=US-ASCII

This text belongs to the e-mail body....

This is a paragraph in the body of the e-mail

This is another paragraph.
```

GATE attempts to detect lines such "*From someone@zzz.zzz.zzz Wed Sep 6 10:35:50 2000*" in the e-mail text. Those lines separate e-mail messages contained in one file. After that, for each field in the e-mail message annotation are created as follows:

The annotation type will be the name of the field, the feature map will be empty and the annotation will span from the end of the filed until the end of the line containing the e-mail field.

Example:

```
a1.type = "date" a1 spans between the two ^ ^. Date:^ Wed,
6Sep2000 10:35:49 +0100 (BST)^

a2.type = "from"; a2 spans between the two ^ ^. From:^ forename1
surname2 <someone1@yyy.yyy.xxx>^
```

The extensions associate with the email reader are:

- eml
- email
- mail

The web server content type associate with plain text documents is: *text/email.*

The magic numbers test searches for keywords like *Subject:*,etc.

Output

Same as plain text output.

6.6 XML Input/Output

Support for input from and output to XML is described in section 6.5.2. In short:

- GATE will read any well-formed XML document (it does not attempt to validate XML documents). Markup will by default be converted into native GATE format.
- GATE will write back into XML in one of two ways:
    1. Preserving the original format and adding selected markup (for example to add the results of some language analysis process to the document).
    2. In GATE's own XML serialisation format, which encodes all the data in a GATE Document (as far as this is possible within a tree-structured paradigm – for 100% non-lossy data storage use GATE's RDBMS or binary serialisation facilities – see section 4.7).

When using the GATE framework, object representations of XML documents such as `DOM` or `jDOM`, or query and transformation languages such as `X-Path` or `XSLT`, may be used in parallel with GATE's own Document representation (`gate.Document`) without conflicts.

Chapter 7

JAPE: Regular Expressions Over Annotations

If Osama bin Laden did not exist, it would be necessary to invent him. For the past four years, his name has been invoked whenever a US president has sought to increase the defence budget or wriggle out of arms control treaties. He has been used to justify even President Bush's missile defence programme, though neither he nor his associates are known to possess anything approaching ballistic missile technology. Now he has become the personification of evil required to launch a crusade for good: the face behind the faceless terror.

The closer you look, the weaker the case against Bin Laden becomes. While the terrorists who inflicted Tuesday's dreadful wound may have been inspired by him, there is, as yet, no evidence that they were instructed by him. Bin Laden's presumed guilt appears to rest on the supposition that he is the sort of man who would have done it. But his culpability is irrelevant: his usefulness to western governments lies in his power to terrify. When billions of pounds of military spending are at stake, rogue states and terrorist warlords become assets precisely because they are liabilities.

The need for dissent, George Monbiot, The Guardian, Tuesday September 18, 2001.

This chapter describes JAPE – a Java Annotation Patterns Engine. JAPE provides finite state transduction over annotations based on regular expressions. JAPE is a version of CPSL – Common Pattern Specification Language[1].

JAPE allows you to recognise regular expressions in annotations on documents. Hang on, there's something wrong here: a regular language can only describe sets of strings, not graphs,

[1]A good description of the original version of this language is in `Doug Appelt's TextPro manual`. Doug was a great help to us in implementing JAPE. Thanks Doug!

and GATE's model of annotations is based on graphs. Hmmm. Another way of saying this: typically, regular expressions are applied to character strings, a simple linear sequence of items, but here we are applying them to a much more complex data structure. The result is that in certain cases the matching process in non-deterministic (i.e. the results are dependent on random factors like the addresses at which data is stored in the virtual machine): when there is structure in the graph being matched that requires more than the power of a regular automaton to recognise, JAPE chooses an alternative arbitrarily. However, this is not the bad news that it seems to be, as it turns out that in many useful cases the data stored in annotation graphs in GATE (and other language processing systems) can be regarded as simple sequences, and matched deterministically with regular expressions.

A JAPE grammar consists of a set of phases, each of which consists of a set of pattern/action rules. The phases run sequentially and constitute a cascade of finite state transducers over annotations. The left-hand-side (LHS) of the rules consist of an annotation pattern that may contain regular expression operators (e.g. `*, ?, +`). The right-hand-side (RHS) consists of annotation manipulation statements. Annotations matched on the LHS of a rule may be referred to on the RHS by means of labels that are attached to pattern elements.

At the beginning of each grammar, several options can be set:

- Control - this defines the method of rule matching (see Section 7.3)
- Debug - when set to true, if the grammar is running in Appelt mode and there is more than one possible match, the conflicts will be displayed on the standard output. See also Section 7.5.

Input annotations must also be defined at the start of each grammar. If no annotations are defined, all annotations will be matched.

There are 3 main ways in which the pattern can be specified:

- specify a string of text, e.g. `{Token.string == "of"}`
- specify the presence or absence of an annotation previously assigned from a gazetteer, tokeniser, or other module, e.g. `{Lookup}` (matches a Lookup annotation) or `{!Lookup}` (matches if there is *not* a Lookup annotation at this location).
- specify the attributes (and values) of an annotation. Several operators are supported - see section 7.1 for full details:
  - `{Token.kind == "number"}`, `{Token.length != 4}` - equality and inequality.
  - `{Token.string > "aardvark"}`, `{Token.length < 10}` - comparison operators. >= and <= are also supported.
  - `{Token.string =~ "[Dd]ogs"}`, `{Token.string !~ "(?i)hello"}` - regular expression. `==~` and `!=~` are also provided, for whole-string matching.

  – `{X contains Y}` and `{X within Y}` for checking annotations within the context of other annotations.

Macros can also be used in the LHS of rules. This means that instead of expressing the information in the rule, it is specified in a macro, which can then be called in the rule. The reason for this is simply to avoid having to repeat the same information in several rules. Macros can themselves be used inside other macros.

New as of September 2008, in addition to referencing annotation features, JAPE allows access to other "meta-properties" of an annotation. This is done by using an "@" symbol rather than a "." symbol after the annotation type name. The three meta-properties that are built in are:

- length - returns the spanning length of the annotation.
- string - returns the string spanned by the annotation in the document.
- cleanString - Like string, but with extra white space stripped out. (i.e. "\s+" goes to a single space and leading or trailing white space is removed).

At this time, you cannot access the value of a "meta-property" from a non-java RHS of a rule. (e.g. You can't write: `{X@length > "5"}:label-->:label.New = {somefeat = :label.X@length`
We hope to add this at some point.

The same union and kleen operators can be used as for the tokeniser rules, i.e.

```
|
*
?
+
```

New as of late-September 2008, a range notation can also be added. e.g.

```
({Token})[1,3]
```

matches one to three Tokens in a row.

```
({Token.kind == number})[3]
```

matches exactly 3 number Tokens in a row.

The pattern description is followed by a label for the annotation. A label is denoted by a preceding colon; in the example below, the label is `:location`.

The RHS of the rule contains information about the annotation. Information about the annotation is transferred from the LHS of the rule using the label just described, and annotated with the entity type (which follows it). Finally, attributes and their corresponding values are added to the annotation. Alternatively, the RHS of the rule can contain Java code to create or manipulate annotations.

In the simple example below, the pattern described will be awarded an annotation of type "Enamex" (because it is an entity name). This annotation will have the attribute "kind", with value "location", and the attribute "rule", with value "GazLocation". (The purpose of the "rule" attribute is simply to ease the process of manual rule validation).

```
Rule: GazLocation
(
{Lookup.majorType == location}
)
:location -->
 :location.Enamex = {kind="location", rule=GazLocation}
```

It is also possible to have more than one pattern and corresponding action, as shown in the rule below. On the LHS, each pattern is enclosed in a set of round brackets and has a unique label; on the RHS, each lable is associated with an action. In this example, the Lookup annotation is labelled "jobtitle" and is given the new annotation JobTitle; the TempPerson annotation is labelled "person" and is given the new annotation "Person".

```
Rule: PersonJobTitle
Priority: 20

(
 {Lookup.majorType == jobtitle}
):jobtitle
(
 {TempPerson}
):person
-->
    :jobtitle.JobTitle = {rule = "PersonJobTitle"},
    :person.Person = {kind = "personName", rule = "PersonJobTitle"}
```

Similarly, labelled patterns can be nested, as in the example below, where the whole pattern is annotated as Person, but within the pattern, the jobtitle is annotated as JobTitle.

```
Rule: PersonJobTitle2
Priority: 20
```

```
(
(
 {Lookup.majorType == jobtitle}
):jobtitle
 {TempPerson}
):person
-->
    :jobtitle.JobTitle = {rule = "PersonJobTitle"},
    :person.Person = {kind = "personName", rule = "PersonJobTitle"}
```

JAPE provides limited support for copying annotation feature values from the left to the right hand side of a rule, for example:

```
Rule: LocationType

(
 {Lookup.majorType == location}
):loc
-->
    :loc.Location = {rule = "LocationType", type = :loc.Lookup.minorType}
```

This will set the "type" feature of the generated location to the value of the "minorType" feature from the "Lookup" annotation bound to the loc label. If the Lookup has no minorType, the Location will have no "type" feature. The behaviour of `newFeat = :bind.Type.oldFeat` is:

- Find all the annotations of type `Type` from the left hand side binding `bind`.
- Find one of them that has a non-`null` value for its `oldFeat` feature (if there is more than one, which one is chosen is up to the JAPE implementation).
- If such a value exists, set the `newFeat` feature of our newly created annotation to this value.
- If no such non-`null` value exists, do not set the `newFeat` feature at all.

Notice that the behaviour is *deliberately underspecified* if there is more than one `Type` annotation in `bind`. If you need more control, or if you want to copy several feature values from the same left hand side annotation, you should consider using Java code on the right hand side of your rule (see section 7.7).

Grammar rules can essentially be of two types. The first type of rule involves no gazetteer lookup, but can be defined using a small set of possible formats. In general, these are fairly straightforward and offer little potential for ambiguity.

The second type of rules rely more heavily on the gazetteer lists, and cover a much wider range of possibilities. This not only means that many rules may be needed to describe all situations, but that there is a much greater potential for ambiguity. This leads to the necessity for rule ordering and prioritisation, as will be discussed below.

For example, a single rule is sufficient to identify an IP address, because there is only one basic format - a series of numbers, each set connected by a dot. The rule for this is given below[2]:

```
Rule: IPAddress
(
 {Token.kind == number}
 {Token.string == "."}
 {Token.kind == number}
 {Token.string == "."}
 {Token.kind == number}
 {Token.string == "."}
 {Token.kind == number}
)
:ipAddress -->
 :ipAddress.Address = {kind = "ipAddress"}
```

To identify a date or time, there are many possible variations, and so many rules are needed. For example, the same date information can appear in the following formats (amongst others):

```
 Wed, 10/7/00
 Wed, 10/July/00
 Wed, 10 July, 2000
 Wed 10th of July, 2000
 Wed. July 10th, 2000
 Wed 10 July 2000
```

Different types of date can also be expressed. For example, the following would also be classified as date entities:

```
 the late '80s
 Monday
 St. Andrew's Day
 99 BC
```

[2]We might be more specific and state the possible lengths of the number, but within the confines of this project we currently have no need to, because there is no ambiguity with anything else

```
 mid-November
 1980-81
 from March to April
```

This also means there is a much greater potential for ambiguity. For example, many of the months of the year can also be girls' Christian names (e.g. May, June). This means that contextual information may be needed to disambiguate them, or we may have to guess which is more likely, based on frequency. For example, while "Friday" could be a person's name (as in "Man Friday"), it is much more likely to be a day of the week.

Finally, macros can also be used on the RHS of rules. In this case, the label (which matches the label on the LHS of the rule) should be included in the macro. Below we give an example of using a macro on the RHS.

```
Macro: UNDERSCORES_OKAY          // separate
:match                                               // lines
{
    gate.AnnotationSet matchedAnns = (gate.AnnotationSet)bindings.get("match");

    int begOffset = matchedAnns.firstNode().getOffset().intValue();
    int endOffset = matchedAnns.lastNode().getOffset().intValue();
    String mydocContent = doc.getContent().toString();
    String matchedString = mydocContent.substring(begOffset, endOffset);

    gate.FeatureMap newFeatures = Factory.newFeatureMap();

    if(matchedString.equals("Spanish"))     {
     newFeatures.put("myrule",  "Lower");
    }
    else    {
     newFeatures.put("myrule",  "Upper");
    }

    newFeatures.put("quality",  "1");
    annotations.add(matchedAnns.firstNode(), matchedAnns.lastNode(),
                              "Spanish_mark", newFeatures);
}

Rule: Lower
(
    ({Token.string == "Spanish"})
:match)-->UNDERSCORES_OKAY   // no label here, only macro name

Rule: Upper
```

```
(
    ({Token.string == "SPANISH"})
:match)-->UNDERSCORES_OKAY   // no label here, only macro name
```

7.1 Matching operators in detail

This section gives more detail on the behaviour of the matching operators used on the left-hand side of JAPE rules.

7.1.1 Equality operators ("==" and "!=")

The basic operator in JAPE is equality. `{Lookup.majorType == "person"}` matches a Lookup annotation whose majorType feature has the value "person". Similarly `{Lookup.majorType != "person"}` would match any Lookup whose majorType feature does *not* have the value "person". If a feature is missing it is treated as if it had an empty string as its value, so this would also match a Lookup annotation that did not have a majorType feature at all.

Certain type coercions are performed:

- If the constraint's attribute is a string, it is compared with the annotation feature value using string equality (`String.equals()`).
- If the constraint's attribute is an integer it is treated as a java.lang.Long. If the annotation feature value is also a Long, or is a string that can be parsed as a Long, then it is compared using `Long.equals()`.
- If the constraint's attribute is a floating-point number it is treated as a java.lang.Double. If the annotation feature value is also a Double, or is a string that can be parsed as a Double, then it is compared using `Double.equals()`.
- If the constraint's attribute is `true` or `false` (without quotes) it is treated as a java.lang.Boolean. If the annotation feature value is also a Boolean, or is a string that can be parsed as a Boolean, then it is compared using `Boolean.equals()`.

The `!=` operator matches exactly when `==` doesn't.

7.1.2 Comparison operators (“<”, “<=”, “>=” and “>”)

Comparison operators have their expected meanings, for example `{Token.length > 3}` matches a Token annotation whose length attribute is an integer greater than 3. The behaviour of the operators depends on the type of the constraint’s attribute:

- If the constraint’s attribute is a string it is compared with the annotation feature value using Unicode-lexicographic order (see `String.compareTo()`).
- If the constraint’s attribute is an integer it is treated as a java.lang.Long. If the annotation feature value is also a Long, or is a string that can be parsed as a Long, then it is compared using `Long.compareTo()`.
- If the constraint’s attribute is a floating-point number it is treated as a java.lang.Double. If the annotation feature value is also a Double, or is a string that can be parsed as a Double, then it is compared using `Double.compareTo()`.

7.1.3 Regular expression operators (“=∼”, “==∼”, “!∼” and “!=∼”)

These operators match regular expressions. `{Token.string =~ "[Dd]ogs"}` matches a Token annotation whose string feature contains a substring that matches the regular expression `[Dd]ogs`, using `!~` would match if the feature value does not contain a substring that matches the regular expression. The `==~` and `!=~` operators are like `=~` and `!~` respectively, but require that the *whole* value match (or not match) the regular expression[3]. As with `==`, missing features are treated as if they had the empty string as their value, so the constraint `{Identifier.name ==~ "(?i)[aeiou]*"}` would match an Identifier annotation which does not have a name feature, as well as any whose name contains only vowels.

The matching uses the standard Java regular expression library, so full details of the pattern syntax can be found in the JavaDoc documentation for java.util.regex.Pattern. There are a few specific points to note:

- To enable flags such as case-insensitive matching you can use the (?*flags*) notation. See the Pattern JavaDocs for details.
- If you need to include a double quote character in a regular expression you must precede it with a backslash, otherwise JAPE will give a syntax error. Quoted strings in JAPE grammars also convert the sequences `\n`, `\r` and `\t` to the characters newline (U+000A), carriage return (U+000D) and tab (U+0009) respectively, but these characters can match literally in regular expressions so it does not make any difference to the result in most cases.[4]

[3]This syntax will be familiar to Groovy users.

[4]However this does mean that it is not possible to include an n, r or t character after a backslash in a

7.1.4 Contextual operators ("contains" and "within")

These operators match annotations within the context of other annotations.

- contains - Written as `{X contains Y}`, returns true if an annotation of type X completely contains an annotation of type Y.
- within - Written as `{X within Y}`, returns true if an annotation of type X is completely covered by an annotation of type Y.

For either operator, the right-hand value (Y in the above examples) can be a full constraint itself. For example `{X contains {Y.foo=bar}}` is also accepted. The operators can be used in a multi-constraint statement just like any of the traditional ones, so `{X.f1 != "something", X contains {Y.foo=bar}}` is valid.

It is possible to add additional custom operators without modifying the JAPE language. There are new init-time parameters to Transducer so that additional annotation "meta-property" accessors and custom operators can be referenced at runtime. To add a custom operator, write a class that implements gate.jape.constraint.ConstraintPredicate, and then list that class name for the Transducer's "operators" property. Similarly, to add a custom "meta-property" accessor, write a class that implements gate.jape.constraint.AnnotationAccessor, and then list that class name in the Transducer's "annotationAccessors" property.

7.2 Use of Context

Context can be dealt with in the grammar rules in the following way. The pattern to be annotated is always enclosed by a set of round brackets. If preceding context is to be included in the rule, this is placed before this set of brackets. This context is described in exactly the same way as the pattern to be matched. If context following the pattern needs to be included, it is placed after the label given to the annotation. Context is used where a pattern should only be recognised if it occurs in a certain situation, but the context itself does not form part of the pattern to be annotated.

For example, the following rule for Time (assuming an appropriate macro for "year") would mean that a year would only be recognised if it occurs preceded by the words "in" or "by":

```
Rule: YearContext1
```

JAPE quoted string, or to have a backslash as the last character of your regular expression. Workarounds include placing the backslash in a character class ([\\]—) or enabling the (?x) flag, which allows you to put whitespace between the backslash and the offending character without changing the meaning of the pattern.

```
({Token.string == "in"}|
 {Token.string == "by"}
)
(YEAR)
:date -->
 :date.Timex = {kind = "date", rule = "YearContext1"}
```

Similarly, the following rule (assuming an appropriate macro for “email”) would mean that an email address would only be recognised if it occurred inside angled brackets (which would not themselves form part of the entity):

```
Rule: Emailaddress1
({Token.string == ‘‘<’’})
(
 (EMAIL)
)
:email
({Token.string == ‘‘>’’})
-->
 :email.Address= {kind = "email", rule = "Emailaddress1"}
```

Also, it is possible to specify the constraint that one annotation must start at the same place as another. For example:

```
Rule: SurnameStartingWithDe
(
  {Token.string == "de",
   Lookup.majorType == "name",
   Lookup.minorType == "surname"}
):de
-->
 :de.Surname = {prefix = "de"}
```

This rule would match anywhere where a Token with string “de” and a Lookup with majorType “name” and minorType “surname” start at the same offset in the text. Both the Lookup and Token annotations would be included in the `:de` binding, so the Surname annotation generated would span the longer of the two. Constraints on the same annotation type must be satisfied by a single annotation, so in this example there must be a single Lookup matching both the major and minor types – the rule would not match if there were two different lookups at the same location, one of them satisfying each constraint.

7.3 Use of Priority

Each grammar has one of 5 possible control styles: “brill”, “all”, “first”, “once” and “appelt”. This is specified at the beginning of the grammar.

The Brill style means that when more than one rule matches the same region of the document, they are all fired. The result of this is that a segment of text could be allocated more than one entity type, and that no priority ordering is necessary. Brill will execute all matching rules starting from a given position and will advance and continue matching from the position in the document where the longest match finishes.

The ”all” style is similar to Brill, in that it will also execute all matching rules, but the matching will continue from the next offset to the current one.

For example, where [] are annotations of type Ann

```
[aaa[bbb]] [ccc[ddd]]
```

then a rule matching {Ann} and creating {Ann-2} for the same spans will generate:

```
BRILL: [aaabbb] [cccddd]
ALL: [aaa[bbb]] [ccc[ddd]]
```

With the “first” style, a rule fires for the first match that’s found. This makes it unappropiate for rules that end in ”+” or ”?” or ”*”. Once a match is found the rule is fired; it does not attempt to get a longer match (as the other two styles do).

With the ”once” style, once a rule has fired, the whole JAPE phase exits after the first match.

With the appelt style, only one rule can be fired for the same region of text, according to a set of priority rules. Priority operates in the following way.

1. From all the rules that match a region of the document starting at some point X, the one which matches the longest region is fired.
2. If more than one rule matches the same region, the one with the highest priority is fired
3. If there is more than one rule with the same priority, the one defined earlier in the grammar is fired.

An optional priority declaration is associated with each rule, which should be a positive integer. The higher the number, the greater the priority. By default (if the priority declaration is missing) all rules have the priority -1 (i.e. the lowest priority).

For example, the following two rules for location could potentially match the same text.

```
Rule:   Location1
Priority: 25

(
 ({Lookup.majorType == loc_key, Lookup.minorType == pre}
  {SpaceToken})?
 {Lookup.majorType == location}
 ({SpaceToken}
  {Lookup.majorType == loc_key, Lookup.minorType == post})?
)
:locName -->
  :locName.Location = {kind = "location", rule = "Location1"}


Rule: GazLocation
Priority: 20
  (
  ({Lookup.majorType == location}):location
  )
  -->   :location.Name = {kind = "location", rule=GazLocation}
```

Assume we have the text "China sea", that "China" is defined in the gazetteer as "location", and that sea is defined as a "loc_key" of type "post". In this case, rule Location1 would apply, because it matches a longer region of text starting at the same point ("China sea", as opposed to just "China"). Now assume we just have the text "China". In this case, both rules could be fired, but the priority for Location1 is highest, so it will take precedence. In this case, since both rules produce the same annotation, so it is not so important which rule is fired, but this is not always the case.

One important point of which to be aware is that prioritisation only operates within a single grammar. Although we could make priority global by having all the rules in a single grammar, this is not ideal due to other considerations. Instead, we currently combine all the rules for each entity type in a single grammar. An index file (main.jape) is used to define which grammars should be used, and in which order they should be fired.

7.4 Use of negation

All the examples in the preceding sections involve constraints that require the presence of certain annotations to match. JAPE also supports "negative" constraints which specify the *absence* of annotations. A negative constraint is signalled in the grammar by a "!" character.

Negative constraints are generally used in combination with positive ones to constrain the locations at which the positive constraint can match. For example:

```
Rule: PossibleName
(
 {Token.orth == "upperInitial", !Lookup}
):name
-->
 :name.PossibleName = {}
```

This rule would match any uppercase-initial Token, but only where there is no Lookup annotation starting at the same location. The general rule is that a negative constraint matches at any location where the corresponding positive constraint would *not* match. Negative constraints do not contribute any annotations to the bindings - in the example above, the `:name` binding would contain only the Token annotation. The exception to this is when a negative constraint is used alone, without any positive constraints in the combination. In this case it binds *all* the annotations at the match position that do not match the constraint. Thus, `{!Lookup}` would bind all the annotations starting at this location except Lookups. In most cases, negative constraints should only be used in combination with positive ones.

Any constraint can be negated, for example:

```
Rule: SurnameNotStartingWithDe
(
 {Surname, !Token.string ==~ "[Dd]e"}
):name
-->
 :name.NotDe = {}
```

This would match any Surname annotation that does not start at the same place as a Token with the string ”de” or ”De”. Note that this is subtly different from `{Surname, Token.string !=~ "[Dd]e"}`, as the second form requires a Token annotation to be present, whereas the first form (!Token...) will match if there is no Token annotation at all at this location.[5]

7.5 Useful tricks

Although the JAPE language has some limitations as to how rules and patterns can be expressed, there are some useful tricks to overcome these problems.

[5]In the Montreal transducer, the two forms were equivalent

- Using priority to resolve ambiguity. If the Appelt style of matching is selected, rule priority operates in the following way.

  1. Length of rule – a rule matching a longer pattern will fire first.
  2. Explicit priority declaration. Use the optional Priority function to assign a ranking. The higher the number, the higher the priority. If no priority is stated, the default is -1.
  3. Order of rules. In the case where the above two factors do not distinguish between two rules, the order in which the rules are stated applies. Rules stated first have higher priority.

  Because priority can only operate within a single grammar, this can be a problem for dealing with ambiguity issues. One solution to this is to create a temporary set of annotations in initial grammars, and then manipulate this temporary set in one or more later phases (for example, by converting temporary annotations from different phases into permanent annotations in a single final phase). See the default set of grammars for an example of this.

- Negative operator. JAPE provides an operator to look for the absence of a single annotation type (see section 7.4), but there is no support for a general negative operator to prevent a rule from firing if a particular *sequence* of annotations is found. One solution to this is to create a "negative rule" which has higher priority than the matching "positive rule". The style of matching must be Appelt for this to work. To create a negative rule, simply state on the LHS of the rule the pattern that should NOT be matched, and on the RHS do nothing. In this way, the positive rule cannot be fired if the negative pattern matches, and vice versa, which has the same end result as using a negative operator. A useful variation for developers is to create a dummy annotation on the RHS of the negative rule, rather than to do nothing, and to give the dummy annotation a rule feature. In this way, it is obvious that the negative rule has fired. Alternatively, use Java code on the RHS to print a message when the rule fires. An example of a matching negative and positive rule follows. Here, we want a rule which matches a surname followed by a comma and a set of initials. But we want to specify that the initials shouldn't have the POS category PRP (personal pronoun). So we specify a negative rule that will fire if the PRP category exists, thereby preventing the positive rule from firing.

```
Rule: NotPersonReverse
Priority: 20
// we don't want to match ''Jones, I''
(
 {Token.category == NNP}
 {Token.string == ","}
 {Token.category == PRP}
)
:foo
```

```
-->
{}

Rule:   PersonReverse
Priority: 5
// we want to match ‘‘Jones, F.W.’’

(
 {Token.category == NNP}
 {Token.string == ","}
 (INITIALS)?
)
:person -->
```

- Matching special characters. To specify a single or double quote as a string, precede it with a backslash, e.g.

  ```
  {Token.string=="\""}
  ```

  will match a double quote. For other special characters, such as "$", enclose it in double quotes, e.g.

  ```
  {Token.category == "PRP\$"}
  ```

- Referring to previous annotations. An annotation generated in one phase can be referred to in a later phase, in exactly the same way as any other kind of annotation (by specifying the name of the annotation within curly braces). The features and values can be referred to or omitted, as with all other annotations. Make sure that if the Input specification is used in the grammar, that the annotation to be referred to is included in the list.

- Using context. Specify left or right context around a pattern by enclosing it in round brackets outside the round brackets of the pattern. In the example below, the context "in" must precede the location to be annotated. Only the location will be annotated, but it is important to remember that context is consumed by the rule, so it cannot be reused in another rule within the same phase. So, for example, right context cannot be used as left context for another rule.

  ```
  Rule:InLoc1
  // in PARIS
  (
   {Token.string == "in"}
  )
  (
   {Lookup.majorType == location}
  )
  :locName
  ```

- Debug. Add the following to the options at the top of the grammar.

  ```
  Options: control = appelt debug = true
  ```

- Avoid conflicts. If two possible ways of matching are found for the same text string, a conflict can arise. Normally this is handled by the priority mechanism (test length, rule priority and finally rule precedence). If all these are equal, Jape will simply choose a match at random and fire it. This leads ot non-deterministic behaviour, which should be avoided.

- Using Java code on the RHS. If you want to be flash, you can use any Java code you like on the RHS of the rule. This is useful for feature percolation (see below), for deleting previous annotations, measuring length of strings, and performing alternative operations depending on particular features of the annotation. See 7.7 for more details.

- Feature percolation. To copy features from previous annotations, where the value of the feature is unknown, some simple Java code can be used. See Section 7.7 for a more detailed explanation of this.

- Adding a feature to the document. Instead of adding a feature to an annotation, a feature can be added to the document as a whole. For example, the following code on the RHS would add the feature "texttype" with value "sport" to the document.

  ```
  doc.getFeatures().put("texttype", ‘‘sport’’);
  ```

- Overlapping annotations. Once JAPE has matched something, that part of the input is "consumed" and it moves on to the next position. This means that it does not recognise overlapping annotations in patterns, e.g. where matching overlapping lookups from the gazetteer. For example, for a string "hALCAM" with the lookups "hAL", "ALCAM" and "CAM", only the first lookup "hAL" will be recognised by the grammar rule that matches lookups. After the grammar has matched "hAL" it will continue with "CAM" hence skipping "ALCAM" completely.

  The trick to handle this kind of situations is to delete the used up lookups and run again the same grammar over the same input. You may need to repeat this several times until you've used up all your lookups. The number of repetitions required needs to be determined experimentally.

7.6 Ontology aware grammar transduction

GATE supports two different methods for ontology aware grammar transduction. Firstly it is possible to use the `ontology` feature both in grammars and annotations, while using the default transducer. Secondly it is possible to use an ontology aware transducer by passing an ontology language resource to one of the subsumes methods in `SimpleFeatureMapImpl`.

This second strategy does not check for ontology features, which will make the writing of grammars easier, as there is no need to specify `ontology` when writing them. More information about the ontology-aware transducer can be found in Section 10.6.

7.7 Using Java code in JAPE rules

The RHS of a JAPE rule can consist of any Java code. This is useful for removing temporary annotations and for percolating and manipulating features from previous annotations. In the example below

The first rule below shows a rule which matches a first person name, e.g. "Fred", and adds a gender feature depending on the value of the minorType from the gazetteer list in which the name was found. We first get the bindings associated with the person label (i.e. the Lookup annotation). We then create a new annotation called "personAnn" which contains this annotation, and create a new FeatureMap to enable us to add features. Then we get the minorType features (and its value) from the personAnn annotation (in this case, the feature will be "gender" and the value will be "male"), and add this value to a new feature called "gender". We create another feature "rule" with value "FirstName". Finally, we add all the features to a new annotation "FirstPerson" which attaches to the same nodes as the original "person" binding.

Note that inputAS and outputAS represent the input and output annotation set. Normally, these would be the same (by default when using ANNIE, these will be the "Default" annotation set). Since the user is at liberty to change the input and output annotation sets in the paramters of the JAPE transducer at runtime, it cannot be guaranteed that the input and output annotation sets will be the same, and therefore we must specify the annotation set we are referring to.

```
Rule: FirstName

(
 {Lookup.majorType == person_first}
):person
-->
{
gate.AnnotationSet person = (gate.AnnotationSet)bindings.get("person");
gate.Annotation personAnn = (gate.Annotation)person.iterator().next();
gate.FeatureMap features = Factory.newFeatureMap();
features.put("gender", personAnn.getFeatures().get("minorType"));
features.put("rule", "FirstName");
outputAS.add(person.firstNode(), person.lastNode(), "FirstPerson",
features);
}
```

The second rule (contained in a subsequent grammar phase) makes use of annotations produced by the first rule described above. Instead of percolating the minorType from the annotation produced by the gazetteer lookup, this time it percolates the feature from the annotation produced by the previous grammar rule. So here it gets the “gender” feature value from the “FirstPerson” annotation, and adds it to a new feature (again called “gender” for convenience), which is added to the new annotation (in outputAS) “TempPerson”. At the end of this rule, the existing input annotations (from inputAS) are removed because they are no longer needed. Note that in the previous rule, the existing annotations were not removed, because it is possible they might be needed later on in another grammar phase.

```
Rule: GazPersonFirst
(
 {FirstPerson}
)
:person
-->
{
gate.AnnotationSet person = (gate.AnnotationSet)bindings.get("person");
gate.Annotation personAnn = (gate.Annotation)person.iterator().next();
gate.FeatureMap features = Factory.newFeatureMap();

features.put("gender", personAnn.getFeatures().get("gender"));
features.put("rule", "GazPersonFirst");
outputAS.add(person.firstNode(), person.lastNode(), "TempPerson",
features);
inputAS.removeAll(person);
}
```

7.7.1 Adding a feature to the document

The following example code shows how to add the feature “genre” with value “email” to the document, using JAVA code on the RHS of a rule:

```
Rule: Email
Priority: 150

(
 {message}
)
-->
{
doc.getFeatures().put("genre", "email");
}
```

7.7.2 Using named blocks

For the common case where a Java block refers just to the annotations from a single left-hand-side binding, JAPE provides a shorthand notation:

```
Rule: RemoveDoneFlag

(
  {Instance.flag == "done"}
):inst
-->
:inst{
  Annotation theInstance = (Annotation)instAnnots.iterator().next();
  theInstance.getFeatures().remove("flag");
}
```

This rule is equivalent to the following:

```
Rule: RemoveDoneFlag

(
  {Instance.flag == "done"}
):inst
-->
{
  AnnotationSet instAnnots = (AnnotationSet)bindings.get("inst");
  if(instAnnots != null && instAnnots.size() != 0) {
    Annotation theInstance = (Annotation)instAnnots.iterator().next();
    theInstance.getFeatures().remove("flag");
  }
}
```

A label `:<label>` on a Java block creates a local variable `<label>Annots` within the Java block which is the `AnnotationSet` bound to the `<label>` label. Also, the Java code in the block is only executed if there is at least one annotation bound to the label, so you do not need to check this condition in your own code. Of course, if you need more flexibility, e.g. to perform some action in the case where the label is not bound, you will need to use an unlabelled block and perform the `bindings.get()` yourself.

7.7.3 Java RHS overview

When a JAPE grammar is parsed, a Jape parser creates action classes for all Java RHSs in the grammar. (one action class per RHS) RHS Java code will be embedded as a body of the method `doIt` and will work in context of this method. When a particular rule is fired, the method `doIt` will be executed.

Method `doIt` is specified by the interface `gate.jape.RhsAction`. Each action class implements this interface and is generated with the following template:

```
import java.io.*;
import java.util.*;
import gate.*;
import gate.jape.*;
import gate.creole.ontology.Ontology;
import gate.annotation.*;
import gate.util.*;
class <AutogeneratedActionClassName>
         implements java.io.Serializable, RhsAction {
    public void doIt(Document doc,
                     java.util.Map bindings,
                     AnnotationSet annotations,
                     AnnotationSet inputAS,
                     AnnotationSet outputAS,
                     Ontology ontology) {
        // your RHS Java code will be embedded here
...
    }
}
```

Method `doIt` has the following parameters that can be used in RHS Java code:

- `Document doc` - a document that is currently processed
- `java.util.Map bindings` - a map of binding variables where a key is a (String) name of binding variable and value is (AnnotationSet) set of annotations corresponding to this binding variable
- `AnnotationSet annotations` - Do not use this (it's a synonym for `outputAS` that is still used in some grammars but is now deprecated).
- `AnnotationSet inputAS` - input annotations
- `AnnotationSet outputAS` - output annotations
- `Ontology ontology` - a GATE's transducer ontology

In your Java RHS you can use short names for all Java classes that are imported by the action class (plus Java classes from the packages that are imported by default according to

JVM specification: java.lang.*, java.math.*). But you need to use fully qualified Java class names for all other classes. For example:

```
-->
{
  // VALID line examples
  AnnotationSet as = ...
  InputStream is = ...
  java.util.logging.Logger myLogger =
          java.util.logging.Logger.getLogger("JAPELogger");
  java.sql.Statement stmt = ...

  // INVALID line examples
  Logger myLogger = Logger.getLogger("JapePhaseLogger");
  Statement stmt = ...
}
```

7.8 Optimising for speed

The way in which grammars are designed can have a huge impact on the processing speed. Some simple tricks to keep the processing as fast as possible are:

- avoid the use of the * and + operators. Replace them with ? where possible. For example, instead of

  ```
  ({Token})*
  ```

  use

  ```
  ({Token})? ({Token})? ({Token})?
  ```

  if you can predict that you won't need to recognise a string of Tokens longer than 3.

- avoid specifying unnecessary elements such as SpaceTokens where you can. To do this, use the Input specification at the beginning of the grammar to stipulate the annotations that need to be considered. If no Input specification is used, all annotations will be considered (so, for example, you cannot match two tokens separated by a space unless you specify the SpaceToken in the pattern). If, however, you specify Tokens but not SpaceTokens in the Input, SpaceTokens do not have to be mentioned in the pattern to be recognised. If, for example, there is only one rule in a phase that requires SpaceTokens to be specified, it may be judicious to move that rule to a separate phase where the SpaceToken can be specified as Input.

- avoid the shorthand syntax for copying feature values (`newFeat = :bind.Type.oldFeat`), particularly if you need to copy multiple features from the left to the right hand side of your rule.

7.9 Serializing JAPE Transducer

JAPE grammars are written as files with the extension ".jape", which are parsed and compiled at run-time to execute them over the GATE document(s). Serialization of the JAPE Transducer adds the capability to serialize such grammar files and use them later to bootstrap new JAPE transducers, where they do not need the original JAPE grammar file. This allows people to distribute the serialized version of their grammars without disclosing the actual contents of their jape files. This is implemented as part of the JAPE Transducer PR. The following sections describe how to serialize and deserialize them.

7.9.1 How to serialize?

Once an instance of a JAPE transducer is created, the option to serialize it appears in the option menu of that instance. The option menu can be activated by right clicking on the respective PR. Having done so, it asks for the file name where the serialized version of the respective JAPE grammar is stored.

7.9.2 How to use the serialized grammar file?

The JAPE Transducer now also has an init-time parameter *binaryGrammarURL*, which appears as an optional parameter to the *grammarURL*. The User can use this parameter (i.e. *binaryGrammarURL*) to specify the serialized grammar file.

7.10 The JAPE Debugger

The Jape debugger helps to find errors in Jape programs enabling the user to see in detail how a Jape rule works when applied to a particular range of text. It was written by `Ontos`, who also provided the original version of this documentation. The debugger allows the user to select a particular part of the text, and then look at the detailed history of processing. This will enable them to see which rules were matched and which were not, and also why particular rules were or were not matched. It is also possible to set breakpoints for particular rules, enabling the user to see how the rule was matched, and what annotations were created.

The Jape debugger could be useful in situations where the old simple DEBUG OUTPUT method does not help. For example when:

- A Rule LHS has not been matched.
- Text did not match the expected template of a rule.
- The rule was overridden by another conflicting rule.
- Annotations are created, but it is not possible to tell which rule created them.

7.10.1 Debugger GUI

The layout of the JAPE-debugger user interface is shown in Figure 7.1.

The debugger's main frame consists of the following primary components:

- Resources tree (appears in the left side of the main frame and contains all the resources available within the current GATE session).
- Debugging panel (located at the center of the main frame and contains three tabs providing all necessary debugging information).
- Document panel (provides you with the document on which you are currently debugging).

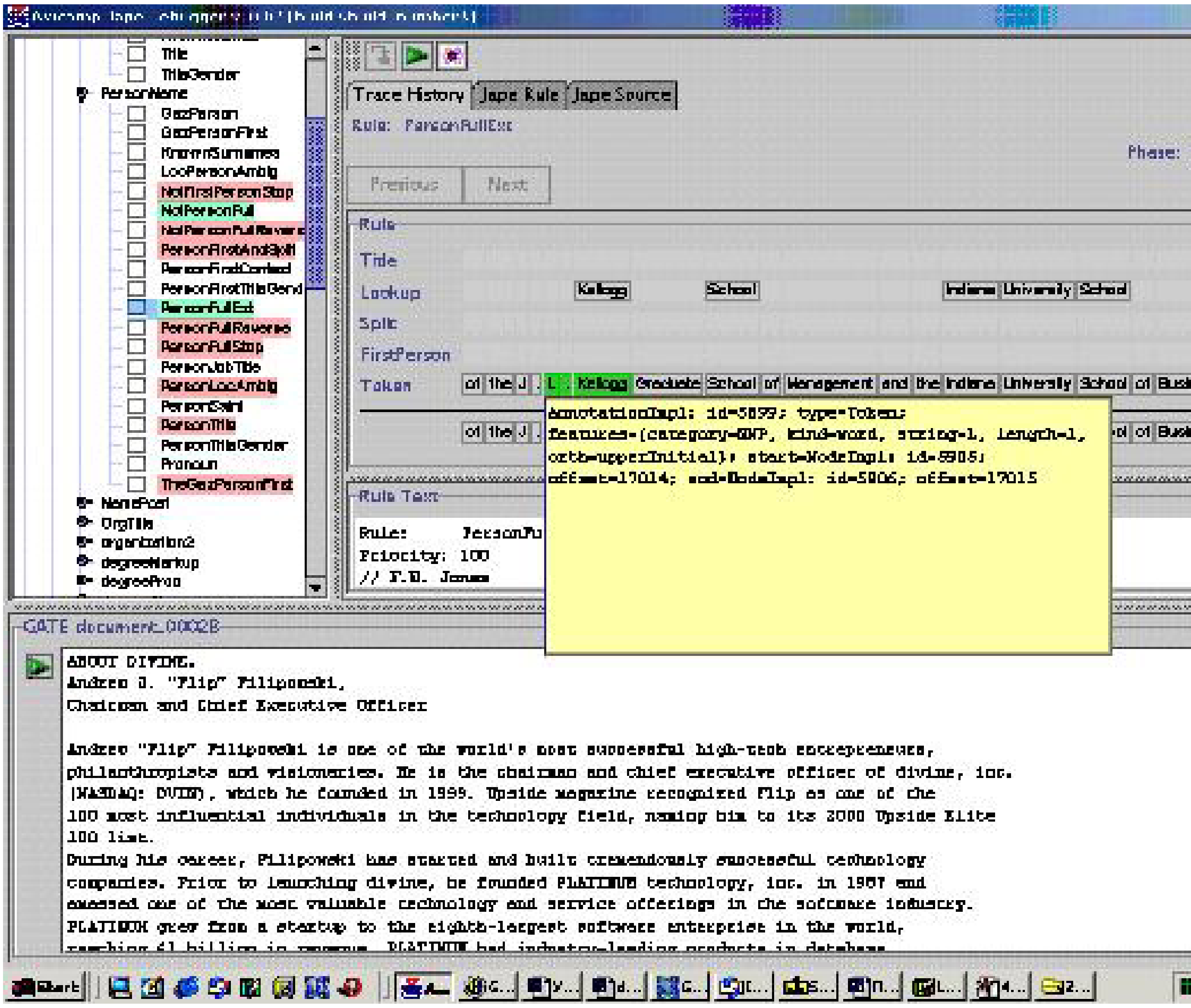
7.10.2 Using the Debugger

In most situations you will use the debugger in trace mode using the following steps:

- initialize JAPE-debugger from the GATE menu (Tools / JAPE Debugger);
- run a GATE serial controller (This can be done either from GATE or from the debugger. Note: for performance reasons, the debugger doesnt gather matching information when its not running, so run GATE serial controller after you open the debugger window);
- select the part of the text that is interesting for debugging purposes, and press the button at the left of the text to update the view of the debugger.

After these steps the following information becomes available. In the Resources tree some of the rules become highlighted in different colors:

- Green means that the rule has matched successfully.



```
AnnotationImpl: id=5899; type=Token;
features={category=NNP, kind=word, string=L, length=1,
orth=upperInitial}; start=NodeImpl: id=5905;
offset=17014; end=NodeImpl: id=5906; offset=17015
```

```
Rule:     PersonFu
Priority: 100
// F.D. Jones
```

```
ABOUT DIVINE.
Andrew J. "Flip" Filipowski,
Chairman and Chief Executive Officer

Andrew "Flip" Filipowski is one of the world's most successful high-tech entrepreneurs,
philanthropists and visionaries. He is the chairman and chief executive officer of divine, inc.
(NASDAQ: DVIN), which he founded in 1999. Upside magazine recognized Flip as one of the
100 most influential individuals in the technology field, naming him to its 2000 Upside Elite
100 list.
During his career, Filipowski has started and built tremendously successful technology
companies. Prior to launching divine, he founded PLATINUM technology, inc. in 1987 and
amassed one of the most valuable technology and service offerings in the software industry.
PLATINUM grew from a startup to the eighth-largest software enterprise in the world,
reaching $1 billion in revenue. PLATINUM had industry-leading products in database
```

Figure 7.1: The JAPE Debugger User Interface

- Yellow means that it matched, but was overridden by another rule.
- Red means that the rule tried to match, but failed.

Trace history is the main debugging tab in Debugging panel. It contains the source of the JAPE rule currently selected, and the selected text in the document panel. All the inputs are shown, and matched inputs are highlighted in green. Annotations, which made the rule fail, are highlighted in red. If a rule tried to match more than one time on the selected text interval, buttons on the top of the panel (Previous and Next) become enabled, and allow one to observe all the matching attempts of the rule. Clicking on any of the inputs shows an annotation window, and the tool tip of the matched words gives the template in the rule.

Step by Step Example

To give an idea of how to use debugger for fixing bugs, lets consider the following example. For instance, there is a rule named PersonFullExt, which should find person names: A. B. Dick, J. F. Kennedy and so on, and create an annotation Person. To test the rule, we run GATE on a text fragment containing the following words: the J.L. Kellogg Graduate School, so we would expect that the part of the text J. L. Kellogg should get an annotation Person. Unfortunately, we encounter a problem (because only L. Kellogg was matched), so we decide to use the debugger to find the reason for this unexpected behavior. With JAPE-debugger, it is possible to observe everything needed during for finding and fixing the error.

The appropriate screenshot is shown in Figure 7.2.

As you can see, the rule NotPersonFull matched the text ‘the J’, so the rule PersonFullExt could start matching only after the pointer has moved to the token ‘.’. Without the debugger, it wouldn’t be so easy to find the reason for this error, because the rule NotPersonFull doesn’t create any annotations.

An additional feature of the debugger is the availability of debugging with breakpoints (Jape Rule Tab). After setting a breakpoint on a given rule (in our case it is the rule named TheOrgXBase), the GATE transducer will be interrupted at the breakpoint, and in the document panel the text that is currently matched by the rule (it is highlighted in cyan) will be displayed. In the tab, a special table representation of the rule (with what it matches on the left side), and the history of annotations created by this rule, will be displayed, as in Figure 7.3.

7.10.3 Known Bugs

1. Debugger doesn’t see processing resource reinitialization. A possible workaround is to close and open the resource again.

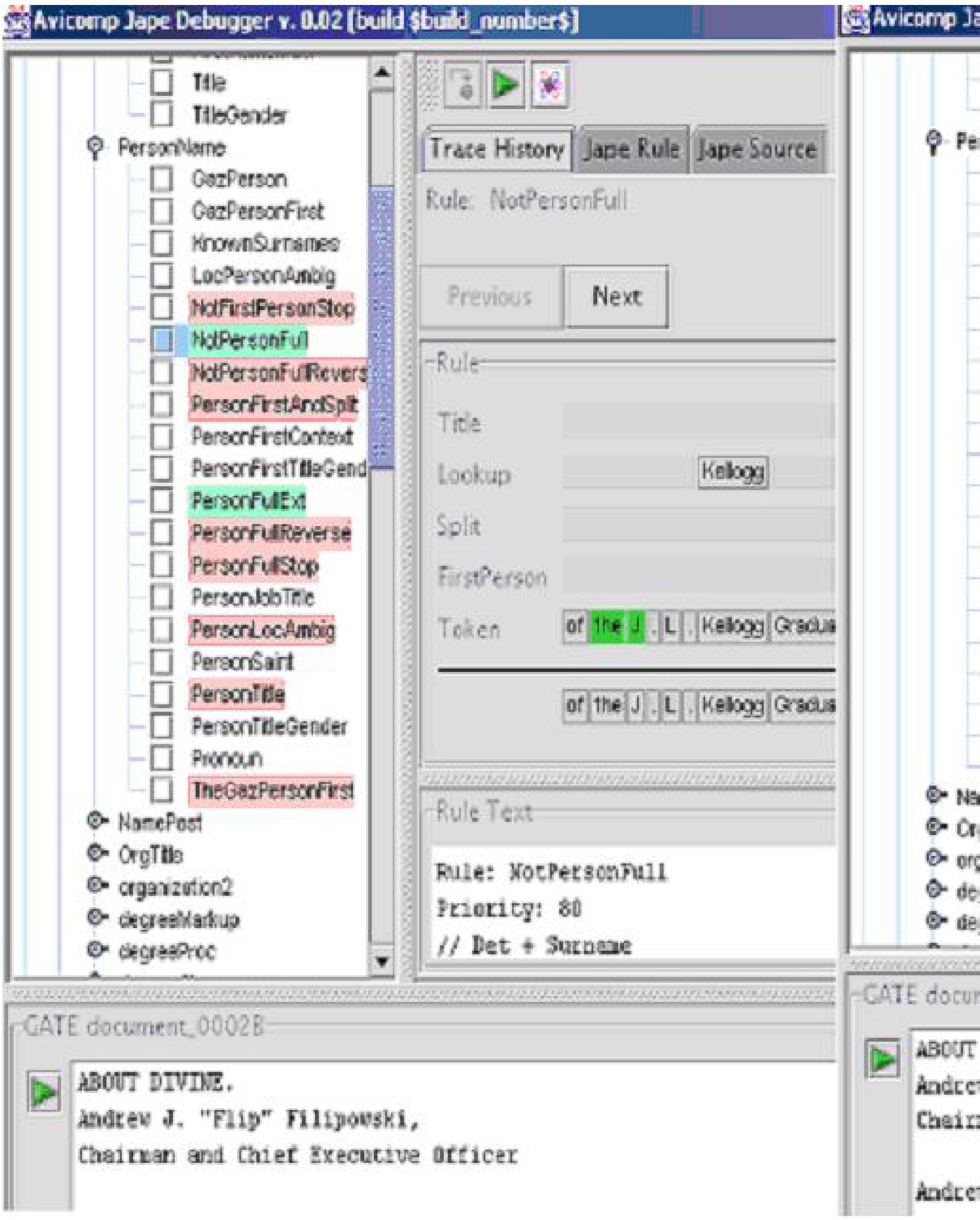
7.11 Notes for Montreal Transducer users

In June 2008, the standard JAPE transducer implementation gained a number of features inspired by Luc Plamondon’s ”Montreal Transducer”, which has been available as a GATE plugin for several years. If you have existing Montreal Transducer grammars and want to update them to work with the standard JAPE implementation you should be aware of the following differences in behaviour:

- Quantifiers (*, + and ?) in the Montreal transducer are always greedy, but this is not necessarily the case in standard JAPE.

- The Montreal Transducer defines `{Type.feature != value}` to be the same as `{!Type.feature == value}` (and likewise the `!~` operator in terms of `=~`). In standard JAPE these constructs have different semantics. `{Type.feature != value}` will only match if there is a `Type` annotation whose `feature` feature does not have the given value, and if it matches it will bind the single `Type` annotation. `{!Type.feature == value}` will match if there is no `Type` annotation at a given place with this feature (including when there is no `Type` annotation at all), and if it matches it will bind every other annotation that starts at that location. If you have used `!=` in your Montreal grammars and want them to continue to behave the same way you must change them to use the prefix-! form instead (see section 7.4).

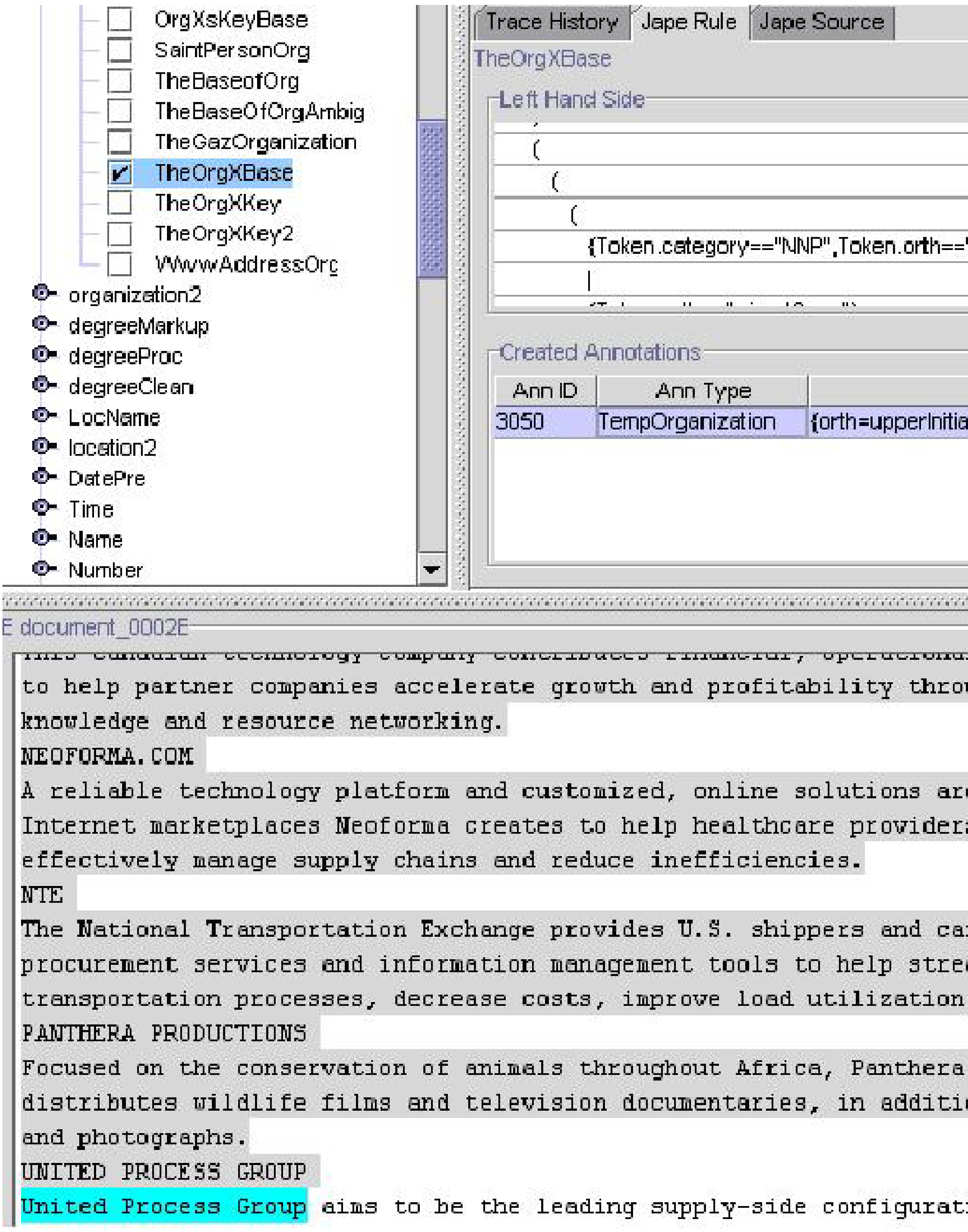
- The `=~` operator in standard JAPE looks for regular expression matches anywhere within a feature value, whereas in the Montreal transducer it requires the whole string to match. To obtain the whole-string matching behaviour in standard JAPE, use the `==~` operator instead (see section 7.1.3).



```
Rule: NotPersonFull
Priority: 80
// Det + Surname
```

```
ABOUT DIVINE.
Andrew J. "Flip" Filipowski,
Chairman and Chief Executive Officer
```

Figure 7.2: Finding Errors



```
(
 (
  (
   {Token.category=="NNP",Token.orth=="
   |
```

Ann ID	Ann Type	
3050	TempOrganization	{orth=upperInitia

```
to help partner companies accelerate growth and profitability thro
knowledge and resource networking.
NEOFORMA.COM
A reliable technology platform and customized, online solutions ar
Internet marketplaces Neoforma creates to help healthcare provider
effectively manage supply chains and reduce inefficiencies.
NTE
The National Transportation Exchange provides U.S. shippers and ca
procurement services and information management tools to help stre
transportation processes, decrease costs, improve load utilization
PANTHERA PRODUCTIONS
Focused on the conservation of animals throughout Africa, Panthera
distributes wildlife films and television documentaries, in additi
and photographs.
UNITED PROCESS GROUP
United Process Group aims to be the leading supply-side configurat
```

Figure 7.3: The Interface of the JAPE Debugger while Running in Breakpoint Mode

Chapter 8

ANNIE: a Nearly-New Information Extraction System

> And so the time had passed predictably and soberly enough in work and routine chores, and the events of the previous night from first to last had faded; and only now that both their days' work was over, the child asleep and no further disturbance anticipated, did the shadowy figures from the masked ball, the melancholy stranger and the dominoes in red, revive; and those trivial encounters became magically and painfully interfused with the treacherous illusion of missed opportunities. Innocent yet ominous questions and vague ambiguous answers passed to and fro between them; and, as neither of them doubted the other's absolute candour, both felt the need for mild revenge. They exaggerated the extent to which their masked partners had attracted them, made fun of the jealous stirrings the other revealed, and lied dismissively about their own. Yet this light banter about the trivial adventures of the previous night led to more serious discussion of those hidden, scarcely admitted desires which are apt to raise dark and perilous storms even in the pureset, most transparent soul; and they talked about those secret regions for which they felt hardly any longing, yet towards which the irrational wings of fate might one day drive them, if only in their dreams. For however much they might belong to one another heart and soul, they knew last night was not the first time they had been stirred by a whiff of freedom, danger and adventure.
>
> *Dream Story*, Arthur Schnitzler, 1926 (pp. 4-5).

GATE was originally developed in the context of Information Extraction (IE) R&D, and IE systems in many languages and shapes and sizes have been created using GATE with the IE components that have been distributed with it (see [Maynard *et al.* 00] for descriptions of some of these projects).[1]

[1]The principal architects of the IE systems in GATE version 1 were Robert Gaizauskas and Kevin Humphreys. This work lives on in the LaSIE system. (A derivative of LaSIE was distributed with GATE

GATE is distributed with an IE system called ANNIE, A Nearly-New IE system (developed by Hamish Cunningham, Valentin Tablan, Diana Maynard, Kalina Bontcheva, Marin Dimitrov and others). ANNIE relies on finite state algorithms and the JAPE language (see chapter 7).

ANNIE components form a pipeline which appears in figure 8.1. ANNIE components are

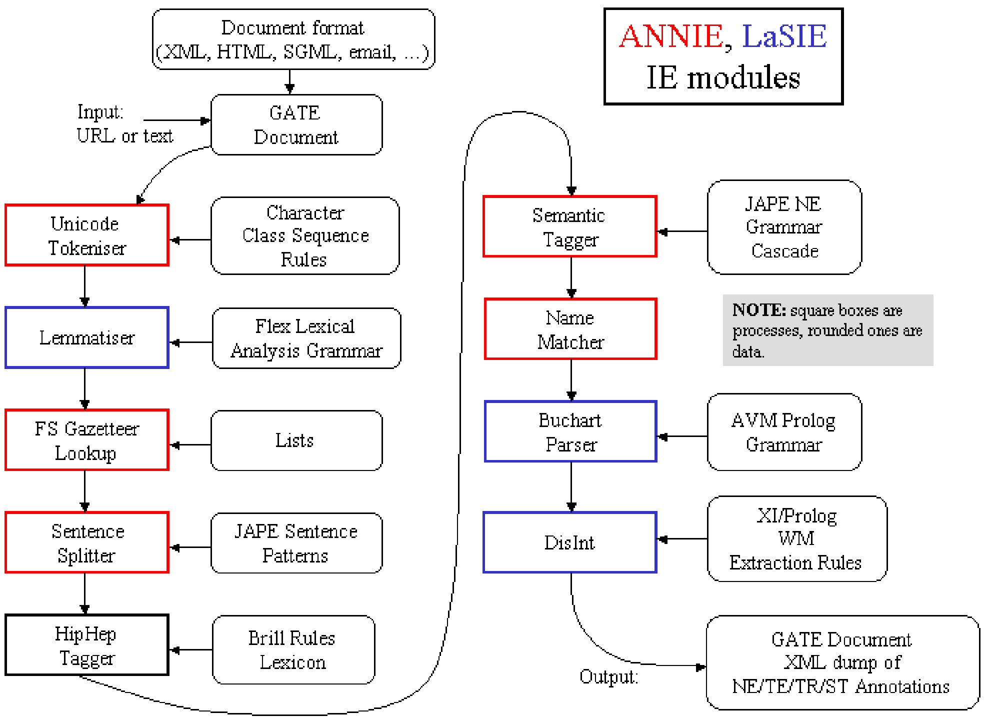


Figure 8.1: ANNIE and LaSIE

included with GATE (though the linguistic resources they rely on are generally more simple than the ones we use in-house). The rest of this chapter describes these components.

8.1 Tokeniser

The tokeniser splits the text into very simple tokens such as numbers, punctuation and words of different types. For example, we distinguish between words in uppercase and lowercase, and between certain types of punctuation. The aim is to limit the work of the tokeniser to maximise efficiency, and enable greater flexibility by placing the burden on the grammar rules, which are more adaptable.

version 1 under the name VIE, a Vanilla IE system.)

8.1.1 Tokeniser Rules

A rule has a left hand side (LHS) and a right hand side (RHS). The LHS is a regular expression which has to be matched on the input; the RHS describes the annotations to be added to the AnnotationSet. The LHS is separated from the RHS by '>'. The following operators can be used on the LHS:

```
| (or)
* (0 or more occurrences)
? (0 or 1 occurrences)
+ (1 or more occurrences)
```

The RHS uses ';' as a separator, and has the following format:

```
{LHS} > {Annotation type};{attribute1}={value1};...;{attribute
n}={value n}
```

Details about the primitive constructs available are given in the tokeniser file (DefaultTokeniser.Rules).

The following tokeniser rule is for a word beginning with a single capital letter:

```
"UPPERCASE_LETTER" "LOWERCASE_LETTER"* >
  Token;orth=upperInitial;kind=word;
```

It states that the sequence must begin with an uppercase letter, followed by zero or more lowercase letters. This sequence will then be annotated as type "Token". The attribute "orth" (orthography) has the value "upperInitial"; the attribute "kind" has the value "word".

8.1.2 Token Types

In the default set of rules, the following kinds of Token and SpaceToken are possible:

Word

A word is defined as any set of contiguous upper or lowercase letters, including a hyphen (but no other forms of punctuation). A word also has the attribute "orth", for which four values are defined:

- upperInitial - initial letter is uppercase, rest are lowercase
- allCaps - all uppercase letters
- lowerCase - all lowercase letters
- mixedCaps - any mixture of upper and lowercase letters not included in the above categories

Number

A number is defined as any combination of consecutive digits. There are no subdivisions of numbers.

Symbol

Two types of symbol are defined: currency symbol (e.g. '$', '£') and symbol (e.g. '&', '^'). These are represented by any number of consecutive currency or other symbols (respectively).

Punctuation

Three types of punctuation are defined: start_punctuation (e.g. '('), end_punctuation (e.g. ')'), and other punctuation (e.g. ':'). Each punctuation symbol is a separate token.

SpaceToken

White spaces are divided into two types of SpaceToken - space and control - according to whether they are pure space characters or control characters. Any contiguous (and homogenous) set of space or control characters is defined as a SpaceToken.

The above description applies to the default tokeniser. However, alternative tokenisers can be created if necessary. The choice of tokeniser is then determined at the time of text processing.

8.1.3 English Tokeniser

The English Tokeniser is a processing resource that comprises a normal tokeniser and a JAPE transducer (see chapter7). The transducer has the role of adapting the generic output of the tokeniser to the requirements of the English part-of-speech tagger. One such adaptation is the joining together in one token of constructs like " '30s", " 'Cause", " 'em", " 'N", " 'S", " 's", " 'T", " 'd", " 'll", " 'm", " 're", " 'til", " 've", etc. Another task of the JAPE transducer is to convert negative constructs like "don't" from three tokens ("don", " ' " and "t") into two tokens ("do" and "n't").

The English Tokeniser should always be used on English texts that need to be processed afterwards by the POS Tagger.

8.2 Gazetteer

The gazetteer lists used are plain text files, with one entry per line. Each list represents a set of names, such as names of cities, organisations, days of the week, etc.

Below is a small section of the list for units of currency:

```
Ecu
European Currency Units
FFr
Fr
German mark
German marks
New Taiwan dollar
New Taiwan dollars
NT dollar
NT dollars
```

An index file (lists.def) is used to access these lists; for each list, a major type is specified and, optionally, a minor type [2]. In the example below, the first column refers to the list name, the second column to the major type, and the third to the minor type. These lists are compiled into finite state machines. Any text tokens that are matched by these machines will be annotated with features specifying the major and minor types. Grammar rules then specify the types to be identified in particular circumstances. Each gazetteer list should reside in the same directory as the index file.

```
currency_prefix.lst:currency_unit:pre_amount
currency_unit.lst:currency_unit:post_amount
date.lst:date:specific
day.lst:date:day
```

So, for example, if a specific day needs to be identified, the minor type "day" should be specified in the grammar, in order to match only information about specific days; if any kind

[2]it is also possible to include a language in the same way, where lists for different languages are used, though ANNIE is only concerned with monolingual recognition

of date needs to be identified,the major type "date" should be specified, to enable tokens annotated with any information about dates to be identified. More information about this can be found in the following section.

In addition, the gazetteer allows arbitrary feature values to be associated with particular entries in a single list. ANNIE does not use this capability, but to enable it for your own gazetteers, set the optional `gazetteerFeatureSeparator` parameter to a single character (or an escape sequence such as `\t` or `\uNNNN`) when creating a gazetteer. In this mode, each line in a `.lst` file can have feature values specified, for example, with the following entry in the index file:

```
software_company.lst:company:software
```

the following `software_company.lst`:

```
Red Hat&stockSymbol=RHAT
Apple Computer&abbrev=Apple&stockSymbol=AAPL
Microsoft&abbrev=MS&stockSymbol=MSFT
```

and `gazetteerFeatureSeparator` set to `&`, the gazetteer will annotate `Red Hat` as a `Lookup` with features `majorType=company`, `minorType=software` and `stockSymbol=RHAT`. Note that you do not have to provide the same features for every line in the file, in particular it is possible to provide extra features for some lines in the list but not others.

Here is a full list of the parameters used by the Default Gazetteer:

Init-time parameters

listsURL A URL pointing to the index file (ususally lists.def) that contains the list of pattern lists.

encoding The character encoding to be used while reading the pattern lists.

gazetteerFeatureSeparator The character used to add arbitrary features to gazetteer entries. See above for an example.

caseSensitive Should the gazetteer be case sensitive during matching.

Run-time parameters

document The document to be preocessed.

annotationSetName The name for annotation set where the resulting Lookup annotations will be created.

wholeWordsOnly Should the gazetteer only match whole words? If set to true, a string segment in the input document will only be matched if it is bordered by characters that are not letters, non spacing marks, or combining spacing marks (as identified by the Unicode standard).

longestMatchOnly Should the gazetteer only match the longest possible string starting from any position. This parameter is only relevant when the list of lookups contains proper prefixes of other entries (e.g when both "Dell" and "Dell Europe" are in the lists). The default behaviour (when this parameter is set to `true`) is to only match the longest entry, "Dell Europe" in this example. This is the default GATE gazetteer behaviour since version 2.0. Setting this parameter to `false` will cause the gazetteer to match all possible prefixes.

8.3 Sentence Splitter

The **sentence splitter** is a cascade of finite-state transducers which segments the text into sentences. This module is required for the tagger. The splitter uses a gazetteer list of abbreviations to help distinguish sentence-marking full stops from other kinds.

Each sentence is annotated with the type Sentence. Each sentence break (such as a full stop) is also given a "Split" annotation. This has several possible types: ".", "punctuation", "CR" (a line break) or "multi" (a series of punctuation marks such as "?!?!".

The sentence splitter is domain and application-independent.

There is an alternative ruleset for the Sentence Splitter which considers newlines and carriage returns differently. In general this version should be used when a new line on the page indicates a new sentence). To use this alternative version, simply load the main-single-nl.jape from the default location instead of main.jape (the default file) when asked to select the location of the grammar file to be used.

8.4 RegEx Sentence Splitter

The RegEx sentence splitter is an alternative to the standard ANNIE Sentence Splitter. Its main aim is to address some performance issues identified in the JAPE-based splitter, mainly do to with improving the execution time and robustness, especially when faced with irregular input.

As its name suggests, the RegEx splitter is based on regular expressions, using the default Java implementation.

The new splitter is configured by three files containing (Java style, see `http://`

`java.sun.com/j2se/1.5.0/docs/api/java/util/regex/Pattern.html`) regular expressions, one regex per line. The three different files encode patterns for:

internal splits sentence splits that are part of the sentence, such as sentence ending punctuation;

external splits sentence splits that are NOT part of the sentence, such as 2 consecutive new lines;

non splits text fragments that might be seen as splits but they should be ignored (such as full stops occurring inside abbreviations).

The new splitter comes with an initial set of patterns that try to emulate the behaviour of the original splitter (apart from the situations where the original one was obviously wrong, like not allowing sentences to start with a number).

Here is a full list of the parameters used by the RegEx Sentence Splitter:

Init-time parameters

encoding The character encoding to be used while reading the pattern lists.

externalSplitListURL URL for the file containing the list of external split patterns;

internalSplitListURL URL for the file containing the list of internal split patterns;

nonSplitListURL URL for the file containing the list of non split patterns;

Run-time parameters

document The document to be preocessed.

outputASName The name for annotation set where the resulting `Split` and `Sentence` annotations will be created.

8.5 Part of Speech Tagger

The **tagger** [Hepple 00] is a modified version of the Brill tagger, which produces a part-of-speech tag as an annotation on each word or symbol. The list of tags used is given in Appendix E. The tagger uses a default lexicon and ruleset (the result of training on a large corpus taken from the Wall Street Journal). Both of these can be modified manually if necessary. Two additional lexicons exist - one for texts in all uppercase (lexicon_cap), and one for texts in all lowercase (lexicon_lower). To use these, the default lexicon should be replaced with the appropriate lexicon at load time. The default ruleset should still be used in this case.

The ANNIE Part-of-Speech tagger requires the following parameters.

- encoding - encoding to be used for reading rules and lexicons (init-time)
- lexiconURL - The URL for the lexicon file (init-time)
- rulesURL - The URL for the ruleset file (init-time)
- document - The document to be processed (run-time)
- inputASName - The name of the annotation set used for input (run-time)
- outputASName - The name of the annotation set used for output (run-time). This is an optional parameter. If user does not provide any value, new annotations are created under the default annotation set.
- baseTokenAnnotationType - The name of the annotation type that refers to Tokens in a document (run-time, default = Token)
- baseSentenceAnnotationType - The name of the annotation type that refers to Sentences in a document (run-time, default = Sentences)
- outputAnnotationType - POS tags are added as category features on the annotations of type "outputAnnotationType" (run-time, default = Token)

If - (inputASName == outputASName) AND (outputAnnotationType == baseTokenAnnotationType)

then - New features are added on existing annotations of type "baseTokenAnnotationType".

otherwise - Tagger searches for the annotation of type "outputAnnotationType" under the "outputASName" annotation set that has the same offsets as that of the annotation with type "baseTokenAnnotationType". If it succeeds, it adds new feature on a found annotation, and otherwise, it creates a new annotation of type "outputAnnotationType" under the "outputASName" annotation set.

8.6 Semantic Tagger

ANNIE's semantic tagger is based on the JAPE language – see chapter 7. It contains rules which act on annotations assigned in earlier phases, in order to produce outputs of annotated entities.

8.7 Orthographic Coreference (OrthoMatcher)

(Note: this component was previously known as a ”NameMatcher”.)

The Orthomatcher module adds identity relations between named entities found by the semantic tagger, in order to perform coreference. It does not find new named entities as such, but it may assign a type to an unclassified proper name, using the type of a matching name.

The matching rules are only invoked if the names being compared are both of the same type, i.e. both already tagged as (say) organisations, or if one of them is classified as ‘unknown’. This prevents a previously classified name from being recategorised.

8.7.1 GATE Interface

Input – entity annotations, with an id attribute.

Output – matches attributes added to the existing entity annotations.

8.7.2 Resources

A lookup table of aliases is used to record non-matching strings which represent the same entity, e.g. “IBM” and “Big Blue”, “Coca-Cola” and “Coke”. There is also a table of spurious matches, i.e. matching strings which do not represent the same entity, e.g. “BT Wireless” and “BT Cellnet” (which are two different organizations). The list of tables to be used is a load time parameter of the orthomatcher: a default list is set but can be changed as necessary.

8.7.3 Processing

The wrapper builds an array of the strings, types and IDs of all `name` annotations, which is then passed to a string comparison function for pairwise comparisons of all entries.

8.8 Pronominal Coreference

The pronominal coreference module performs anaphora resolution using the JAPE grammar formalism. Note that this module is not automatically loaded with the other ANNIE modules, but can be loaded separately as a Processing Resource. The main module consists of three submodules:

- quoted text module
- pleonastic it module
- pronominal resolution module

The first two modules are helper submodules for the pronominal one, because they do not perform anything related to coreference resolution except the location of quoted fragments and pleonastic it occurrences in text. They generate temporary annotations which are used by the pronominal submodule (such temporary annotations are removed later).

The main coreference module can operate successfully only if all ANNIE modules were already executed. The module depends on the following annotations created from the respective ANNIE modules:

- Token (English Tokenizer)
- Sentence (Sentence Splitter)
- Split (Sentence Splitter)
- Location (NE Transducer, OrthoMatcher)
- Person (NE Transducer, OrthoMatcher)
- Organization (NE Transducer, OrthoMatcher)

For each pronoun (anaphor) the coreference module generates an annotation of type "Coreference" containing two features:

- antecedent offset - this is the offset of the starting node for the annotation (entity) which is proposed as the antecedent, or null if no antecedent can be proposed.
- matches - this is a list of annotation IDs that comprise the coreference chain comprising this anaphor/antecedent pair.

8.8.1 Quoted Speech Submodule

The quoted speech submodule identifies quoted fragments in the text being analysed. The identified fragments are used by the pronominal coreference submodule for the proper resolution of pronouns such as I, me, my, etc. which appear in quoted speech fragments. The module produces "Quoted Text" annotations.

The submodule itself is a JAPE transducer which loads a JAPE grammar and builds an FSM over it. The FSM is intended to match the quoted fragments and generate appropriate annotations that will be used later by the pronominal module.

The JAPE grammar consists of only four rules, which create temporary annotations for all punctuation marks that may enclose quoted speech, such as ", ', “, etc. These rules then try to identify fragments enclosed by such punctuation. Finally all temporary annotations generated during the processing, except the ones of type "Quoted Text", are removed (because no other module will need them later).

8.8.2 Pleonastic It submodule

The pleonastic it submodule matches pleonastic occurrences of "it". Similar to the quoted speech submodule, it is a JAPE transducer operating with a grammar containing patterns that match the most commonly observed pleonastic it constructs.

8.8.3 Pronominal Resolution Submodule

The main functionality of the coreference resolution module is in the pronominal resolution submodule. This uses the result from the execution of the quoted speech and pleonastic it submodules. The module works according to the following algorithm:

- Preprocess the current document. This step locates the annotations that the submodule need (such as Sentence, Token, Person, etc.) and prepares the appropriate data structures for them.
- For each pronoun do the following:
  - inspect the proper appropriate context for all candidate antecedents for this kind of pronoun;
  - choose the best antecedent (if any);
- Create the coreference chains from the individual anaphor/antecedent pairs and the coreference information supplied by the OrthoMatcher (this step is performed from the main coreference module).

8.8.4 Detailed description of the algorithm

Full details of the pronominal coreference algorithm are as dollows.

Preprocessing

The preprocessing task includes the following subtasks:

- Identifying the sentences in the document being processed. The sentences are identified with the help of the Sentence annotations generated from the Sentence Splitter. For each sentence a data structure is prepared that contains three lists. The lists contain the annotations for the person/organization/location named entities appearing in the sentence. The named entities in the sentence are identified with the help of the Person, Location and Organization annotations that are already generated from the Named Entity Transducer and the OrthoMatcher.
- The gender of each person in the sentence is identified and stored in a global data structure. It is possible that the gender information is missing for some entities - for example if only the person family name is observed then the Named Entity transducer will be unable to deduce the gender. In such cases the list with the matching entities generated by the OrhtoMatcher is inspected and if some of the orthographic matches contains gender information it is assigned to the entity being processed.
- The identified pleonastic it occurrences are stored in a separate list. The ”Pleonastic It” annotations generated from the pleonastic submodule are used for the task.
- For each quoted text fragment, identified by the quoted text submodule, a special structure is created that contains the persons and the 3rd person singular pronouns such as ”he” and ”she” that appear in the sentence containing the quoted text, but not in the quoted text span (i.e. the ones preceding and succeeding the quote).

Pronoun resolution

This task includes the following subtasks:

etrieving all the pronouns in the document. Pronouns are represented as annotations of type ”Token” with feature ”category” having value ”PRP” or ”PRP$”. The former classifies possessive adjectives such as my, your, etc. and the latter classifies personal, reflexive etc. pronouns. The two types of pronouns are combined in one list and sorted according to their offset in the text.

For each pronoun in the list the following actions are performed:

- If the pronoun is it then a check is performed if this is a pleonastic occurrence and if so then no further attempt for resolution is made.
- The proper context is determined. The context size is expressed in the number of sentences it will contain. The context always includes the current sentence (the one containing the pronoun), the preceding sentence and zero or more preceding sentences.

- Depending on the type of pronoun, a set of candidate antecedents is proposed. The candidate set includes the named entities that are compatible with this pronoun. For example if the current pronoun is she then only the Person annotations with "gender" feature equal to "female" or "unknown" will be considered as candidates.
- From all candidates, one is chosen according to evaluation criteria specific for the pronoun.

Coreference chain generation

This step is actually performed by the main module. After executing each of the submodules on the current document, the coreference module follows the steps:

- Retrieves the anaphor/antecedent pairs generated from them.
- For each pair, the orthographic matches (if any) of the antecedent entity is retrieved and then extended with the anaphor of the pair (i.e. the pronoun). The result is the coreference chain for the entity. The coreference chain contains the IDs of the annotations (entities) that co-refer.
- A new Coreference annotation is created for each chain. The annotation contains a single feature "matches" which value is the coreference chain (the list with IDs). The annotations are exported in a pre-specified annotation set.

The resolution for she, her, her$, he, him, his, herself and himself is similar because the analysis of the corpus showed that these pronouns are related to their antecedents in similar manner. The characteristics of the resolution process are:

- Context inspected is not very big - cases where the antecedent is found more than 3 sentences further back than the anaphor are rare.
- Recency factor is heavily used - the candidate antecedents that appear closer to the anaphor in the text are scored better.
- Anaphora have higher priority than cataphora. If there is an anaphoric candidate and a cataphoric one, then the anaphoric one is preferred, even if the recency factor scores the cataphoric candidate better.

The resolution process performs the following steps:

- Inspect the context of the anaphor for candidate antecedents. A candidate is considered every Person annotation. Cases where she/her refers to inanimate entity (ship for example) are not handled.

- For each candidate perform a gender compatibility check - only candidates having "gender" feature equal to "unknown" or compatible with the pronoun are considered for further evaluation.
- Evaluate each candidate with the best candidate so far. If the two candidates are anaphoric for the pronoun then choose the one that appears closer. The same holds for the case where the two candidates are cataphoric relative to the pronoun. If one is anaphoric and the other is cataphoric then choose the former, even if the latter appears closer to the pronoun.

Resolution of it, its, itself

This set of pronouns also shares many common characteristics. The resolution process contains certain differences with the one for the previous set of pronouns. Successful resolution for it, its, itself is more difficult because of the following factors:

- There is no gender compatibility restriction. In the case when there are several candidates in the context, the gender compatibility restriction is very useful for rejecting some of the candidates. When no such restriction exists, and with the lack of any syntactic or ontological information about the entities in the context, the recency factor plays the major role for choosing the best antecedent.
- The number of nominal antecedents (i.e. entities that are referred not by name) is much higher compared to the number of such antecedents for she, he, etc. In this case trying to find antecedent only amongst named entities degrades the precision a lot.

Resolution of I, me, my, myself

Resolution of these pronouns is dependent on the work of the quoted speech submodule. One important difference from the resolution process of other pronouns is that the context is not measured in sentences but depends solely on the quote span. Another difference is that the context is not contiguous - the quoted fragment itself is excluded from the context, because it is unlikely that an antecedent for I, me, etc. appears there. The context itself consists of:

- the part of the sentence where the quoted fragment originates, that is not contained in the quote - i.e. the text prior to the quote;
- the part of the sentence where the quoted fragment ends, that is not contained in the quote - i.e. the text following the quote;
- the part of the sentence preceding the sentence where the quote originates, which is not included in other quote.

It is worth noting that contrary to other pronouns, the antecedent for I, me, my and myself is most often cataphoric or if anaphoric it is not in the same sentence with the quoted fragment.

The resolution algorithm consists of the following steps:

- Locate the quoted fragment description that contains the pronoun. If the pronoun is not contained in any fragment then return without proposing an antecedent.
- Inspect the context for the quoted fragment (as defined above) for candidate antecedents. Candidates are considered annotations of type Pronoun or annotations of type Token with features category = "PRP", string = "she" or category = "PRP", string = "he".
- Try to locate a candidate in the text succeeding the quoted fragment (first pattern). If more than one candidate is present, choose the closest to the end of the quote. If a candidate is found then propose it as antecedent and exit.
- Try to locate candidate in the text preceding the quoted fragment (third pattern). Choose the closes one to the beginning of the quote. If found then set as antecedent and exit.
- Try to locate antecedents in the unquoted part of the sentence preceding the sentence where the quote starts (second pattern). Give preference to the one closest to the end of the quote (if any) in the preceding sentence or closest to the sentence beginning.

8.9 A Walk-Through Example

Let us take an example of a 3-stage procedure using the tokeniser, gazetteer and named-entity grammar. Suppose we wish to recognise the phrase “800,000 US dollars” as an entity of type “Number”, with the feature “money”.

First of all, we give an example of a grammar rule (and corresponding macros) for money, which would recognise this type of pattern.

```
Macro: MILLION_BILLION
({Token.string == "m"}|
{Token.string == "million"}|
{Token.string == "b"}|
{Token.string == "billion"}
)

Macro: AMOUNT_NUMBER
```

```
({Token.kind == number}
(({Token.string == ","}|
  {Token.string == "."})
{Token.kind == number})*
(({SpaceToken.kind == space})?
 (MILLION_BILLION)?)
)

Rule: Money1
// e.g. 30 pounds
  (
      (AMOUNT_NUMBER)
      (SpaceToken.kind == space)?
      ({Lookup.majorType == currency_unit})
  )
 :money  -->
  :money.Number = {kind = "money", rule = "Money1"}
```

8.9.1 Step 1 - Tokenisation

The tokeniser separates this phrase into the following tokens. In general, a word is comprised of any number of letters of either case, including a hyphen, but nothing else; a number is composed of any sequence of digits; punctuation is recognised individually (each character is a separate token), and any number of consecutive spaces and/or control characters are recognised as a single spacetoken.

```
Token, string = ‘‘800’’, kind = number, length = 3
Token, string = ‘‘,’’, kind = punctuation, length = 1
Token, string = ‘‘000’’, kind = number, length = 3
SpaceToken, string = ‘‘ ’’, kind = space, length = 1
Token, string = ‘‘US’’, kind = word, length = 2, orth = allCaps
SpaceToken, string = ‘‘ ’’, kind = space, length = 1
Token, string = ‘‘dollars’’, kind = word, length = 7, orth = lowercase
```

8.9.2 Step 2 - List Lookup

The gazetteer lists are then searched to find all occurrences of matching words in the text. It finds the following match for the string “US dollars”:

```
Lookup, minorType = post_amount, majorType = currency_unit
```

8.9.3 Step 3 - Grammar Rules

The grammar rule for money is then invoked. The macro MILLION_BILLION recognises any of the strings "m", "million", "b", "billion". Since none of these exist in the text, it passes onto the next macro. The AMOUNT_NUMBER macro recognises a number, optionally followed by any number of sequences of the form"dot or comma plus number", followed by an optional space and an optional MILLION_BILLION. In this case, "800,000" will be recognised. Finally, the rule Money1 is invoked. This recognises the string identified by the AMOUNT_NUMBER macro, followed by an optional space, followed by a unit of currency (as determined by the gazetteer). In this case, "US dollars" has been identified as a currency unit, so the rule Money1 recognises the entire string "800,000 US dollars". Following the rule, it will be annotated as a Number entity of type Money:

```
Number, kind = money, rule = Money1
```

Chapter 9

(More CREOLE) Plugins

For the previous reader was none other than myself. I had already read this book long ago.

The old sickness has me in its grip again: amnesia in litteris, the total loss of literary memory. I am overcome by a wave of resignation at the vanity of all striving for knowledge, all striving of any kind. Why read at all? Why read this book a second time, since I know that very soon not even a shadow of a recollection will remain of it? Why do anything at all, when all things fall apart? Why live, when one must die? And I clap the lovely book shut, stand up, and slink back, vanquished, demolished, to place it again among the mass of anonymous and forgotten volumes lined up on the shelf.

...

But perhaps - I think, to console myself - perhaps reading (like life) is not a matter of being shunted on to some track or abruptly off it. Maybe reading is an act by which consciousness is changed in such an imperceptible manner that the reader is not even aware of it. The reader suffering from amnesia in litteris is most definitely changed by his reading, but without noticing it, necause as he reads, those critical faculties of his brain that could tell him that change is occurring are changing as well. And for one who is himself a writer, the sickness may conceivably be a blessing, indeed a necessary precondition, since it protects him against that crippling awe which every great work of literature creates, and because it allows him to sustain a wholly uncomplicated relationship to plagiarism, without which nothing original can be created.

Three Stories and a Reflection, Patrick Suskind, 1995 (pp. 82, 86).

This chapter describes additional CREOLE resources which do not form part of ANNIE.

9.1 Document Reset

The document reset resource enables the document to be reset to its original state, by removing all the annotation sets and their contents, apart from the one containing the document format analysis (Original Markups). An optional parameter, keepOriginalMarkupsAS, allows users to decide whether to keep the Original Markups AS or not while reseting the document. This resource is normally added to the beginning of an application, so that a document is reset before an application is rerun on that document.

9.2 Verb Group Chunker

The rule-based verb chunker is based on a number of grammars of English [Cobuild 99, Azar 89]. We have developed 68 rules for the identification of non recursive verb groups. The rules cover finite ('is investigating'), non-finite ('to investigate'), participles ('investigated'), and special verb constructs ('is going to investigate'). All the forms may include adverbials and negatives. The rules have been implemented in JAPE. The finite state analyser produces an annotation of type 'VG' with features and values that encode syntactic information ('type', 'tense', 'voice', 'neg', etc.). The rules use the output of the POS tagger as well as information about the identity of the tokens (e.g. the token 'might' is used to identify modals).

The grammar for verb group identification can be loaded as a Jape grammar into the GATE architecture and can be used in any application: the module is domain independent.

9.3 Noun Phrase Chunker

The NP Chunker application is a Java implementation of the Ramshaw and Marcus BaseNP chunker (in fact the files in the resources directory are taken straight from their original distribution) which attempts to insert brackets marking noun phrases in text which have been marked with POS tags in the same format as the output of Eric Brill's transformational tagger. The output from this version should be identical to the output of the oringinal C++/Perl version released by Ramshaw and Marcus.

For more information about baseNP structures and the use of tranformation-based learning to derive them, see [Ramshaw & Marcus 95].

9.3.1 Differences from the Original

The major difference is the assumption is made that if a POS tag is not in the mapping file then it is tagged as 'I'. The original version simply failed if an unknown POS tag was encountered. When using the GATE wrapper the chunk tag can be changed from 'I' to any other legal tag (B or O) by setting the unknownTag parameter.

9.3.2 Using the Chunker

The Chunker requires the Creole plugin "NP_Chunking" to be loaded. The two loadtime parameters are simply urls pointing at the POS tag dictionary and the rules file, which should be set automatically. There are five runtime parameters which should be set prior to executing the chunker.

- annotationName: name of the annotation the chunker should create to identify noun phrases in the text.
- inputASName: The chunker requires certain types of annotations (e.g. Tokens with part of speech tags) for identifying noun chunks. This parameter tells the chunker which annotation set to use to obtain such annotations from.
- outputASName: This is where the results (i.e. new noun chunk annotations will be stored).
- posFeature: Name of the feature that holds POS tag information. '
- unknownTag: it works as specified in the previous section.

The chunker requires the following PRs to have been run first: tokeniser, sentence splitter, POS tagger.

9.4 OntoText Gazetteer

The OntoText Gazetteer is a Natural Gazetteer, implemented from the OntoText Lab (http://www.ontotext.com/). Its implementaion is based on simple lookup in several java.util.HashMap, and is inspired by the strange idea of Atanas Kiryakov, that searching in HashMaps will be faster than a search in a Finite State Machine (FSM).

Here follows a description of the algorithm that lies behind this implementation:

Every phrase i.e. every list entry is separated into several parts. The parts are determined by the whitespaces lying among them. e.g. the phrase : "form is emptiness" has three parts

: form, is & emptiness. There is also a list of HashMaps: mapsList which has as many elements as the longest (in terms of "count of parts") phrase in the lists. So the first part of a phrase is placed in the first map. The first part + space + second part is placed in the second map, etc. The full phrase is placed in the appropriate map, and a reference to a Lookup object is attached to it.

On first sight it seems that this algorithm is certainly much more memory-consuming than a finite state machine (FSM) with the parts of the phrases as transitions, but this is actually not so important since the average length of the phrases (in parts) in the lists is 1.1. On the other hand, one advantage of the algorithm is that, although unconventional, on average it takes four times less memory and works three times faster than an optimized FSM implementation.

The lookup part is implemented in execute() so a lot of tokenization takes place there. After defining the candidates for phrase-parts, we build a candidate phrase and try to look it up in the maps (in which map again depends on the count of parts in the current candidate phrase).

9.4.1 Prerequisites

The phrases to be recognised should be listed in a set of files, one for each type of occurrence (as for the standard gazetteer).

The gazetteer is built with the information from a file that contains the set of lists (which are files as well) and the associated type for each list. The file defining the set of lists should have the following syntax: each list definition should be written on its own line and should contain:

- the file name (required)
- the major type (required)
- the minor type (optional)
- the language(s) (optional)

The elements of each definition are separated by ":". The following is an example of a valid definition:

```
personmale.lst:person:male:english
```

Each file named in the lists definition file is just a list containing one entry per line.

When this gazetter is run over some input text (a GATE document) it will generate annotations of type Lookup having the attributes specified in the definition file.

9.4.2 Setup

In order to use this gazetteer from within GATE the following should reside in the creole setup file (creole.xml):

```
    <RESOURCE>
      <NAME>OntoText Gazetteer</NAME>
      <CLASS>com.ontotext.gate.gazetteer.NaturalGazetteer</CLASS>
      <COMMENT>A list lookup component. for documentation please refer to
(www.ontotext.com/gate/gazetteer/documentation/index.html). For licence
information please refer to
(www.ontotext.com/gate/gazetteer/documentation/licence.ontotext.html) or to
licence.ontotext.html in the lib folder of
GATE</COMMENT>
      <PARAMETER NAME="document" RUNTIME="true" COMMENT="The document to be
processed">gate.Document</PARAMETER>
      <PARAMETER NAME="annotationSetName" RUNTIME="true" COMMENT="The
annotation set to be used for the generated
annotations" OPTIONAL="true">java.lang.String</PARAMETER>
      <PARAMETER NAME="listsURL"
DEFAULT="gate:/creole/gazeteer/default/lists.def" COMMENT="The URL to the
file with list of
lists" SUFFIXES="def">java.net.URL</PARAMETER>
      <PARAMETER DEFAULT="UTF-8" NAME="encoding" COMMENT="The encoding used
for reading the
definitions">java.lang.String</PARAMETER>
      <PARAMETER DEFAULT="true" NAME="caseSensitive" COMMENT="Should this
gazetteer diferentiate on case. Currently the
Gazetteer works only in case sensitive mode.">java.lang.Boolean</PARAMETER>
      <ICON>shefGazetteer.gif</ICON>
    </RESOURCE>
```

9.5 Flexible Gazetteer

The Flexible Gazetteer provides users with the flexibility to choose their own customized input and an external Gazetteer. For example, the user might want to replace words in the text with their base forms (which is an output of the Morphological Analyser) or to segment a Chinese text (using the Chinese Tokeniser) before running the Gazetteer on the Chinese text.

The Flexible Gazetteer performs lookup over a document based on the values of an arbitrary feature of an arbitrary annotation type, by using an *externally provided* gazetteer. It is important to use an external gazetteer as this allows the use of any type of gazetteer (e.g. an Ontological gazetteer).

Input to the Flexible Gazetteer:

Runtime parameters:

- Document – the document to be processed
- **inputAnnotationSetName** The annotationSet where the Flexible Gazetteer should search for the AnnotationType.feature specified in the inputFeatureNames.
- **outputAnnotationSetName** The AnnotationSet where Lookup annotations should be placed.

  Creation time parameters:
- **inputFeatureNames** – when selected, these feature values are used to replace the corresponding original text. A temporary document is created from the values of the specified features on the specified annotation types. For example: for Token.string the temporary document will have the same content as the original one but all the SpaceToken annotations will have been replaced by single spaces.
- **gazetteerInst** – the actual gazetteer instance, which should run over a temporary document. This generates the Lookup annotations with features. This must be an instance of `gate.creole.gazetteer.Gazetteer` which has already been created.

Once the external gazetteer has annotated text with Lookup annotations, Lookup annotations on the temporary document are converted to Lookup annotations on the original document. Finally the temporary document is deleted.

9.6 Gazetteer List Collector

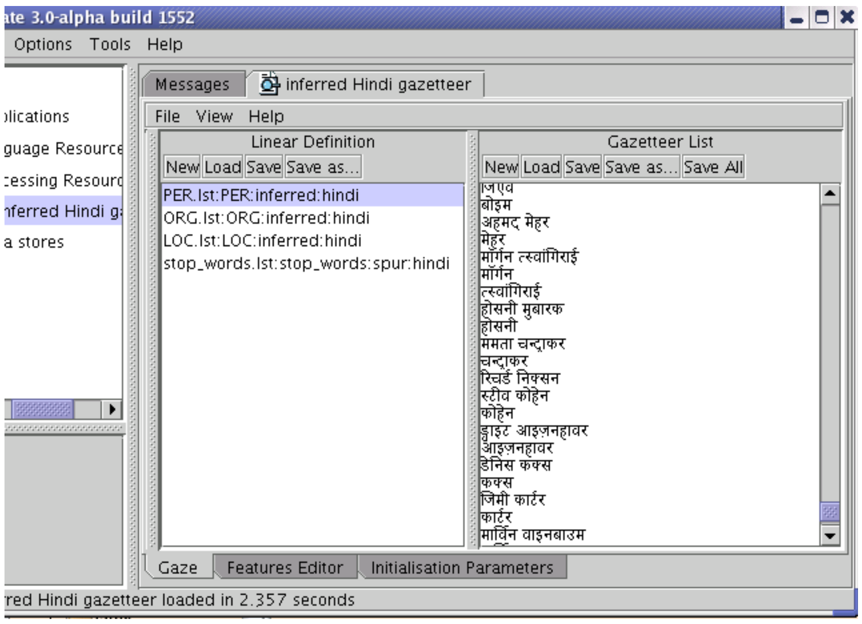
The gazetteer list collector collects occurrences of entities directly from a set of annotated training texts, and populates gazetteer lists with the entities. The entity types and structure of the gazetteer lists are defined as necessary by the user. Once the lists have been collected, a semantic grammar can be used to find the same entities in new texts.

An empty list must be created first for each annotation type, if no list exists already. The set of lists must be loaded into GATE before the PR can be run. If a list already exists, the list will simply be augmented with any new entries. The list collector will only collect one occurrence of each entry: it first checks that the entry is not present already before adding a new one.

There are 4 runtime parameters:

- annotationTypes: a list of the annotation types that should be collected
- gazetteer: the gazetteer where the results will be stored (this must be already loaded in GATE)
- markupASname: the annotation set from which the annotation types should be collected
- theLanguage: sets the language feature of the gazetteer lists to be created to the appropriate language (in the case where lists are collected for different languages)

Figure 9.1 shows a screenshot of a set of lists collected automatically for the Hindi language. It contains 4 lists: Person, Organisation, Location and a list of stopwords. Each list has a majorType whose value is the type of list, a minorType "inferred" (since the lists have been inferred from the text), and the language "Hindi".



PER.lst:PER:inferred:hindi
ORG.lst:ORG:inferred:hindi
LOC.lst:LOC:inferred:hindi
stop_words.lst:stop_words:spur:hindi

जिएव
बोइम
अहमद मेहर
मेहर
मॉर्गन त्स्वांगिराई
मॉर्गन
त्स्वांगिराई
होसनी मुबारक
होसनी
ममता चन्द्राकर
चन्द्राकर
रिचर्ड निक्सन
स्टीव कोहेन
कोहेन
ड्वाइट आइज़नहावर
आइज़नहावर
डेनिस कक्स
कक्स
जिमी कार्टर
कार्टर
मार्विन वाइनबाउम

red Hindi gazetteer loaded in 2.357 seconds

Figure 9.1: Lists collected automatically for Hindi

The list collector also has a facility to split the Person names that it collects into their individual tokens, so that it adds both the entire name to the list, and adds each of the tokens to the list (i.e. each of the first names, and the surname) as a separate entry. When the grammar annotates Persons, it can require them to be at least 2 tokens or 2 consecutive Person Lookups. In this way, new Person names can be recognised by combining a known first name with a known surname, even if they were not in the training corpus. Where only a single token is found that matches, an Unknown entity is generated, which can later be matched with an existing longer name via the orthomatcher component which performs orthographic coreference between named entities. This same procedure can also be used for other entity types. For example, parts of Organisation names can be combined together in different ways. The facility for splitting Person names is hardcoded in the file gate/src/gate/creole/GazetteerListsCollector.java and is commented.

9.7 Tree Tagger

The TreeTagger is a language-independent part-of-speech tagger, which currently supports English, French, German, Spanish, Italian and Bulgarian (although the latter two are not available in GATE). It is integrated with GATE using a GATE CREOLE wrapper, originally designed by the CLaC lab (Computational Linguistics at Concordia), Concordia University, Montreal (http://www.cs.concordia.ca/research/researchgroups/clac.php).

The GATE wrapper calls TreeTagger as an external program, passing gate Tokens as input, and adding two new features to them, which hold the features as described below:

- Features of the TreeTaggerToken:
  i) category: the part-of-speech tag of the token;
  ii) lemma: the lemma of the token
- Runtime parameters:
  i) document: the document to be processed
  ii) treeTaggerBinary: a URL indicating the location of a (language-specific) GATE TreeTagger wrapper shell script. Note that the scripts used by GATE are different from the original TreeTagger scripts (in cmd), since the latter perform their own tokenisation, whereas the GATE scripts rely on Token annotations as they have been computed by a Tokeniser component. The GATE scripts reside in plugins/TreeTagger/resources. Currently available are command scripts for German, French, and Spanish.
  iii) encoding: The character encoding to use when passing data to and from the tagger. This must be `ISO-8859-1` to work with the standard TreeTagger distribution – do not change it unless you know what you are doing.
  iv) failOnUnmappableChar: What to do if a character is encountered in the document which cannot be represented in the selected encoding. If the parameter is `true` (the default), unmappable characters cause the wrapper to throw an exception and fail. If set to `false`, unmappable characters are replaced by question marks when the document is passed to the tagger. This is useful if your documents are largely OK but contain the odd character from outside the Latin-1 range.
- Requirement: The TreeTagger, which is available from http://www.ims.uni-stuttgart.de/projekte/corplex/TreeTagger/DecisionTreeTagger.html, must be correctly installed on the same machine as GATE. It must be installed in a directory

that does not contain any spaces in its path, otherwise the scripts will fail. Once the TreeTagger is installed, the first two lines of the shell script may need to be modified to indicate the installed location of the bin and lib directories of the tagger, as shown below:

```
# THESE VARIABLES HAVE TO BE SET:
BIN=/usr/local/clactools/TreeTagger/bin
LIB=/usr/local/clactools/TreeTagger/lib
```

The TreeTagger plugin works on any platform that supports the tree tagger tool, including Linux, Mac OS X and Windows, but the GATE-specific scripts require a POSIX-style Bourne shell with the `gawk`, `tr` and `grep` commands, plus Perl for the Spanish tagger. For Windows this means that you will need to install the appropriate parts of the Cygwin environment from http://www.cygwin.com and set the system property `treetagger.sh.path` to contain the path to your `sh.exe` (typically `C:\cygwin\bin\sh.exe`). If this property is set, the TreeTagger plugin runs the shell given in the property and passes the tagger script as its first argument; without the property, the plugin will attempt to run the shell script directly, which fails on Windows with a cryptic "error=193". For the GATE GUI, put the following line in `build.properties` (see section 3.3, and note the extra backslash before each backslash and colon in the path):

```
run.treetagger.sh.path: C\:\\cygwin\\bin\\sh.exe
```

Figure 9.2 shows a screenshot of a French document processed with the TreeTagger.

9.7.1 POS tags

For English the POS tagset is a slightly modified version of the Penn Treebank tagset, where the second letter of the tags for verbs distinguishes between "be" verbs (B), "have" verbs (H) and other verbs (V).

The tagsets for French, German, Italian, Spanish and Bulgarian can be found in the original TreeTagger documenation at http://www.ims.uni-stuttgart.de/projekte/corplex/TreeTagger/Decision

9.8 Stemmer

The stemmer plugin consists of a set of stemmers PRs for the following 11 European languages: Danish, Dutch, English, Finnish, French, German, Italian, Norwegian, Portuguese, Russian, Spanish and Swedish. These take the form of wrappers for the Snowball stemmers freely available from http://snowball.tartarus.org. Each Token is annotated with a new feature "stem", with the stem for that word as its value. The stemmers should be run as other PRs, on a document that has been tokenised.

There are three runtime parameters which should be set prior to executing the stemmer on a document.

- annotationType: This is the type of annotations that represent tokens in the document. Default value is set to "Token".
- annotationFeature: This is the name of a feature that contains tokens' strings. The stemmer uses value of this feature as a string to be stemmed. Default value is set to "string".
- annotationSetName: This is where the stemmer expects the annotations of type as specified in the annotationType parameter to be.

9.8.1 Algorithms

The stemmers are based on the Porter stemmer for English [Porter 80], with rules implemented in Snowball e.g.

```
define Step_1a as
( [substring] among (
 'sses' (<-'ss')
'ies' (<-'i')
'ss' () 's'  (delete)
 )
```

9.9 GATE Morphological Analyzer

The Morphological Analyser PR can be found in the Tools plugin. It takes as input a tokenized GATE document. Considering one token and its part of speech tag, one at a time, it identifies its lemma and an affix. These values are than added as features on the Token annotation. Morpher is based on certain regular expression rules. These rules were originally implemented by *Kevin Humphreys* in GATE1 in a programming language called *Flex*. Morpher has a capability to interepret these rules with an extension of allowing users to add new rules or modify the existing ones based on their requirements. In order to allow these operations with as little effort as possible, we changed the way these rules are written. More information on how to write these rules is explained later in Section 9.9.1.

Two types of parameters, Init-time and run-time, are required to instantiate and execute the PR.

- rulesFile (Init-time) The rule file has several regular expression patterns. Each pattern has two parts, L.H.S. and R.H.S. L.H.S. defines the regular expression and R.H.S. the function name to be called when the pattern matches with the word under consideration. Please see 9.9.1 for more information on rule file.
- caseSensitive (init-time) By default, all tokens under consideration are converted into lowercase to identify their lemma and affix. If the user selects *caseSensitive* to be *true*, words are no longer converted into lowercase.
- document (run-time) Here the document must be an instance of a GATE document.
- affixFeatureName Name of the feature that should hold the affix value.
- rootFeatureName Name of the feature that should hold the root value.
- annotationSetName Name of the annotationSet that contains Tokens.
- considerPOSTag Each rule in the rule file has a separate tag, which specifies which rule to consider with what part-of-speech tag. If this option is set to false, all rules are considered and matched with all words. This option is very useful. For example if the word under consideration is ”singing”. ”singing” can be used as a noun as well as a verb. In the case where it is identified as a verb, the lemma of the same would be ”sing” and the affix ”ing”, but otherwise there would not be any affix.

9.9.1 Rule File

GATE provides a default rule file, called *default.rul*, which is available under the *gate/plugins/Tools/morph/resources* directory. The rule file has two sections.

1. Variables
2. Rules

Variables

The user can define various types of variables under the section *defineVars*. These variables can be used as part of the regular expressions in rules. There are three types of variables:

1. Range With this type of variable, theuser can specify the range of characters. e.g. A ==> [-a-z0-9]

2. Set With this type of variable, user can also specify a set of characters, where one character at a time from this set is used as a value for the given variable. When this variable is used in any regular expression, all values are tried one by one to generate the string which is compared with thecontents of the document. e.g. A ==> [abcdqurs09123]

3. Strings Where in the two types explained above, variables can hold only one character from the given set or range at a time, this allows specifying strings as possibilities for the variable. e.g. A ==> "bb" OR "cc" OR "dd"

Rules

All rules are declared under the section *defineRules*. Every rule has two parts, LHS and RHS. The LHS specifies the regular expresssion and the RHS the function to be called when the LHS matches with the given word. "==>" is used as delimeter between the LHS and RHS.

The LHS has the following syntax:

$<"*"—"verb"—"noun"><regularexpression>$.

User can specify which rule to be considered when the word is identified as "verb" or "noun". "*" indicates that the rule should be considered for all part-of-speech tags. If the part-of-speech should be used to decide if the rule should be considered or not can be enabled or disabled by setting the value of *considerPOSTags* option. Combination of any string along with any of the variables declared under the *defineVars* section and also the Klene operators, "+" and "*", can be used to generate the regular expressions. Below we give few examples of L.H.S. expressions.

- <verb>"bias"
- <verb>"canvas"{ESEDING} "ESEDING" is a variable defined under the *defineVars* section. Note: variables are enclosed with "{" and "}".
- <noun>({A}*"metre") "A" is a variable followed by the Klene operator "*", which means "A" can occur zero or more times.
- <noun>({A}+"itis") "A" is a variable followed by the Klene operator "+", which means "A" can occur one or more times.
- $<*>$"aches" "$<*>$" indicates that the rule should be considered for all part-of-speech tags.

On the RHS of the rule, the user has to specify one of the functions from those listed below. These rules are hard-coded in the Morph PR in GATE and are invoked if the regular expression on the LHS matches with any particular word.

- stem(*n*, *string*, *affix*) Here,
  - *n* = number of characters to be truncated from the end of the string.
  - *string* = the string that should be concatenated after the word to produce the root.
  - *affix* = affix of the word
- irreg_stem(*root*, *affix*) Here,
  - *root* = root of the word
  - *affix* = affix of the word
  - null_stem() This means words are themselves the base forms and should not be analyzed.
- semi_reg_stem(*n*,*string*) *semir_reg_stem* function is used with the regular expressions that end with any of the {EDING} or {ESEDING} variables defined under the variable section. If the regular expression matches with the given word, this function is invoked, which returns the value of variable (i.e. {EDING} or {ESEDING}) as an affix. To find a lemma of the word, it removes the *n* characters from the back of the word and adds the *string* at the end of the word.

9.10 MiniPar Parser

MiniPar is a shallow parser. In its shipped version, it takes one sentence as an input and determines the dependency relationships between the words of a sentence. It parses the sentence and brings out the information such as:

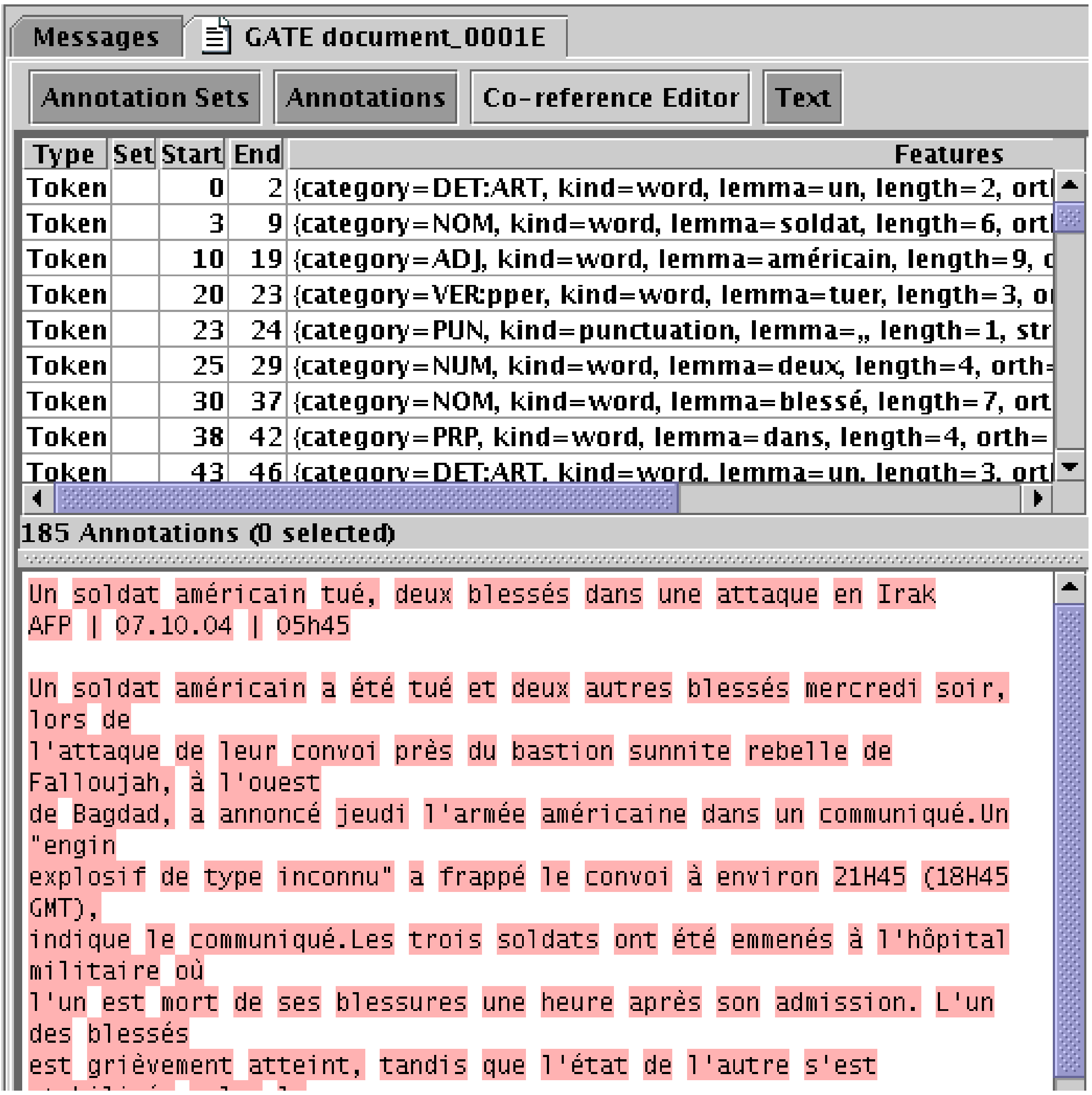
- the lemma of the word;
- the part of speech of the word;
- the head modified by this word;
- name of the dependency relationship between this word and the head;
- the lemma of the head.

In the version of MiniPar integrated in GATE, it generates annotations of type "DepTreeNode" and the annotations of type "[relation]" that exists between the head and the child node. The document is required to have annotations of type "Sentence", where each annotation consists of a string of the sentence.

Minipar takes one sentence at a time as an input and generates the tokens of type "DepTreeNode". Later it assigns relation between these tokens. Each DepTreeNode consists of feature called "word": this is the actual text of the word.

For each and every annotation of type "[Rel]", where 'Rel' is obj, pred etc. This is the name of the dependency relationship between the child word and the head word (see Section 9.10.5). Every "[Rel]" annotation is assigned four features:

- **child_word**: this is the text of the child annotation;
- **child_id**: IDs of the annotations which modify the current word (if any).
- **head_word**: this is the text of the head annotation;
- **head_id**: ID of the annotation modified by the child word (if any);



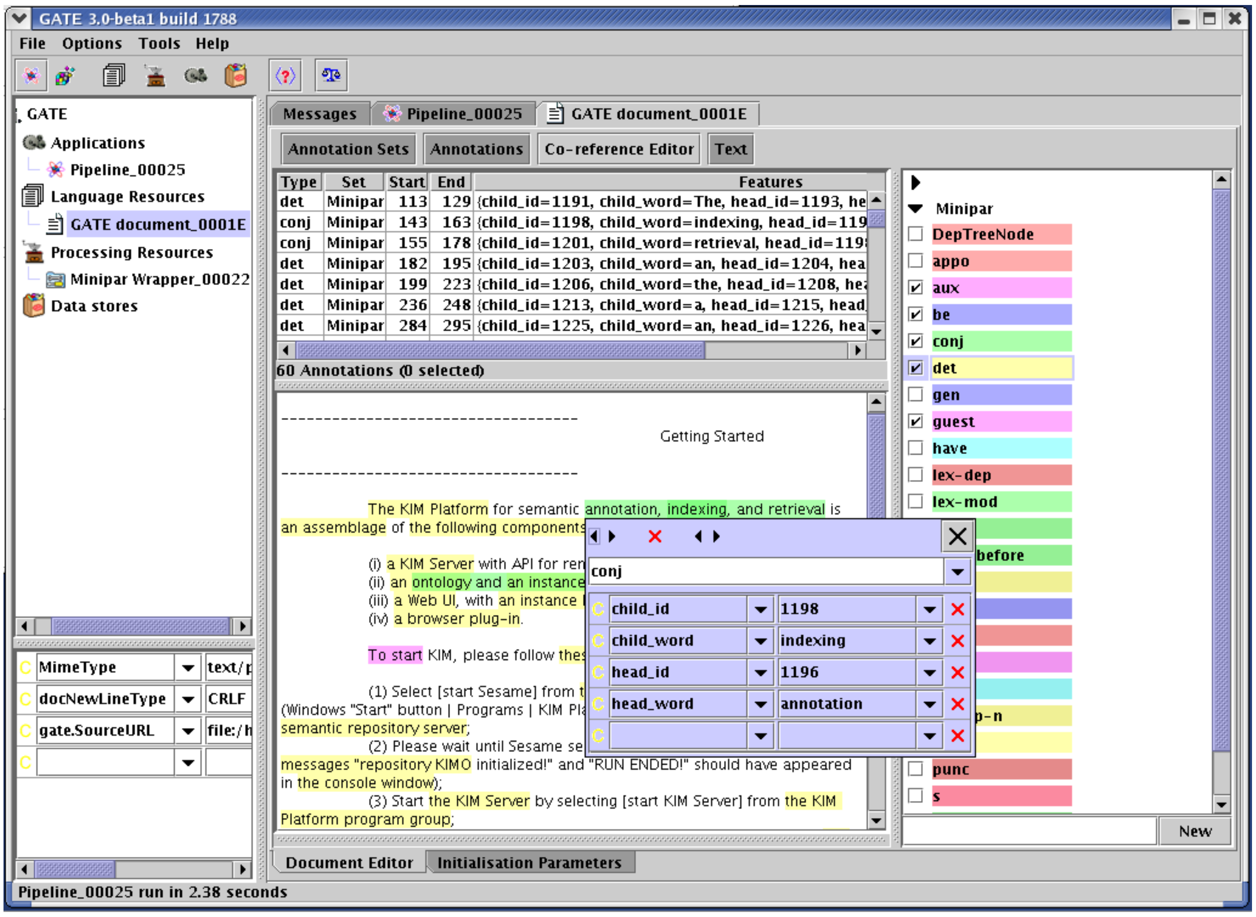
Type	Set	Start	End	Features
Token		0	2	{category=DET:ART, kind=word, lemma=un, length=2, ort
Token		3	9	{category=NOM, kind=word, lemma=soldat, length=6, ort
Token		10	19	{category=ADJ, kind=word, lemma=américain, length=9, c
Token		20	23	{category=VER:pper, kind=word, lemma=tuer, length=3, o
Token		23	24	{category=PUN, kind=punctuation, lemma=,, length=1, str
Token		25	29	{category=NUM, kind=word, lemma=deux, length=4, orth=
Token		30	37	{category=NOM, kind=word, lemma=blessé, length=7, ort
Token		38	42	{category=PRP, kind=word, lemma=dans, length=4, orth=
Token		43	46	{category=DET:ART, kind=word, lemma=un, length=3, ort

185 Annotations (0 selected)

```
Un soldat américain tué, deux blessés dans une attaque en Irak
AFP | 07.10.04 | 05h45

Un soldat américain a été tué et deux autres blessés mercredi soir,
lors de
l'attaque de leur convoi près du bastion sunnite rebelle de
Falloujah, à l'ouest
de Bagdad, a annoncé jeudi l'armée américaine dans un communiqué.Un
"engin
explosif de type inconnu" a frappé le convoi à environ 21H45 (18H45
GMT),
indique le communiqué.Les trois soldats ont été emmenés à l'hôpital
militaire où
l'un est mort de ses blessures une heure après son admission. L'un
des blessés
est grièvement atteint, tandis que l'état de l'autre s'est
```

Figure 9.2: a TreeTagger processed document

Figure 9.3 shows a MiniPar annotated document in GATE.



Type	Set	Start	End	Features
det	Minipar	113	129	{child_id=1191, child_word=The, head_id=1193, he
conj	Minipar	143	163	{child_id=1198, child_word=indexing, head_id=119
conj	Minipar	155	178	{child_id=1201, child_word=retrieval, head_id=119
det	Minipar	182	195	{child_id=1203, child_word=an, head_id=1204, hea
det	Minipar	199	223	{child_id=1206, child_word=the, head_id=1208, he
det	Minipar	236	248	{child_id=1213, child_word=a, head_id=1215, head
det	Minipar	284	295	{child_id=1225, child_word=an, head_id=1226, hea

60 Annotations (0 selected)

```
-----------------------------------
                                        Getting Started
-----------------------------------

            The KIM Platform for semantic annotation, indexing, and retrieval is
an assemblage of the following components

            (i) a KIM Server with API for ren
            (ii) an ontology and an instance
            (iii) a Web UI, with an instance l
            (iv) a browser plug-in.

            To start KIM, please follow thes

            (1) Select [start Sesame] from t
(Windows "Start" button | Programs | KIM Pla
semantic repository server;
            (2) Please wait until Sesame se
messages "repository KIMO initialized!" and "RUN ENDED!" should have appeared
in the console window);
            (3) Start the KIM Server by selecting [start KIM Server] from the KIM
Platform program group;
```

conj

child_id	1198
child_word	indexing
head_id	1196
head_word	annotation

MimeType	text/p
docNewLineType	CRLF
gate.SourceURL	file:/h

Pipeline_00025 run in 2.38 seconds

Figure 9.3: a MiniPar annotated document

9.10.1 Platform Supported

MiniPar in GATE is supported for the Linux and Windows operating systems. Trying to instantiate this PR on any other OS will generate the ResourceInstantiationException.

9.10.2 Resources

MiniPar in GATE is shipped with four basic resources:

- **MiniparWrapper.jar**: this is a JAVA Wrapper for MiniPar;
- **creole.XML**: this defines the required parameters for MiniPar Wrapper;

- **minipar.linux**: this is a modified version of pdemo.cpp.
- **minipar-windows.exe** : this is a modified version of pdemo.cpp compiled to work on windows.

9.10.3 Parameters

The MiniPar wrapper takes six parameters:

- **annotationTypeName**: new annotations are created with this type, default is ”DepTreeNode”;
- **annotationInputSetName**: annotations of Sentence type are provided as an input to MiniPar and are taken from the given annotationSet;
- **annotationOutputSetName**: All annotations created by Minipar Wrapper are stored under the given annotationOutputSet;
- **document**: the GATE document to process;
- **miniparBinary**: location of the MiniPar Binary file (i.e. either minipar.linux or minipar-windows.exe. These files are available under gate/plugins/minipar/ directory);
- **miniparDataDir**: location of the ”data” directory under the installation directory of MINIPAR. default is ”%MINIPAR_HOME%/data”.

9.10.4 Prerequisites

The MiniPar wrapper requires the MiniPar library to be available on the underlying Linux/Windows machine. It can be downloaded from the MiniPar homepage.

9.10.5 Grammatical Relationships

```
appo    "ACME president, --appo-> P.W. Buckman"
aux "should <-aux-- resign"
be  "is <-be-- sleeping"
c   "that <-c-- John loves Mary"
comp1   first complement
det "the <-det ‘-- hat"
gen "Jane’s <-gen-- uncle"
i   the relationship between a C clause and its I clause
inv-aux     inverted auxiliary: "Will <-inv-aux-- you stop it?"
```

```
inv-be      inverted be: "Is <-inv-be-- she sleeping"
inv-have    inverted have: "Have <-inv-have-- you slept"
mod the relationship between a word and its adjunct modifier
pnmod       post nominal modifier
p-spec      specifier of prepositional phrases
pcomp-c     clausal complement of prepositions
pcomp-n     nominal complement of prepositions
post        post determiner
pre         pre determiner
pred        predicate of a  clause
rel         relative clause
vrel        passive verb modifier of nouns
wha, whn, whp:  wh-elements at C-spec positions
obj         object of verbs
obj2    second object of ditransitive verbs
subj    subject of verbs
s   surface subjec
```

9.11 RASP Parser

RASP (Robust Accurate Statistical Parsing) is a robust parsing system for English, developed by the Natural Language and Computational Linguistics group at the University of Sussex.

This plugin, developed by DigitalPebble, provides four wrapper PRs that call the RASP modules as external programs, as well as a JAPE component that translates the output of the ANNIE POS Tagger (section 8.5).

RASP2 Tokenizer This PR requires `Sentence` annotations and creates `Token` annotations with a `string` feature. Note that sentence-splitting must be carried out before tokenization; the the RegEx Sentence Splitter (see section 8.4) is suitable for this. (Alternatively, you can use the ANNIE Tokenizer (section 8.1) and then the ANNIE Sentence Splitter (section 8.3); their output is compatible with the other PRs in this plugin).

RASP2 POS Tagger This requires `Token` annotations and creates `WordForm` annotations with `pos`, `probability`, and `string` features.

RASP2 Morphological Analyser This requires `WordForm` annotations (from the POS Tagger) and adds `lemma` and `suffix` features.

RASP2 Parser This requires the preceding annotation types and creates multiple `Dependency` annotations to represent a parse of each sentence.

RASP POS Converter This PR requires `Token` annotations with a `category` feature as produced by the ANNIE POS Tagger (see section 8.5 and creates `WordForm` annotations in the RASP Format. The ANNIE POS Tagger and this Converter can together be used as a substitute for the RASP2 POS Tagger.

Here are some examples of corpus pipelines that can be correctly constructed with these PRs.

1. RegEx Sentence Splitter
2. RASP2 Tokenizer
3. RASP2 POS Tagger
4. RASP2 Morphological Analyser
5. RASP2 Parser

1. RegEx Sentence Splitter
2. RASP2 Tokenizer
3. ANNIE POS Tagger
4. RASP POS Converter
5. RASP2 Morphological Analyser
6. RASP2 Parser

1. ANNIE Tokenizer
2. ANNIE Sentence Splitter
3. RASP2 POS Tagger
4. RASP2 Morphological Analyser
5. RASP2 Parser

1. ANNIE Tokenizer
2. ANNIE Sentence Splitter
3. ANNIE POS Tagger
4. RASP POS Converter

5. RASP2 Morphological Analyser
6. RASP2 Parser

Futher documentation is included in the directory `gate/plugins/rasp/doc/`.

The RASP package, which provides the external programs, is available from the RASP web page.

RASP is only supported for Linux operating systems. Trying to run it on any other operating systems will generate an exception with the message: "The RASP cannot be run on any other operating systems except Linux."

It must be correctly installed on the same machine as GATE, and must be installed in a directory whose path does not contain any spaces (this is a requirement of the RASP scripts as well as the wrapper). Before trying to run scripts for the first time, edit `rasp.sh` and `rasp_parse.sh` to set the correct value for the shell variable RASP, which should be the file system pathname where you have installed the RASP tools (for example, `RASP=/opt/RASP` or `RASP=/usr/local/RASP`. You will need to enter the same path for the initialization parameter `raspHome` for the POS Tagger, Morphological Analyser, and Parser PRs.

(On some systems the `arch` command used in the scripts is not available; a work-around is to comment that line out and add `arch='ix86_linux'`, for example.)

(The previous version of the RASP plugin can now be found in `plugins/Obsolete/rasp`.)

9.12 SUPPLE Parser (formerly BuChart)

The BuChart parser has been removed and replaced by SUPPLE: The Sheffield University Prolog Parser for Language Engineering. If you have an application which uses BuChart and wish to upgrade to a later version of GATE than 3.1 you must upgrade your application to use SUPPLE.

SUPPLE is a bottom-up parser that constructs syntax trees and logical forms for English sentences. The parser is complete in the sense that every analysis licensed by the grammar is produced. In the current version only the 'best' parse is selected at the end of the parsing process. The English grammar is implemented as an attribute-value context free grammar which consists of subgrammars for noun phrases (NP), verb phrases (VP), prepositional phrases (PP), relative phrases (R) and sentences (S). The semantics associated with each grammar rule allow the parser to produce logical forms composed of unary predicates to denote entities and events (e.g., *chase(e1)*, *run(e2)*) and binary predicates for properties (e.g. *lsubj(e1,e2)*). Constants (e.g., $e1$, $e2$) are used to represent entity and event identifiers. The GATE SUPPLE Wrapper stores syntactic infomation produced by the parser in the gate document in the form of `parse` annotations containing a bracketed representation of the parse; and `semantics` annotations that contains the logical forms produced by the parser. It also produces `SyntaxTreeNode` annotations that allow viewing of the parse tree for a sentence (see section 9.12.4).

9.12.1 Requirements

The SUPPLE parser is written in Prolog, so you will need a Prolog interpreter to run the parser. A copy of PrologCafe (http://kaminari.scitec.kobe-u.ac.jp/PrologCafe/), a pure Java Prolog implementation, is provided in the distribution. This should work on any platform but it is not particularly fast. SUPPLE also supports the open-source SWI Prolog (http://www.swi-prolog.org) and the commercially licenced SICStus prolog (http://www.sics.se/sicstus, SUPPLE supports versions 3 and 4), which are available for Windows, Mac OS X, Linux and other Unix variants. For anything more than the simplest cases we recommend installing one of these instead of using PrologCafe.

9.12.2 Building SUPPLE

The SUPPLE plugin must be compiled before it can be used, so you will require a suitable Java SDK (GATE itself requires only the JRE to run). To build SUPPLE, first edit the file `build.xml` in the `SUPPLE` directory under `plugins`, and adjust the user-configurable options at the top of the file to match your environment. In particular, if you are using SWI or SICStus Prolog, you will need to change the `swi.executable` or `sicstus.executable` property to the correct name for your system. Once this is done, you can build the plugin by opening a command prompt or shell, going to the `SUPPLE` directory and runing:

```
../../bin/ant swi
```

(on Windows, use `..\..\bin\ant`). For PrologCafe or SICStus, replace `swi` with `plcafe` or `sicstus` as appropriate.

9.12.3 Running the parser in GATE

In order to parse a document you will need to construct an application that has:

- tokeniser
- splitter
- POS-tagger

- Morphology
- SUPPLE Parser with parameters

  mapping file (config/mapping.config)

  feature table file (config/feature_table.config)

  parser file (supple.plcafe or supple.sicstus or supple.swi)

  prolog implementation (shef.nlp.supple.prolog.PrologCafe, shef.nlp.supple.prolog.SICStusPro
  shef.nlp.supple.prolog.SICStusProlog4, shef.nlp.supple.prolog.SWIProlog or shef.nlp.supple.prolo

  You can take a look at build.xml to see examples of invocation for the different implementations.

Note that prior to GATE 3.1, the parser file parameter was of type `java.io.File`. From 3.1 it is of type `java.net.URL`. If you have a saved application (.gapp file) from before GATE 3.1 which includes SUPPLE it will need to be updated to work with the new version. Instructions on how to do this can be found in the README file in the SUPPLE plugin directory.

9.12.4 Viewing the parse tree

GATE provides a syntax tree viewer in the `Tools` plugin which can display the parse tree generated by SUPPLE for a sentence. To use the tree viewer, be sure that the `Tools` plugin is loaded, then open a document that has been processed with SUPPLE and view its `Sentence` annotations. Right-click on the relevant `Sentence` annotation in the annotations table and select "Edit with syntax tree viewer". This viewer can also be used with the constituency output of the Stanford Parser PR (section 9.13).

9.12.5 System properties

The `SICStusProlog` (3 and 4) and `SWIProlog` implementations work by calling the native prolog executable, passing data back and forth in temporary files. The location of the prolog executable is specified by a system property:

- for SICStus: `supple.sicstus.executable` - default is to look for `sicstus.exe` (Windows) or `sicstus` (other platforms) on the PATH.
- for SWI: `supple.swi.executable` - default is to look for `plcon.exe` (Windows) or `swipl` (other platforms) on the PATH.

[1]shef.nlp.supple.prolog.SICStusProlog exists for backwards compatibility and behaves the same as SICStusProlog3.

If your prolog is installed under a different name, you should specify the correct name in the relevant system property. For example, when installed from the source distribution, the Unix version of SWI prolog is typically installed as `pl`, most binary packages install it as `swipl`, though some use the name `swi-prolog`. You can also use the properties to specify the full path to prolog (e.g. `/opt/swi-prolog/bin/pl`) if it is not on your default PATH.

For details of how to pass system properties to the GATE GUI, see the end of section 3.3.

9.12.6 Configuration files

Two files are used to pass information from GATE to the SUPPLE parser: the *mapping* file and the *feature table* file.

Mapping file

The mapping file specifies how annotations produced using Gate are to be passed to the parser. The file is composed of a number of pairs of lines, the first line in a pair specifies a Gate annotation we want to pass to the parser. It includes the AnnotationSet (or default), the AnnotationType, and a number of features and values that depend on the AnnotationType. The second line of the pair specifies how to encode the Gate annotation in a SUPPLE syntactic category, this line also includes a number of features and values. As an example consider the mapping:

```
Gate;AnnotationType=Token;category=DT;string=&S
SUPPLE;category=dt;m_root=&S;s_form=&S
```

It specifies how a determinant ('DT') will be translated into a category 'dt' for the parser. The construct '&S' is used to represent a variable that will be instantiated to the appropriate value during the mapping process. More specifically a token like 'The' recognised as a DT by the POS-tagging will be mapped into the following category:

```
dt(s_form:'The',m_root:'The',m_affix:'_',text:'_').
```

As another example consider the mapping:

```
Gate;AnnotationType=Lookup;majorType=person_first;minorType=female;string=&S
SUPPLE;category=list_np;s_form=&S;ne_tag=person;ne_type=person_first;gender=female
```

It specified that an annotation of type 'Lookup' in Gate is mapped into a category 'list_np' with specific features and values. More specifically a token like 'Mary' identified in Gate as a Lookup will be mapped into the following SUPPLE category:

```
list_np(s_form:'Mary',m_root:'_',m_affix:'_',
text:'_',ne_tag:'person',ne_type:'person_first',gender:'female').
```

Feature table

The feature table file specifies SUPPLE 'lexical' categories and its features. As an example an entry in this file is:

```
n;s_form;m_root;m_affix;text;person;number
```

which specifies which features and in which order a noun category should be writen. In this case:

```
n(s_form:...,m_root:...,m_affix:...,text:...,person:...,number:....).
```

9.12.7 Parser and Grammar

The parser builds a semantic representation compositionally, and a 'best parse' algorithm is applied to each final chart, providing a partial parse if no complete sentence span can be constructed. The parser uses a feature valued grammar. Each `Category` entry has the form:

```
Category(Feature1:Value1,...,FeatureN:ValueN)
```

where the number and type of features is dependent on the category type (see Section 6.1). All categories will have the features `s_form` (surface form) and `m_root` (morphological root); nominal and verbal categories will also have `person` and `number` features; verbal categories will also have `tense` and `vform` features; and adjectival categories will have a `degree` feature. The `list_np` category has the same features as other nominal categories plus `ne_tag` and `ne_type`.

Syntactic rules are specifed in Prolog with the predicate $rule(LHS, RHS)$ where LHS is a syntactic category and RHS is a list of syntactic categories. A rule such as $BNP\_HEAD \Rightarrow N$ ("a basic noun phrase head is composed of a noun") is writen as follows:

```
rule(bnp_head(sem:E^[[R,E],[number,E,N]],number:N),
[n(m_root:R,number:N)]).
```

where the feature 'sem' is used to construct the semantics while the parser processes input, and E, R, and N are variables tobe instantiated during parsing.

The full grammar of this distribution can be found in the prolog/grammar directory, the file load.pl specifies which grammars are used by the parser. The grammars are compiled when the system is built and the compied version is used for parsing.

9.12.8 Mapping Named Entities

SUPPLE has a prolog grammar which deals with named entities, the only information required is the Lookup annotations produced by Gate, which are specified in the mapping file. However, you may want to pass named entities identified with your own Jape grammars in Gate. This can be done using a special syntactic category provided with this distribution. The category sem_cat is used as a bridge between Gate named entities and the SUPPLE grammar. An example of how to use it (provided in the mapping file) is:

```
Gate;AnnotationType=Date;string=&S
SUPPLE;category=sem_cat;type=Date;text=&S;kind=date;name=&S
```

which maps a named entity 'Date' into a syntactic category 'sem_cat'. A grammar file called semantic_rules.pl is provided to map sem_cat into the appropriate syntactic category expected by the phrasal rules. The following rule for example:

```
rule(ne_np(s_form:F,sem:X^[[name,X,NAME],[KIND,X]]),[
sem_cat(s_form:F,text:TEXT,type:'Date',kind:KIND,name:NAME)]).
```

is used to parse a 'Date' into a named entity in SUPPLE which in turn will be parsed into a noun phrase.

9.12.9 Upgrading from BuChart to SUPPLE

In theory upgrading from BuChart to SUPPLE should be relatively straightforward. Basically any instance of BuChart needs to be replaced by SUPPLE. Specific changes which must be made are:

- The compiled parser files are now supple.swi, supple.sicstus, or supple.plcafe
- The GATE wrapper parameter buchartFile is now SUPPLEFile, and it is now of type `java.net.URL` rather than `java.io.File`. Details of how to compensate for this in existing saved applications are given in the SUPPLE README file.

- The Prolog wrappers now start shef.nlp.supple.prolog instead of shef.nlp.buchart.prolog
- The mapping.conf file now has lines starting SUPPLE; instead of Buchart;
- Most importantly the main wrapper class is now called nlp.shef.supple.SUPPLE

Making these changes to existing code should be trivial and allow application to benefit from future improvements to SUPPLE.

9.13 Stanford Parser

The Stanford Parser is a probabilistic parsing system implemented in Java by Stanford University's Natural Language Processing Group. Data files are available from Stanford for parsing Arabic, Chinese, English, and German.

This plugin, developed by the GATE team, provides a PR (`gate.stanford.Parser`) that acts as a wrapper around the Stanford Parser (version 1.6.1) and translates GATE annotations to and from the data structures of the parser itself. The plugin is supplied with the unmodified `jar` file and one English data file obtained from Stanford. Stanford's software itself is subject to the full GPL.

The parser itself can be trained on other corpora and languages, as documented on the website, but this plugin does not provide a means of doing so. Trained data files are not compatible between different versions of the parser; in particular, note that you need version 1.6.1 data files for GATE builds numbered above 3120 (when we upgraded the plugin to Stanford version 1.6.1 on 22 January 2009) but version 1.6 files for earlier versions, including *Release 5.0 beta 1*.

Creating multiple instances of this PR in the same JVM with different trained data files does not work—the PRs can be instantiated, but runtime errors will almost certainly occur.

9.13.1 Input requirements

Documents to be processed by the Parser PR must already have `Sentence` and `Token` annotations, such as those produced by either ANNIE Sentence Splitter (sections 8.3 and 8.4) and the ANNIE English Tokeniser (section 8.1).

If the `reusePosTags` parameter is true, then the `Token` annotations must have `category` features with compatible POS tags. The tags produced by the ANNIE POS Tagger are compatible with Stanford's parser data files for English (which also use the Penn treebank tagset).

9.13.2 Initialization parameters

parserFile the path to the trained data file; the default value points to the English data file[2] included with the GATE distribution. You can also use other files downloaded from the Stanford Parser website or produced by training the parser.

mappingFile the optional path to a mapping file: a flat, two-column file which the wrapper can use to "translate" tags. A sample file is included.[3] By default this value is `null` and mapping is ignored.

tlppClass an implementation of `TreebankLangParserParams`, used by the parser itself to extract the dependency relations from the constituency structures. The default value is compatible with the English data file supplied. Please refer to the Stanford NLP Group's documentation and the parser's javadoc for a further explanation.

9.13.3 Runtime parameters

annotationSetName the name of the annotationSet used for input (`Token` and `Sentence` annotations) and output (`SyntaxTreeNode` and `Dependency` annotations, and `category` and `dependencies` features added to `Tokens`).

debug a boolean value which controls the verbosity of the wrapper's output.

reusePosTags if true, the wrapper will read `category` features (produced by an earlier POS-tagging PR) from the `Token` annotations and force the parser to use them.

useMapping if this is true and a mapping file was loaded when the PR was initialized, the POS and syntactic tags produced by the parser will be translated using that file. If no mapping file was loaded, this parameter is ignored.

The following boolean parameters switch on and off the various types of output that the parser can produce. Any or all of them can be true, but if all are false the PR will simply print a warning to save time (instead of running the parser).

addPosTags if this is true, the wrapper will add `category` features to the `Token` annotations.

addConstituentAnnotations if true, the wrapper will mark the syntactic constituents with `SyntaxTreeNode` annotations that are compatible with the Syntax Tree Viewer (see section 9.12.4).

addDependencyAnnotations if true, the wrapper will add `Dependency` annotations to indicate the dependency relations in the sentence.

[2]`resources/englishPCFG.ser.gz`

[3]`resources/english-tag-map.txt`

addDependencyFeatures if true, the wrapper will add `dependencies` features to the `Token` annotations to indicate the dependency relations in the sentence.

The parser will derive the dependency structures only if either or both of the dependency output options is enabled, so if you do not need the dependency analysis, you can disable both of them and the PR will run faster.

Two sample GATE applications for English are included in the `plugins/Stanford` directory: `sample_parser_en.gapp` runs the Regex Sentence Splitter and ANNIE Tokenizer and then this PR to annotate constituency and dependency structures, whereas `sample_pos+parser_en.gapp` also runs the ANNIE POS Tagger and makes the parser re-use its POS tags.

9.14 Montreal Transducer

Many of the key features introduced in the Montreal Transducer (MT) have now been ported in some form into the standard JAPE transducer. If you are considering using the MT, you should first check the documentation for the standard transducer in chapter 7 to see if that is suitable for your needs. Being such a core part of GATE, the standard JAPE transducer is likely to be more stable and bugs will be fixed more rapidly than with the MT.

The Montreal Transducer is an improved Jape Transducer, developed by Luc Plamondon, Université de Montréal. It is intended to make grammar authoring easier by providing a more flexible version of the JAPE language and it also fixes a few bugs. Full details of the transducer can be found at http://www.iro.umontreal.ca/ plamondl/mtltransducer/. We summarise the main features below.

9.14.1 Main Improvements

- While only == constraints were allowed on annotation attributes, the grammar now accepts constraints such as {MyAnnot.attrib != value}, {MyAnnot.attrib > value}, {MyAnnot.attrib < value}, {MyAnnot.attrib = value} and {MyAnnot.attrib ! value} *(a similar feature has now been incorporated in the standard JAPE transducer, see section 7.1)*

- The grammar now accepts negated constraints such as {!MyAnnot} (true if no annotation starting from current node has the MyAnnot type) and {!MyAnnot.attrib == value} (true if {MyAnnot.attrib == value} fails), where the == constraint can be any other operator *(this feature has now been incorporated into the standard transducer, see section 7.4)*

- Because the transducer compiles rules at run-time, the classpath must include the transducer jar file (unless the transducer is bundled in the GATE jar file). The Montreal Transducer updates the classpath automatically when it is initialised.

9.14.2 Main Bug fixes

- Constraints on more than one annotation types for a same node now work. For example, {MyAnnot1, MyAnnot2} was allowed by the Jape Transducer but not implemented yet *(this is also supported by the standard transducer)*
- The ∗ and + Kleene operators were not greedy when they occurred inside a rule *(the standard transducer still behaves this way).* The document region parsed by a rule is correct but ambiguous labels inside the rule were not resolved the expected way. In the following rule for example, a node that would match both constraints should be part of the ":titles" label and not ":names" because the first + is expected to be greedy:

```
({Lookup.majorType == title})+:titles ({Token.orth == upperInitial})*:names
```

9.15 Language Plugins

There are plugins available for processing the following languages: French, German, Spanish, Italian, Chinese, Arabic, Romanian, Hindi and Cebuano. Some of the applications are quite basic and just contain some useful processing resources to get you started when developing a full application. Others (Cebuano and Hindi) are more like toy systems built as part of an exercise in language portability.

Note that if you wish to use individual language processing resources without loading the whole application, you will need to load the relevant plugin for that language in most cases. The plugins all follow the same kind of format. Load the plugin using the plugin manager, and the relevant resources will be available in the Processing Resources set.

Some plugins just contain a list of resources which can be added ad hoc to other applications. For example, the Italian plugin simply contains a lexicon which can be used to replace the English lexicon in the default English POS tagger: this will provide a reasonable basic POS tagger for Italian.

In most cases you will also find a directory in the relevant plugin directory called data which contains some sample texts (in some cases, these are annotated with NEs).

9.15.1 French Plugin

The French plugin contains two applications for NE recognition: one which includes the TreeTagger for POS tagging in French (french+tagger.gapp) , and one which does not (french.gapp). Simply load the application required from the plugins/french directory. You do not need to load the plugin itself from the plugins menu. Note that the TreeTagger must first be installed and set up correctly (see Section 9.7 for details). Check that the runtime parameters are set correctly for your TreeTagger in your application. The applications both contain resources for tokenisation, sentence splitting, gazetteer lookup, NE recognition (via JAPE grammars) and orthographic coreference. Note that they are not intended to produce high quality results, they are simply a starting point for a developer working on French. Some sample texts are contained in the plugins/french/data directory.

9.15.2 German Plugin

The German plugin contains two applications for NE recognition: one which includes the TreeTagger for POS tagging in German (german+tagger.gapp) , and one which does not (german.gapp). Simply load the application required from the plugins/german/resources directory. You do not need to load the plugin itself from the plugins menu. Note that the TreeTagger must first be installed and set up correctly (see Section 9.7 for details). Check that the runtime parameters are set correctly for your TreeTagger in your application. The applications both contain resources for tokenisation, sentence splitting, gazetteer lookup, compound analysis, NE recognition (via JAPE grammars) and orthographic coreference. Some sample texts are contained in the plugins/german/data directory. We are grateful to Fabio Ciravegna and the Dot.KOM project for use of some of the components for the German plugin.

9.15.3 Romanian Plugin

The Romanian plugin contains an application for Romanian NE recognition (romanian.gapp). Simply load the application from the plugins/romanian/resources directory. You do not need to load the plugin itself from the plugins menu. The application contains resources for tokenisation, gazetteer lookup, NE recognition (via JAPE grammars) and orthographic coreference. Some sample texts are contained in the plugins/romanian/corpus directory.

9.15.4 Arabic Plugin

The Arabic plugin contains a simple application for Arabic NE recognition (arabic.gapp). Simply load the application from the plugins/arabic/resources directory. You do not need to load the plugin itself from the plugins menu. The application contains resources for tokenisation, gazetteer lookup, NE recognition (via JAPE grammars) and orthographic coreference. Note that there are two types of gazetteer used in this application: one which was derived automatically from training data (Arabic inferred gazetteer), and one which was created manually. Note that there are some other applications included which perform quite specific tasks (but can generally be ignored). For example, arabic-for-bbn.gapp and arabic-for-muse.gapp make use of a very specific set of training data and convert the result to a special format. There is also an application to collect new gazetteer lists from training data (arabic_lists_collector.gapp). For details of the gazetteer list collector please see Section 9.6.

9.15.5 Chinese Plugin

The Chinese plugin contains a simple application for Chinese NE recognition (chinese.gapp). Simply load the application from the plugins/chinese/resources directory. You do not need to load the plugin itself from the plugins menu. The application contains resources for tokenisation, gazetteer lookup, NE recognition (via JAPE grammars) and orthographic coreference. The application makes use of some gazetteer lists (and a grammar to process them) derived automatically from training data, as well as regular hand-crafted gazetteer lists. There are also applications (listscollector.gapp, adj_collector.gapp and nounperson_collector.gapp) to create such lists, and various other application to perform special tasks such as coreference evaluation (coreference_eval.gapp) and converting the output to a different format (ace-to-muse.gapp).

9.15.6 Hindi Plugin

The Hindi plugin contains a set of resources for basic Hindi NE recognition which mirror the ANNIE resources but are customised to the Hindi language. You need to have the ANNIE plugin loaded first in order to load any of these PRs. With the Hindi, you can create an application similar to ANNIE but replacing the ANNIE PRs with the default PRs from the plugin.

9.16 Chemistry Tagger

This GATE module is designed to tag a number of chemistry items in running text. Currently the tagger tags compound formulas (e.g. SO2, H2O, H2SO4 ...) ions (e.g. Fe3+, Cl-) and element names and symbols (e.g. Sodium and Na). Limited support for compound names is also provided (e.g. sulphur dioxide) but only when followed by a compound formula (in parenthesis or commas).

9.16.1 Using the tagger

The Tagger requires the Creole plugin "Chemistry_Tagger" to be loaded. It requires the following PRs to have been run first: tokeniser and sentence splitter. There are four init parameters giving the locations of the two gazetteer list definitions, the element mapping file and the JAPE grammar used by the tagger (in previous versions of the tagger these files were fixed and loaded from inside the `ChemTagger.jar` file). Unless you know what you are doing you should accept the default values.

The annotations added to documents are "ChemicalCompound", "ChemicalIon" and "ChemicalElement" (currently they are always placed in the default annotation set). By default "ChemicalElement" annotations are removed if they make up part of a larger compound or ion annotation. This behaviour can be changed by setting the removeElements parameter to false so that all recognised chemical elements are annotated.

9.17 Flexible Exporter

The Flexible Exporter enables the user to save a document (or corpus) in its original format with added annotations. The user can select the name of the annotation set from which these annotations are to be found, which annotations from this set are to be included, whether features are to be included, and various renaming options such as renaming the annotations and the file.

At load time, the following parameters can be set for the flexible exporter:

- includeFeatures - if set to true, features are included with the annotations exported; if false (the default status), they are not.
- useSuffixForDumpFiles - if set to true (the default status), the output files have the suffix defined in suffixForDumpFiles; if false, no suffix is defined, and the output file simply overwrites the existing file (but see the outputFileUrl runtime parameter for an alternative).
- suffixForDumpFiles - this defines the suffix if useSuffixForDumpFiles is set to true. By default the suffix is .gate.

The following runtime parameters can also be set (after the file has been selected for the application):

- annotationSetName - this enables the user to specify the name of the annotation set which contains the annotations to be exported. If no annotation set is defined, it will use the Default annotation set.

- annotationTypes - this contains a list of the annotations to be exported. By default it is set to Person, Location and Date.
- dumpTypes - this contains a list of names for the exported annotations. If the annotation name is to remain the same, this list should be identical to the list in annotationTypes. The list of annotation names must be in the same order as the corresponding annotation types in annotationTypes.
- outputDirectoryUrl - this enables the user to specify the export directory where the file is exported with its original name and an extension (provided as a parameter) appended at the end of filename. Note that you can also save a whole corpus in one go.

9.18 Annotation Set Transfer

The Annotation Set Transfer allows copying or moving annotations to a new annotation set if they lie between the beginning and the end of an annotation of a particular type (the covering annotation). For example, this can be used when a user only wants to run a processing resource over a specific part of a document, such as the Body of an HTML document. The user specifies the name of the annotation set and the annotation which covers the part of the document they wish to transfer, and the name of the new annotation set. All the other annotations corresponding to the matched text will be transferred to the new annotation set. For example, we might wish to perform named entity recognition on the body of an HTML text, but not on the headers. After tokenising and performing gazetteer lookup on the whole text, we would use the Annotation Set Transfer to transfer those annotations (created by the tokeniser and gazetteer) into a new annotation set, and then run the remaining NE resources, such as the semantic tagger and coreference modules, on them.

The Annotation Set Transfer has no loadtime parameters. It has the following runtime parameters:

- `inputASName` - this defines the annotation set from which annotations will be transferred (copied or moved). If nothing is specified, the Default annotation set will be used.
- `outputASName` - this defines the annotation set to which the annotations will be transferred. This default value for this parameter is "Filtered". If it is left blank the Default annotation set will be used.
- `tagASName` - this defines the annotation set which contains the annotation covering the relevant part of the document to be transferred. This default value for this parameter is "Original markups". If it is left blank the Default annotation set will be used.

- `textTagName` - this defines the type of the annotation covering the annotations to be transferred. The default value for this parameter is "BODY". If this is left blank, then all annotations from the inputASName annotation set will be transferred. If more than one covering annotation is found, the annotation covered by each of them will be transferrred. If no covering annotation is found, the processing depends on the `copyAllUnlessFound` parameter (see below).
- `copyAnnotations` - this specifies whether the annotations should be moved or copied. The default value `false` will move annotations, removing them from the `inputASName` annotation set. If set to `true` the annotations will be copied.
- `transferAllUnlessFound` - this specifies what should happen if no covering annotation is found. The default value is `true`. In this case, all annotations will be copied or moved (depending on the setting of parameter `copyAnnotations`) if no covering annotation is found. If set to `false`, no annotation will be copied or moved.

For example, suppose we wish to perform named entity recognition on only the text covered by the BODY annotation from the Original Markups annotation set in an HTML document. We have to run the gazetteer and tokeniser on the entire document, because since these resources do not depend on any other annotations, we cannot specify an input annotation set for them to use. We therefore transfer these annotations to a new annotation set (Filtered) and then perform the NE recognition over these annotations, by specifying this annotation set as the input annotation set for all the following resources. In this example, we would set the following parameters (assuming that the annotations from the tokenise and gazetteer are initially placed in the Default annotation set).

- inputASName: Default
- outputASName: Filtered
- tagASName: Original markups
- textTagName: BODY
- copyAnnotations: true or false (depending on whether we want to keep the Token and Lookup annotations in the Default annotation set)
- copyAllUnlessFound: true

9.19 Information Retrieval in GATE

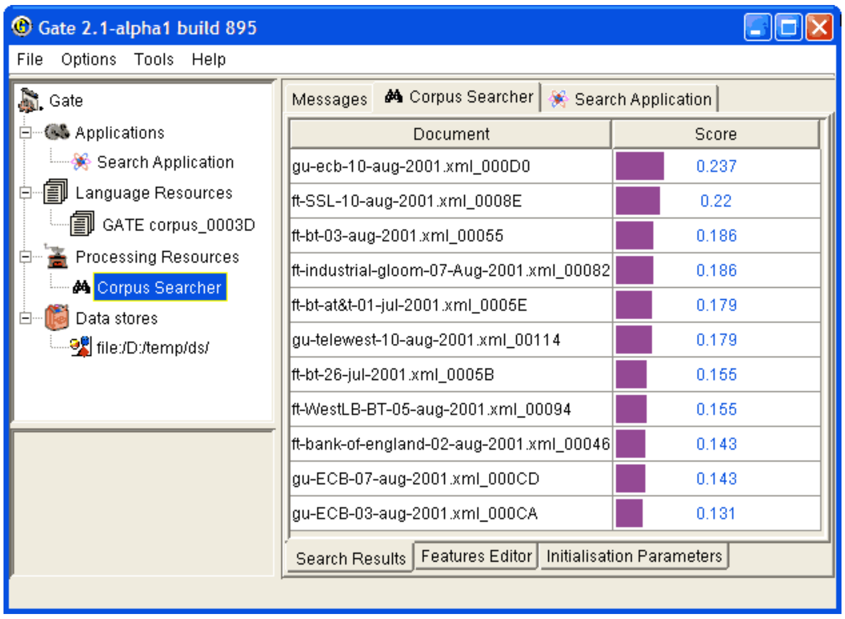
GATE comes with a full-featured Information Retrieval (IR) subsystem that allows queries to be performed against GATE corpora. This combination of IE and IR means that documents can be retrieved from the corpora not only based on their textual content but also according to their features or annotations. For example, a search over the Person annotations for ”Bush” will return documents with higher relevance, compared to a search in the content for the string ”bush”. The current implementation is based on the most popular open source full-text search engine - Lucene (available at http://jakarta.apache.org/lucene/) but other implementations may be added in the future.

An Information Retrieval system is most often considered a system that accepts as input a set of documents (corpus) and a query (combination of search terms) and returns as input only those documents from the corpus which are considered as relevant according to the query. Usually, in addition to the documents, a proper relevance measure (score) is returned for each document. There exist many relevance metrics, but usually documents which are considered more relevant, according to the query, are scored higher.

Figure 9.4 shows the results from running a query against an indexed corpus in GATE.

	$Term_1$	$Term_2$	...	...	$term_k$
Doc1	$w_{1,1}$	$w_{1,2}$	...	...	$w_{1,k}$
Doc2	$w_{2,1}$	$w_{2,1}$	...	...	$w_{2,k}$
...	...	...	...	...	...
...	...	...	...	...	...
doc_n	$w_n, 1$	$w_{n,2}$	...	...	$w_{n,k}$

Table 9.1:



Document	Score
gu-ecb-10-aug-2001.xml_000D0	0.237
ft-SSL-10-aug-2001.xml_0008E	0.22
ft-bt-03-aug-2001.xml_00055	0.186
ft-industrial-gloom-07-Aug-2001.xml_00082	0.186
ft-bt-at&t-01-jul-2001.xml_0005E	0.179
gu-telewest-10-aug-2001.xml_00114	0.179
ft-bt-26-jul-2001.xml_0005B	0.155
ft-WestLB-BT-05-aug-2001.xml_00094	0.155
ft-bank-of-england-02-aug-2001.xml_00046	0.143
gu-ECB-07-aug-2001.xml_000CD	0.143
gu-ECB-03-aug-2001.xml_000CA	0.131

Figure 9.4: Documents with scores, returned from a search over a corpus

Information Retrieval systems usually perform some preprocessing one the input corpus in order to create the document-term matrix for the corpus. A document-term matrix is usually presented as:

where doc_i is a document from the corpus, $term_j$ is a word that is considered as important and representative for the document and wi, j is the weight assigned to the term in the document. There are many ways to define the term weight functions, but most often it depends on the term frequency in the document and in the whole corpus (i.e. the local and the global frequency). Note that the machine learning plugin described in Chapter 11 can produce such document-term matrix (for detailed description of the matrix produced see Section 11.5.4).

Note that not all of the words appearing in the document are considered terms. There are many words (called "stop-words") which are ignored, since they are observed too often and are not representative enough. Such words are articles, conjunctions, etc. During the preprocessing phase which identifies such words, usually a form of stemming is performed in order to minimize the number of terms and to improve the retrieval recall. Various forms of the same word (e.g. ”play”, ”playing” and ”played”) are considered identical and multiple occurrences of the same term (probably ”play”) will be observed.

It is recommended that the user reads the relevant Information Retrieval literature for a detailed explanation of stop words, stemming and term weighting.

IR systems, in a way similar to IE systems, are evaluated with the help of the precision and recall measures (see Section 13.4 for more details).

9.19.1 Using the IR functionality in GATE

In order to run queries against a corpus, the latter should be ”indexed”. The indexing process first processes the documents in order to identify the terms and their weights (stemming is performed too) and then creates the proper structures on the local filesystem. These file structures contain indexes that will be used by Lucene (the underlying IR engine) for the retrieval.

Once the corpus is indexed, queries may be run against it. Subsequently the index may be removed and then the structures on the local filesytem are removed too. Once the index is removed, queries cannot be run against the corpus.

Indexing the corpus

In order to index a corpus, the latter should be stored in a serial datastore. In other words, the IR functionality is unavailable for corpora that are transient or stored in a RDBMS datastores (though support for the lattr may be added in the future).

To index the corpus, follow these steps:

- Select the corpus from the resource tree (top-left pane) and from the context menu (right button click) choose ”Index Corpus”. A dialogue appears that allows you to specify the index properties.
- In the index properties dialogue, specify the underlying IR system to be used (only Lucene is supported at present), the directory that will contain the index structures, and the set of properties that will be indexed such as document features, content, etc (the same properties will be indexed for each document in the corpus).
- Once the corpus in indexed, you may start running queries against it. Note that the directory specified for the index data should exist and be empty. Otherwise an error will occur during the index creation.

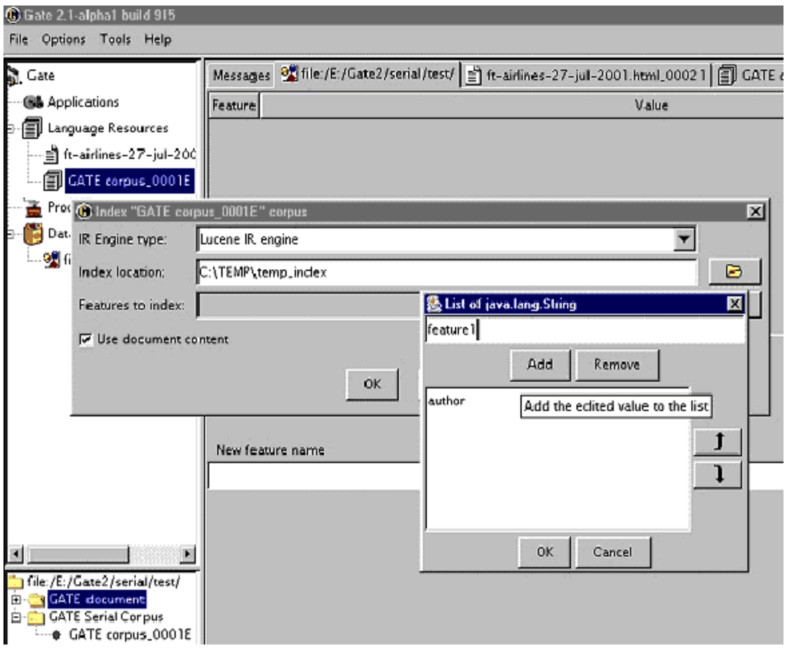


Figure 9.5: Indexing a corpus by specifying the index location and indexed features (and content)

Querying the corpus

To query the corpus, follow these steps:

- Create a SearchPR processing resource. All the parameters of SearchPR are runtime so theyare set later.
- Create a pipeline application containing the SearchPR.
- Set the following SearchPR parameters:
  - The corpus that will be queried.
  - The query that will be executed.
  - The maximum number of documents returned.

  A query looks like the following:

  ```
  {+/-}field1:term1 {+/-}field2:term2 ? {+/-}fieldN:termN
  ```

  where field is the name of a index field, such as the one specified at index creation (the document content field is body) and term is a term that should appear in the field.

  For example the query:

```
+body:government +author:CNN
```

will inspect the document content for the term "government" (together with variations such as "governments" etc.) and the index field named "author" for the term "CNN". The "author" field is specified at index creation time, and is either a document feature or another document property.

- After the SearchPR is initialized, running the application executes the specified query over the specified corpus.
- Finally, the results are displayed (see fig.1) after a double-click on the SearchPR processing resource.

Removing the index

An index for a corpus may be removed at any time from the "Remove Index" option of the context menu for the indexed corpus (right button click).

9.19.2 Using the IR API

The IR API within GATE makes it possible for corpora to be indexed, queried and results returned from any Java application, without using the GATE GUI. The following sample indexes a corpus, runs a query against it and then removes the index.

```
// open a serial data store
SerialDataStore sds =
Factory.openDataStore("gate.persist.SerialDataStore",
"/tmp/datastore1");
sds.open();

//set an AUTHOR feature for the test document
Document doc0 = Factory.newDocument(new URL("/tmp/documents/doc0.html"));
doc0.getFeatures().put("author","John Smit");

Corpus corp0 = Factory.newCorpus("TestCorpus");
corp0.add(doc0);

//store the corpus in the serial datastore
Corpus serialCorpus = (Corpus) sds.adopt(corp0,null);
sds.sync(serialCorpus);
```

```
//index the corpus -  the content and the AUTHOR feature

IndexedCorpus indexedCorpus = (IndexedCorpus) serialCorpus;

DefaultIndexDefinition did = new DefaultIndexDefinition();
did.setIrEngineClassName(gate.creole.ir.lucene. LuceneIREngine.class.getName());
did.setIndexLocation("/tmp/index1");
did.addIndexField(new IndexField("content", new DocumentContentReader(), false));
did.addIndexField(new IndexField("author", null, false));
indexedCorpus.setIndexDefinition(did);

indexedCorpus.getIndexManager().createIndex();
//the corpus is now indexed

//search the corpus
Search search = new LuceneSearch();
search.setCorpus(ic);

QueryResultList res = search.search("+content:government +author:John");

//get the results
Iterator it = res.getQueryResults();
while (it.hasNext()) {
QueryResult qr = (QueryResult) it.next();
System.out.println("DOCUMENT_ID="+ qr.getDocumentID() +",   scrore="+qr.getScore());
}
```

9.20 Crawler

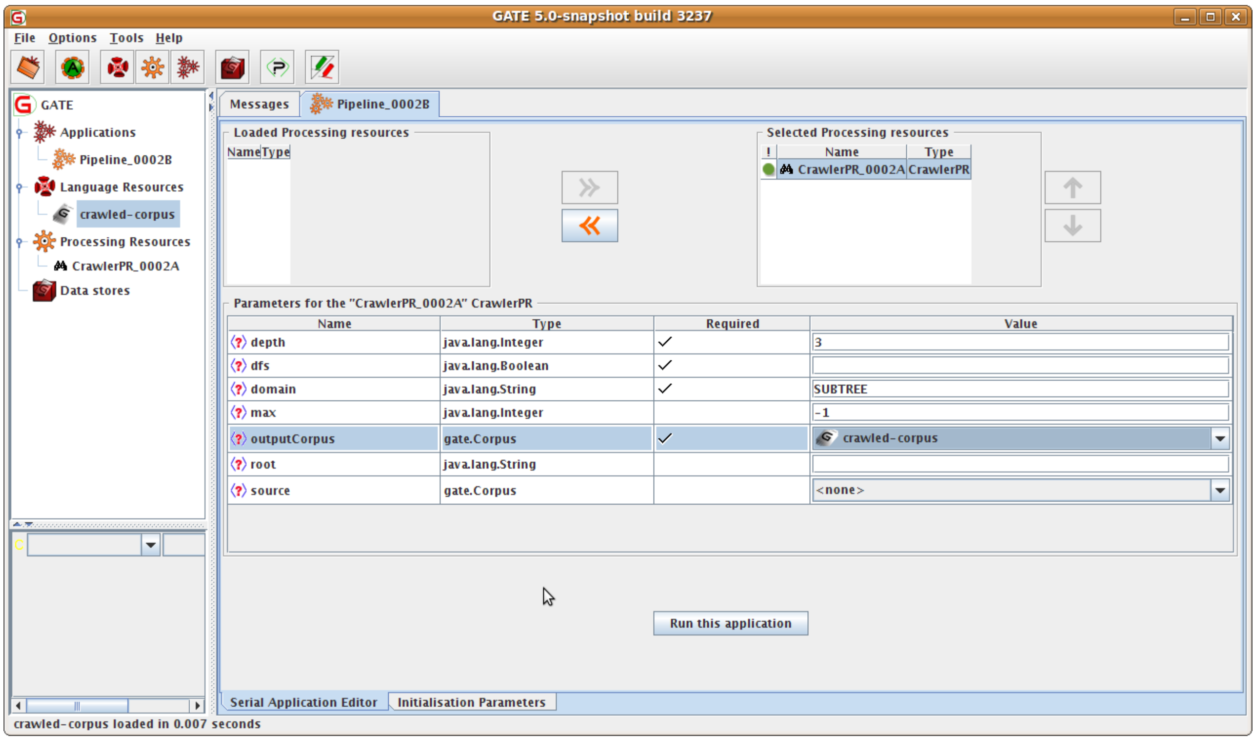
The crawler plugin enables GATE to be used for a corpus that is built using a web crawl. The crawler itself is Websphinx.This is a JAVA based multi-threaded web crawler that can be customized for any application. In order to use this plugin it may be required that the websphinx.jar file be added in the required libraries in JBuilder.

The basic idea is to be able to specify a source URL and a depth to build the initial corpus upon which further processing could be done. The PR itself provides a number of helpful features to set various parameters of the crawl.

9.20.1 Using the Crawler PR

In order to use the processing resource you first need to load the plugin using the plugin manager. Then load the crawler from the list of processing resources. User needs to create a corpus in which he or she wants to store crawled documents. In order to use the crawler, create a simple pipeline (note: do not create a corpus pipeline) and add the crawl PR to the pipeline.

Once the crawl PR is created there will be a number of parameters that can be set based on the PR required (see also Figure 9.6).



Parameters for the "CrawlerPR_0002A" CrawlerPR

Name	Type	Required	Value
depth	java.lang.Integer	✓	3
dfs	java.lang.Boolean	✓	
domain	java.lang.String	✓	SUBTREE
max	java.lang.Integer		-1
outputCorpus	gate.Corpus	✓	crawled-corpus
root	java.lang.String		
source	gate.Corpus		<none>

crawled-corpus loaded in 0.007 seconds

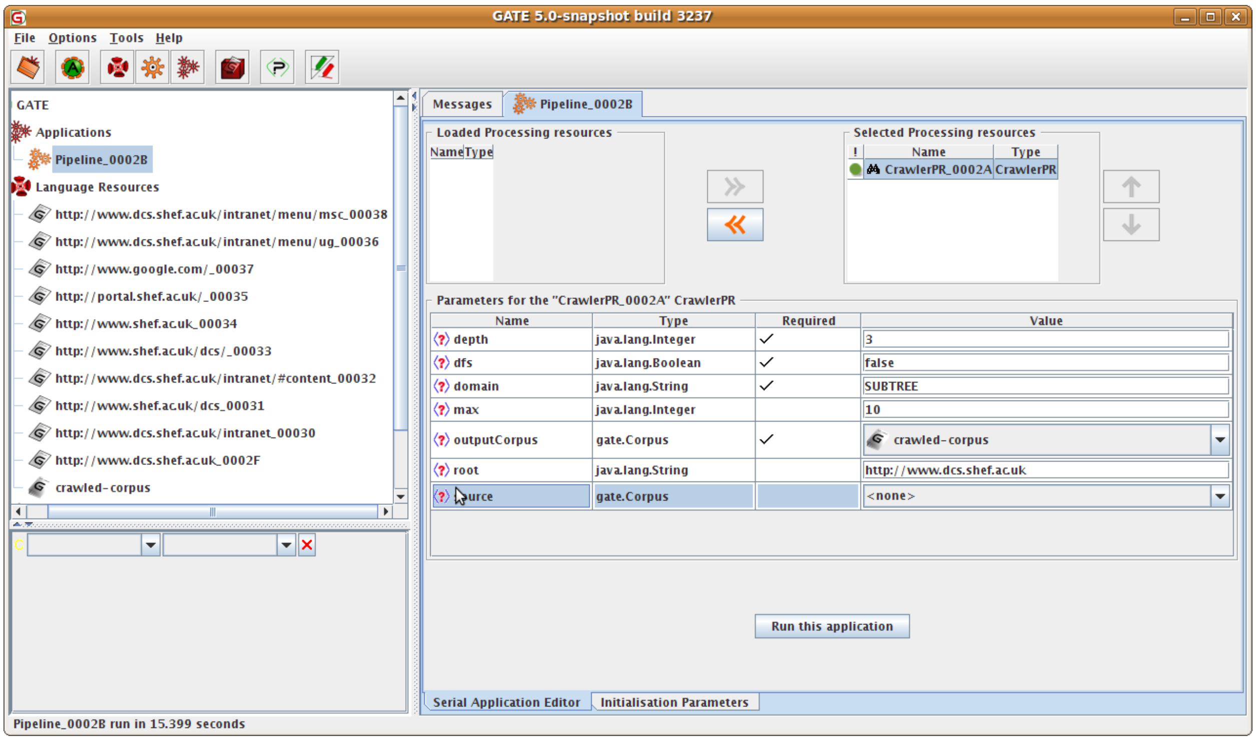
Figure 9.6: Crawler parameters

- depth: the depth to which the crawl should proceed.
- dfs / bfs: dfs if true bfs if false
  - Dfs : the crawler uses the depth first strategy for the crawl.
    - ∗ Visits the nodes in dfs order until the specified depth limit is reached.
  - Bfs: the crawler used the breadth first strategy for the crawl.
    - ∗ Visits the nodes on bfs order until the specified depth limit is reached.
- domain

  - SUBTREE: Crawler visits only the descendents of the page specified as the root for the crawl.
  - WEB: Crawler visits all the pages on the web.
  - SERVER: Crawler visits only the pages that are present on the server where the root page is located.

- max number of pages to be fetched
- outputCorpus an instance of Corpus to be used for storing crawled web pages
- root the starting URL to be used for the crawl to begin
- source is the corpus to be used that contains the documents from which the crawl must begin. Source is useful when the documents are fetched first from the google PR and then need to be crawled to expand the web graph further. At any time either the source or the root needs to be set.

Once the parameters are set, the crawl can be run and the documents fetched are added to the specified corpus. Figure 9.7 shows the crawled pages added to the corpus.


GATE 5.0-snapshot build 3237
File
Options
Tools
Help
Messages
Pipeline_0002B
Loaded Processing resources
Selected Processing resources
Run this application
Serial Application Editor
Initialisation Parameters

GATE
Applications
Pipeline_0002B
Language Resources
http://www.dcs.shef.ac.uk/intranet/menu/msc_00038
http://www.dcs.shef.ac.uk/intranet/menu/ug_00036
http://www.google.com/_00037
http://portal.shef.ac.uk/_00035
http://www.shef.ac.uk_00034
http://www.shef.ac.uk/dcs/_00033
http://www.dcs.shef.ac.uk/intranet/#content_00032
http://www.shef.ac.uk/dcs_00031
http://www.dcs.shef.ac.uk/intranet_00030
http://www.dcs.shef.ac.uk_0002F
crawled-corpus

!	Name	Type
●	CrawlerPR_0002A	CrawlerPR

Parameters for the "CrawlerPR_0002A" CrawlerPR

Name	Type	Required	Value
depth	java.lang.Integer	✓	3
dfs	java.lang.Boolean	✓	false
domain	java.lang.String	✓	SUBTREE
max	java.lang.Integer		10
outputCorpus	gate.Corpus	✓	crawled-corpus
root	java.lang.String		http://www.dcs.shef.ac.uk
source	gate.Corpus		<none>

Pipeline_0002B run in 15.399 seconds

Figure 9.7: Crawled pages added to the corpus

9.21 Google Plugin

The Google API is now integrated with GATE, and can be used as a PR-based plugin. This plugin allows the user to query Google and build the document corpus that contains the search results returned by Google for the query. There is a limit of 1000 queries per day as set by Google. For more information about the Google API please refer to http://www.google.com/apis/. In order to use the Google PR, you need to register with Google to obtain a license key.

The Google PR can be used for a number of different application scenarios. For example, one use case is where a user wants to find out what are the different named entities that can be associated with a particular individual. In this example, the user could build the collection of documents by querying Google and then running ANNIE over the collection. This would annotate the results and show what are the different Organization, Location and other entities that can be associated with the query.

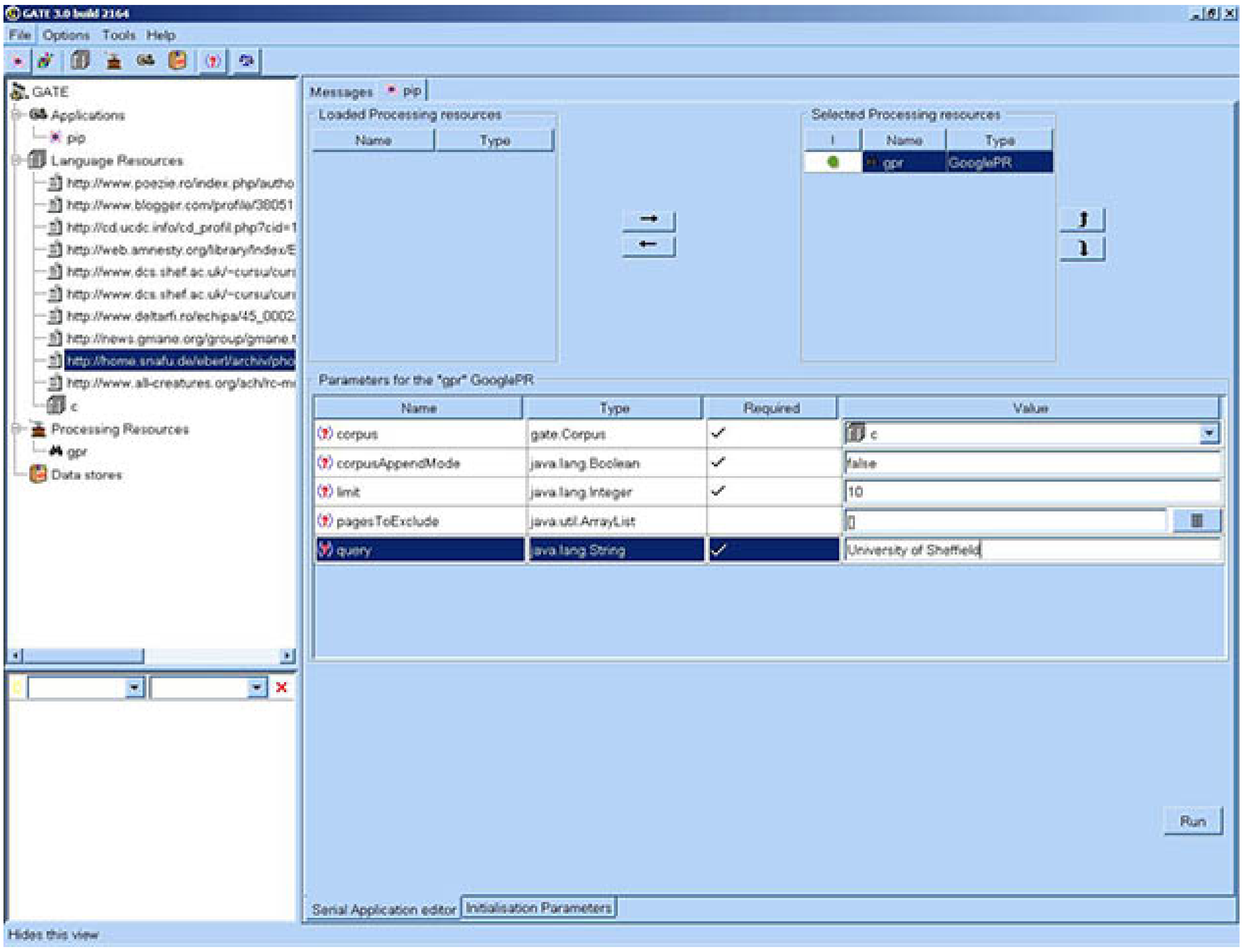
9.21.1 Using the GooglePR

In order to use the PR, you first need to load the plugin using the plugin manager. Once the PR is loaded, it can be initialized by creating an instance of a new PR. Here you need to specify the Google API License key. Please use the license key assigned to you by registering with Google.

Once the Google PR is initialized, it can be placed in a pipeline or a conditional pipeline application. This pipeline would contain the instance of the Google PR just initialized as above. There are a number of parameters to be set at runtime:

- `corpus`: The corpus used by the plugin to add or append documents from the Web.
- `corpusAppendMode`: If set to `true`, will append documents to the corpus. If set to `false`, will remove preexisting documents from the corpus, before adding the documents newly fetched by the PR
- `limit`: A limit on the results returned by the search. Default set to 10.
- `pagesToExclude`: This is an optional parameter. It is a list with URLs not to be included in the search.
- `query`: The query sent to Google. It is in the format accepted by Google.

Once the required parameters are set we can run the pipeline. This will then download all the URLs in the results and create a document for each. These documents would be added to the corpus as shown in Figure 9.8.


GATE 3.0 build 2164
File Options Tools Help
GATE
Applications
pip
Language Resources
http://www.poezie.ro/index.php/autho
http://www.blogger.com/profile/38051
http://cd.ucdc.info/cd_profil.php?cid=1
http://web.amnesty.org/library/index/E
http://www.dcs.shef.ac.uk/~cursu/curs
http://www.dcs.shef.ac.uk/~cursu/curs
http://www.deltarfi.ro/echipa/45_0002
http://news.gmane.org/group/gmane.t
http://home.snafu.de/ebert/archiv/pho
http://www.all-creatures.org/ach/rc-m
c
Processing Resources
gpr
Data stores
Messages
pip
Loaded Processing resources
Name
Type
Selected Processing resources
!
Name
Type
gpr
GooglePR
Run
Serial Application editor
Initialisation Parameters
Hides this view

Parameters for the "gpr" GooglePR

Name	Type	Required	Value
corpus	gate.Corpus	✓	c
corpusAppendMode	java.lang.Boolean	✓	false
limit	java.lang.Integer	✓	10
pagesToExclude	java.util.ArrayList		[]
query	java.lang.String	✓	University of Sheffield

Figure 9.8: URLs added to the corpus

9.22 Yahoo Plugin

The Yahoo API is now integrated with GATE, and can be used as a PR-based plugin. This plugin allows the user to query Yahoo and build the document corpus that contains the search results returned by Yahoo for the query. For more information about the Yahoo API please refer to http://developer.yahoo.com/search/. In order to use the Yahoo PR, you need to obtain an application ID.

The Yahoo PR can be used for a number of different application scenarios. For example, one use case is where a user wants to find out what are the different named entities that can be associated with a particular individual. In this example, the user could build the collection of documents by querying Yahoo and then running ANNIE over the collection. This would annotate the results and show what are the different Organization, Location and other entities that can be associated with the query.

9.22.1 Using the YahooPR

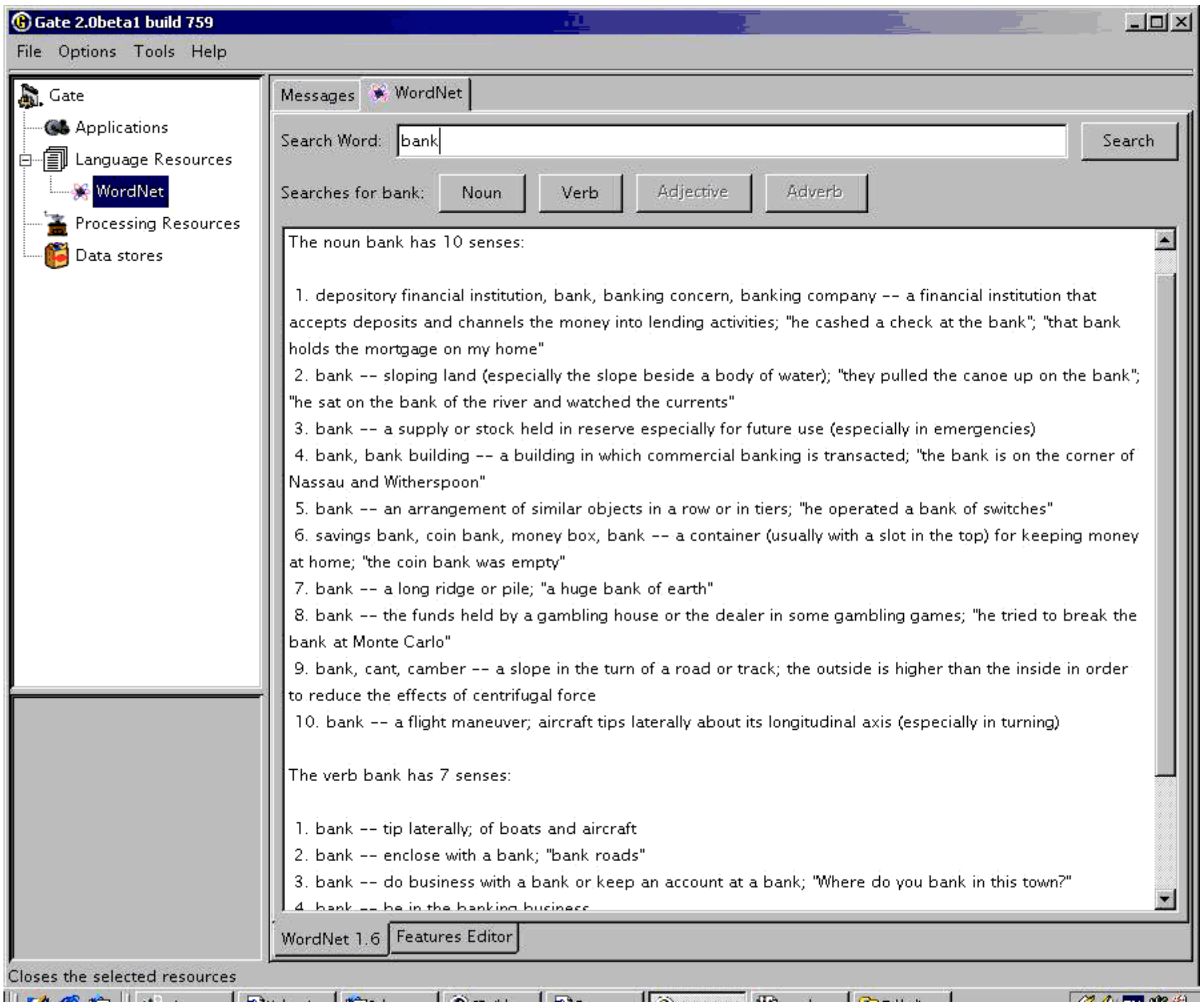
In order to use the PR, you first need to load the plugin using the plugin manager. Once the PR is loaded, it can be initialized by creating an instance of a new PR. Here you need to specify the Yahoo Application ID. Please use the license key assigned to you by registering with Yahoo.

Once the Yahoo PR is initialized, it can be placed in a pipeline or a conditional pipeline application. This pipeline would contain the instance of the Yahoo PR just initialized as above. There are a number of parameters to be set at runtime:

- `corpus`: The corpus used by the plugin to add or append documents from the Web.
- `corpusAppendMode`: If set to `true`, will append documents to the corpus. If set to `false`, will remove preexisting documents from the corpus, before adding the documents newly fetched by the PR
- `limit`: A limit on the results returned by the search. Default set to 10.
- `pagesToExclude`: This is an optional parameter. It is a list with URLs not to be included in the search.
- `query`: The query sent to Yahoo. It is in the format accepted by Yahoo.

Once the required parameters are set we can run the pipeline. This will then download all the URLs in the results and create a document for each. These documents would be added to the corpus.

9.23 WordNet in GATE



The noun bank has 10 senses:

1. depository financial institution, bank, banking concern, banking company -- a financial institution that accepts deposits and channels the money into lending activities; "he cashed a check at the bank"; "that bank holds the mortgage on my home"
2. bank -- sloping land (especially the slope beside a body of water); "they pulled the canoe up on the bank"; "he sat on the bank of the river and watched the currents"
3. bank -- a supply or stock held in reserve especially for future use (especially in emergencies)
4. bank, bank building -- a building in which commercial banking is transacted; "the bank is on the corner of Nassau and Witherspoon"
5. bank -- an arrangement of similar objects in a row or in tiers; "he operated a bank of switches"
6. savings bank, coin bank, money box, bank -- a container (usually with a slot in the top) for keeping money at home; "the coin bank was empty"
7. bank -- a long ridge or pile; "a huge bank of earth"
8. bank -- the funds held by a gambling house or the dealer in some gambling games; "he tried to break the bank at Monte Carlo"
9. bank, cant, camber -- a slope in the turn of a road or track; the outside is higher than the inside in order to reduce the effects of centrifugal force
10. bank -- a flight maneuver; aircraft tips laterally about its longitudinal axis (especially in turning)

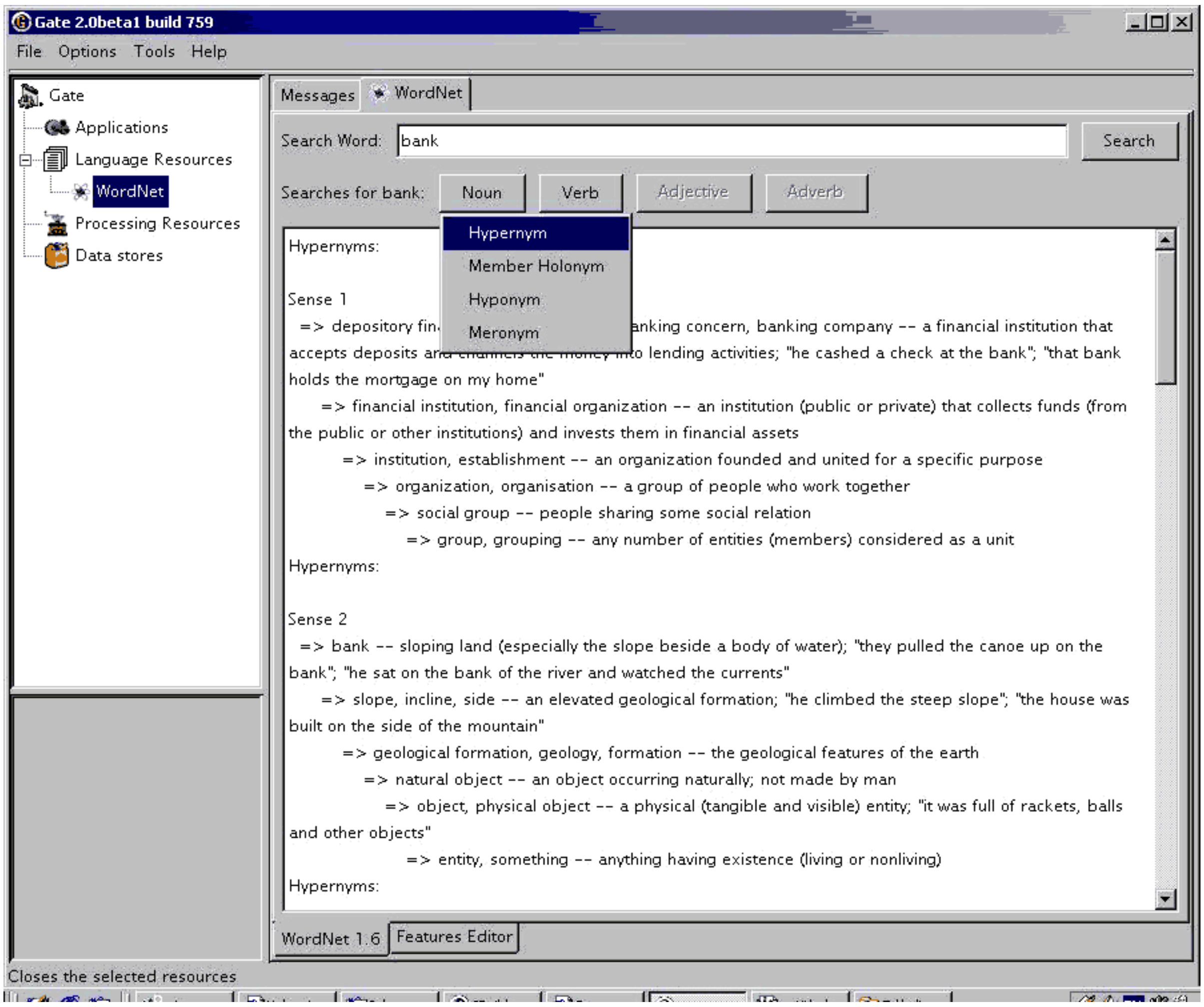
The verb bank has 7 senses:

1. bank -- tip laterally; of boats and aircraft
2. bank -- enclose with a bank; "bank roads"
3. bank -- do business with a bank or keep an account at a bank; "Where do you bank in this town?"
4. bank -- be in the banking business

Figure 9.9: WordNet in GATE – results for “bank”

At present GATE supports only WordNet 1.6, so in order to use WordNet in GATE, you must first install WordNet 1.6 on your computer. WordNet is available at http://wordnet.princeton.edu/. The next step is to configure GATE to work with your local WordNet installation. Since GATE relies on the Java WordNet Library (JWNL) for WordNet access, this step consists of providing one special xml file that is used internally by JWNL. This file describes the location of your local copy of the WordNet 1.6 index files. An example of this wn-config.xml file is shown below:

```
<?xml version="1.0" encoding="UTF-8"?>

<jwnl_properties language="en">
        <version publisher="Princeton" number="1.6" language="en"/>
        <dictionary class="net.didion.jwnl.dictionary.FileBackedDictionary">
                <param name="morphological_processor" value="net.didion.jwnl.diction
```



```
Hypernyms:

Sense 1
  => depository fin[...]                    anking concern, banking company -- a financial institution that
accepts deposits an[...]                     o lending activities; "he cashed a check at the bank"; "that bank
holds the mortgage on my home"
      => financial institution, financial organization -- an institution (public or private) that collects funds (from
the public or other institutions) and invests them in financial assets
         => institution, establishment -- an organization founded and united for a specific purpose
            => organization, organisation -- a group of people who work together
               => social group -- people sharing some social relation
                  => group, grouping -- any number of entities (members) considered as a unit
Hypernyms:

Sense 2
  => bank -- sloping land (especially the slope beside a body of water); "they pulled the canoe up on the
bank"; "he sat on the bank of the river and watched the currents"
      => slope, incline, side -- an elevated geological formation; "he climbed the steep slope"; "the house was
built on the side of the mountain"
         => geological formation, geology, formation -- the geological features of the earth
            => natural object -- an object occurring naturally; not made by man
               => object, physical object -- a physical (tangible and visible) entity; "it was full of rackets, balls
and other objects"
                  => entity, something -- anything having existence (living or nonliving)
Hypernyms:
```

Figure 9.10: WordNet in GATE

```
                <param name="file_manager" value="net.didion.jwnl.dictionary.file_ma
                        <param name="file_type" value="net.didion.jwnl.princeton.fil
                        <param name="dictionary_path" value="e:\wn16\dict"/>
                </param>
        </dictionary>
        <dictionary_element_factory class="net.didion.jwnl.princeton.data.PrincetonW
        <resource class="PrincetonResource"/>
</jwnl_properties>
```

All you have to do is to replace the value of the `dictionary_path` parameter to point to your local installation of WordNet 1.6.

After configuring GATE to use WordNet, you can start using the built-in WordNet browser or API. In GATE, load the WordNet plugin via the plugins menu. Then load WordNet by selecting it from the set of available language resources. Set the value of the parameter to the path of the xml properties file which describes the WordNet location (wn-config).

Once Word Net is loaded in GATE, the well-known interface of WordNet will appear. You can search Word Net by typing a word in the box next to to the label "SearchWord"' and then pressing "Search". All the senses of the word will be displayed in the window below. Buttons for the possible parts of speech for this word will also be activated at this point. For instance, for the word "play", the buttons "Noun", "Verb" and "Adjective" are activated. Pressing one of these buttons will activate a menu with hyponyms, hypernyms, meronyms for nouns or verb groups, and cause for verbs, etc. Selecting an item from the menu will display the results in the window below.

More information about WordNet can be found at http://www.cogsci.princeton.edu/wn/index.shtml

More information about the JWNL library can be found at http://sourceforge.net/projects/jwordnet

An example of using the WordNet API in GATE is available on the GATE examples page at http://gate.ac.uk/GateExamples/doc/index.html

9.23.1 The WordNet API

GATE offers a set of classes that can be used to access the WordNet 1.6 Lexical Base. The implementation of the GATE API for WordNet is based on Java WordNet Library (JWNL). There are just a few basic classes, as shown in Figure 9.11. Details about the properties and methods of the interfaces/classes comprising the API can be obtained from the JavaDoc. Below is a brief overview of the interfaces:

- **WordNet**: the main WordNet class. Provides methods for getting the synsets of a lemma, for accessing the unique beginners, etc.

- **Word**: offers access to the word's lemma and senses
- **WordSense**: gives access to the synset, the word, POS and lexical relations.
- **Synset**: gives acess to the word senses (synonyms) in the synset, the semantic relations, POS etc.
- **Verb**: gives access to the verb frames (not working properly at present)
- **Adjective**: gives access to the adj. position (attributive, predicative, etc.).
- **Relation**: abstract relation such as type, symbol, inverse relation, set of POS tags, etc. to which it is applicable.
- **LexicalRelation**
- **SemanticRelation**
- **VerbFrame**

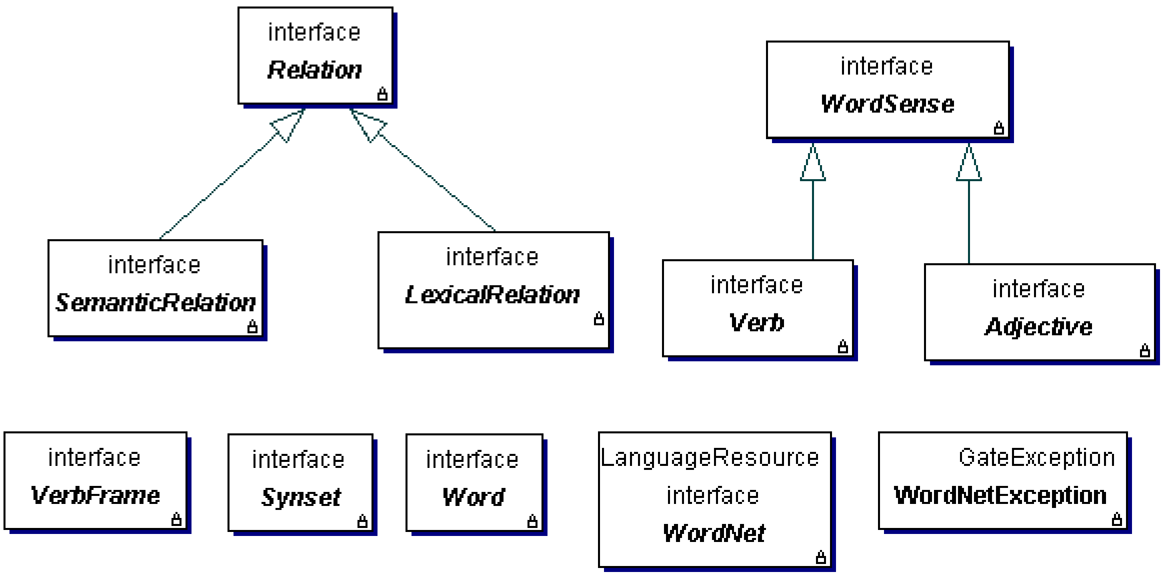


Figure 9.11: The Wordnet API

9.24 Machine Learning in GATE

Note: A brand new machine learning layer specifically targetted at NLP tasks including text classification, chunk learning (e.g. for named entity recognition) and relation learning has been added to GATE. See chapter 11 for more details.

9.24.1 ML Generalities

This section describes the use of Machine Learning (ML) algorithms in GATE.

An ML algorithm "learns" about a phenomenon by looking at a set of occurrences of that phenomenon that are used as examples. Based on these, a model is built that can be used to predict characteristics of future (and unforeseen) examples of the phenomenon.

Classification is a particular example of machine learning in which the set of training examples is split into multiple subsets (classes) and the algorithm attempts to distribute the new examples into the existing classes.

This is the type of ML that is used in GATE and all further references to ML actually refer to classification.

Some definitions

- **instance**: an example of the studied phenomenon. An ML algorithm learns from a set of known instances, called a dataset.
- **attribute**: a characteristic of the instances. Each instance is defined by the values of its attributes. The set of possible attributes is well defined and is the same for all instances in a dataset.
- **class**: an attribute for which the values need to be found through the ML mechanism.

GATE-specific interpretation of the above definitions

- **instance**: an annotation. In order to use ML in GATE the users will need to choose the type of annotations used as instances. Token annotations are a good candidate for this, but any type of annotation could be used (e.g. things that were found by a previously run JAPE grammar).
- **attribute**: an attribute can be either:
  - the presence (or absence) of a particular annotation type [partially] covering the instance annotation
  - the value of a named feature of a particular annotation type.

  The value of the attribute can refer to the current instance or to an instance situated at a specified location relative to the current instance.
- **class**: any attribute can be marked as class attribute.

An ML implementation has two modes of functioning: training and application. The training phase consists of building a model (e.g. statistical model, a decision tree, a rule set, etc.) from a dataset of already classified instances. During application, the model built while training is used to classify new instances.

There are ML algorithms which permit the incremental building of the model (e.g. the Updateable Classifiers in the WEKA library). These classifiers do not require the entire training dataset to build a model; the model improves with each new training instance that the algorithm is provided with.

9.24.2 The Machine Learning PR in GATE

Access to ML implementations is provided in GATE by the "Machine Learning PR" that handles both the training and application of ML model on GATE documents. This PR is a Language Analyser so it can be used in all default types of GATE controllers.

In order to allow for more flexibility, all the configuration parameters for the ML PR are set through an external XML file and not through the normal PR parameterisation. The root element of the file needs to be called "ML-CONFIG" and it contains two elements: "DATASET" and "ENGINE". An example XML configuration file is given in Appendix F.

The DATASET element

The DATASET element defines the type of annotation to be used as instance and the set of attributes that characterise all the instances.

An "INSTANCE-TYPE" element is used to select the annotation type to be used for instances, and the attributes are defined by a sequence of "ATTRIBUTE" elements.

For example, if an "INSTANCE-TYPE" has a "Token" for value, there will one instance in the dataset per "Token". This also means that the **positions** (see below) are defined in relation to Tokens. The "INSTANCE-TYPE" can be seen as the smallest unit to be taken into account for the Machine Learning.

An ATTRIBUTE element has the following sub-elements:

- **NAME**: the name of the attribute
- **TYPE**: the annotation type used to extract the attribute.
- **FEATURE** (optional): if present, the value of the attribute will be the value of the named feature on the annotation of specified type.
- **POSITION**: the position of the annotation used to extract the feature relative to the current instance annotation.

- **VALUES**(optional): includes a list of VALUE elements.
- **<CLASS/>**: an empty element used to mark the class attribute. There can only be one attribute marked as class in a dataset definition.

The **VALUES** being defined as XML entities, the characters <, > and & must be replaced by &lt;, &rt; and &amp;. It is recommended to write the XML configuration file in UTF-8 in order to have some uncommon character correctly parsed.

Semantically, there are three types of attributes:

- **nominal attributes**: both type and features are defined and a list of allowed values is provided;
- **numeric**: both type and features are defined but no list of allowed values is provided; it is assumed that the feature can be converted to a number (a double value).
- **boolean**: no feature or list of values is provided; the attribute will take one of the "true" or "false" values based on the presence (or absence) of the specified annotation type at the required position.

Figure 9.12 gives some examples of what the values of specified attributes would be in a situation when "Token" annotations are used as instances.

An ATTRIBUTELIST element is similar to ATTRIBUTE except that it has no POSITION sub-element but a RANGE element. This will be converted into several ATTRIBUTELIST with position ranging from the value of the attribute "from" to the value of the attribute "to". This can be used in order to avoid the duplication of ATTRIBUTE elements.

The ENGINE element

The ENGINE element defines which particular ML implementation will be used, and allows the setting of options for that particular implementation.

The ENGINE element has three sub-elements:

- **WRAPPER**: defines the class name for the ML implementation (or implementation wrapper). The specified class needs to extend gate.creole.ml.MLEngine.
- **BATCH-MODE-CLASSIFICATION**: this element is optional. If present (as an empty element `<BATCH-MODE-CLASSIFICATION />`), the training instances will be passed to the engine in a single batch. If absent, the instances are passed to the engine one at a time. Not every engine supports this option, but for those that do, it can greatly improve performance.

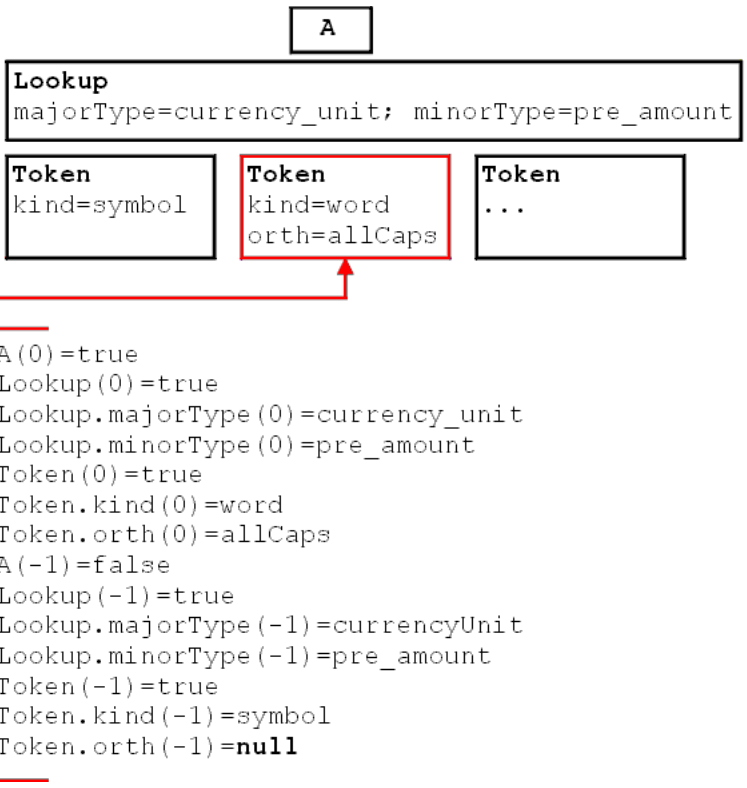


Figure 9.12: Sample attributes and their values

- **OPTIONS**: the contents of the OPTIONS element will be passed verbatim to the ML engine used.

9.24.3 The WEKA Wrapper

GATE provides a wrapper for the WEKA ML Library (http://www.cs.waikato.ac.nz/ml/weka/) in the form of the gate.creole.ml.weka.Wrapper class.

Options for the WEKA wrapper

The WEKA wrapper accepts the following options:

- **CLASSIFIER**: the class name for the classifier to be used.
- **CLASSIFIER-OPTIONS**: the options string as required for the classifier.
- **CONFIDENCE-THRESHOLD**: a double value. If the classifier can provide a probability distribution rather than a simple classification then all possible classifica-

tions that have a probability value larger or equal to the confidence threshold will be considered.

- **DATASET-FILE**: location of the weka arff file. This item is not mandatory, it is possible to specify the file using the saving option on the GUI.

9.24.4 Training an ML model with the ML PR and WEKA wrapper

The ML PR has a Boolean runtime parameter named "training". When the value of this parameter is set to true, the PR will collect a dataset of instances from the documents on which it is run. If the classifier used is an updatable classifier then the ML model will be built while collecting the dataset. If the selected classifier is not updatable, then the model will be built the first time a classification is attempted.

Training a model consists of designing a definition file for the ML PR, and creating an application containing an ML PR. When the application is run over a corpus, the dataset (and the model if possible) is built.

9.24.5 Applying a learnt model

Using the same ML PR, set the "training" parameter to false and run your application.

Depending on the type of the attribute that is marked as class, different actions will be performed when a classification occurs:

- if the attribute is boolean, a new annotation of the specified type will be created with no features;
- if the attribute is nominal or numeric, a new annotation of the specified type will be created with the feature named in the attribute definition having the value predicted by the classifier.

Once a model is learnt, it can be saved and reloaded at a later time. The WEKA wrapper also provides an operation for saving only the dataset in the ARFF format, which can be used for experiments in the WEKA interface. This could be useful for determining the best algorithm to be used and the optimal options for the selected algorithm.

9.24.6 The MAXENT Wrapper

GATE also provides a wrapper for the Open NLP MAXENT library (http://maxent.sourceforge.net/ab
The MAXENT library provides an implementation of the maximum entropy learning algo-

rithm, and can be accessed using the gate.creole.ml.maxent.MaxentWrapper class.

The MAXENT library requires all attributes except for the class attribute to be boolean, and that the class attribute be boolean or nominal. (It should be noted that, within maximum entropy terminology, the class attribute is called the 'outcome'.) Because the MAXENT library does not provide a specific format for data sets, there is no facility to save or load data sets separately from the model, but if there should be a need to do this, the WEKA wrapper can be used to collect the data.

Training a MAXENT model follows the same general procedure as for WEKA models, but the following difference should be noted. MAXENT models are not updateable, so the model will always be created and trained the first time a classification is attempted. The training of the model might take a considerable amount of time, depending on the amount of training data and the parameters of the model.

Options for the MAXENT Wrapper

- **CUT-OFF**: MAXENT features will only be included in the model if they occur at least this many times. (The default value of this parameter is zero.)
- **ITERATIONS**: The number of times the training procedure should iterate when finding the model's parameters (default is 10). In general no more than about 100 iterations should be needed to train a model, and it is recommended that less are used during development to allow for shorter training times.
- **CONFIDENCE-THRESHOLD**: Same as for the WEKA wrapper (see above). However, if this parameter is not set, or is set to zero, the model will not use a confidence threshold, but will simply return the most likely classification.
- **SMOOTHING**: Use smoothing when training the model. Smoothing can improve the accuracy of the learned models, but it will result in longer training times, and training will use more memory. The size of the learned models will also be larger. Generally smoothing will only improve performance for those models trained from small data sets with a few outcomes. With larger data sets with lots of outcomes, it may make performance worse.
- **SMOOTHING-OBSERVATION**: When using smoothing, this will specify the number of times that trainer will imagine that it has seen features which it did not see (default value is 0.1).
- **VERBOSE**: If selected, this will cause the classifier to output more details of its operation during execution.

9.24.7 The SVM Light Wrapper

From version 3.0, GATE provides a wrapper for the SVM Light ML system (http://svmlight.joachims.org). SVM Light is a support vector machine implementation, written in C, which is provided as a set of command line programs. The GATE wrapper takes care of the mundane work of converting the data structures between GATE and SVM Light formats, and calls the command line programs in the right sequence, passing the data back and forth in temporary files. The <WRAPPER> value for this engine is `gate.creole.ml.svmlight.SVMLightWrapper`.

The SVM Light binaries themselves are not distributed with GATE – you should download the version for your platform from http://svmlight.joachims.org and place `svm_learn` and `svm_classify` on your path.

Classifying documents using the SVMLightWrapper is a two phase procedure. In its first phase, SVMWrapper collects data from the pre-annotated documents and builds the SVM model using the collected data to classify the unseen documents in its second phase. Below we describe briefly an example of classifying the start time of the seminar in a corpus of email announcing seminars and provide more details later in the section.

Figure 9.13 explains step by step the process of collecting training data for the SVM classifier. GATE documents, which are pre-annotated with the annotations of type *Class* and feature *type='stime'*, are used as the training data. In order to build the SVM model, we require start and end annotations for each *stime* annotation. We use pre-processor JAPE transduction script to mark the *sTimeStart* and *sTimeEnd* annotations on *stime* annotations. Following this step, the Machine Learning PR (SVMLightWrapper) with training mode set to true collects the training data from all training documents. GATE corpus pipeline, given a set of documents and PRs to execute on them, executes all PRs one by one only on one document at a time. Unless provided in a separate pipleline, it makes it impossible to send all training data (i.e. collected from all documents) altogether to the SVMWrapper using the same pipeline to build the SVM model. This results into the model not being built at the time of collecting training data. The state of the SVMWrapper can be saved to an external file once the training data is collected.

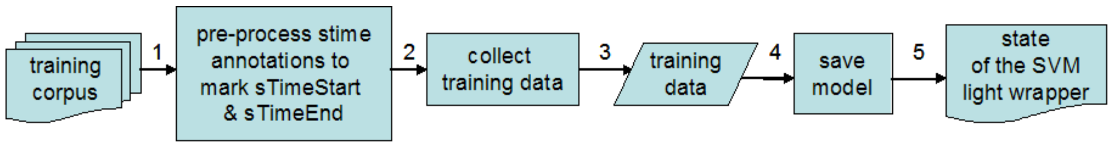

training corpus
1
pre-process stime annotations to mark sTimeStart & sTimeEnd
2
collect training data
3
training data
4
save model
5
state of the SVM light wrapper

Figure 9.13: Flow diagram explaining the SVM training data collection

Before classifying any unseen document, SVM requires the SVM model to be available. In the absence of an up-to-date SVM model, SVMWrapper builds a new one using a command line *SVM_learn* utility and the training data collected from the training corpus. In other words, the first SVM model is built when user tries to classify the first document. At this point the user has an option to save the model somewhere on the external storage. This is in

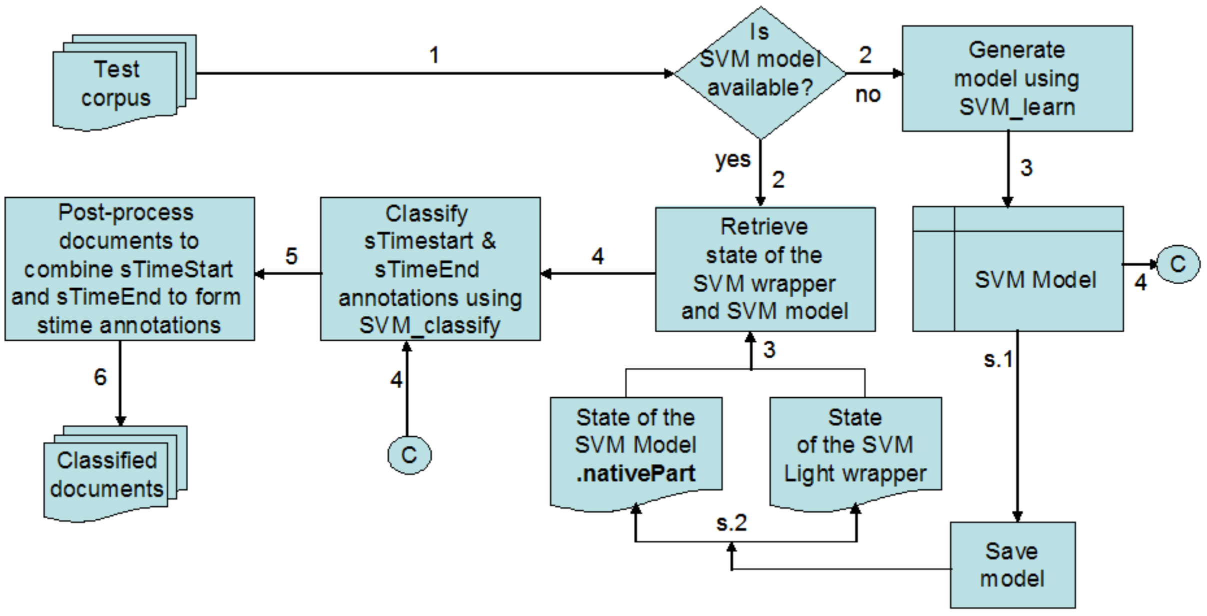

Test corpus
1
Is SVM model available?
2
no
Generate model using SVM_learn
yes
2
3
Post-process documents to combine sTimeStart and sTimeEnd to form stime annotations
5
Classify sTimestart & sTimeEnd annotations using SVM_classify
4
Retrieve state of the SVM wrapper and SVM model
SVM Model
4
C
3
s.1
6
4
State of the SVM Model .nativePart
State of the SVM Light wrapper
Classified documents
C
s.2
Save model

Figure 9.14: Flow diagram explaining document classifying process

order to reload the model prior to classifying other documents and to avoid rebuilding of the SVM model everytime the user classifies a new set of documents. Once the model becomes available, SVMWrapper classifies the unseen documents which creates new *sTimeStart* and *sTimeEnd* annotations over the text. Finally, a post-processor JAPE transduction script is used to combine them into the *sTime* annotation. Figure 9.14 explains this process.

The wrapper allows support vector machines to be created which either do boolean classification or regression (estimation of numeric parameters), and so the class attribute can be boolean or numeric. Additionally, when learning a classifier, SVM Light supports *transduction*, whereby additional examples can be presented during training which do not have the value of the class attribute marked. Presenting such examples can, in some circumstances, greatly improve the performance of the classifier. To make use of this within GATE, the class attribute can be a three value nominal, in which case the first value specified for that nominal in the configuration file will be interpreted as *true*, the second as *false* and the third as *unknown*. Transduction will be used with any instances for which this attribute is set to the *unknown* value. It is also possible to use a two value nominal as the class attribute, in which case it will simply be interpreted as *true* or *false*.

The other attributes can be boolean, numeric or nominal, or any combination of these. If an attribute is nominal, each value of that attribute maps to a separate SVM Light feature. Each of these SVM Light features will be given the value 1 when the nominal attribute has the corresponding value, and will be omitted otherwise. If the value of the nominal is not specified in the configuration file or there is no value for an instance, then no feature will be added.

An extension to the basic functionality of SVM Light is that each attribute can receive a weighting. These weighting can be specified in the configuration file by adding `<WEIGHTING>` tags to the parts of the XML file specifying each attribute. The weighting for the attribute must be specified as a numeric value, and be placed between an opening `<WEIGHTING>` tag and a closing `</WEIGHTING>` one. Giving an attribute a greater weighting, will cause it to play a greater role in learning the model and classifying data. This is achieved by multiplying the value of the attribute by the weighting before creating the training or test data that is passed to SVM Light. Any attribute left without an explicitly specified weighting is given a default weighting of one. Support for these weightings is contained in the Machine Learning PR itself, and so is available to other wrappers, though at time of writing only the SVM Light wrapper makes use of weightings.

As with the MAXENT wrapper, SVM Light models are not updateable, so the model will be trained at the first classification attempt. The SVM Light wrapper supports `<BATCH-MODE-CLASSIFICATION />`, which should be used unless you have a very good reason not to.

The SVM Light wrapper allows both data sets and models to be loaded and saved to files in the same formats as those used by SVM Light when it is run from the command line. When a model is saved, a file will be created which contains information about the state of the SVM Light Wrapper, and which is needed to restore it when the model is loaded again. This file does not, however, contain any information about the SVM Light model itself. If an SVM Light model exists at the time of saving, and that model is up to date with respect to the current state of the training data, then it will be saved as a separate file, with the same name as the file containing information about the state of the wrapper, but with `.NativePart` appended to the filename. These files are in the standard SVM Light model format, and can be used with SVM Light when it is run from the command line. When a model is reloaded by GATE, both of these files must be available, and in the same directory, otherwise an error will result. However, if an up to date trained model does not exist at the time the model is saved, then only one file will be created upon saving, and only that file is required when the model is reloaded. So long as at least one training instance exists, it is possible to bring the model up to date at any point simply by classifying one or more instances (i.e. running the model with the *training* parameter set to false).

Options for the SVM Light engine

Only one `<OPTIONS>` subelement is currently supported:

- `<CLASSIFIER-OPTIONS>` a string of options to be passed to `svm_learn` on the command line. The only difference is that the user should not specify whether regression or classification is to be used, as the wrapper will detect this automatically, based on the type of the class attribute, and set the option accordingly.

9.25 MinorThird

MinorThird is a collection of Java classes for storing text, annotating text, and learning to extract entities and categorize text. It was written primarily by William W. Cohen, a professor at Carnegie Mellon University in the Center for Automated Learning and Discovery, though contributions have been made by many other colleagues and students.

Minorthird's toolkit of learning methods is integrated tightly with the tools for manually and programmatically annotating text. In Minorthird, a collection of documents are stored in a database called a "TextBase". Logical assertions about documents in a TextBase can be made, and stored in a special "TextLabels" object. "TextLabels" are a type of "stand off annotation" — unlike XML markup (for instance), the annotations are completely independent of the text. This means that the text can be stored in its original form, and that many different types of (perhaps incompatible) annotations can be associated with the same TextBase.

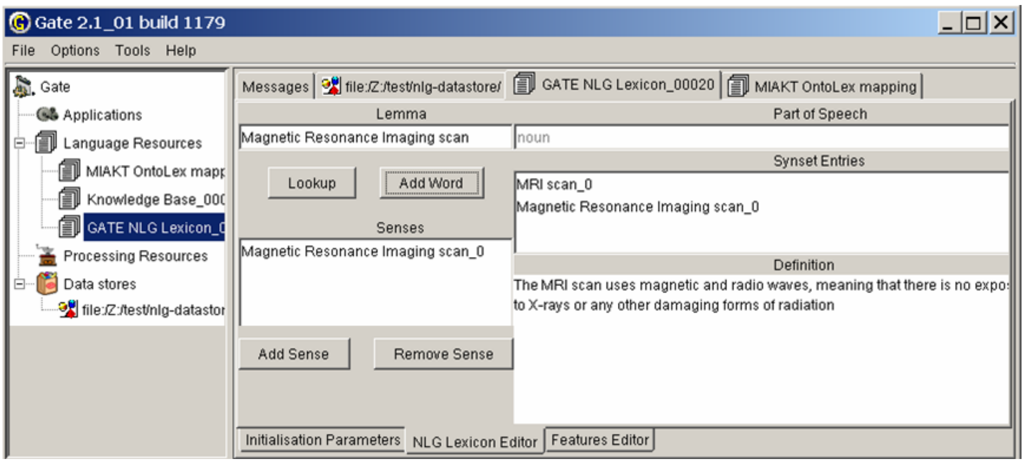
Each TextLabels annotation asserts a category or property for a word, a document, or a subsequence of words. (In Minorthird, a sequence of adjacent words is called a "span".) As an example, these annotations might be produced by human labelers; they might be produced by a hand-writted program, or annotations by a learned program. TextLabels might encode syntactic properties (like shallow parses or part of speech tags) or semantic properties (like the functional role that entities play in a sentence). TextLabels can be nested, much like variable-binding environments can be nested in a programming language, which enables sets of hypothetical or temporary labels to be added in a local scope and then discarded.

Annotated TextBases are accessed in a single uniform way. However, they are stored in one of several schemes. A Minorthird "repository" can be configured to hold a bunch of TextLabels and their associated TextBases.

Moderately complex hand-coded annotation programs can be implemented with a special-purpose annotation language called Mixup, which is part of Minorthird. Mixup is based on a the widely used notion of cascaded finite state transducers, but includes some powerful features, including a GUI debugging environment, escape to Java, and a kind of subroutine call mechanism. Mixup can also be used to generate features for learning algorithms, and all the text-based learning tools in Minorthird are closely integrated with Mixup. For instance, feature extractors used in a learned named-entity recognition package might call a Mixup program to perform initial preprocessing of text.

Minorthird contains a number of methods for learning to extract and label spans from a document, or learning to classify spans (based on their content or context within a document). A special case of classifying spans is classifying entire documents. Minorthird includes a number of state-of-the-art sequential learning methods (like conditional random fields, and discriminative training methods for training hidden Markov models).

One practical difficulty in using learning techniques to solve NLP problems is that the input



Lemma: Magnetic Resonance Imaging scan

Part of Speech: noun

Synset Entries:
MRI scan_0
Magnetic Resonance Imaging scan_0

Senses: Magnetic Resonance Imaging scan_0

Definition: The MRI scan uses magnetic and radio waves, meaning that there is no expos
to X-rays or any other damaging forms of radiation

Figure 9.15: Example NLG Lexicon

to learners is the result of a complex chain of transformations, which begin with text and end with very low-level representations. Verifying the correctness of this chain of derivations can be difficult. To address this problem, Minorthird also includes a number of tools for visualizing transformed data and relating it to the text from which it was derived.

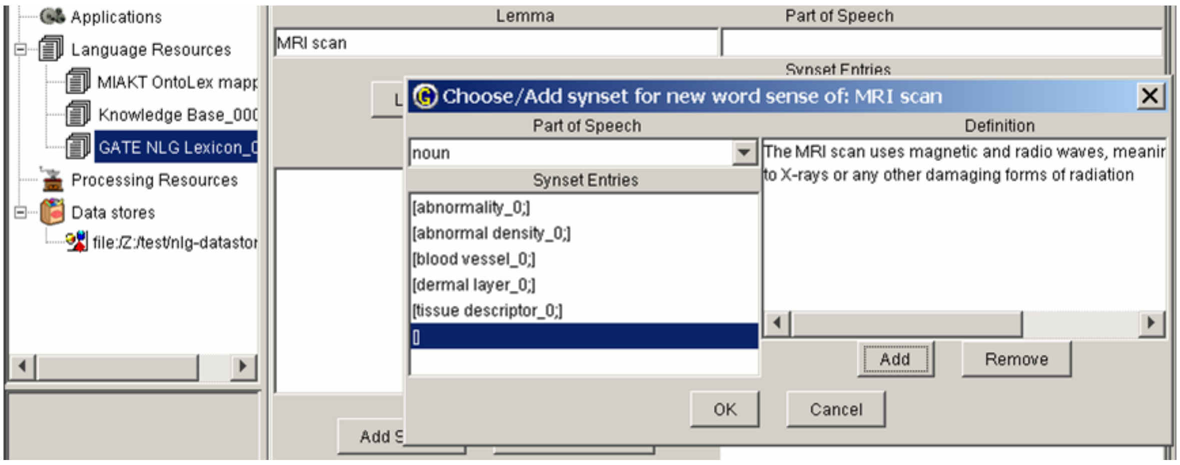
More information about MinorThird can be found at http://minorthird.sourceforge.net/.

9.26 MIAKT NLG Lexicon

In order to lower the overhead of NLG lexicon development, we have created graphical tools for editing, storage, and maintenance of NLG lexicons, combined with a model which connects lexical entries to concepts and instances in the ontology. GATE also provides access to existing general-purpose lexicons such as WordNet and thus enables their use in NLG applications.

The structure of the NLG lexicons is similar to that of WordNet. Each lexical entry has a lemma, sense number, and syntactic information associated with it (e.g., part of speech, plural form). Each lexical entry also belongs to a *synonym set* or *synset*, which groups together all word senses which are synonymous. For example, as shown in Figure 9.15, the lemma "Magnetic Resonance Imaging scan" has one sense, its part of speech is noun, and it belongs to the synset containing also the first sense of the "MRI scan" lemma. Each synset also has a definition, which is shown in order to help the user when choosing the relevant synset for new word senses.

When the user adds a new lemma to the lexicon, it needs to be assigned to an existing synset. The editor also provides functionality for creating a new synset with part of speech and definition. (see Figure 9.16).



MRI scan

noun

[abnormality_0;]
[abnormal density_0;]
[blood vessel_0;]
[dermal layer_0;]
[tissue descriptor_0;]
[]

The MRI scan uses magnetic and radio waves, meanir
to X-rays or any other damaging forms of radiation

Figure 9.16: Editing synset information

The advantage of a synset-based lexicon is that while there can be a one-to-one mapping between concepts and instances in the ontology and synsets, the generator can still use different lexicalisations by choosing them among those listed in the synset (e.g., MRI or Magnetic Resonance Imaging). In other words, synsets effectively correspond to concepts or instances in the ontology and their entries specify possible lexicalisations of these concepts/instances in natural language.

At present, the NLG lexicon encodes only synonymy, while other non-lexical relations present in WordNet like hypernymy and hyponymy (i.e., superclass and subclass relations) are instead derived from the ontology, using the mapping between the synsets and concepts/instances. The reason behind this architectural choice comes from the fact that ontology-based generators ultimately need to use the ontology as the knowledge source. In this framework, the role of the lexicon is to provide lexicalisations for the ontology classes and instances.

9.26.1 Complexity and Generality

The lexicon model was kept as generic as possible by making it incorporate only minimal lexical information. Additional, generator-specific information can be stored in a hash table, where values can be retrieved by their name. Since these are generator specific, the current lexicon user interface does not support editing of this information, although it can be accessed and modified programmatically.

On the other hand, the NLG lexicon is based on synonym sets, so generators which subscribe to a different model of synonymy might be able to access GATE-based NLG lexicons only via a wrapper mapping between the two models.

Given that the lexicon structure follows the WordNet synset model, such a lexicon can potentially be used for language analysis, if the application only requires synonymy. Our

NLG lexicon model does not support yet the richer set of relations in WordNet such as hypernymy, although it is possible to extend the current model with richer relations. Since we used the lexicon in conjunction with the ontology, such non-linguistic relations were instead taken from the ontology.

The NLG lexicon itself is also independent from the generator's input knowledge and its format.

9.27 Kea - Automatic Keyphrase Detection

Kea is a tool for automatic detection of key phrases developed at the University of Waikato in New Zealand. The home page of the project can be found at `http://www.nzdl.org/Kea/`.

This user guide section only deals with the aspects relating to the integration of Kea in GATE. For the inner workings of Kea, please visit the Kea web site and/or contact its authors.

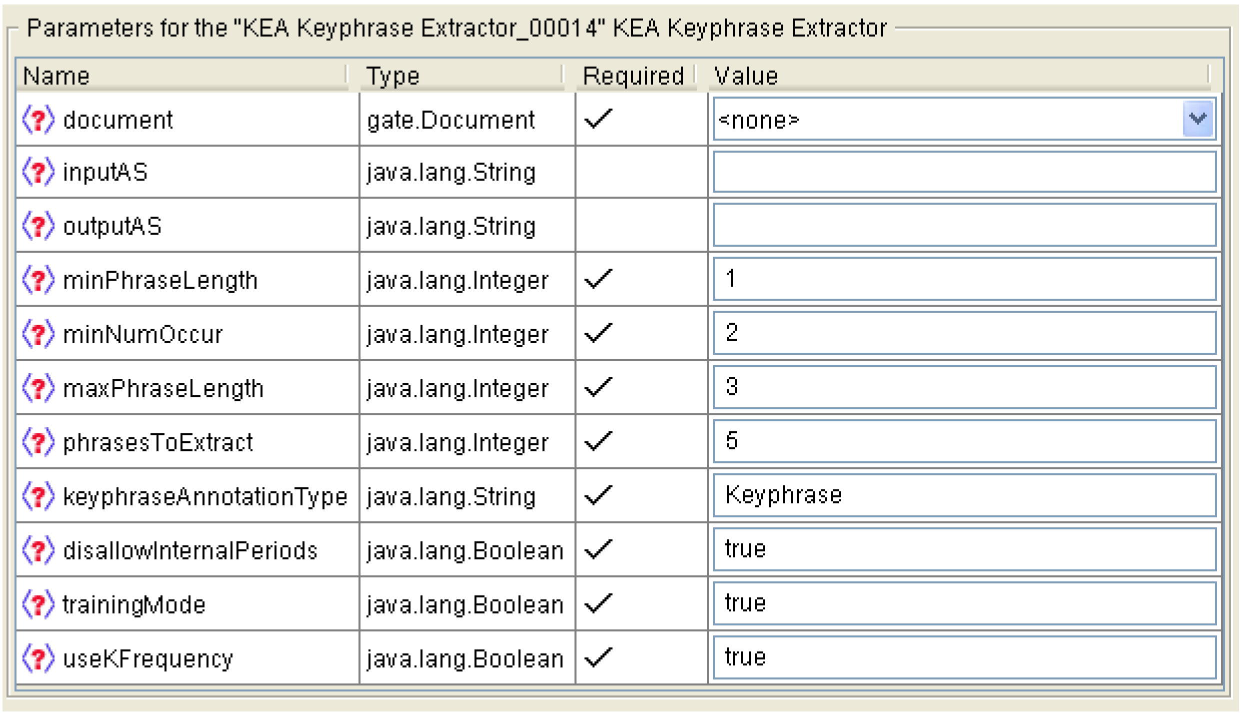
In order to use Kea in GATE, the "Kea" plugin needs to be loaded using the plugins management console. After doing that, two new resource types are available for creation: the "KEA Keyphrase Extractor" (a processing resource) and the "KEA Corpus Importer" (a visual resource associated with the PR).

9.27.1 Using the "KEA Keyphrase Extractor" PR

Kea is based on machine learning and it needs to be trained before it can be used to extract keyphrases. In order to do this, a corpus is required where the documents are annotated with keyphrases. Corpora in the Kea format (where the text and keyphrases are in separate files with the same name but different extensions) can be imported into GATE using the "KEA Corpus Importer" tool. The usage of this tool is presented in a sub-section below.

Once an annotated corpus is obtained, the "KEA Keyphrase Extractor" PR can be used to build a model:

1. load a "KEA Keyphrase Extractor"
2. create a new "Corpus Pipeline" controller.
3. set the corpus for the controller
4. set the 'trainingMode' parameter for the PR to 'true'
5. run the application.



Parameters for the "KEA Keyphrase Extractor_00014" KEA Keyphrase Extractor

Name	Type	Required	Value
document	gate.Document	✓	<none>
inputAS	java.lang.String		
outputAS	java.lang.String		
minPhraseLength	java.lang.Integer	✓	1
minNumOccur	java.lang.Integer	✓	2
maxPhraseLength	java.lang.Integer	✓	3
phrasesToExtract	java.lang.Integer	✓	5
keyphraseAnnotationType	java.lang.String	✓	Keyphrase
disallowInternalPeriods	java.lang.Boolean	✓	true
trainingMode	java.lang.Boolean	✓	true
useKFrequency	java.lang.Boolean	✓	true

Figure 9.17: Parameters used by the Kea PR

After these steps, the Kea PR contains a trained model. This can be used immediately by switching the 'trainingMode' parameter to 'false' and running the PR over the documents that need to be annotated with keyphrases. Another possiblity is to save the model for later use, by right-clicking on the PR name in the right hand side tree and choosing the "Save model" option.

When a previously built model is availalbe, the training procedure does not need to be repeated, the exisiting model can be loaded in memory by selecting the "Load model" option in the PR's pop-up menu.

The Kea PR uses several parameters as seen in Figure 9.17:

document The document to be processed.

inputAS The input annotation set. This parameter is only relevant when the PR is running in training mode and it specifies the annotation set containing the keyphrase annotations.

outputAS The output annotation set. This parameter is only relevant when the PR is running in application mode (i.e. when the 'trainingMode' parameter is set to false) and it specifies the annotation set where the generated keyphrase annotations will be saved.

minPhraseLength the minimum length (in number of words) for a keyphrase.

minNumOccur the minimum number of occurences of a phrase for it to be a keyphrase.

maxPhraseLength the maximum length of a keyphrase.

phrasesToExtract how many different keyphrases should be generated.

keyphraseAnnotationType the type of annotations used for keyphrases.

dissallowInternalPeriods should internal periods be dissallowed.

trainingMode if 'true' the PR is running in training mode; otherwise it is running in application mode.

useKFrequency should the K-frequency be used.

9.27.2 Using Kea corpora

The authors of Kea provide on the project web page a few manually annotated corpora that can be used for training Kea. In order to do this from within GATE, these corpora need to be converted to the format used in GATE (i.e. GATE documents with annotations). This is possible using the "KEA Corpus Importer" tool which is available as a visual resource associated with the Kea PR. The importer tool can be made visible by double-clicking on the Kea PR's name in the resources tree and then selecting the "KEA Corpus Importer" tab, see Figure 9.18.

The tool will read files from a given directory, converting the text ones into GATE documents and the ones containing keyphrases into annotations over the documents.

The user needs to specify a few values:

Source Directory the directory containing the text and key files. This can be typed in or selected by pressing the folder button next to the text field.

Extension for text files the extension used for text fiels (by default .txt).

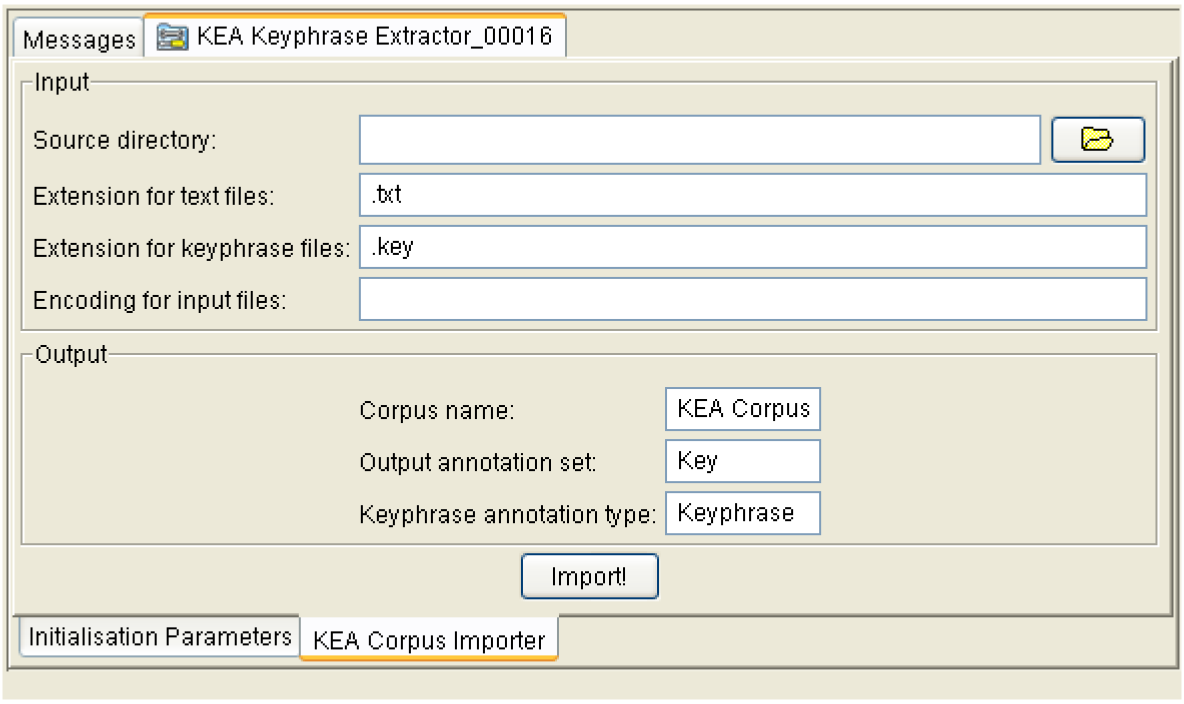
Extension for keyphrase files the extension for the files listing keyphrases.

Encoding for input files the encoding to be used when reading the files.

Corpus name the name for the GATE corpus that will be created.

Output annotaion set the name for the anntoation set that will contain the keyphrases read from the input files.

Keyphrase annotation type the type for the generated annotations.


Messages | KEA Keyphrase Extractor_00016

Input

Source directory:

Extension for text files: .txt

Extension for keyphrase files: .key

Encoding for input files:

Output

Corpus name: KEA Corpus

Output annotation set: Key

Keyphrase annotation type: Keyphrase

Figure 9.18: Options for the "KEA Corpus Importer"

9.28 Ontotext JapeC Compiler

Note: the JapeC compiler does not currently support the new JAPE language features introduced in July–September 2008. If you need to use negation, the `@length` *and* `@string` *accessors, the contextual operators* `within` *and* `contains`*, or any comparison operators other than* `==`*, then you will need to use the standard JAPE transducer instead of JapeC.*

Japec is an alternative implementation of the JAPE language which works by compiling JAPE grammars into Java code. Compared to the standard implementation, these compiled grammars can be several times faster to run. At Ontotext, a modified version of the ANNIE sentence splitter using compiled grammars has been found to run up to five times as fast as the standard version. The compiler can be invoked manually from the command line, or used through the "Ontotext Japec Compiler" PR in the *Jape_Compiler* plugin.

The "Ontotext Japec Transducer" (com.ontotext.gate.japec.JapecTransducer) is a processing resource that is designed to be an alternative to the original Jape Transducer. You can simply replace gate.creole.Transducer with com.ontotext.gate.japec.JapecTransducer in your gate application and it should work as expected.

The Japec transducer takes the same parameters as the standard JAPE transducer:

grammarURL the URL from which the grammar is to be loaded. Note that the Japec Transducer will *only* work on `file:` URLs. Also, the alternative *binaryGrammarURL* parameter of the standard transducer is not supported.

encoding the character encoding used to load the grammars.

ontology the ontology used for ontolog-aware transduction.

Its runtime parameters are likewise the same as those of the standard transducer:

document the document to process.

inputASName name of the AnnotationSet from which input annotations to the transducer are read.

outputASName name of the AnnotationSet to which output annotations from the transducer are written.

The Japec compiler itself is written in Haskell. Compiled binaries are provided for Windows, Linux (x86) and Mac OS X (PowerPC), so no Haskell interpreter is required to run Japec on these platforms. For other platforms, or if you make changes to the compiler source code, you can build the compiler yourself using the Ant build file in the Jape_Compiler plugin directory. You will need to install the latest version of the Glasgow Haskell Compiler[4] and associated libraries. The japec compiler can then be built by running:

```
../../bin/ant japec.clean japec
```

from the Jape_Compiler plugin directory.

9.29 ANNIC

ANNIC (ANNotations-In-Context) is a full-featured annotation indexing and retrieval system. It is provided as part of an extension of the Serial Data-stores, called Searchable Serial Data-store (SSD).

ANNIC can index documents in any format supported by the GATE system (i.e., XML, HTML, RTF, e-mail, text, etc). Compared with other such query systems, it has additional features addressing issues such as extensive indexing of linguistic information associated with document content, independent of document format. It also allows indexing and extraction of information from overlapping annotations and features. Its advanced graphical user interface provides a graphical view of annotation markups over the text, along with an ability to build new queries interactively. In addition, ANNIC can be used as a first step in rule development for NLP systems as it enables the discovery and testing of patterns in corpora.

[4]GHC version 6.4.1 was used to build the supplied binaries for Windows, Linux and Mac

ANNIC is built on top of the Apache Lucene[5] – a high performance full-featured search engine implemented in Java, which supports indexing and search of large document collections. Our choice of IR engine is due to the customisability of Lucene. For more details on how Lucene was modified to meet the requirements of indexing and querying annotations, please refer to [Aswani *et al.* 05].

As explained earlier, SSD is an extension of the serial data-store. In addition to the persist location, SSD asks user to provide some more information (explained later) that it uses to index the documents. Once the SSD has been initiated, user can add/remove documents/-corpora to the SSD in a similar way it is done with other data-stores. When documents are added to the SSD, it automatically tries to index them. It updates the index whenever there is a change in any of the documents stored in the SSD and removes the document from the index if it is deleted from the SSD. Be warned that only the annotation sets, types and features initially indexed will be updated when adding/removing documents to the datastore. This mean, for example, that if you add a new annotation type in one of the indexed document, it will not appear in the results when searching for it.

SSD has an advanced graphical interface that allows users to issue queries over the SSD. Below we explain the parameters required by SSD and how to instantiate it, how to use its graphical interface and how to use SSD from programmatically.

9.29.1 Instantiating SSD

Steps:

1. Right click on "Data Stores" and select "Create datastore".
2. From a drop-down list select "Lucene Based Searchable DataStore".
3. Here, you will see an input window. Please provide these parameters:
   (a) DataStore URL: Select an empty folder where the DS is created.
   (b) Index Location: Select an empty folder. This is where the index will be created.
   (c) Annotation Sets: Here, you can provide one or more annotation sets that you wish to index or exclude from being indexed. In order to be able to index the default annotation set, you must click on the edit list icon and add an empty field to the list. If there are no annotation sets provided, all the annotation sets in all documents are indexed. In addition to all annotation sets a new combined annotation set is created in memory which is a union of all annotations from all the annotation sets of the document being indexed. This set is also indexed in order to allow users to issue queries across various annotation sets.

[5]http://lucene.apache.org

(d) Base-Token Type: (e.g. Token or Key.Token) These are the basic tokens of any document. Your documents must have the annotations of Base-Token-Type in order to get indexed. These basic tokens are used for displaying contextual information while searching patterns in the corpus. In case of indexing more than one annotation set, user can specify the annotation set from which the tokens should be taken (e.g. Key.Token- annotations of type Token from the annotation set called Key). In case user does not provide any annotation set name (e.g. Token), the system searches in all the annotation sets to be indexed and the base-tokens from the first annotation set with the base token annotations are taken. Please note that the documents with no base-tokens are not indexed. However, if the "create tokens automatically" option is selected, the SSD creates base-tokens automatically. Here, each string delimited with white space is considered as a token.

(e) Index Unit Type: (e.g. Sentence, Key.Sentence) This specifies the unit of Index. In other words, annotations lying within the boundaries of these annotations are indexed (e.g. in the case of "Sentences", no annotations that are spanned across the boundaries of two sentences are considered for indexing). User can specify from which annotation set the index unit annotations should be considered. If user does not provide any annotation set, the SSD searches among all annotation sets for index units. If this field is left empty or SSD fails to locate index units, the entire document is considered as a single unit.

(f) Features: Finally, users can specify the annotation types and features that should be indexed or excluded from being indexed. (e.g. SpaceToken and Split). If user wants to exclude only a specific feature of a specific annotation type, he/she can specify it using a '.' separator between the annotation type and its feature (e.g. Person.matches).

4. Click OK. If all parameters are OK, a new empty DS will be created.

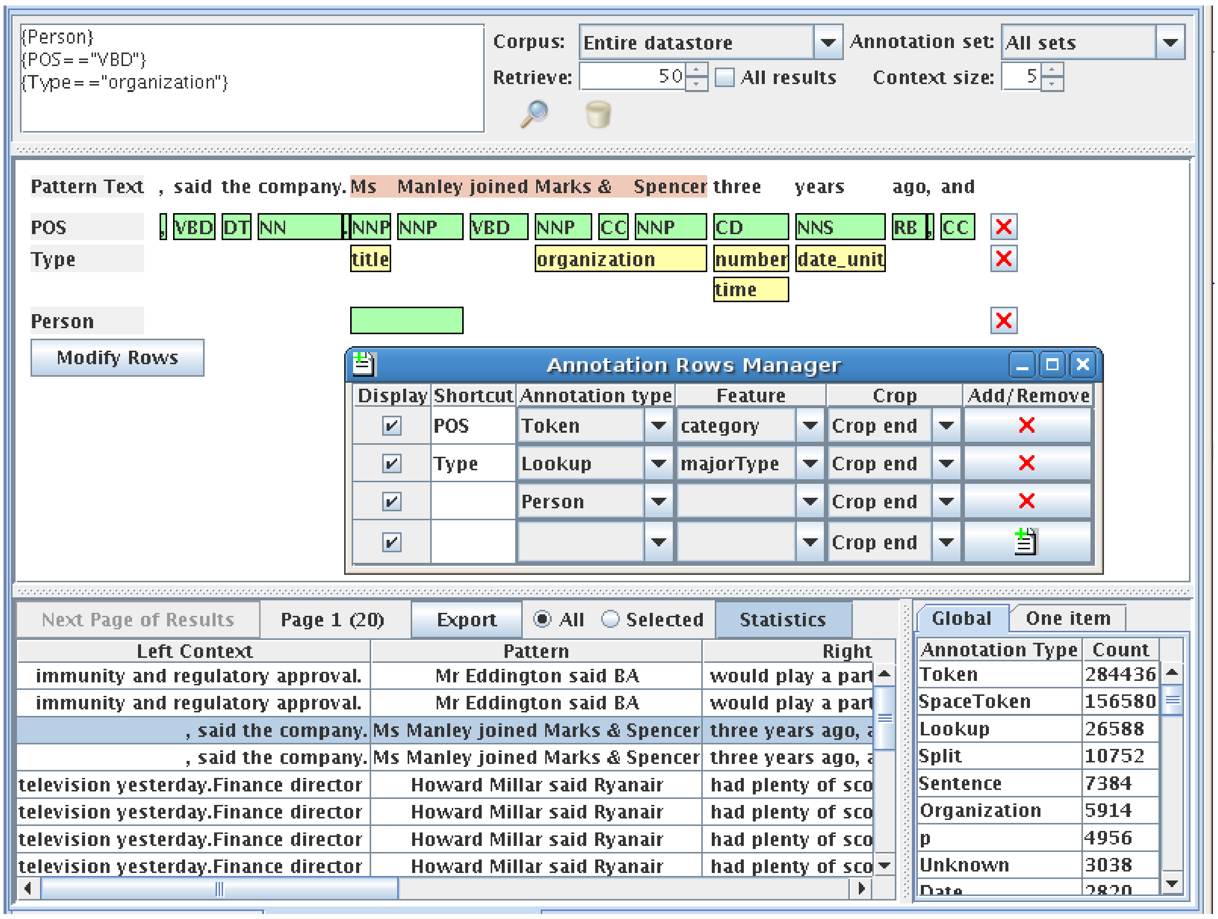
5. Create an empty corpus and save it to the SSD.

6. Populate it with some documents. Each document added to the corpus and eventually to the SSD is indexed automatically. If the document does not have the required annotations, that document is skipped and not indexed.

9.29.2 Search GUI

Overview

Figure 9.19 gives a snapshot of the GUI. The top section contains a text area to write a query, options to select the input data and the output format and two icons to execute and delete a query. The central section shows a graphical visualisation of annotations and values of the result selected in the bottom results table. You can also see the annotation



```
{Person}
{POS=="VBD"}
{Type=="organization"}
```

Corpus: Entire datastore
Annotation set: All sets
Retrieve: 50 ☐ All results
Context size: 5

Pattern Text	, said the company. Ms Manley joined Marks & Spencer three years ago, and
POS	, VBD DT NN . NNP NNP VBD NNP CC NNP CD NNS RB , CC
Type	title organization number date_unit time
Person	🟩

Modify Rows

Annotation Rows Manager

Display	Shortcut	Annotation type	Feature	Crop	Add/Remove
✓	POS	Token	category	Crop end	✗
✓	Type	Lookup	majorType	Crop end	✗
✓		Person		Crop end	✗
✓				Crop end	

Next Page of Results | Page 1 (20) | Export | ● All ○ Selected | Statistics

Left Context	Pattern	Right
immunity and regulatory approval.	Mr Eddington said BA	would play a part
immunity and regulatory approval.	Mr Eddington said BA	would play a part
, said the company.	Ms Manley joined Marks & Spencer	three years ago, a
, said the company.	Ms Manley joined Marks & Spencer	three years ago, a
television yesterday.Finance director	Howard Millar said Ryanair	had plenty of sco
television yesterday.Finance director	Howard Millar said Ryanair	had plenty of sco
television yesterday.Finance director	Howard Millar said Ryanair	had plenty of sco
television yesterday.Finance director	Howard Millar said Ryanair	had plenty of sco

Global | One item

Annotation Type	Count
Token	284436
SpaceToken	156580
Lookup	26588
Split	10752
Sentence	7384
Organization	5914
p	4956
Unknown	3038
Date	2820

Figure 9.19: Searchable Serial Datastore Viewer.

rows manager window where you define which annotation type and feature to display in the central section. The bottom section contains the results table of the query, i.e. the text that matches the query with their left and right contexts, the annotation set and the document. The bottom section contains also a tabbed panes of statistics.

Syntax of queries

SSD enables you to formulate versatile queries using JAPE patterns. JAPE patterns support various query formats. Below we give a few examples of JAPE pattern clauses which can be used as SSD queries. Actual queries can also be a combination of one or more of the following pattern clauses:

1. String
2. {AnnotationType}
3. {AnnotationType == String}
4. {AnnotationType.feature == feature value}
5. {AnnotationType1, AnnotationType2.feature == featureValue}
6. {AnnotationType1.feature == featureValue, AnnotationType2.feature == featureValue}

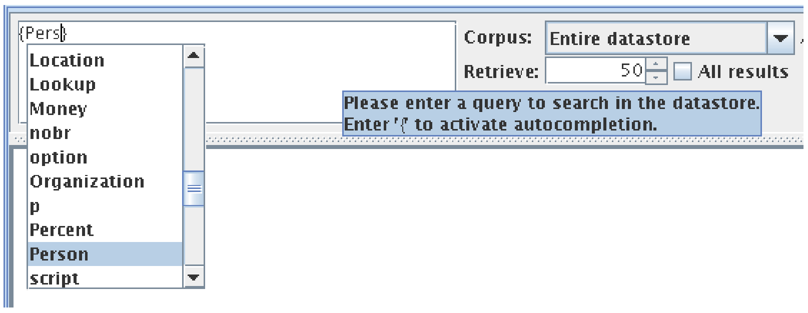
JAPE patterns also support the | (OR) operator. For instance, {A} ({B} | {C}) is a pattern of two annotations where the first is an annotation of type A followed by the annotation of type either B or C. ANNIC supports two operators, + and *, to specify the number of times a particular annotation or a sub pattern should appear in the main query pattern. Here, ({A})+n means one and up to n occurrences of annotation {A} and ({A})*n means zero or up to n occurrences of annotation {A}.

Below we explain the steps to search in SDS.

1. Double click on SSD. You will see an extra tab "Lucene DataStore Searcher". Click on it to activate the searcher GUI.
2. Here you can specify a query to search in your SSD. The query here is a L.H.S. part of the JAPE grammar. Please refer to the following example queries:
   (a) {Person} – This will return annotations of type Person from the SSD
   (b) {Token.string == "Microsoft"} – This will return all occurrences of "Microsoft" from the SSD.

(c) {Person}({Token})*2{Organization} – Person followed by zero or upto two tokens followed by Organization.

(d) {Token.orth==“upperInitial”, Organization} – Token with feature orth with value set to “upperInitial” and which is also annotated as Organization.



Please enter a query to search in the datastore.
Enter '{' to activate autocompletion.

Figure 9.20: Searchable Serial Datastore Viewer - Auto-completion.

Top section

A text-area located in the top left part of the GUI is used to input a query. You can copy/cut/paste with Control+C/X/V, undo/redo your changes with Control+Z/Y as usual. To add a new line, use Control+Enter combination keys.

Auto-completion shown on the figure 9.20 for annotation type is triggered when typing '{' and for feature when typing '.' after a valid annotation type. It shows only the annotation types and features related to the selected corpus and annotation set. If you right-click on an expression it will automatically select the shortest valid enclosing brace and if you click on a selection it will propose you to add quantifiers for allowing the expression to appear zero, one or more times.

To execute the query, click on the magnifying glass icon, use Enter key or Alt+Enter combination keys. To delete the query, click on the trash icon or use Alt+Backspace combination keys.

It is possible to have more than one corpus, each containing a different set of documents, stored in a single data-store. ANNIC, by providing a drop down box with a list of stored corpora, also allows searching within a specific (selected) corpus. Similarly a document can have more than one annotation set indexed and therefore ANNIC also provides a drop down box with a list of indexed annotation sets for the selected corpus.

A large corpus can have many hits for a given query. This may take a long time to refresh the GUI and may create inconvenience while browsing through patterns. ANNIC therefore allows you to specify a number of patterns that you wish to retrieve at once and provides a way to iterate through next pages with the *Next Page of Results* button. Due to technical complexities, it is not possible to visit a previous page. It is however possible to tick a check-box for retrieving all the results at the same time.

Central section

Annotation types and features to show can be selected from the annotation rows manager in clicking on the *Modify Rows* button in the central section. When you choose to show a feature of an annotation (e.g. feature *category* for annotation type *Token*), the central section shows colored rectangles where the annotation type are existing containing values of those features. When you choose to show only one annotation type in letting the feature column empty then all its features are displayed with empty rectangles that show their features values in a pop-up window when the mouse is over the rectangles.

Shortcuts are expression that stand for an "AnnotationType.Feature" expression. For example, on the figure 9.19, the shortcut "POS" stands for the expression "Token.category". The purpose is to make the query more readable.

When you left-clicks on any of the rectangles of the annotations rows, the respective query expression is placed at the caret position in the query text area or replace the selected expression, if any. You can also click on a word on the first line to add it to the query.

Bottom section

In the table of results, ANNIC shows each pattern retrieve from the last query executed on a row and provides a tool tip that shows the query that the selected pattern refers to.

Along with its left and right context texts, it also lists the names of the document and the annotation set that the patterns come from. When the focus changes from one row to another, the central section is updated accordingly. You can sort a table column in clicking on its header.

You can remove a result from the results table or open the document containing it in right-clicking on a result in the results table.

ANNIC provides an *Export* button to export in an HTML file all the results or only the selected results.

A statistics tabbed pane can be displayed on the bottom-right when clicking on the *Statistics* button. There is always a global statistics pane that list the count the occurrences of all annotation types for the selected corpus and annotation set.

Statistics can be obtain in 16 different ways for the datastore, matched spans of the query in the results, with or without contexts and for an annotation type, a annotation type + feature or an annotation type + feature + value. A second pane contains the one item statistics that you can add in right-clicking on a non empty rectangle or on the header of a row in the central section. You can sort a table column in clicking on its header.

9.29.3 Using SSD from your code

```
//how to instantiate a searchabledatastore
===============================

// create an instance of datastore
LuceneDataStoreImpl ds = (LuceneDataStoreImpl)
Factory.createDataStore(‘‘gate.persist.LuceneDataStoreImpl’’, dsLocation);

// we need to set Indexer
Indexer indexer = new LuceneIndexer(new URL(indexLocation));

// set the parameters
Map parameters = new HashMap();

// specify the index url
parameters.put(Constants.INDEX_LOCATION_URL, new URL(indexLocation));

// specify the base token type
// and specify that the tokens should be created automatically
// if not found in the document
parameters.put(Constants.BASE_TOKEN_ANNOTATION_TYPE, ‘‘Token’’);
parameters.put(Constants.CREATE_TOKENS_AUTOMATICALLY, new Boolean(true));

// specify the index unit type
parameters.put(Constants.INDEX_UNIT_ANNOTATION_TYPE, ‘‘Sentence’’);

// specifying the annotation sets "Key" and "Default Annotation Set"
// to be indexed
List<String> setsToInclude = new ArrayList<String>();
setsToInclude.add("Key");
setsToInclude.add("<null>");
parameters.put(Constants.ANNOTATION_SETS_NAMES_TO_INCLUDE, setsToInclude);
parameters.put(Constants.ANNOTATION_SETS_NAMES_TO_EXCLUDE, new ArrayList<String>());

// all features should be indexed
parameters.put(Constants.FEATURES_TO_INCLUDE, new ArrayList<String>());
parameters.put(Constants.FEATURES_TO_EXCLUDE, new ArrayList<String>());

// set the indexer
ds.setIndexer(indexer, parameters);

// set the searcher
```

```
ds.setSearcher(new LuceneSearcher());

//how to search in this datastore
//=====================
// obtain the searcher instance
Searcher searcher = ds.getSearcher();
Map parameters  = new HashMap();

// obtain the url of index
String indexLocation =
new File(((URL) ds.getIndexer().getParameters().get(Constants.INDEX_LOCATION_URL))
.getFile()).getAbsolutePath();
ArrayList indexLocations = new ArrayList();
indexLocations.add(indexLocation);


// corpus2SearchIn = mention corpus name that was indexed here.

// the annotation set to search in
String annotationSet2SearchIn = "Key";

// set the parameter
parameters.put(Constants.INDEX_LOCATIONS,indexLocations);
parameters.put(Constants.CORPUS_ID, corpus2SearchIn);
parameters.put(Constants.ANNOTATION_SET_ID, annotationSet);
parameters.put(Constants.CONTEXT_WINDOW, contextWindow);
parameters.put(Constants.NO_OF_PATTERNS, noOfPatterns);

// search
String query = ‘‘{Person}’’;
Hit[] hits = searcher.search(query, parameters);
```

9.30 Annotation Merging

If we got annotations about the same subject on the same document from different annotators, we need to merge those annotations to form a unified annotations. The merging is applied to the annotations with the same annotation type but in different annotation sets of the same document. We implemented two approaches for annotation merging. The first method takes a parameter *numMinK* and selects the annotation on which at least *numMinK* annotators agree. If two or more merged annotations have the same span, then the annotation with the most supporters is kept and other annotations with the same span are discarded. The second method selects one annotation from those annotations with the same span, which the majority of the annotators support. Note that if one annotator did not create the annotation with the particular span, we count it as one non-support of the annotation with the span. If it turns out that the majority of the annotators did not support the annotation with that span, then no annotation with the span would be put into the merged annotations.

9.30.1 Two implemented methods

The following two static methods in the class **gate.util.AnnotationMerging** are for the merging methods. The two methods have very similar input and output parameters. Each of the methods takes an array of annotation sets, which should be the same annotation type on the same document from different annotators, as input. If there is one annotation feature indicating the annotation label, the name of the annotation feature is another input. Otherwise, set the input parameter for the annotation feature as *null*. The output is a map the key of which is one merged annotation and the value of which represents the annotators (in terms of the indices of the array of annotation sets) who support the annotation. The method also has a boolean input parameter to indicate if or not the annotations from different annotators are based on the same set of instances, which can be determined by the static method *public boolean isSameInstancesForAnnotators(AnnotationSet[] annsA)* in the class **gate.util.IaaCalculation**. One instance corresponds to all the annotations with the same span. If the annotation sets are based on the same set of instances, the merging methods will ensure that the merged annotations are on the same set of instances.

- The Method *public static void mergeAnnogation(AnnotationSet[] annsArr, String nameFeat,*
  HashMap<Annotation,String>mergeAnns, int numMinK, boolean isTheSameInstances) merges the annotations stored in the array *annsArr*. The merged annotation is put into the map *mergeAnns*, which key is the merged annotation and which value is a string containing the indices of elements in the annotation set array *annsArr* which contain that annotation. *NumMinK* specifies the minimal number of the annotators supporting one merged annotation. The boolean parameter *isTheSameInstances* indicate if or not those annotation sets for merging are based on the same instances.

- Method *public static void mergeAnnogationMajority(AnnotationSet[] annsArr, String nameFeat, HashMap<Annotation, String>mergeAnns, boolean isTheSameInstances)* selects the annotations which the majority of the annotators agree on. The meanings of parameters are the same as those in the above method.

9.30.2 Annotation Merging Plugin

The annotation merging methods are wrapped in the plugin such that they can be used in a PR in the GATE GUI. The plugin can be used as a PR in an application of pipeline or corpus pipeline. To use the PR, each document in the pipeline or the corpus pipeline should have the annotation sets for merging. The annotation merging PR has no loading parameter but has several run-time parameters specifying the annotation merging task, explained in the following.

- *annSetOutput*: the annotation set in the current document for storing the merged annotations. For the sake of clearance, it had better not be an existing annotation set.
- *annSetsForMerging*: the annotation sets in the document for merging. It is an optional parameter. If it is not assigned with any value, the annotation sets for merging would be all the annotation sets in the document except the default annotation set. If specified, it is a sequence of the names of the annotation sets for merging, separated by ";". For example, the value "a-1;a-2;a-3" represents three annotation set, "a-1", "a-2" and "a-3".
- *annTypeAndFeats*: the annotation types in the annotation set for merging. It is an optional parameter. It specifies the annotation types in the annotation sets for merging. For each type specified, it may also specify an annotation feature of the type and the values of the feature define the labels of the annotation type. If the parameter is not set a value, the annotation types for merging are all the types in the annotation sets for merging, and no annotation feature for each type is specified. If the parameter is specified, it is a sequence of names of annotation types, separated by ";". If one annotation type has one particular annotation feature to indicate the label of the annotation, the annotation feature will immediately follow the annotation type's name and is separated by "->" in the sequence. For example, the value "SENT->senRel;OPINION_OPR;OPINION_SRC->type" specifies three annotation types, "SENT", "OPINION_OPR" and "OPINION_SRC" and specifies the annotation feature "senRel" and "type" for the two types SENT and OPINION_SRC, respectively but does not specify any feature for the type OPINION_OPR.
- *keepSourceForMergedAnnotations*: should source annotations be kept in the *annSetsForMerging* annotation sets when merged? True by default.
- *mergingMethod*: specifies the method used for merging. Currently it has two values *MajorityVoting* and *MergingByAnnotatorNum*, referring to the two merging methods described above, respectively.
- *minimalAnnNum*: specifies the minimal number of annotators who agree on one annotation in order to put the annotation into merged set, which is needed by the merging method *MergingByAnnotatorNum*. If the value of the parameter is smaller than 1, set the parameter as 1. If the value is bigger than total number of annotation sets

for merging, set the parameter as the total number of annotation sets. If not assigning anything to the parameter in the GUI, it use the default value 1. Note that the parameter does not have any effect on another merging method *MajorityVoting*.

9.31 OntoRoot Gazetteer

OntoRoot Gazetteer is a type of a dynamically created gazetteer that is, in combination with few other generic GATE resources, capable of producing ontology-aware annotations over the given content with regards to given ontology. This gazetteer is a part of Ontology_Based_Gazetteer plugin that has been developed as a part of TAO project.

9.31.1 How does it work?

To produce ontology-aware annotations i.e. annotations that link to the specific concepts or relations from the ontology, it is essential to pre-process the Ontology Resources (e.g., Classes, Instances, Properties) and extract their human-understandable lexicalisations.

As a precondition for extracting human-understandable content from the ontology first a list of the following is being created:

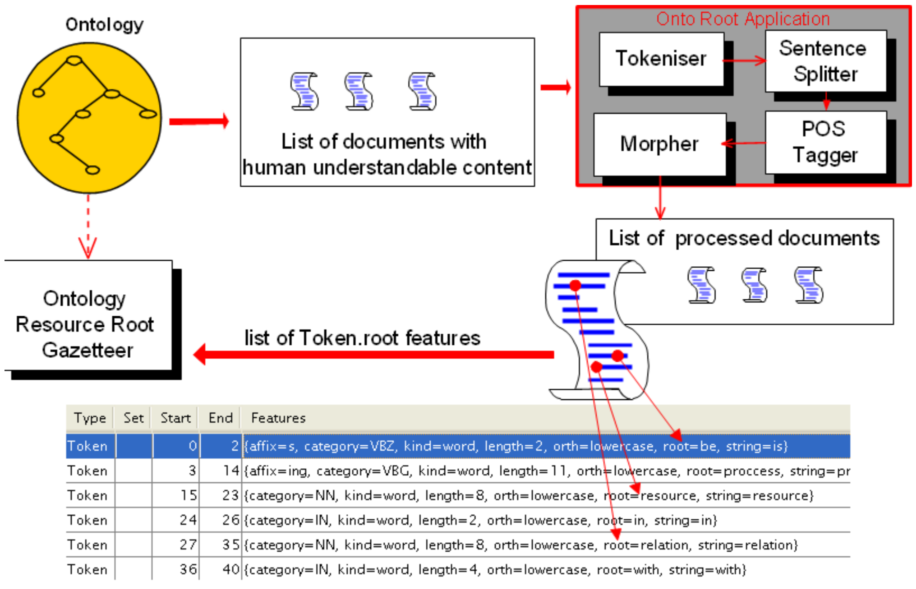
- names of all ontology resources i.e. fragment identifiers [6] and
- assigned property values for all ontology resources (e.g., label and datatype property values)

Each item from the list is further processed so that:

- any name containing dash (`"-"`) or underline (`"_"`) character(s) is processed so that each of these characters is replaced by a blank space. For example, `Project_Name` or `Project-Name` would become a `Project Name`.
- any name that is written in *camelCase* style is actually split into its constituent words, so that `ProjectName` becomes a `Project Name` (optional).
- any name that is a compound name such as 'POS Tagger for Spanish' is split so that both 'POS Tagger' and 'Tagger' are added to the list for processing. In this example, 'for' is a stop word, and any words after it are ignored (optional).

[6] An ontology resource is usually identified by an URI concatenated with a set of characters starting with '#'. This set of characters is called *fragment identifier*. For example, if the URI of a class representing *GATE POS Tagger* is: 'http://gate.ac.uk/ns/gate-ontology#POSTagger', the fragment identifier will be 'POSTagger'.

Each item from this list is analysed separately by the Onto Root Application (ORA) on execution (see figure 9.21). The Onto Root Application first tokenises each linguistic term, then assigns part-of-speech and lemma information to each token.


Ontology
List of documents with human understandable content
Onto Root Application
Tokeniser
Sentence Splitter
POS Tagger
Morpher
List of processed documents
Ontology Resource Root Gazetteer
list of Token.root features

Type	Set	Start	End	Features
Token		0	2	{affix=s, category=VBZ, kind=word, length=2, orth=lowercase, root=be, string=is}
Token		3	14	{affix=ing, category=VBG, kind=word, length=11, orth=lowercase, root=proccess, string=pr
Token		15	23	{category=NN, kind=word, length=8, orth=lowercase, root=resource, string=resource}
Token		24	26	{category=IN, kind=word, length=2, orth=lowercase, root=in, string=in}
Token		27	35	{category=NN, kind=word, length=8, orth=lowercase, root=relation, string=relation}
Token		36	40	{category=IN, kind=word, length=4, orth=lowercase, root=with, string=with}

Figure 9.21: Building Ontology Resource Root (OntoRoot) Gazetteer from the Ontology

As a result of that pre-processing, each token in the terms will have additional feature named 'root', which contains the lemma as created by the morphological analyser. It is this lemma or a set of lemmas which are then added to the dynamic gazetteer list, created from the ontology.

For instance, if there is a resource with a short name (i.e., fragment identifier) *ProjectName*, without any assigned properties the created list before executing the OntoRoot gazetteer collection will contain following the strings:

- '*ProjectName*',
- '*Project Name*' after separating camelCased word and
- '*Name*' after applying heuristic rules.

Each of the item from the list is then analysed separately and the results would be the same as the input strings, as all of entries are nouns given in singular form.

9.31.2 Initialisation of OntoRoot Gazetteer

To initialise the gazetteer there are few mandatory parameters:

- *Ontology* to be processed;

- *Tokeniser*, *POS Tagger* and *GATE Morphological Analyser* to be used during processing.

and few optional ones:

- *useResourceUri*, default is set to true - should this gazetteer analyse resource URIs or not;
- *considerProperties*, default is set to true - should this gazetteer consider properties or not;
- *propertiesToInclude* - checked only if *considerProperties* is set to true - this parameter contains the list of property names (URIs) to be included, comma separated;
- *propertiesToExclude* - checked only if *considerProperties* is set to true - this parameter contains the list of property names to be excluded, comma separated;
- *caseSensitive*, default set to be false -should this gazetteer diferentiate on case;
- *separateCamelCasedWords*, default set to true - should this gazetteer separate emphcamelCased words, e.g. 'ProjectName' into 'Project Name';
- *considerHeuristicRules*, default set to false - should this gazetteer consider several heuristic rules or not. Rules include splitting the words containing spaces, and using prepositions as stop words; for example, if 'pos tagger for spanish' would be analysed, 'for' would be considered as a stop word; heuristically derived would be 'pos tagger' and this would be further used to add 'pos tagger' to the gazeetteer list, with a feature emphheuristical level set to be 0, and 'tagger' with emphheuristical level 1; at runtime lower heuristical level should be prefered. NOTE: setting *considerHeuristicRules* to true can cause a lot of noise for some ontologies and is likely to require implementing an additional filterring resource that will prefer the annotations with the lower heuristic level;

9.32 Chinese Word Segmentation

Unlike English, Chinese text does not have a symbol (or delimiter) such as blank space to explicitly separate a word from the surrounding words. Therefore, for automatic Chinese text processing, we may need a system to recognise the words in Chinese text, a problem known as Chinese word segmentation. The plugin described in this section performs the task of Chinese word segmentation. It is based on our work using the Perceptron learning algorithm for Chinese word segmentation task of the Sighan 2005[7]. [Li *et al.* 05c]. Our Perceptron based system has achieved very good performance in the Sighan-05 task.

[7]See http://www.sighan.org/bakeoff2005/ for the Sighan-05 task

The plugin has the name as *ChineseSegmenter* and is available in the GATE distribution. The corresponding processing resource's name is *Chinese Segmenter PR*. Once you load the PR into GATE, you may put it into a *Pipeline* application[8]. The plugin can be used to learn a model from the segmented Chinese text as training data. It can also use the learned model to segment Chinese text. The plugin can use different learning algorithms to learn different models. It can deal with different codes for Chinese text, such as UTF-8, GB2312 or BIG5. All those difference options can be selected by setting the run-time parameters of the plugin.

The plugin has five run-time parameters, which are described in the following.

- **learningAlg** is a String variable, which specifies the learning algorithm used for producing the model. Currently it has two values, *PAUM* and *SVM*, representing the two popular learning algorithms Perceptron and SVM, respectively. The default value is *PAUM*.
  Generally speaking, SVM may perform better than Perceptron, in particular for small training data. On the other hand, Perceptron's learning is much faster than SVM's. Hence, if you have a small training data, you may want to use SVM to obtain a better model. However, if you have a big training data which is typical for the Chinese word segmentation task, you may want to use Perceptron for learning, because the SVM's learning may take too long time. In addition, using a big training data, the performance of the Perceptron model is quite similar to that of the SVM model. See [Li *et al.* 05c] for the experimental comparison of SVM and Perceptron on the Chinese word segmentation.

- **learningMode** determines the two modes of using the plugin, eitherlearning a model from training data or applying a learned model to segment Chinese text. Accordingly it has two values, *SEGMENTING* and *LEARNING*. The default value is *SEGMENTING*, meaning segmenting the Chinese text.
  Note that you first need to learn a model and then you can use the learned model to segment the text. Several models using the training data used in the Sighan-05 Bakeoff are available for this plugin. So you can use one of those models to segment your Chinese text. More descriptions about the provided models will be given below.

- **modelURL** specifies an URL referring to a directory containing the model. If the plugin is in the *LEARNING* runmode, the model learned will be put into the directory. If it is in the *SEGMENTING* runmode, the plugin will use the model stored in the directory to segment the text. The models learned from the Sighan-05 bakeoff training data will be discussed below.

- **textCode** specifies the code of the text used. For example it can be UTF-8, BIG5, GB2312 or any other code for Chinese text. Note that, when you segment some Chinese

[8]You may put the plugin into a *Corpus Pipeline*. Note that the plugin does not process the documents contained in the corpus which is assigned to the corpus pipeline. In stead, it will process the documents in a directory specified by a run-time parameter. However, you still need to make sure that the assigned corpus contains at least one document in order to allow the plugin process the specified documents.

text using a learned model, the Chinese text should use the same code as the one used by the training text for obtaining the model.

- **textFilesURL** specifies an URL referring to a directory containing the Chinese documents. All the documents contained in this directory (but not those documents contained in its sub-directory if there is any) will be used as input data. In the *LEARNING* runmode, those documents contain the segmented Chinese text as training data. In the *SEGMENTING* runmode, the text in those documents will be segmented. The segmented text will be stored in the corresponding documents in the sub-directory called *segmented*.

The following PAUM models are available for the plugin and can be downloaded from the website http://www.dcs.shef.ac.uk/∼yaoyong/. In detail, those models were learned using the PAUM learning algorithm from the corpora provided by Sighan-05 bakeoff task.

- the PAUM model learned from the PKU training data, using the PAUM learning algorithm and the *UTF-8* code, can be downloaded from *http://www.gate.ac.uk/resources/chineseSeg* *paum-pku-utf8.zip*.
- the PAUM model learned from the PKU training data, using the PAUM learning algorithm and the *GB2312* code, can be downloaded from *http://www.gate.ac.uk/resources/chineseSe* *paum-pku-gb.zip*.
- the PAUM model learned from the AS training data, using the PAUM learning algorithm and the *UTF-8* code, can be downloaded from *http://www.gate.ac.uk/resources/chineseSeg* *as-utf8.zip*.
- the PAUM model learned from the AS training data, using the PAUM learning algorithm and the *BIG5* code, can be downloaded from *http://www.gate.ac.uk/resources/chineseSegm* *as-big5.zip*.

As you can see, those models were learned using different training data and different Chinese text codes of the same training data. The PKU training data are the news articles published in the mainland China and use the simplified Chinese, while the AS training data are the news articles published in Taiwan and use the traditional Chinese. Hence, if your text are in the simplified Chinese, you can use the models trained by the PKU data. For example, if your text are in the traditional Chinese, you need use the models trained by the AS data. If your data are in GB2312 code or any compatible code, you need use the model trained by the corpus in GB2312 code.

Note that the segmented Chinese text (either used as training data or produced by this plugin) use the blank space to separate a word from its surrounding words. Hence, if your data are in the Unicde such as UTF-8, you can use the *GATE Unicode Tokeniser* to process the segmented text to add the Token annotations into your text to represent the Chinese words. Once you get the annotations for all the Chinese words, you can perform further processing such as POS tagging and named entity recogntion.

9.33 Copying Annotations Between Documents

Sometimes a document has two copies, each of which was annotated by two annotators for the same task. Then we want to copy the annotations in one copy to another copy of document, in order to save them using less memory or to use the annotation merging plugin or IAA plugin to process them. This plugin does exactly this task – it copies the specified annotations from one document to another document.

The plugin is named as **copyAS2AnoDoc** and is available with the GATE distribution. When loading the plugin into GATE, it represented as the processing resource **Copy Anns to Another Doc PR**. You need to put the PR into a *Corpus Pipeline* to use it. The plugin does not have any initialisation parameter. It has several run-time parameters, which specify the annotations to be copied, the source documents and target documents. In detail, the run-time parameters are:

- **sourceFilesURL** specifies a directory in which the source documents are in. The plugin copy the annotations from source documents to target documents.
- **inputASName** specifies the name of the annotation set in the source documents. Whole or a part of annotations in the annotation set will be copied.
- **annotationTypes** specifies one or more annotation types in the annotation set *inputASName* which will be copied into target documents. If does not give any value to this parameter, the plugin will copy all annotations in the annotation set.
- **outputASName** specifies the name of the annotation set in the target documents, into which the annotations will be copied. If there is no such annotation set in the target documents, the annotation set will be created automatically.

The **Corpus** parameter of the *Corpus Pipeline* application containing the plugin specifies a corpus which contains the target documents. Given one (target) document in the corpus, the plugin tries to find a source document in the source directory specified by the parameter *sourceFilesURL*, according to the similarity of the names of the source and target documents. The similarity of two file names is calculated by comparing the two strings of names from the start to the end of the strings. The two names have greater similarity if they share more characters from the beginning of the strings. For example, suppose two target documents have the names *aabcc.xml* and *abcab.xml* and the three source files have the names *abacc.xml*, *abcbb.xml* and *aacc.xml*, respectively. Then the target document *aabcc.xml* has the corresponding source document *aacc.xml*, and *abcab.xml* has the corresponding source document *abcbb.xml*. The plugin should copy the annotations within the document if the source and target directories are the same.

Chapter 10

Working with Ontologies

An increasing number of NLP projects make use of taxonomic data structures and of ontologies. The use of NLP techniques for (semi-)automatically generating Semantic Web meta-data is also a growing trend. The advancements in the Semantic Web research area have led to a variety of standards for representing ontologies and an increasing number of tools and programming libraries for managing ontologies are becoming available. All this underlines the need for NLP systems to access ontological information and has led to the addition of support for ontologies in GATE.

The various ontology representation formalisms (such as RDF-Schema [Lassila & Swick 99], OWL and its variants [Dean *et al.* 04], DAML-OIL [Horrocks & vanHarmelen 01]) have their advantages and disadvantages as well as their idiosyncrasies. Rather than attempting to choose one of the formalisms based on what can only be subjective criteria and running the risk of obsolescence when that particular formalism falls out of grace with the research community or is superseded by a newer one, the GATE ontology support is aiming at providing an abstraction layer between the actual representation mechanism and the NLP modules making use of it. It consists of an in-memory data model for ontologies, an API providing access to that representation, a visual resource displaying the information, and input/output capabilities for accessing files containing ontologies using various standards. This approach has well-proved benefits, because it enables each application to use this format-independent model when dealing with ontologies, thus making the application immune to changes in the underlining ontology formats. If a new format needs to be supported, the application can automatically start using ontologies in this format, by simply including the correct tool that converts the format into the common model. From a language engineer's perspective the advantage is that they only need to learn one API and model, rather than having to deal with many different ontology formats. This approach is similar to the way we deal with document formats.

10.1 Data Model for Ontologies

In order to work as an abstraction layer, the GATE ontology implementation supports only those features common to all formalisms, which are also the features most widely used. All the information that is specific to a given representation model and cannot be represented in GATE is ignored. Currently, the ontology data model has support for hierarchies of classes and restrictions, hierarchies of properties and instances (also known as individuals).

10.1.1 Hierarchies of classes and restrictions

The central role in the ontology data model is played by the class hierarchy (or taxonomy). This consists of a set of classes linked by subClassOf, superClassOf and equivalentClassAs relations. Each ontology class has a name and a URI; in most cases the name is the local part of the URI, though this is not enforced. The URIs of all the resources in a given ontology must be unique.

Each class can have a set of superclasses and a set of subclasses; these are used to build the class hierarchy. The subClassOf and superClassOf relations are transitive and methods are provided by the API for calculating the transitive closure for each of these relations given a class. The transitive closure for the set of superclasses for a given class is a set containing all the superclasses of that class, as well as all the superclasses of its direct superclasses, and so on until no more are found. This calculation is finite, the upper bound being the set of all the classes in the ontology. A class that has no superclasses is called a top class. An ontology can have several top classes. Although the GATE ontology API can deal with cycles in the hierarchy graph, these can cause problems for processes using the API and probably indicate an error in the definition of the ontology. Care should be taken to avoid such situations.

A pair of classes can also have an equivalentClassAs (known as sameClassAs in Gate 3.1) relation, which indicates that the two classes are virtually the same and all their properties and instances should be shared.

Restriction is an anonymous type of class and is set on a object or datatype property to restrict some instances of the specified domain of the property to have only certain values (also known as value constraint) or certain number of values (also known as cardinality restriction) for the property. Thus for each restriction there exists atleast three tripples in the repository. One that defines resource as a restriction, another one that indicates on which property the restriction is specified and finally the third one that indicates what is the constraint set on the cardinality or value on the property. There are six types of restrictions:

1. Cardinality Restriction
2. MinCardinality Restriction
3. MaxCardinality Restriction

4. HasValue Restriction
5. AllValuesFrom Restriction
6. SomeValuesFrom Restriction

Please visit the OWL Reference for more detailed information on restrictions.

10.1.2 Instances

Instances are objects that belong to classes. Like classes, each instance has a name and a URI. Each instance can belong to one or more classes and can have properties with values. API methods are provided for getting all the instances of ontology, all the ones that belong to a given class and all the property values for a given instance. There is also a method to retrieve a list of classes that the instance belongs to, using either transitive or direct closure. Like classes, two instances can also have the sameInstanceAs relation, which indicates that the property values assigned to both instances should be shared and that all the properties applicable to one instance are also valid for the other. In addition, there is a differentInstanceAs relation, which declares the instances as disjoint.

10.1.3 Hierarchies of properties

The last part of the data model is made up of hierarchies of properties that can be associated with objects in the ontology. Unlike some other representation models, in GATE, properties do not 'belong' to classes and they are instead first-class citizens of the data model. The specification of the type of objects that properties apply to is done through the means of domains. Similarly, the types of values that a property can take are restricted through the definition of a range. A property with a domain that is an empty set can apply to instances of any type (i.e. there are no restrictions given). Like classes, properties can also have superPropertyOf, subPropertyOf and equivalentPropertyAs relations among them.

GATE defines five types (three in Gate 3.1) of properties:

1. **RDF Property (used to known as Property in Gate 3.1):**

   Each property in an ontology is either an RDF property or a subtype of the RDF property. Each property has constraints on the type of values it can have for its domain and range. In case of the RDF property, it can have any ontology resource as its domain and range. In other words, it is associated to an ontology resource and has an ontology resource as value. For each property, the ontology API provides methods not only to set and get values of its domain and range, but also to check if a given value is valid for it. It also provides methods which can be used for specifying a particular property as its super or subproperty. These methods are inherited by all subtypes of the RDF Property (i.e. Annotation, Datatype and Object properties).

2. **Annotation Property (new in Gate 4):**

   An annotation property is associated with an ontology resource (i.e. a class, property or instance) and can have a Literal (new in Gate 4) as value. A Literal is a Java object that can refer to the URI of any ontology resource or a string (http://www.w3.org/2001/XMLSchema#string) with the specified language or a data type (discussed below) with a compatible value. No two annotation properties can be declared as equivalent. It is also not possible to specify a domain or range for a annotation property or a super or subproperty relation between two annotation properties. Five annotation properties, predefined by OWL, are made available to user whenever a new ontology instance is created: owl:versionInfo, rdfs:label, rdfs:comment, rdfs:seeAlso and rdfs:isDefinedBy. In other words, even when user creates an empty ontology, these annotation properties are created automatically and provided to users.

3. **Datatype Property:**

   A datatype properties is associated with an ontology instance and can have a Literal value that is compatible with its data type (new in Gate 4—Gate 3.1 allowed users to specify a java class as a range value, which is no longer valid in Gate 4). A data type can be one of the pre-defined data types in the GATE ontology API (as listed below).

```
http://www.w3.org/2001/XMLSchema#boolean
http://www.w3.org/2001/XMLSchema#byte
http://www.w3.org/2001/XMLSchema#date
http://www.w3.org/2001/XMLSchema#decimal
http://www.w3.org/2001/XMLSchema#double
http://www.w3.org/2001/XMLSchema#duration
http://www.w3.org/2001/XMLSchema#float
http://www.w3.org/2001/XMLSchema#int
http://www.w3.org/2001/XMLSchema#integer
http://www.w3.org/2001/XMLSchema#long
http://www.w3.org/2001/XMLSchema#negativeInteger
http://www.w3.org/2001/XMLSchema#nonNegativeInteger
http://www.w3.org/2001/XMLSchema#nonPositiveInteger
http://www.w3.org/2001/XMLSchema#positiveInteger
http://www.w3.org/2001/XMLSchema#short
http://www.w3.org/2001/XMLSchema#string
http://www.w3.org/2001/XMLSchema#time
http://www.w3.org/2001/XMLSchema#unsignedByte
http://www.w3.org/2001/XMLSchema#unsignedInt
http://www.w3.org/2001/XMLSchema#unsignedLong
http://www.w3.org/2001/XMLSchema#unsignedShort
```

A set of ontology classes can be specified as a property's domain; then the property can be associated only with the instance belonging to all of the classes specified in the domain.

4. **Object Property:**

   An object properties is associated with an ontology instance and has an instance as value. A set of ontology classes can be specified as property's domain and range. Then the property can be associated only with the instance belonging to all of the classes specified in the domain. Similarly, only the instances that belong to all the classes specified in the range can be set as values. Symmetric and Transitive properties are the two subtypes of Object properties. A symmetric property's domain and range are the same (new in Gate 4).

All properties (except the annotation properties) can be marked as functional properties, which means that for a given instance in their domain, they can only take at most one value, i.e. they define a function in the algebraic sense. Properties inverse to functional properties are marked as inverse functional. If one likes ontology properties with algebraic relations, the semantics of these become apparent.

10.2 Ontology Event Model (new in Gate 4)

An Ontology Event Model (OEM) is implemented and incorporated into the new GATE Ontology API. Under the new OEM, events are fired when a resource is added, modified or deleted from the ontology.

An interface called OntologyModificationListener is created with five methods (see below) that need to be implemented by the listeners of ontology events.

```
public void resourcesRemoved(Ontology ontology, String[] resources);
```

This method is invoked whenever an ontology resource (a class, property or instance) is removed from the ontology. Deleting one resource can also result into the deletion of other dependent resources. For example, deleting a class should also delete all its instances (more details on how deletion works are explained later). The second parameter, an array of strings, provides a list of URIs of resources deleted from the ontology.

```
public void resourceAdded(Ontology ontology, OResource resource);
```

This method is invoked whenever a new resource is added to the ontology. The parameters provide references to the ontology and the resource being added to it.

```
public void ontologyRelationChanged(Ontology ontology, OResource resource1, OResourc
```

This method is invoked whenever a relation between two resources (e.g. OClass and OClass, RDFPRoeprty and RDFProeprty etc) is changed. Example events are the addition or removal of a subclass or a subproperty, two classes or properties being set as equivalent or different and two instances being set as same or different. The first parameter is the reference to the ontology, the next two parameters are the resources being affected and the final parameters is the event type. Please refer to the list of events specified below for different types of events.

```
public void resourcePropertyValueChanged(Ontology ontology, OResource resource, RDFP
```

This method is invoked whenever any property value is added or removed to a resource. The first parameter provides a reference to the ontology in which the event took place. The second provides a reference to the resource affected, the third parameter provides a reference to the property for which the value is added or removed, the fourth parameter is the actual value being set on the resource and the fifth parameter identifies the type of event.

```
public void ontologyReset(Ontology ontology)
```

This method is called whenever ontology is reset. In other words when all resources of the ontology are deleted using the ontology.cleanup method.

The OConstants class defines the static constants, listed below, for various event types.

```
public static final int OCLASS_ADDED_EVENT;
public static final int ANONYMOUS_CLASS_ADDED_EVENT;
public static final int CARDINALITY_RESTRICTION_ADDED_EVENT;
public static final int MIN_CARDINALITY_RESTRICTION_ADDED_EVENT;
public static final int MAX_CARDINALITY_RESTRICTION_ADDED_EVENT;
public static final int HAS_VALUE_RESTRICTION_ADDED_EVENT;
public static final int SOME_VALUES_FROM_RESTRICTION_ADDED_EVENT;
public static final int ALL_VALUES_FROM_RESTRICTION_ADDED_EVENT;
public static final int SUB_CLASS_ADDED_EVENT;
public static final int SUB_CLASS_REMOVED_EVENT;
public static final int EQUIVALENT_CLASS_EVENT;
public static final int ANNOTATION_PROPERTY_ADDED_EVENT;
public static final int DATATYPE_PROPERTY_ADDED_EVENT;
public static final int OBJECT_PROPERTY_ADDED_EVENT;
public static final int TRANSTIVE_PROPERTY_ADDED_EVENT;
public static final int SYMMETRIC_PROPERTY_ADDED_EVENT;
public static final int ANNOTATION_PROPERTY_VALUE_ADDED_EVENT;
public static final int DATATYPE_PROPERTY_VALUE_ADDED_EVENT;
public static final int OBJECT_PROPERTY_VALUE_ADDED_EVENT;
public static final int RDF_PROPERTY_VALUE_ADDED_EVENT;
public static final int ANNOTATION_PROPERTY_VALUE_REMOVED_EVENT;
public static final int DATATYPE_PROPERTY_VALUE_REMOVED_EVENT;
public static final int OBJECT_PROPERTY_VALUE_REMOVED_EVENT;
public static final int RDF_PROPERTY_VALUE_REMOVED_EVENT;
public static final int EQUIVALENT_PROPERTY_EVENT;
public static final int OINSTANCE_ADDED_EVENT;
public static final int DIFFERENT_INSTANCE_EVENT;
public static final int SAME_INSTANCE_EVENT;
public static final int RESOURCE_REMOVED_EVENT;
public static final int RESTRICTION_ON_PROPERTY_VALUE_CHANGED;
public static final int SUB_PROPERTY_ADDED_EVENT;
public static final int SUB_PROPERTY_REMOVED_EVENT;
```

An ontology is responsible for firing various ontology events. Object wishing to listen to the ontology events must implement the methods above and must be registered with the ontology using the following method.

```
addOntologyModificationListener(OntologyModificationListener oml);
```

The following method cancels the registration.

```
removeOntologyModificationListener(OntologyModificationListener oml);
```

10.2.1 What happens when a resource is deleted?

Resources in an ontology are connected with one other. For example, one class can be a sub or superclass of other classes. A resource can have multiple properties attached to it. Taking these various relations into account, change in one resource can affect other resources in the ontology. Below we describe what happens (in terms of what does the GATE ontology API do) when a resource is deleted.

- When a class is deleted
  - A list of all its super classes is obtained. For each class in this list, a list of its subclasses is obtained and the deleted class is removed from it.
  - All subclasses of the deleted class are removed from the ontology. A list of all its equivalent classes is obtained. For each class in this list, a list of its equivalent classes is obtained and the deleted class is removed from it.
  - All instances of the deleted class are removed from the ontology.

  - All properties are checked to see if they contain the deleted class as a member of their domain or range. If so, the respective property is also deleted from the ontology.

- When an instance is deleted
  - A list of all its same instances is obtained. For each instance in this list, a list of its same instances is obtained and the deleted instance is removed from it.
  - A list of all instances set as different from the deleted instance is obtained. For each instance in this list, a list of instances set as different from it is obtained and the deleted instance is removed from it.
  - All the instances of ontology are checked to see if any of their set properties have the deleted instance as value. If so, the respective set property is altered to remove the deleted instance from it.

- When a property is deleted
  - A list of all its super properties is obtained. For each property in this list, a list of its sub properties is obtained and the deleted property is removed from it.
  - All sub properties of the deleted property are removed from the ontology.
  - A list of all its equivalent properties is obtained. For each property in this list, a list of its equivalent properties is obtained and the deleted property is removed from it.
  - All instances and resources of the ontology are checked to see if they have the deleted property set on them. If so the respective property is deleted.

10.3 OWLIM Ontology LR

Ontologies in GATE are classified as language resources. In order to make use of the ontology implementation included in the main distribution, one needs to load the 'Ontology_Tools' CREOLE plug-in; this makes a new language resource 'OWLIM Ontology' available.

The implementation is based on the OWL In Memory (OWLIM), a high performance semantic repository developed at Ontotext (in Bulgaria) as part of the SEKT project. OWLIM is packaged as a Storage and Inference Layer (SAIL) for the Sesame RDF database. OWLIM uses the TRREE engine to perform RDFS, OWL DLP, and OWL Horst reasoning. The most expressive language supported is a combination of limited OWL Lite and unconstrained RDFS. OWLIM offers configurable reasoning support and performance. In the "standard" version of OWLIM (referred to as SwiftOWLIM) reasoning and query evaluation are performed in-memory, while a reliable persistence strategy assures data preservation, consistency and integrity.

OWLIM asks users to provide an XML configuration for the ontology they wish to load into the Sesame RDF database. In order to understand OWL statements, an ontology describing relations between the OWL constructs and the rdfs schema is imported. For example, owl:class is a subclass of the rdfs:class. This allows users to load OWL data into the sesame RDF database.

In order to load an ontology in an OWLIM repository, the user has to provide certain configuration parameters. These include the name of the repository, the URL of the ontology, the default name space, the format of the ontology (RDF/XML, N3, NTriples and Turtle), the URLs or absolute locations of the other ontologies to be imported, their respective name spaces and so on. Ontology files, based on their format, are parsed and persisted in the NTriples format.

In order to utilize the power of OWLIM and the simplicity of GATE Ontology API, GATE provides an implementation of the OWLIM Ontology. Its basic purpose is to hide all the complexities of OWLIM and Sesame and provide an easy to use API and interface to create, load, save and update ontologies. Based on certain parameters that the user provides when instantiating the ontology, a configuration file is dynamically generated to create a dummy repository in memory (unless persistence is specified).

When creating a new ontology, one can use an existing file to pre-populate it with data. If no such file is provided, an empty ontology is created. A detailed description for all the parameters that are available for new ontologies follows:

1. **defaultNameSpace** is the base URI to be used for all new items that are only mentioned using their local name. This can safely be left empty, in which case, while adding new resources to the ontology, users are asked to provide name spaces for each new resource.

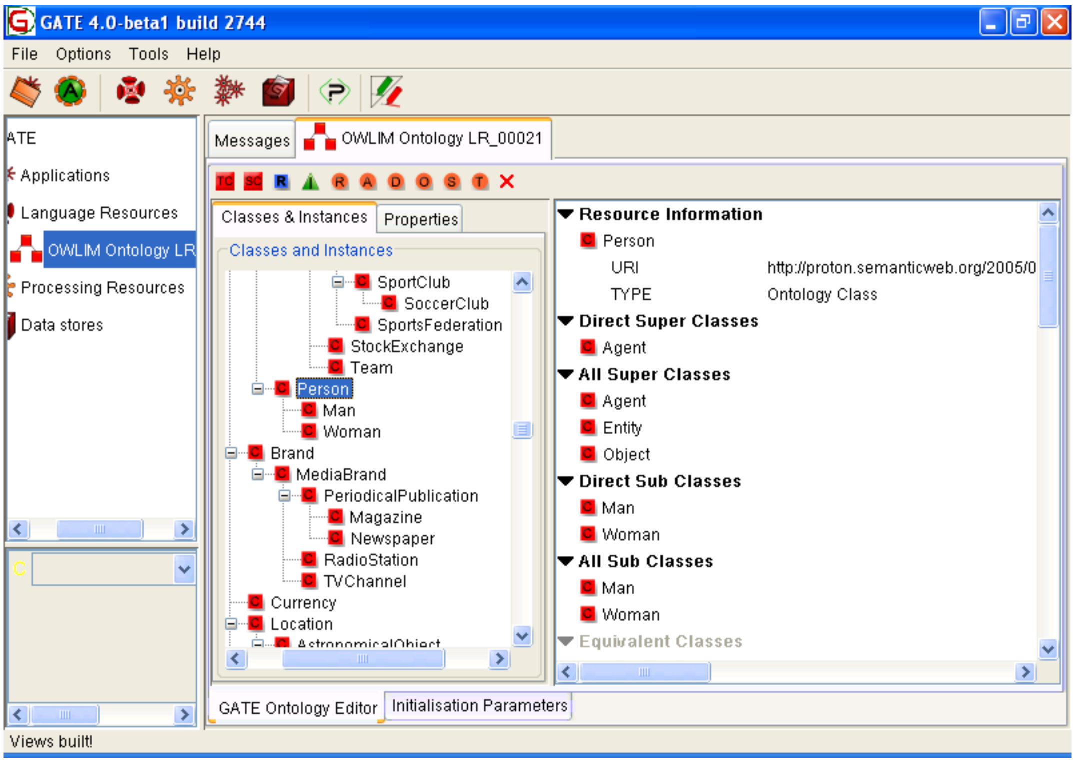
2. As indicated earlier, OWLIM supports four different formats: RDF/XML, NTriples, Turtle and N3. According to the format of the ontology file, user should select one of the four URL options **(rdfXmlURL, ntriplesURL, turtleURL and n3URL (not supported yet))** and provide a URL pointing to the ontology data.

Once an ontology is created, additional data can be loaded that will be merged with the existing information. This can be done by right-clicking on the ontology in the resources tree and selecting 'Load ... data' where "..." is one of the supported formats.

Other options available are cleaning the ontology (deleting all the information from it) and saving it to a file in one of the supported formats.

10.4 GATE's Ontology Editor

GATE's ontology support also includes a viewer/editor that can be used to navigate an ontology and quickly inspect the information relating to any of the objects defined in it—classes and restrictions, instances and their properties. Also, resources can be deleted and new resources can be added through the viewer.



URI	http://proton.semanticweb.org/2005/0
TYPE	Ontology Class

Figure 10.1: The GATE Ontology Viewer

The viewer is divided into two areas. One on the left shows separate tabs for hierarchy of classes and instances and for (new in Gate 4) hierarchy of properties. The view on right hand side shows the details pertaining of the object currently selected in the other two.

First tab on the left view displays a tree which shows all the classes and restrictions defined in the ontology. The tree can have several root nodes—one for each top class in the ontology. The same tree also shows each class's instances. Instances that belong to several classes are shown as children of all the classes they belong to.

Second tab on the left view displays a tree of all the properties defined in the ontology. This tree can also have several root nodes—one for each top property in the ontology. The different types of properties are distinguished with different icons.

Whenever an item is selected in the tree view, the right-hand view is populated with the details that are appropriate for the selected object. For an ontology class, the details include the brief information about the resource such as the URI of the selected class, type of the selected class etc., set of direct superclasses, the set of all superclasses using the transitive closure, the set of direct subclasses, the set of all the subclasses, the set of equivalent classes, the set of applicable property types, the set of property values set on the selected class and the set of instances that belong to the selected class. For a restriction, in addition to the above information, it displays on which property the restriction is applicable to and the what type of the restriction it is.

For an instance, the details displayed include the brief information about the instance, set of direct types (the list of classes this instance is known to belong to), the set of all types this instance belongs to (through the transitive closure of the set of direct types), the set of same instances, the set of different instances and the values for all the properties that are set.

When a property is selected, different information is displayed in the right-hand view according to the property type. It includes the brief information about the property itself, set of direct superproperties, the set of all superproperties (obtained through the transitive closure), the set of direct subproperties, the set of all subproperties (obtained through the transitive closure), the set of equivalent properties, and domain and range information.

As mentioned in the description of the data model, properties are not directly linked to the classes, but rather define their domain of applicability through a set of domain restrictions. This means that the list of properties should not really be listed as a detail for class objects but only for instances. It is however quite useful to have an indication of the types of properties that could apply to instances of a given class. Because of the semantics of property domains, it is not possible to calculate precisely the list of applicable properties for a given class, but only an estimate of it. If a property for instance requires its domain instances to belong to two different classes then it cannot be known with certitude whether it is applicable to either of the two classes—it does not apply to all instances of any of those classes, but only to those instances the two classes have in common. Because of this, such properties will not be listed as applicable to any class.

The information listed in the details pane is organised in sub-lists according to the type of the items. Each sub-list can be collapsed or expanded by clicking on the little triangular button next to the title. The ontology viewer is dynamic and will update the information displayed whenever the underlying ontology is changed through the API.

When double clicked on any resource in the details table the respective resource is selected in the class or in the property tree and the selected resource's details are shown in the details table. To change a property value, user can double click on a value of the property (second column) and the relevant window is shown where user is asked to provide a new value. Along with each property value, a button (with red X caption) is provided. If user wants to remove a property value he or she can click on the button and the property value is deleted.

A new toolbar has been added at the top of the ontology viewer, which contains the following buttons to add and delete ontology resources:

- Add new top class (TC)
- Add new subclass (SC)
- Add new instance (I)
- Add new restriction (R)
- Add new RDF property (R)
- Add new Annotation property (A)
- Add new Datatype property (D)
- Add new Object property (O)
- Add new Symmetric property (S)
- Add new Transitive property (T)
- Remove the selected resource(s) (X)

The tree components allow the user to select more than one node, but the details table on the right-hand side of the GUI only shows the details of the first selected node. The buttons in the toolbar are enabled and disabled based on users' selection of nodes in the tree.

1. **Creating a new top class:**

   A window appears which asks the user to provide details for its namespace (default name space if specified), and class name. If there is already a class with same name in ontology, the GUI shows an appropriate message.

2. **Creating a new subclass:**

   A class can have multiple super classes. Therefore, selecting multiple classes in the ontology tree and then clicking on the "SC" button, automatically considers the selected classes as the super classes. The user is then asked for details for its namespace and class name.

3. **Creating a new instance:**

   An instance can belong to more than one class. Therefore, selecting multiple classes in the ontology tree and then clicking on the "I" button, automatically considers the selected classes as the type of new instance. The user is then prompted to provide details such as namespace and instance name.

4. **Creating a new restriction:**

   As described above, restriction is a type of an anonymous class and is specified on a property with a constraint set on either the number of values it can take or the type of value allowed for instances to have for that property. User can click on the blue "R" square button which shows a window for creating a new restriction. User can select a type of restriction, property and a value constraint for the same. Please note that restrictions are considered as anonymous classes and therefore user does not have to specify any URI for the same but restrictions are named automatically by the system.

5. **Creating a new property:**

   An RDF property can have any ontology resource as its domain and range, so selecting multiple resources and then clicking on the new RDF Property icons shows a window where the selected resources in the tree are already taken as domain for the property. The user is then asked to provide information such as the namespace and the property name. Two buttons are provided to select resources for domain and range—clicking on them brings up a window with drop down box containing a list of resources that can be selected for domain or range and a list of resources selected by the user.

   - Since an annotation property cannot have any domain or range constraints, clicking on the new annotation property button brings up a dialog that asks the user for information such as the namespace and the annotation property name.
   - A datatype property can have one or more ontology classes as its domain and one of the pre-defined datatypes as its range, so selecting one or more classes and then clicking on the new Datatype property icon, brings up a window where the selected classes in the tree are already taken as the domain. The user is then asked to provide information such as the namespace and the property name. A drop down box allows users to select one of the data types from the list.
   - Object, symmetric and transitive properties can have one or more classes as their domain and range. For a symmetric property the domain and range are the same. Clicking on any of these options brings up a window where user is asked to provide information such as the namespace and the property name. The user is also given two buttons to select one or more classes as values for domain and range.

6. **Removing the selected resources:**

   All the selected nodes are removed when user clicks on the "X" button. Please note that since ontology resources are related in various ways, deleting a resource can affect other resources in the ontology; for example, deleting a resource can cause other resources in the same ontology to be deleted too.

7. **Setting properties on instances/classes:**

   Right-clicking on an instance brings up a menu that provides a list of properties that are inherited and applicable to its classes. Selecting a specific property from the menu allows the user to provide a value for that property. For example, if the property is an Object property, a new window appears which allows the user to select one or more instances which are compatible to the range of the selected property. The selected instances are then set as property values. For classes, all the properties (e.g. annotation and RDF properties) are listed on the menu.

8. **Setting relations among resources:**

   Two or more classes, or two or more properties, can be set as equivalent; similarly two or more instances can be markes as the same. Right-clicking on a resource brings up a menu with an appropriate option (Equivalent Class for ontology classes, Same As Instance for instances and Equivalent Property for properties) which when clicked then brings up a window with a drop down box containing a list of resources that the user can select to specify them as equivalent or the same.

Ontology can be saved in different formats (rdf/xml, ntriples, n3 and turtle) using the options provided in the options menu that can be invoked by right clicking on the instance of an ontology in the GATE GUI. All the changes made to the ontology are logged and stored as an ontology feature. User can also export these changes to a file by selecting the "Save Ontology Event Log" option from the options menu. Similarly, users can also load the exported event log and apply the changes on a different ontology by using the "Load Ontology Event Log" option. Any change made to the ontology can be described by a set of triples either added or deleted from the repository. For example, In GATE API implementation, addition of a new instance results into addition of two statements into the repository:

```

// Adding a new instance "Rec1" of type "Recognized"
// Here + indicates the addition
+ <http://proton.semanticweb.org/2005/04/protons#Rec1>
        <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
        <http://proton.semanticweb.org/2005/04/protons#Recognized>

// Adding a label (annotation property) to the instance with value "Rec Instanc
+ <http://proton.semanticweb.org/2005/04/protons#Rec1>
        <http://www.w3.org/2000/01/rdf-schema#label>
        <Rec Instance>
        <http://www.w3.org/2001/XMLSchema#string>
```

The event log therefore contains a list of such triples, the latest change being at the bottom of the change log. Each triple consists of a subject followed by a predicate followed by an object. Below we give an illustration explaining the syntax used for recording the changes.

```

// Adding a new instance "Rec1" of type "Recognized"
// Here + indicates the addition
+ <http://proton.semanticweb.org/2005/04/protons#Rec1>
        <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
        <http://proton.semanticweb.org/2005/04/protons#Recognized>

// Adding a label (annotation property) to the instance with value "Rec Instanc
+ <http://proton.semanticweb.org/2005/04/protons#Rec1>
        <http://www.w3.org/2000/01/rdf-schema#label>
        <Rec Instance>
        <http://www.w3.org/2001/XMLSchema#string>

// Adding a new class called TrustSubClass
```

```
+ <http://proton.semanticweb.org/2005/04/protons#TrustSubClass>
        <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
        <http://www.w3.org/2002/07/owl#Class>

// TrustSubClass is a subClassOf the class Trusted
+ <http://proton.semanticweb.org/2005/04/protons#TrustSubClass>
        <http://www.w3.org/2000/01/rdf-schema#subClassOf>
        <http://proton.semanticweb.org/2005/04/protons#Trusted>

// Deleting a property called hasAlias and all relevant statements
// Here - indicates the deletion
// * indicates any value in place
- <http://proton.semanticweb.org/2005/04/protons#hasAlias> <*> <*>
- <*> <http://proton.semanticweb.org/2005/04/protons#hasAlias> <*>
- <*> <*> <http://proton.semanticweb.org/2005/04/protons#hasAlias>

// Deleting a label set on the instance Rec1
- <http://proton.semanticweb.org/2005/04/protons#Rec1>
        <http://www.w3.org/2000/01/rdf-schema#label>
        <Rec Instance>
        <http://www.w3.org/2001/XMLSchema#string>

// Reseting the entire ontology (Deleting all statements)
- <*> <*> <*>
```

10.5 Instantiating OWLIM Ontology using GATE API

The following code demonstrates how to use the GATE API to create an instance of OWLIM Ontology.

```
// step 1: initialize GATE
Gate.init();

// step 2: load the Ontology_Tools plugin
File ontoHome = new File(Gate.getPluginsHome(),"Ontology_Tools");
Gate.getCreoleRegister().addDirectory(ontoHome.toURL());

// step 3: set the parameters
FeatureMap fm = Factory.newFeatureMap();
fm.put("rdfXmlURL", url-of-the-ontology);

// step 4: finally create an instance of ontology
Ontology ontology = (Ontology)
Factory.createResource("gate.creole.ontology.owlim.OWLIMOntologyLR", fm);

// retrieving a list of top classes
Set<OClass> topClasses = ontology.getOClasses(true);

// for all top classes, printing their direct sub classes
Iterator<OClass> iter = topClasses.iterator();
```

```java
while(iter.hasNext()) {
   Set<OClass> dcs = iter.next().getSubClasses(OConstants.DIRECT_CLOSURE);
   for(OClass aClass : dcs) {
        System.out.println(aClass.getURI().toString());
   }
}

// creating a new class
// false indicates that it is not an anonymous URI
URI aURI = new URI("http://sample.en/owlim#Organization", false);
OClass organizationClass = ontology.addOClass(aURI);

// creating a new Datatype property called name
// with domain set to Organization
// with datatype set to string
URI dURI = new URI("http://sample.en/owlim#Name", false);
Set<OClass> domain = new HashSet<OClass>();
domain.add(organizationClass);
DatatypeProperty dp = ontology.addDatatypeProperty(dURI, domain,
                                        Datatype.getStringDataType());

// creating a new instance of class organization called IBM
URI iURI = new URI("http://sample.en/owlim#IBM", false);
OInstance ibm = Ontology.addOInstance(iURI, organizationClass);

// assigning a Datatype property, name to ibm
ibm.addDatatypePropertyValue(dp, new Literal("IBM Corporation",
                                        dp.getDataType());

// get all the set values of all Datatype properties on the instance ibm
Set<DatatypeProperty> dps = Ontology.getDatatypeProperties();
for(DatatypeProperty dp : dps) {
 List<Literal> values = ibm.getDatatypePropertyValues(dp);
 System.out.println("DP : "+dp.getURI().toString());
 for (Literal l : values) {
   System.out.println("Value : "+l.getValue());
   System.out.println("Datatype : "+ l.getDataType().getXmlSchemaURI().toString
 }
}

// export data to a file in the ntriples format
BufferedWriter writer = new BufferedWriter(new FileWriter(someFile));
String output = ontology.getOntologyData(
                   OConstants.ONTOLOGY_FORMAT_NTRIPLES);
writer.write(output);
writer.flush();
writer.close();
```

10.6 Ontology-Aware JAPE Transducer

One of the GATE components that makes use of the ontology support is the JAPE transducer (see Chapter 7). Combining the power of ontologies with JAPE's pattern matching mechanisms can ease the creation of applications.

In order to use ontologies with JAPE, one needs to load an ontology in GATE before loading the JAPE transducer. Once the ontology is known to the system, it can be set as the value for the optional `ontology` parameter for the JAPE grammar. Doing so alters slightly the way the matching occurs when the grammar is executed. If a transducer is ontology-aware (i.e. it has a value set for the 'ontology' parameter) it will treat all occurrences of the feature named `class` differently from the other features of annotations. The values for the feature `class` on any type of annotation will be considered to be the names of classes belonging the ontology and the matching between two values will not be based on simple equality but rather hierarchical compatibility. For example if the ontology contains a class named '`Politician`', which is a sub class of the class '`Person`', then a pattern of `{Entity.class == ``Person''}` will successfully match an annotation of type `Entity` with a feature `class` having the value "`Politician`". If the JAPE transducer were not ontology-aware, such a test would fail.

This behaviour allows a larger degree of generalisation when designing a set of rules. Rules that apply several types of entities mentioned in the text can be written using the most generic class they apply to and need not be repeated for each subtype of entity. One could have rules applying to `Location`s without needing to know whether a particular location happens to be a country or a city.

If a domain ontology is available at the time of building an application, using it in conjunction with the JAPE transducers can significantly simplify the set of grammars that need to be written.

The ontology does not normally affect actions on the right hand side of JAPE rules, but when Java is used on the right hand side, then the ontology becomes accessible via a local variable named `ontology`, which may be referenced from within the right-hand-side code.

In Java code, the `class` feature should be referenced using the static final variable, `LOOKUP_CLASS_FEATURE_NAME`, that is defined in `gate.creole.ANNIEConstants`.

10.7 Annotating text with Ontological Information

The ontology-aware JAPE transducer enables the text to be linked to classes in an ontology by means of annotations. Essentially this means that each annotation can have a class and ontology feature. To add the relevant class feature to an annotation is very easy: simply add a feature "class" with the classname as its value. To add the relevant ontology, use `ontology.getURL()`.

Below is a sample rule which looks for a location annotation and identifies it as a "Mention" annotation with the class "Location" and the ontology loaded with the ontology-aware JAPE transducer (via the runtime parameter of the transducer).

```
Rule: Location

({Location}):mention

-->
{
// create an annotation set consisting of all the annotations for each tag
gate.AnnotationSet mentionSet = (gate.AnnotationSet)bindings.get(``mention'');

// create the ontology and class features
   FeatureMap features = Factory.newFeatureMap();
   features.put(``ontology'', ontology.getURL());
   features.put(``class'', ``Location'');

// create the new annotation
 annotations.add(mentionSet.firstNode(), mentionSet.lastNode(), ''Mention'',
 features);
}
```

10.8 Populating Ontologies

Another typical application that combines the use of ontologies with NLP techniques is finding mentions of entities in text. The scenario is that one has an existing ontology and wants to use Information Extraction to populate it with instances whenever entities belonging to classes in the ontology are mentioned in the input texts.

Let us assume we have an ontology and an IE application that marks the input text with annotations of type '`Mention`' having a feature '`class`' specifying the class of the entity mentioned. The task we are seeking to solve is to add instances in the ontology for every `Mention` annotation.

The example presented here is based on a JAPE rule that uses Java code on the action side in order to access directly the ontology API:

```
Rule: FindEntities
({Mention}):mention
-->
{
  //find the annotation matched by LHS
  //we know the annotation set returned
  //will always contain a single annotation
  Annotation mentionAnn = (Annotation)
    ((AnnotationSet)bindings.get("mention")).
    iterator().next();

12     //find the class of the mention
13     String className = (String)mentionAnn.getFeatures().
14       get(gate.creole.ANNIEConstants.LOOKUP_CLASS_FEATURE_NAME);
15
16     //find the text covered by the annotation
17     String mentionName;
18     try {
19       mentionName = doc.getContent().
20         getContent(
21           mentionAnn.getStartNode().getOffset(),
22           mentionAnn.getEndNode().getOffset()).
23         toString();
24     } catch (InvalidOffsetException e) {
25       throw new GateRuntimeException(e); //this should never happen
26     }
27
28     //add the instance to the ontology
29     //get the first class with that name
30     gate.creole.ontology.OClass aClass = null;
31     for (gate.creole.ontology.OResource aResource :
32          ontology.getOResourcesByName(className)) {
33       if (aResource instanceof gate.creole.ontology.OClass) {
34         aClass = (gate.creole.ontology.OClass) aResource;
35         break;
36       }
37     }
38     if (aClass == null) {
39       System.err.println("Error class \"" + className + "\" does not exist!");
40
41     } else {
42       //check if the instance already exists
43       //assume that the mentionName instances are unique instances
44       gate.creole.ontology.URI uri = gate.creole.ontology.OntologyUtilities
45         .createURI(ontology, mentionName, false);
46       if (!ontology.containsOInstance(uri)) {
47         // create the instance in the ontology
48         ontology.addOInstance(uri, aClass);
49       }
50     }
51   }
```

This will match each annotation of type `Mention` in the input and assign it to a label '`mention`'. That label is then used in the right hand side to find the annotation that was matched by the pattern (lines 5–10); the value for the `class` feature of the annotation is used to identify the ontological class name (lines 12–14); and the annotation span is used to extract the text covered in the document (lines 16–26). Once all these pieces of information are available, the addition to the ontology can be done. First the right class in the ontology is identified using the class name (lines 28–37) and then a new instance for that class is created (lines 38–50).

Beside JAPE, another tool that could play a part in this application is the Ontological

Gazetteer see Section 5.2, which can be useful in bootstrapping the IE application that finds entity mentions.

The solution presented here is purely pedagogical as it does not address many issues that would be encountered in a real life application solving the same problem. For instance, it is naïve to assume that the name for the entity would be exactly the text found in the document. In many cases entities have several aliases – for example the same person name can be written in a variety of forms depending on whether titles, first names, or initials are used. A process of name normalisation would probably need to be employed in order to make sure that the same entity, regardless of the textual form it is mentioned in, will always be linked to the same ontology instance.

For a detailed description of the ontology API, please consult the JavaDoc documentation.

10.9 Ontology Annotation Tool

The Ontology Annotation Tool (OAT) is a GATE plugin available from the Ontology Tools plugin set, which enables a user to manually annotate a text with respect to one or more ontologies. The required ontology must be selected from a pull-down list of available ontologies.

The OAT tool supports annotation with information about the ontology classes, instances and properties.

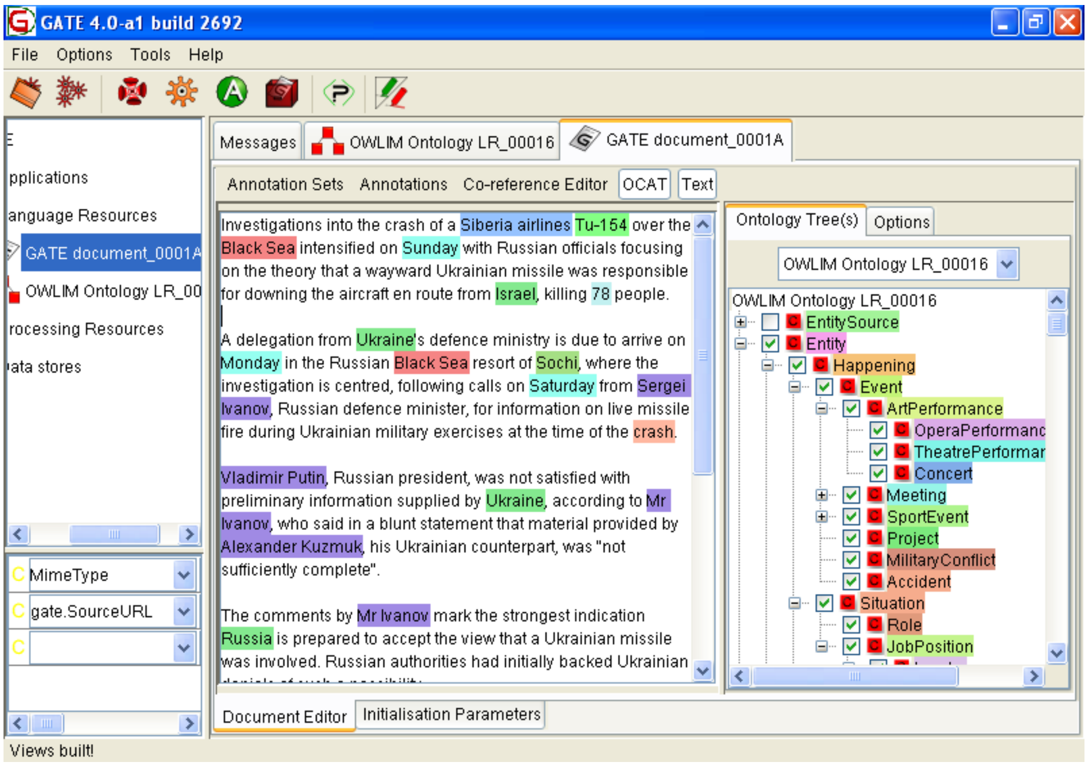
10.9.1 Viewing Annotated Texts

Ontology-based annotations in the text can be viewed by selecting in the ontology tree the desired classes or instances (see Figure 10.2). By default, when a class is selected, all of its sub-classes and instances are also automatically selected and their mentions are highlighted in the text. There is an option to disable this default behaviour (see Section 10.9.4).

Figure 10.2 shows the mentions of each class and instance in a different colour. These colours can be customised by the user by clicking on the class/instance names in the ontology tree. It is also possible to expand and collapse branches of the ontology.

10.9.2 Editing Existing Annotations

In order to view the class/instance of a highlighted annotation in the text (e.g., United States - see Figure 10.3), hover the mouse over it and an edit dialogue will appear. It shows the current class or instance (Country in our example) and allows the user to delete it or change it. To delete an existing annotation, press the Delete button.



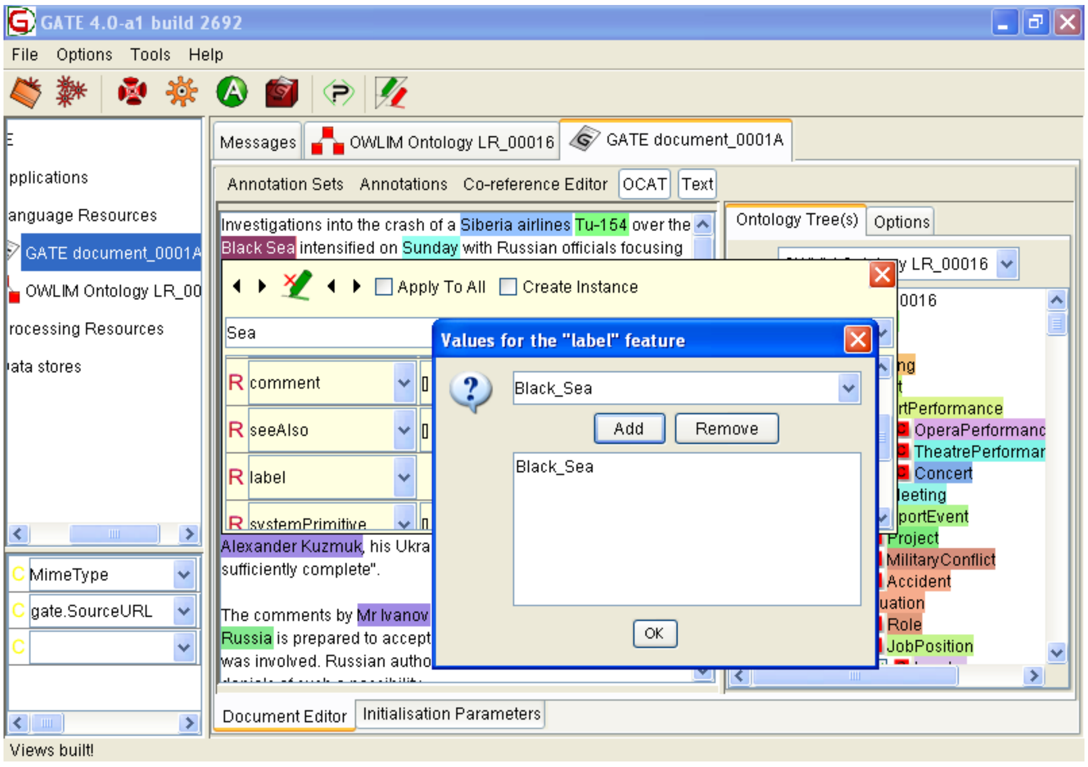
Investigations into the crash of a Siberia airlines Tu-154 over the Black Sea intensified on Sunday with Russian officials focusing on the theory that a wayward Ukrainian missile was responsible for downing the aircraft en route from Israel, killing 78 people.

A delegation from Ukraine's defence ministry is due to arrive on Monday in the Russian Black Sea resort of Sochi, where the investigation is centred, following calls on Saturday from Sergei Ivanov, Russian defence minister, for information on live missile fire during Ukrainian military exercises at the time of the crash.

Vladimir Putin, Russian president, was not satisfied with preliminary information supplied by Ukraine, according to Mr Ivanov, who said in a blunt statement that material provided by Alexander Kuzmuk, his Ukrainian counterpart, was "not sufficiently complete".

The comments by Mr Ivanov mark the strongest indication Russia is prepared to accept the view that a Ukrainian missile was involved. Russian authorities had initially backed Ukrainian

Figure 10.2: Viewing Ontology-Based Annotations



Investigations into the crash of a Siberia airlines Tu-154 over the Black Sea intensified on Sunday with Russian officials focusing

Alexander Kuzmuk, his Ukra sufficiently complete".

The comments by Mr Ivanov Russia is prepared to accept was involved. Russian autho

Figure 10.3: Editing Existing Annotations

A class or instance can be changed by starting to type the name of the new class in the combo-box. Then it displays a list of available classes and instances, which start with the typed string. For example, if we want to change the type from Country to Location, we can type “Lo” and all classes and instances which names start with Lo will be displayed. The more characters are typed, the fewer matching classes remain in the list. As soon as one sees the desired class in the list, it is chosen by clicking on it.

It is possible to apply the changes to all occurrences of the same string and the same previous class/instance, not just to the current one. This is useful when annotating long texts. The user needs to make sure that they still check the classes and instances of annotations further down in the text, in case the same string has a different meaning (e.g., bank as a building vs. bank as a river bank).

The edit dialogue also allows correcting annotation offset boundaries. In other words, user can expand or shrink the annotation offsets' boundaries by clicking on the relevant arrow buttons.

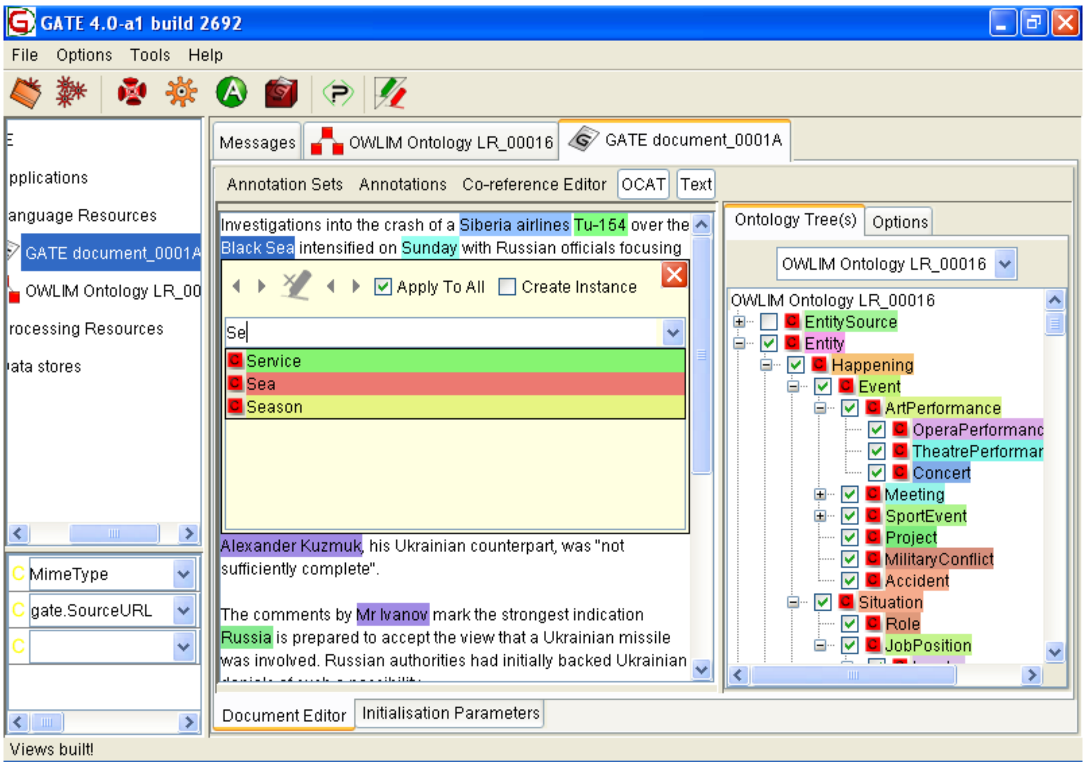
OAT also allows users to assign property values as annotation features to the existing class and instance annotations. In the case of class annotation, all annotation properties from the ontology are displayed in the table. In the case of instance annotations, all properties from the ontology applicable to the selected instance are shown in the table. The table also shows existing features of the selected annotation. User can then add, delete or edit any value(s) of the selected feature. In the case of a property, user is allowed to provide an arbitrary number of values. User can, by clicking on the editList button, add, remove or edit any value to the property. In case of object properties, users are only allowed to select values from a pre-selected list of values (i.e. instances which satisify the selected property's range constraints).

10.9.3 Adding New Annotations

New annotations can be added in two ways: using a dialogue (see Figure 10.4) or by selecting the text and clicking on the desired class or instance in the ontology tree.

When adding a new annotation using the dialogue, select a text and after a very short while, if the mouse is not moved, a dialogue will appear (see Figure 10.4). Start typing the name of the desired class or instance, until you see it listed in the combo-box, then select it with the mouse. This operation is the same, as in changing the class/instance of an existing annotation. One has the option of applying this choice to the current selection only or to all mentions of the selected string in the current document (Apply to All check box).

User can also create an instance from the selected text. If user checks the “create instance” checkbox prior to selecting the class, the selected text is annotated with the selected class and a new instance of the selected class (with the name equivalent to the selected text) is created (provided there isn't any existing instance available in the ontology with that name).


Investigations into the crash of a Siberia airlines Tu-154 over the Black Sea intensified on Sunday with Russian officials focusing

Alexander Kuzmuk, his Ukrainian counterpart, was "not sufficiently complete".

The comments by Mr Ivanov mark the strongest indication Russia is prepared to accept the view that a Ukrainian missile was involved. Russian authorities had initially backed Ukrainian

Figure 10.4: Add New Annotation

10.9.4 Options

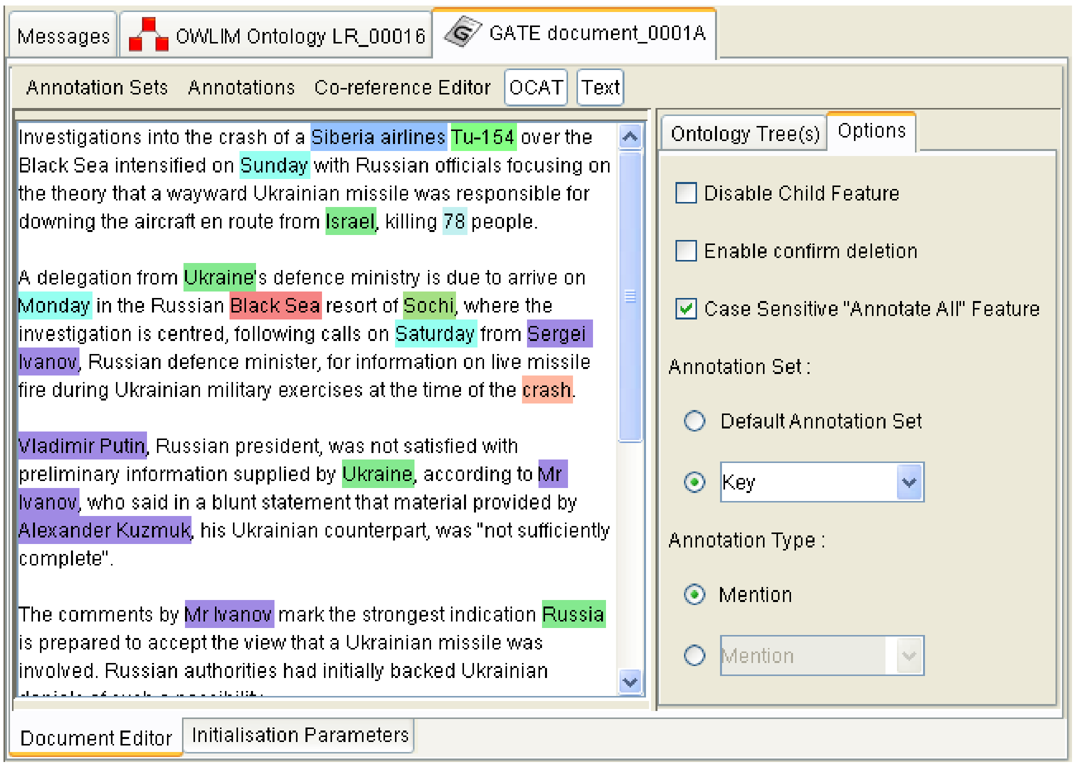
There are several options that control the OAT behaviour (see Figure 10.5):

- **Disable child feature**: By default, when a class is selected, all of its sub-classes are also automatically selected and their mentions are highlighted in the text. This option disables that behaviour, so only mentions of the selected class are highlighted.

- **Delete confirmation**: By default, OAT deletes ontological information without asking for confirmation, when the delete button is pressed. However, if this leads to too many mistakes, it is possible to enable delete confirmations from this option.

- **Disable Case-Sensitive Feature**: When user decides to annotate all occurrences of the selected text ("apply to all" option) in the document and if the "disable case-sensitive feature" is selected, the tool, when searching for the identical strings in the document text, ignores the case-sensitivity.

- **Setting up a filter to disable resources from the OAT GUI**: When user wants to annotate the text of a document with certain classes/instances of the ontology, s/he may disable the resources which s/he is not going to use. This option allows users to select a file which contains class or instance names, one per line. These names are case sensitive. After selecting a file, when user turns on the "filter" check box, the resources



Investigations into the crash of a Siberia airlines Tu-154 over the Black Sea intensified on Sunday with Russian officials focusing on the theory that a wayward Ukrainian missile was responsible for downing the aircraft en route from Israel, killing 78 people.

A delegation from Ukraine's defence ministry is due to arrive on Monday in the Russian Black Sea resort of Sochi, where the investigation is centred, following calls on Saturday from Sergei Ivanov, Russian defence minister, for information on live missile fire during Ukrainian military exercises at the time of the crash.

Vladimir Putin, Russian president, was not satisfied with preliminary information supplied by Ukraine, according to Mr Ivanov, who said in a blunt statement that material provided by Alexander Kuzmuk, his Ukrainian counterpart, was "not sufficiently complete".

The comments by Mr Ivanov mark the strongest indication Russia is prepared to accept the view that a Ukrainian missile was involved. Russian authorities had initially backed Ukrainian

☐ Disable Child Feature

☐ Enable confirm deletion

☑ Case Sensitive "Annotate All" Feature

Annotation Set :

○ Default Annotation Set

● Key

Annotation Type :

● Mention

○ Mention

Figure 10.5: Tool Options

specified in the filter file are disabled and removed from the annotation editor window. User can also add new resources to this list or remove some or all from the list by right clicking on the respective resource and by selecting the relevant option. Once modified, the "save" button allows users to export this list to a file.

- **Annotation Set**: GATE stores information in annotation sets and OAT allows you to select which set to use as input and output.
- **Annotation Type**: By default, this is annotation of type Mention, but that can be changed to any other name. This option is required because OAT uses Gate annotations to store and read the ontological data. However, to do that, it needs a type (i.e. name) so ontology-based annotations can be distinguished easily from other annotations (e.g. tokens, gazetteer lookups).

Chapter 11

Machine Learning API

This chapter describes a new machine learning layer in GATE. The current implementation are mainly for the three types of NLP learning, namely chunk recognition (e.g. named entity recognition), text classification and relation extraction, which cover many NLP learning problems. The implementation for chunk recognition is based on our work using the support vector machines (SVM) for information extraction [Li *et al.* 05a]. The text classification is based on our works on opinionated sentence classification and patent document classification (see [Li *et al.* 07b] and [Li *et al.* 07c], respectively). The relation extraction is based on our work on named entity relation extraction [Wang *et al.* 06].

The machine learning API, given a set of documents, can also produce several feature files containing linguistic features and feature vectors, respectively, and labels if there are any in the documents. It can also produce the so-called document-term matrix and n-gram based language model. Those features files are in text format and can be used outside of the GATE. Hence user can use those features off-line for her/his own purpose, e.g. evaluating the new learning algorithms.

The learning API also provides the facilities for active learning based on the learning algorithm Support Vector Machines (SVM), mainly ranking the unlabelled documents according to the confidence scores of the current SVM models for those documents.

The primary learning algorithm implemented is SVM, which has achieved state of the art performances for many NLP learning tasks. The training of the SVM uses the Java version of the SVM package LibSVM [CC001]. The application of the SVM is implemented by ourselves. Moreover, the ML implementation provides an interface to the open-source machine learning package Weka [Witten & Frank 99]. Therefore it can use the machine learning algorithms implemented in Weka. The three widely used learning algorithms, naive Bayes method, KNN and the decision tree algorithm C4.5 are available in the current implementation.

In order to use the machine learning (ML) API, user mainly has to do three things. First user has to annotate some documents with the labels that s/he wants the learning system to annotate in new documents. Those label annotations should be the GATE annotations. Secondly, user may need to pre-process the documents to obtain the linguistic features for the learning. Again these features should be in the form of the GATE annotations. The GATE's plug-in ANNIE would be very helpful for producing the linguistic features. Other plug-ins such as NP Chunker and parser may also be very helpful. Finally user has to create a configuration file for setting the ML API, e.g. selecting the learning algorithm and defining the linguistic features used by learning. Note that user may not need to create the configuration file from scratch. Instead user can copy one of the three example files presented below and make modifications on it for one particular problem.

The rest of the chapter is organised as follows. Section 11.1 explains the ML in general and the specifications in GATE. Section 11.2 describes all the configuration settings of the ML API one by one, in particular all the elements in the configuration file for setting the ML API (e.g. the learning algorithm to be used and the options for the learning) and defining the NLP features for the user's particular problem. Section 11.3 presents three exemplary settings respectively for the three types of NLP learning problems to illustrate the usage of this ML plug-in. Section 11.4 lists the steps of using the ML API. Finally Section 11.5 explains the outputs of the ML API for the four usage modes, namely the training, application, evaluation and producing feature files only, respectively, and in particular the format of the feature files and label list file produced by the ML API.

11.1 ML Generalities

There are two main types of ML, supervised learning and unsupervised learning. The supervised learning is more effective and much more widely used in the NLP. Classification is a particular example of supervised learning in which the set of training examples is split into multiple subsets (classes) and the algorithm attempts to distribute the new examples into the existing classes. This is the type of ML that is used in GATE and all further references to ML actually refer to classification.

An ML algorithm "learns" about a phenomenon by looking at a set of occurrences of that phenomenon that are used as examples. Based on these, a model is built that can be used to predict characteristics of future (and unforeseen) examples of the phenomenon.

An ML implementation has two modes of functioning: training and application. The training phase consists of building a model (e.g. statistical model, a decision tree, a rule set, etc.) from a dataset of already classified instances. During application, the model built during training is used to classify new instances.

The ML API in GATE is designed particularly for NLP learning. It can be used for the three types of NLP learning, text classification, chunk recognition, and relation extraction, which cover many NLP learning tasks.

- **Text classification** classifies text into pre-defined categories. The text can be at different level, such as document, sentence, or token. Some typical examples of text classification are document classification, opinionated sentence recognition, token's POS tagging, and word sense disambiguation.
- **Chunk recognition** often consists of two steps. First it identifies the interested chunks from text. It then assigns some label(s) to the extracted chunks. However it may just need the first step only in some cases. The examples of chunk recognition includes named entity recognition (and generally information extraction), NP chunking, and Chinese word segmentation.
- **Relation extraction** determines if or not a pair of terms from text has some type(s) of pre-defined relations. Two examples are named entity relation extraction and co-reference resolution.

From the ML's point of view, typically the three types of NLP learning use different linguistic features and feature representations. For example, it has been recognised that for text classification the so-called $tf - idf$ representation of the n-grams in the text is very effective by using some learning algorithms such as SVM. For chunk recognition identifying the first token and the end token of chunk by using the linguistic features of the token itself and the surrounding tokens is effective and efficient. Relation extraction needs considering both the linguistic features from each of the two terms involved in the relation and those features combined from the two terms.

The ML API implements the suitable feature representations for the three types of NLP learning. It also implements the Java codes or wrappers for the widely used ML algorithms including SVM, KNN, Naive Bayes, and the decision tree algorithm C4.5. In addition, given some documents, it can produce the NLP features and the feature vectors, which will be stored in the files. Hence, the users can use those features for evaluating the learning algorithms of their own for some NLP learning task or for any other further processing.

The rest of this section explains some basic definitions in ML and their specification in this GATE plug-in.

11.1.1 Some definitions

- **instance**: an example of the studied phenomenon. An ML algorithm learns a model from a set of known instances, called a (training) dataset. It can then apply the learned model to another (application) dataset.
- **attribute**: a characteristic of the instances. Each instance is defined by the values of its attributes. The set of possible attributes is well defined and is the same for all instances in the training and application datasets.
- **class**: an attribute for which the values are available in the training dataset for learning and need to be found in the application dataset through the ML mechanism.

11.1.2 GATE-specific interpretation of the above definitions

- **instance**: an annotation. In order to use ML in GATE the users will need to choose the type of annotations used as instances. Token annotations are a good candidate for many NLP learning such as information extraction and POS tagging, but any type of annotation could be used (e.g. things that were found by a previously run JAPE grammar, such as the sentence annotations and document annotations for sentence and document classifications, respectively).
- **attribute**: an attribute is the value of a named feature of a particular annotation type, which can either (partially) cover the instance annotation considered or another instance annotation which is related to the instance annotation considered. The value of the attribute can refer to the current instance or to an instance either situated at a specified location relative to the current instance or having special relation with the current instance.
- **class**: any attribute referring to the current instance can be marked as class attribute.

11.2 The Batch Learning PR in GATE

Access to ML implementations is provided in GATE by the "Batch Learning PR" that handles the four usage modes, the training and application of ML model, the evaluation of learning on GATE documents[1], and producing the feature files only. This PR is a Language Analyser so it can be used in all default types of GATE controllers.

In order to allow for more flexibility, all the configuration parameters for the PR are set through one external XML file, except the three learning modes which are selected through the normal PR parameterisation. The XML file contains both the configuration parameters of the ML API itself and the linguistic data (namely the definitions of the instance and attributes) used by the ML API. The XML file would be required to be specified when loading ML API plug-in into GATE.

The parent directory of the XML configuration file is called as working directory. A subdirectory of the working directory, named as "savedFiles", will be created (if it does not exist when loading the ML API). All the files produced by the ML API, including the NLP features files, label list file, feature vector file and learned model file, will be stored in that subdirectory. The log file recording the information of one learning session is also in this directory.

In the following we first describe a few settings which are the parameters of the ML API plug-in. Then we explain those settings specified in the configuration file.

[1]For the evaluation mode the system divides the corpus into two parts, uses one part to learn a model, applies the model to another part, and finally output the results on the testing part (see Sub-section 11.2.2 for the evaluation methods and Section 11.5 for an explanation of the evaluation results

11.2.1 The settings not specified in the configuration file

For the sake of convenience, a few settings are not specified in the configuration file. Instead the user should specify them as loading or run-time parameters of the ML API plug-in, as in many other PRs.

- **URL (or path and name) of the configuration file**. The user is required to give the URL of the configuration file when loading the ML API into GATE GUI. The configuration file should be in the XML format with the extension name *.xml*. It contains most of learning settings and will be explained in detail in the next sub-section.
- **Corpus**. It is a run-time parameter, meaning that the user should specify it before running the ML API in one session. Corpus contains the documents as the learning or application data. The documents should include all the annotations specified in the configuration file, except the annotation as the class attribute. The annotations for class attribute should be available in the documents used for training or evaluation, and may not be presented in the documents which the ML API is applying the learned model to.
- **inputASName** is the annotation set containing the annotations for the linguistic features used and the class labels.
- **outputASName** is the annotation set in which the result annotations of applying the models will be put. Note that it should be set as the same as the *inputASName* when doing the evaluation (namely setting the *learningMode* as "EVALUATION").
- **learningMode**. It is a run-time Enum type parameter. It can be set as one of the following values, "TRAINING", "APPLICATION", "EVALUATION", "ProduceFeatureFilesOnly", "MITRAINING", and "VIEWPRIMALFORMMODELS". The first four values correspond to the common learning modes of the ML API. Other learning modes are for the specific purpose only. The default learning mode is "TRAINING".
  - In *TRAINING* mode, the ML API learns from the data provided and saves the models into a file called "learnedModels.save" under the sub-directory "savedFiles" of the working directory.
  - If user wants to apply the learned model to the data, s/he should select the *APPLICATION* mode. In the application mode, the ML API reads the learned model from the file "learnedModels.save" in the subdirectory "savedFiles" and then applies the model to the data.
  - In *EVALUATION* mode, the ML API will do the k-fold or hold-out test set evaluation on the corpus provided (the method of the evaluation is specified in the configuration file, see below), and output the evaluation results to the Messages Window of the GATE GUI and into the log file. Note that when using the "EVALUATION" mode, please make sure that the *outputASName* is set with the same annotation set as the *inputASName*.

- If user only wants ML API to produce the NLP feature data and feature vectors data but does not want the training or the application of the learned model, select the *ProduceFeatureFilesOnly* mode. The feature files that the ML API produces will be explained in detail in Sub-section 11.5.4. Otherwise (namely user wants to use the whole procedure of learning), select one of other learning modes.
- *MITRAINING* mode is a specific training mode. In this mode, the training data obtained in the session are appended to the end of the feature file. In contrast, in *training* mode the training data obtained in the session overrides the previous data (if there any) in the feature file. Consequently, the *MITRAINING* mode uses both the training data obtained in this session and the data existed in the feature file before starting the session for training. Hence, the *TRAINING* mode is for batch learning, while the *MITRAINING* mode can be used for on-line (or adaptive, or mixed-initiative) learning. There is one parameter for the *MITRAINING* mode specifying the minimal number of newly added documents before starting the learning procedure to update the learned model. The parameter can be defined in the configuration file.
- *VIEWPRIMALFORMMODELS* mode is used for displaying the most salient NLP features in the learned modes. In the current implementation the mode is only valid for linear SVM model in which the most salient NLP features correspond to the biggest (absolute values of) weights in the weight vector. In the configuration file one can specify two parameters to determine the number of displayed NLP features for positive and negative weights, respectively. Note that if e.g. the number for negative weight is set as 0, then no NLP feature is displayed for negative weight.
- *RankingDocsForAL* applies the current learned SVM models (namely in the sub-directory "savedFiles") on the feature vectors storing in the file *fvsDataSelecting.save* in the sub-directory "savedFiles" and ranks the documents stored in the data file according to the margins of the examples in one document to the SVM models. The ranked list of documents will be put into the file *ALRankedDocs.save*.

Please note that, if the ML API is added into a GATE application as corpus pipeline, it does not process the documents in the corpus until the last document goes through the corpus pipeline, except in the *APPLICATION* mode[2]. When the last document in the corpus is going through the pipeline, the ML PR processes all the documents in the corpus. This kind of batch processing is necessary for batch learning algorithms such as the SVM. It also makes the application of learned model much more faster than the normal behaviour of a PR in corpus pipeline, namely every PR in the pipeline process the same document in

[2]In the *APPLICATION* mode the number of documents processed by one time of application can be specified by one parameter in the configuration file and the default value of the parameter is 1. If the parameter is set as 1, then the ML API applies the learned model to the documents in the corpus one by one. In other word, in this case the ML API has the normal behaviour of a GATE PR and can be followed by other PRs in a pipeline. On the other hand, if assigning the parameter a larger value, usually the application would be faster but may probably consume more computer memory.

the corpus before processing the next document. One important consequence of the batch implementation is that one cannot put some post-processing PRs directly after the ML PR in the same corpus pipeline. Instead, s/he should use the ML PR as the last PR in the corpus pipeline, and put all the post-processing PRs into another GATE application which post-processes the documents processed by the ML. However, this kind of limitation can be alleviated for the *APPLICATION* mode (see Footnote above for more details).

11.2.2 All the settings in the XML configuration file

The root element of the XML configuration file needs to be called "ML-CONFIG" and it must contain two basic elements, "DATASET" and "ENGINE", and optionally other optional settings. In the following we first describe the optional settings, then the "ENGINE" element, and finally the "DATASET" element. In next section some examples of the XML configuration file are given for illustration. Please also refer to the configuration files in the test directory (i.e. plugs/learning/test/ under the main gate directory) for more examples.

The optional settings in the configuration file

The ML API provides a variety of optional settings, which facilitates different tasks. Every optional setting has a default value — if one optional setting is not specified in the configuration file, the ML API will adopt its default value. Each of the following optional settings can be set as an element in the XML configuration file.

- **Surround mode**. Set its value as "true" if user wants ML API to learn the start token and the end token of chunk, which often results in better performance than learning every token of chunk for chunk learning such as named entity recognition. For the classification problem and relation extraction, set its value as "false". The corresponding element in the configuration file is as
  <SURROUND VALUE="`X`"/>
  where the variable `X` has two possible values: "true" or "false". The default value is "false".

- **FILTERTING**. In some applications user may want to filter out some training examples from the original training data before running learning algorithm. The filtering option allows user to remove some examples without class label (called as negative examples) from training set. Those negative examples were selected according to their distances to the SVM classification hyper-plane which is learned from the original training data for separating the positive examples from the negative ones. If the item *dis* is set as "near", the ML API selects and removes the negative examples which are closest to the SVM hyper-plane. If it is set as "far", those negative examples that are furthest from the SVM hyper-plane are removed. The value of the item *ratio* determines how many (namely the ratio) of negative examples will be filtered out. The element in the

configuration file is as
< FILTERING ratio="X" dis="Y"/>
where X represents a number between 0 and 1 and Y can be set as "near" or "far". If the filtering element is not presented in the configuration file, or the value of *ratio* is set as 0.0, the ML API would not do the filtering. The default value of *ratio* is 0.0. The default value of *dis* is "far".

- **Evaluation setting**. As said above, if the learning mode parameter *learningMode* is set as "EVALUATION", ML API will do evaluation on the corpus. Basically it will split the documents in the corpus into two parts, training dataset and testing dataset, learn a model from the training dataset, apply the model to the testing dataset, and finally compare the annotations assigned by the model with the true annotations of the testing data and output some evaluation results such as the overall F-measures. The evaluation setting element specifies the method of splitting the corpus. The item *method* determines which method to use for evaluation. Currently two commonly used methods are implemented, namely the *k-fold cross-validation* and the *hold-out test*[3]. The value of the item *runs* specifies the number "k" for the k-fold cross-validation or the number of runs for hold-out test. The value of the item *ratio* specifies the ratio of the data used for training in the hold-out test method. The element in the configuration file is as
<EVALUATION method="X" runs="Y" ratio="Z"/>
where the variable X has two possible values "kfold" and "holdout", Y is a positive integer, and Z is a float number between 0 and 1. The default value of *method* is "holdout". The default value of *runs* is "1". The default value of *ratio* is "0.66".

- **multiClassification2Binary**. In many cases an NLP learning problem can be transformed into a multi-class classification problem. On the other hand, some learning algorithm such as the SVM is often used as a binary classifier. ML API implements two common methods for converting a multi-class problem into several binary class problems, namely *one against others* and *one against another one*[4]. User can select one of the two methods by specifying the value of the item *method* of the element. The element is as
<multiClassification2Binary method="X" thread-pool-size="N"/>
where the variable X has two values, "one-vs-others" and "one-vs-another'. The default method is one-vs-others method. If the configuration file does not have the element

[3]For the k-fold cross-validation the system first gets a ranked list of documents in alphabetic order of the documents' names, then segments the document list into k partitions of equal size, and finally uses each of the partitions as testing set and other documents as training set. For hold-out test, the system randomly selected some documents as testing data and uses all other documents as training data.

[4]Probably the two methods have different names in some publications. But the methods are the same. Suppose we have a multi-class classification problem with n classes. For the *one against others* method, one binary classification problem is derived for each of the n classes, which has the examples belonging to the class considered as positive examples and all other examples in training set as negative examples. In contrast, for the *one against another one* method, one binary classification problem is derived for each pair $(c1, c2)$ of the n classes, in which the training examples belonging to the class $c1$ are the positive examples and those belonging to another class $c2$ are the negative examples.

or the item *method* is missed, then ML API will use the one-vs-others method in the evaluation mode. Since the derived binary classifiers are independent it is possible to learn several of them in parallel. The "thread-pool-size" attribute gives the number of threads that will be used to learn and apply the binary classifiers - if omitted, a single thread will be used to process all the classifiers in sequence.

- Parameter **thresholdProbabilityBoundary** sets the threshold of the probability of start (or end) token for chunk learning. It is used in the post-processing of the learning results. Only those boundary tokens which confidence level is above the threshold are selected as candidates of the entities. The element in configuration file is as
  <PARAMETER name="thresholdProbabilityBoundary" value="X"/>
  The value `X` is between 0 and 1. The default value is 0.4.

- Parameter **thresholdProbabilityEntity** set the threshold of the probability of a chunk (which is the multiplication of the probabilities of the start token and end token of the chunk) for chunk learning. Only those entities which confidence level is above the threshold are selected as candidates of the entities for further post-processing. The element in configuration file is as
  <PARAMETER name="thresholdProbabilityEntity" value="X"/>
  The value `X` is between 0 and 1. The default value is 0.2.

- The threshold parameter **thresholdProbabilityClassification** is for the classification (e.g. text classification and relation extraction tasks. In contrast, the above two probabilities are for the chunking recognition task.) The corresponding element in configuration file is as
  <PARAMETER name="thresholdProbabilityClassification" value="X"/>
  The value `X` is between 0 and 1. The default value is 0.5.

- **IS-LABEL-UPDATABLE** is a Boolean parameter. If its value is set as "true", the label list is updated from the labels in the training data. Otherwise, a pre-defined label list will be used and cannot be updated from the training data. The configuration element is as
  <IS-LABEL-UPDATABLE value="X"/>
  The value `X` is "true" or "false". The default value is "true".

- **IS-NLPFEATURELIST-UPDATABLE** is a Boolean parameter. If its value is set as "true", the NLP feature list is updated from the features in the training or application data. Otherwise, a pre-defined NLP feature list will be used and cannot be updated. The configuration element is as
  <IS-NLPFEATURELIST-UPDATABLE value="X"/>
  The value `X` is "true" or "false". The default value is "true".

- The parameter **VERBOSITY** specifies the maximal verbosity level of the output of the system, both to the Message Window of the GATE GUI and into the log file. Currently there are three verbosity levels. Level 0 only allows the output of warning messages. Level 1 outputs some important setting information and the results for

evaluation mode. Level 2 is used for debug purpose. The element in configuration file is as
<VERBOSITY level="`X`*"/>*
The value `X` can be set as 0, 1 or 2. The default value is 1.

- Option **MI-TRAINING-INTERVAL** specifies the minimal number of newly added documents needed for triggering the learning, which is used in the *MITRAINING* mode of the ML API. The number is specified by the value of the feature "num" as showed in the following.
<MI-TRAINING-INTERVAL num="`X`*"/>*
The default value of `X` is 1.

- Option **BATCH-APP-INTERVAL** is used in the *APPLICATION* mode and specifies the number of documents in the corpus processed by batch application. Please refer to Section 11.2.1 for detailed explanation about the option. The corresponding element in the configuration file is as
<BATCH-APP-INTERVAL num="`X`*"/>*
The default value of `X` is 1.

- Option **DISPLAY-NLPFEATURES-LINEARSVM** specifies two numbers of the NLP features respectively for the positive and negative weights of a linear model for displaying. It is used in the *VIEWPRIMALFORMMODELS* mode. For more detailed about the mode see Section 11.2.1. It has the following form in configuration file
<DISPLAY-NLPFEATURES-LINEARSVM numP="`X`*" numN="*`Y`*"/>*
where `X` and `Y` represent the numbers for the positive and negative weights, respectively. The default values of `X` and `Y` are 10 and 0, respectively.

- Optin **ACTIVELEARNING** specifies the settings for active learning. It has the following form
<ACTIVELEARNING numExamplesPerDoc="`X`*"/>*
where `X` represents the number of examples in one document used for obtaining the confidence score of the document (by averaging) with respect to the learned model. The default value of *numExamplesPerDoc* is 3.

The ENGINE element

The ENGINE element specifies which particular ML algorithm will be used and also allows the setting of options for that algorithm.

Note that for the SVM learning, user can choose one of the two learning engines. We will discuss the two SVM learning engines below. Another note is that the current implementation only allows the SVM learning engine use linear and polynomial kernels but not other types of kernels, despite the fact that the original SVM packages implemented other types of kernel. The main reason for that is that linear and polynomial kernels are popular in natural language learning and other types of kernel has been used in rare case. However, if you want to experiment with other types of kernel, you can do it by first running the learning plugin in GATE to produce the training and testing data, then convert those data into the specific format that a SVM package requires for input data, and finally run the SVM package on the data.

The configuration files in the test directory (i.e. plugs/learning/test/ under the main gate directory) contain the examples for setting the learning engine.

The ENGINE element in the configuration file is as
<ENGINE nickname="`X`**" implementationName="**`Y`**" options="**`Z`**"/>**
It has three items:

- **nickname** can be the normal name of the learning algorithm or whatever user wants it to be.
- **implementationName** refers to the implementation of the particular learning algorithm that user wants to use. Its value should be strictly the same as the one defined in ML API for the particular algorithm. Currently it can be specified as one of the following values.
  - **SVMLibSvmJava**, the binary classification SVM algorithm implemented in the Java version of the SVM package *LibSVM*.
  - **SVMExec**, the binary classification SVM algorithm which could be implemented in the C/C++ or other language and will be run as a separated process outside the GATE. Since it run the learning process outside GATE, it can potentially cope with large training data better that running the learning process inside GATE. Currently it can use the SVM^{light} SVM package[5]. See the xml file in the GATE distribution (at gate/plugins/learning/test/chunklearning/engines-svm-svmlight.xml) for how to specify the learning engine for it.
    Note that the learning engine *SVMExec* and another one *SVMLibSvmJava* use exactly the same learning algorithm SVM but different implementations of SVM. They should have the same results in theory but may get slightly different results in practice because of different implementations. SVMExec can deal with large training data and SVMLibSvmJava may have the memory problem for large data[6]. On the other hand, it's more convenient to use SVMLibSvmJava than SVMExec, because the latter require some extra work from the user, namely downloading the package SVM^{light} into the user's computer, compiling the package, and the extra settings in the configuration file (see below). SVMLibSvmJava tends to be faster than SVMExec for small or moderate size of training data. Therefore, if the training data is not too big to cause memory problem or makes the learning very slow, it's convenient to use SVMLibSvmJava. If the training data is very big, then SVMExec may probably be the only feasible option.

[5]The SVM package SVM^{light} can be downloaded from http://svmlight.joachims.org/.

[6]Because SVMLibSvmJava runs the SVM learning inside the GATE, while SVMExec runs the SVM learning outside of the GATE as a separated process.

- **PAUM**, the Perceptron with uneven margins, a simple and fast classification learning algorithms (for details about learning algorithm PAUM itself, see [Li *et al.* 02]).
- **PAUMExec**, the binary classification PAUM algorithm which could be implemented in the C/C++ or other language and will be run as a separated process outside the GATE. The relation between the PAUM and PAUMExec is similar to that of SVMLibSvmJava and SVMExec. You may download and use one implementation in c from the website *http://www.dcs.shef.ac.uk/∼yaoyong/paum/paum-learning.zip*. See the xml file in the GATE distribution (at gate/plugins/learning/test/chunklearning/engines-paum-exec.xml) for how to specify the learning engine for it.
- **NaiveBayesWeka**, the Naive Bayes learning algorithm implemented in Weka.
- **KNNWeka**, the K nearest neighbour (KNN) algorithm implemented in Weka.
- **C4.5Weka**, the decision tree algorithm C4.5 implemented in Weka.

- **Options**: the value of this item, which is dependent on the particular learning algorithm, will be passed verbatim to the ML engine used. If one option is missed in the specification or some options items are not presented in the configuration file at all, the default settings of the learning algorithm will be used.
  - The options for the *SVMLibSvmJava* are similar as that for the LibSVM but has some specifications as described in follows. Moreover *SVMLibSvmJava* implements the uneven margins SVM algorithms (see [Li & Shawe-Taylor 03]). Hence it also has the uneven margins parameter as one option. The specifications of the LibSVM options for the *SVMLibSvmJava* and some other important settings are as
    - **-s svm_type**, type of the SVM (default value is 0). Since we only implement the binary classification of the SVM algorithm, always set it as 0 (or do not specify this item and use the default value).
    - **-t kernel_type**, the *kernel_type* should be 0 for linear kernel, and 1 for polynomial kernel. Default value is 0. Note that the current implementation does not support other kernel types such as radial kernel and sigmoid function kernel.
    - **-d degree**, the degree in polynomial kernel, e.g. 2 for quadratic kernel. Default value is 3.
    - **-c cost**, the cost parameter C in the SVM. Default value is 1.
    - **-m cachesize**, set the cache memory size in MB (default 100).
    - **-tau value**, setting the value of uneven margins parameter of the SVM. $\tau = 1$ corresponds to the standard SVM. If the training data has just a small number of positive examples and a big number of negative examples, which may occur in many NLP learning problems particularly when using a few documents for

training, set the parameter τ as a value less than 1 (e.g. $\tau = 0.4$) often results in better F-measure than the standard SVM (see [Li & Shawe-Taylor 03]).

- the options for the SVMExec is similar to those for the training program of SVM^{light} for setting type of kernel, the paramters in the kernel function, the cost parameter, the memory used, etc. It also has the parameter tau to set the uneven margins parameter, as explained above. Note that the last two terms in the value of the parameter options are the file names referring to the two data files in your computers that the SVM learning program can write and read[7]. Another parameter for the SVMExec is **executableTraining**, which specifies the svm learning program svm_learn.exe in the SVM^{light}[8].
- the PAUM algorithm has three options, "-p" for the positive margin, "-n" fo the negative margin, and "-optB " for the modification of the bias term. For example, options=" -p 50 -n 5 -optB 0.3 " means $\tau_+ = 50$, $\tau_- = 5$ and $b = b + 0.3$ in the PAUM algorithm.
- The KNN algorithm has one option, the number of neighbours used. It is set via "**-k** `X`". The default value is 1.
- There is no option currently for Naive Bayes and C4.5 algorithms.

The DATASET element

The DATASET element defines the type of annotation to be used as instance and the set of attributes that characterise all the instances.

An "INSTANCE-TYPE" sub-element is used to select the annotation type to be used for instances, and the attributes are defined by a sequence of attribute elements.

For example, if an "INSTANCE-TYPE" has a "Token" as its value, there will be one instance in the document per "Token". This also means that the **positions** (see below) are defined in relation to Tokens. The "INSTANCE-TYPE" can be seen as the basic unit to be taken into account for machine learning.

Different NLP learning tasks may have different instance types and use different kinds of attribute elements. Chunking recognition often uses the token as instance type and the linguistic features of "Token" and other annotations as features. Text classification's instance type is the text unit for classification, e.g. the whole document, or sentence, or token. If

[7]An example of the options for SVMExec is "-c 0.7 -t 0 -m 100 -v 0 -tau 0.6 /yaoyong/software/svm-light/data_svm.dat /yaoyong/software/svm-light/model_svm.dat", meaning that the svm learning uses linear kernel, the uneven margins parameter is set as 0.6, and two data files /yaoyong/software/svm-light/data_svm.dat and /yaoyong/software/svm-light/model_svm.dat for writing and reading data. Note that both the data files specified here are tempory files which are used only by the svm-light training program, can be in anywhere in your computer, and are independent of the data files produced by the GATE learning plugin.

[8]For example, executableTraining="/yaoyong/software/svm-light/svm_learn.exe " specifies one particular svm_learn.exe obtained from the package SVM^{light}.

classifying a sequence of tokens, the n-grams representation of the tokens is often a good feature representation for many statistical learning algorithms. For relation extraction, the instance type is a pair of terms for the relation, and the features come from not only the linguistic features of each of the two terms but also those related to both terms.

A DATASET element should define an instance type sub-element, and an “ATTRIBUTE” sub-element or an “ATRRIBUTE_REL” sub-element as “Class”, and some linguistic feature related sub-elements. All the annotation types involved in the data set definition should be in the same Annotation Set. Each sub-element defining the linguistic features or class label is associated with one or more annotation types. Each annotation type used should specify one of its annotation feature — the values of that annotation feature are the linguistic features used as input to learning algorithm or the class labels for learning[9]. Note that if blank spaces are contained in the values of the annotation features, they will be replaced by the character “_” in each occurrence. So it is advisable that the values of the annotation features used, in particular for the class label, do not contain any blank space.

In the following we explain all the sub-elements one by one. Please also refer to the examples of configuration files presented in next section for the examples of the dataset definition. Note that each sub-element should have a unique name (which is different from other sub-element's names) if it requires name, unless we state explicitly the other case.

- The **instance** type sub-element is defined as *<INSTANCE-TYPE>`X`</INSTANCE-TYPE>* where `X` is the annotation type used as instance unit for learning. For relation extraction, user has to specify the two arguments of the relation too, as
  <INSTANCE-ARG1>`A`</INSTANCE-ARG1>
  <INSTANCE-ARG2>`B`</INSTANCE-ARG2>
  which are the two features of the Instance Type. The values of `A` and `B` should be some identification of the first and second terms of the relation, respectively.
- An **ATTRIBUTE** element has the following sub-elements:
  - **NAME**, the name of the attribute. Its value should not end with “gram”(see below for the name of NGRAM feature).
  - **SEMTYPE**, type of the attribute value, it can be “NOMINAL” or “NUMERIC”. Currently only *nominal* is implemented.
  - **TYPE**, the annotation type used to extract the attribute.
  - **FEATURE**, the value of the attribute will be the value of the named feature on the annotation of specified type.

[9]For an “‘ATTRIBUTE” sub-element, if you do not specify the annotation feature, this sub-element will be ignored by the ML API. Therefore, if one annotation type you want to use does not have any annotation feature, you should add one annotation feature to it and assign one and the same value to the feature for all annotations of that type.

  - **POSITION**: the position of an instance annotation relative to the current instance annotation. The instance annotation is used for extracting the feature defined in this element[10]. The default value of the parameter is 0.
  - **<CLASS/>**: an empty element used to mark the class attribute. There can only be one attribute marked as class in a dataset definition.

  Note that, in the current implementation, the component value in the feature vector for one *attribute* feature is 1 if the attribute's position p is 0. Otherwise its value is $1.0/|p|$.

- An **ATTRIBUTELIST** element is similar to ATTRIBUTE except that it has no POSITION sub-element but a RANGE element. This will be converted into several ATTRIBUTEs with position ranging from the value of the attribute "from" to the value of the attribute "to". Actually it defines a window containing several consecutive examples. We often call the window as context window. The *ATTRIBUTELIST* should be used when defining a context window for features, because not only it can avoid the duplication of ATTRIBUTE elements, but also for speeding up the processing (see the discussion for the element *WINDOWSIZE* below).

- An **WINDOWSIZE** element specifies the size of context window, which will override the context window size defined in every *ATTRIBUTELIST*. In detail, if the element is not presented in the configuration file, the window size defined in each element ATTRIBUTELIST will be used for that feature. Otherwise, the window size specified by this element will be used for each ATTRIBUTELIST if the latter contains one ATTRIBUTE at position 0 (otherwise the ATTRIBUTELIST will be ignored). This element can be used for speeding up the process of extracting the feature vectors from documents speficied by ATTRIBUTELIST. The element has two features specifying the length of left and right sides of context window, respectively. It has the following form
  / / / / / / <WINDOWSIZE windowSizeLeft="`X`*" windowSizeRight="*`Y`*"/>*
  where `X` and `Y` represent the the length of left and right sides of context window, respectively. For example, if `X` = 2 and `Y` = 1, then the context window will be from the position -2 to 1 (e.g. from the second token in the left through the current token to the first token in the right).

[10]Given the current instance annotation A, the position information is used to determine another instance annotation B. The linguistic feature for the instance A obtained by this element is the feature from the annotation C which is defined by this element and overlaps with the annotation B. The position 0 means that the instance annotation B is the A itself, -1 means that B is the first preceding annotation of A, and 1 means that B is the first annotation following A.
Please note that, if the annotation A contains more than one annotation C, e.g. C refers to token and A refers to named entity which contains more than one tokens, then there are more than one annotations C in the position 0 relative to the annotation A, in which case the current implementation just picks the first annotation of C. Therefore, if there is a possibility that the annotation A contains more than one annotations C, then it would be better of using e.g. a NGRAM type feature to characterise the feature than using an ATTRIBUTE feature.

- An **NGRAM** feature is used for characterising the annotation which consists of more than one instance annotations, e.g. a sentence consists of many tokens. It has the following sub-elements.
  - **NAME**, name of the n-gram. Its value should end with "gram" to denote the n-gram feature.
  - **NUMBER**, the "n" of the n-gram, with value 1 for uni-gram, and 2 for bi-gram, etc.
  - **CONSNUM**, number of the features for each of the n-grams. Given a value "k" of the *CONSUM*, the NGRAM element should have the "k" **CONS-`X`** sub-elements, where `X`= 1, ..., k. Each *CONS-`X`* element has one **TYPE** sub-element and one **FEATURE** sub-element, which define one feature of the term in the n-gram.
  - The **WEIGHT** sub-element specifies a weight for the n-gram feature. The n-gram part of the feature vector for one instance is normalised. If a user want to adjust the contributions of the n-gram to the whole feature vector, s/he can do it by setting this *WEIGHT* parameter[11]. Then every component of the n-gram part of the feature vector would be multiplied by the parameter. The default value of the parameter is 1.0.
- The **ValueTypeNgram** element specifies the type of value used in the n-gram. Currently it can take one of the three types, binary, tf, and tf-idf, which are explained in Section 11.5.4. The value is specified by the `X` in
  <ValueTypeNgram>`X`</ValueTypeNgram>
  `X` = 1 for binary, = 2 for tf, and = 3 for tf-idf. The default value is 3.
- **FEATURES-ARG1**[12] element defines the features related to the first argument of relation for relation learning. It should include one *ARG* sub-element referring to the GATE annotation of the argument (see below for detailed explanation). It may include other sub-elements, such as *ATTRIBUTE*, *ATTRIBUTELIST* and/or *NGRAM*, to define the linguistic features related to the argument.
- **FEATURES-ARG2** element defines the features related to the second argument of relation. Like the element *FEATURES-ARG1*, it should include one *ARG* sub-element. It may also include some other sub-elements for linguistic feature. Note that, the *ARG* sub-element in the *FEATURES-ARG2* should have a unique name which is different from the name for the *ARG* sub-element in the *FEATURES-ARG1*. However, the other

[11]For example, if a user does sentence classification and s/he uses two features, the uni-gram of tokens in one sentence and the length of sentence and thinks that the uni-gram feature is more important than the length of sentence, then s/he can set the weight sub-element of the n-gram element with a number bigger than 1.0 (like 10.0).

[12]If a feature of a relation instance is determined only by one argument of the relation but not the two arguments, then the feature could be defined in the element *FEATURES-ARG1* or the element *FEATURES-ARG2*. On the other hand, if a feature is related to both arguments, then it should be defined in the element *ATTRIBUTE_REL*.

sub-elements could have the same name as the corresponding ones in the *FEATURES-ARG1*, if they refer to the same annotation type and feature in the text.

- **ARG** element is used in both *FEATURES-ARG1* and *FEATURES-ARG2*. It specifies the annotation corresponding to one argument of relation. It has four sub-elements as the following
  - **NAME**, an unique name of the argument.
  - **SEMTYPE**: type of the arg value, it can be "NOMINAL" or "NUMERIC". Currently only *nominal* is implemented.
  - **TYPE**, the annotation type for the argument.
  - **FEATURE**, the value of the named feature on the annotation of specified type is the identification of the argument. Only if the value of the feature is same as the value of the feature specified in the sub-element *<INSTANCE-ARG1>A</INSTANCE-ARG1>* (or *<INSTANCE-ARG2>B</INSTANCE-ARG2>*), the argument is regarded as one argument of the relation instance considered.
- **ATTRIBUTE_REL** element is similar to the *ATTRIBUTE* element. But it does not have the *POSITION* sub-element, and has two other sub-elements *ARG1* and *ARG2*, relating to the two argument features of the (relation) instance type. In other words, if and only if the value `X` in the sub-element *<ARG1>X</ARG1>* is same as the value `A` in the first argument instance *<INSTANCE-ARG1>A</INSTANCE-ARG1>* and the value `Y` in the sub-element *<ARG2>Y</ARG2>* is same as the value `B` in the second argument instance *<INSTANCE-ARG2>B</INSTANCE-ARG2>*, the feature defined in this *ATTRIBUTE_REL* sub-element is assigned to the instance considered. Note that for relation learning a *ATTRIBUTE_REL* is used as the class attribute by having an empty element *<CLASS/>*.

11.3 Examples of configuration file for the three learning types

The following are three illustrated examples of configuration file for information extraction, sentence classification, and relation extraction, respectively. Note that the configuration file is in the XML format and should be stored in a file with name having "XML" extension.

The first example is for information extraction. The optional settings are in the first part. It first specifies the surround mode as "true", as it is a kind of chunk learning. Then it specifies the filtering settings. The *ratio*'s value is "0.1" and the *dis*'s value is "near", meaning that the 10% of negative examples which are closest to the learned SVM hyperplane will be removed in the filtering stage before the learning for the information extraction. The thresholds of probabilities for the boundary tokens and information entity are set as

"0.4" and "0.2",respectively. The threshold of probability for the classification is also set as "0.5", which, however, will not be used in the problem for chunk learning with the *surround* mode set as "true". The *multiClassification2Binary* is set as "one-vs-others", meaning that the ML API will convert the multi-class classification problem into some binary classification problems by using the *one against all others* approach. For the *EVALUATION* setting, the "2-fold" cross-validation will be used.

The second part is the sub-element *ENGINE* specifying the learning algorithm. The ML API will use the SVM learning implemented in the LibSVM. From the option settings it will use the linear kernel with the cost C as 0.7 and the cache memory as 100M. Additionally it will use the uneven margins SVM with the uneven margins parameter τ as 0.4.

The last part is the *DATASET* sub-element, defining the linguistic features used. It first specifies the "Token" annotation as instance type. The first *ATTRIBUTELIST* allows the token's string as the feature of an instance. The range from "-5" to "5" means that the strings of the current token instance as well as its five preceding tokens and its five following tokens would be used as features for the current token instance. The next two attribute lists define the features based on the token's capitalisation information and types, respectively. The *ATTRIBUTELIST* named as "Gaz" uses as features the values of the feature "majorType" of the annotation type "Lookup". Finally the *ATTRIBUTE* feature defines the attribute class, because it has a sub-element <**CLASS/**>. The values of the feature "class" of the annotation type "Mention" are the class labels.

```
<?xml version="1.0"?>
<ML-CONFIG>
  <SURROUND value="true"/>
  <FILTERING ratio="0.1" dis="near"/>
  <PARAMETER name="thresholdProbabilityEntity" value="0.2"/>
  <PARAMETER name="thresholdProbabilityBoundary" value="0.4"/>
  <PARAMETER name="thresholdProbabilityClassification" value="0.5"/>
  <multiClassification2Binary method="one-vs-others"/>
  <EVALUATION method="kfold" runs="2"/>
  <ENGINE nickname="SVM" implementationName="SVMLibSvmJava"
        options=" -c 0.7 -t 0 -m 100 -tau 0.4  "/>
  <DATASET>
    <INSTANCE-TYPE>Token</INSTANCE-TYPE>
    <ATTRIBUTELIST>
       <NAME>Form</NAME>
       <SEMTYPE>NOMINAL</SEMTYPE>
       <TYPE>Token</TYPE>
       <FEATURE>string</FEATURE>
       <RANGE from="-5" to="5"/>
    </ATTRIBUTELIST>
    <ATTRIBUTELIST>
```

```
       <NAME>Orthography</NAME>
       <SEMTYPE>NOMINAL</SEMTYPE>
       <TYPE>Token</TYPE>
       <FEATURE>orth</FEATURE>
       <RANGE from="-5" to="5"/>
    </ATTRIBUTELIST>
    <ATTRIBUTELIST>
       <NAME>Tokenkind</NAME>
       <SEMTYPE>NOMINAL</SEMTYPE>
       <TYPE>Token</TYPE>
       <FEATURE>kind</FEATURE>
       <RANGE from="-5" to="5"/>
     </ATTRIBUTELIST>
     <ATTRIBUTELIST>
       <NAME>Gaz</NAME>
       <SEMTYPE>NOMINAL</SEMTYPE>
       <TYPE>Lookup</TYPE>
       <FEATURE>majorType</FEATURE>
       <RANGE from="-5" to="5"/>
     </ATTRIBUTELIST>
     <ATTRIBUTE>
        <NAME>Class</NAME>
        <SEMTYPE>NOMINAL</SEMTYPE>
        <TYPE>Mention</TYPE>
        <FEATURE>class</FEATURE>
        <POSITION>0</POSITION>
        <CLASS/>
     </ATTRIBUTE>
   </DATASET>
</ML-CONFIG>
```

The following is a configuration file for sentence classification. It first specifies the surround mode as "false", because it is a text classification problem. The next two options allows the label list and the NLP feature list being updated from the training data. It also specifies the thresholds for entity and boundary of entity. Note that these two specifications will not be used in the text classification problems. However, the presences of them in the configuration file is not harmful to the ML API. The threshold of probability for classification is set as "0.5", which would be used in the application of the learned model. The evaluation will use the hold-out test method. It will randomly select 66% documents from the corpus for training and other 34% documents for testing. It will have two runs of evaluation and the results are averaged over the two runs. Note that it does not specify the method of converting a multi-class classification problem into several binary class problem, meaning that it will adopt the default one (namely one against all others) if it needs.

The configuration file specifies the KNN as learning algorithm. It also specifies the number of neighbours used as 5. Of course other learning algorithms can be used as well. For example, the *ENGINE* element in the previous example, which specifies the SVM as learning algorithm, can be put into this configuration file to replace the current one.

In the **DATASET** element, the annotation "Sentence" is used as instance type. Two kinds of linguistic features are defined. One is the *NGRAM*. Another is the *ATTRIBUTE*. The n-gram is based on the annotation "Token". It is uni-gram as its *NUMBER* element has the value 1. It is based on the two features "root" and "category" of the annotation "Token". In another words, two tokens will be considered as the same term of the uni-gram if and only if they have the same "root" feature and the same "category" feature. The weight of the ngram was set as 10.0, meaning its contribution is ten times of the contribution of another feature, the sentence length. By the *ATTRIBUTE* the feature "sent_size" of the annotation "Sentence" is used as another NLP feature. Finally the values of the feature "class" of the annotation "Sentence" are the class labels.

```
<?xml version="1.0"?>
<ML-CONFIG>
  <SURROUND value="false"/>
  <IS-LABEL-UPDATABLE value="true"/>
  <IS-NLPFEATURELIST-UPDATABLE value="true"/>
  <PARAMETER name="thresholdProbabilityEntity" value="0.2"/>
  <PARAMETER name="thresholdProbabilityBoundary" value="0.42"/>
  <PARAMETER name="thresholdProbabilityClassification" value="0.5"/>
  <EVALUATION method="holdout" runs="2" ratio="0.66"/>
  <ENGINE nickname="KNN" implementationName="KNNWeka" options = " -k 5 "/>
  <DATASET>
     <INSTANCE-TYPE>Sentence</INSTANCE-TYPE>
     <NGRAM>
        <NAME>Sent1gram</NAME>
        <NUMBER>1</NUMBER>
        <CONSNUM>2</CONSNUM>
        <CONS-1>
            <TYPE>Token</TYPE>
            <FEATURE>root</FEATURE>
        </CONS-1>
        <CONS-2>
            <TYPE>Token</TYPE>
            <FEATURE>category</FEATURE>
        </CONS-2>
        <WEIGHT>10.0</WEIGHT>
     </NGRAM>
     <ATTRIBUTE>
        <NAME>Class</NAME>
```

```
            <SEMTYPE>NOMINAL</SEMTYPE>
            <TYPE>Sentence</TYPE>
            <FEATURE>sent_size</FEATURE>
            <POSITION>0</POSITION>
         </ATTRIBUTE>
         <ATTRIBUTE>
            <NAME>Class</NAME>
            <SEMTYPE>NOMINAL</SEMTYPE>
            <TYPE>Sentence</TYPE>
            <FEATURE>class</FEATURE>
            <POSITION>0</POSITION>
            <CLASS/>
         </ATTRIBUTE>
      </DATASET>
</ML-CONFIG>
```

The last configuration file is for relation extraction. It does not specify any optional setting, meaning that it uses all the default values of those settings (see Section 11.2.2 for the default values of all possible settings). For example,

- it sets the *surround mode* as "false";
- both the label list and NLP feature list are updatable.
- the threshold of the probability for classification is set as 0.5.
- it uses the one against all others method for converting the multi-class problem into binary class problems for the SVM learning.
- for evaluation it uses the hold-out testing with ratio as 0.66 and only one run.

The configuration file specifies the learning algorithm as the Naive Bayes method implemented in Weka. However, other learning algorithms can be used as well.

The linguistic features used for relation extraction are more complicated than other two types of NLP learning. For relation learning, user must specify an annotation type covering the two arguments of relation as the instance type. User may also need to specify the annotation type for one argument if s/he wants to use the feature related only to the argument. In the **DATASET** sub-element, the instance type is specified as the annotation type "RE_INS". Two arguments of the instance are specified by the two features "arg1" and "arg2" of the annotation type "RE_INS", which should refer to the same values of the argument IDs as those specified in the *ARG* sub-elements of the sub-element *FEATURE-ARG1* and *FEATURE-ARG2*. The linguistic features related to the first argument of relation are defined in the sub-element *FEATURE-ARG1*, in which the feature "string" of the annotation type "Token" of the first argument is used as one NLP feature for relation learning. Similarly the token's form feature related to the second argument is also used as another NLP feature, as specified in the sub-element *FEATURE-ARG2*. By the first *ATTRIBUTE_REL* the feature "t12" of the annotation type *RE_INS* related to the relation instance defines another feature for relation learning. The second *ATTRIBUTE_REL* defines the label. It specifies the class labels as the values of the feature "Relation_type" of the annotation type *ACERelation*. The feature "MENTION_ARG1" of the type *ACERelation* should refer to the identification feature (namely the values of feature "MENTION_ID" of the annotation type "ACEEntity") of the first argument annotation. The feature "MENTION_ARG2" should refer to the second argument annotation of the relation.

```
<?xml version="1.0"?>
<ML-CONFIG>
   <ENGINE nickname="NB" implementationName="NaiveBayesWeka"/>
   <DATASET>
     <INSTANCE-TYPE>RE_INS</INSTANCE-TYPE>
     <INSTANCE-ARG1>arg1</INSTANCE-ARG1>
     <INSTANCE-ARG2>arg2</INSTANCE-ARG2>
     <FEATURES-ARG1>
       <ARG>
         <NAME>ARG1</NAME>
         <SEMTYPE>NOMINAL</SEMTYPE>
         <TYPE>ACEEntity</TYPE>
         <FEATURE>MENTION_ID</FEATURE>
       </ARG>
       <ATTRIBUTE>
         <NAME>Form</NAME>
         <SEMTYPE>NOMINAL</SEMTYPE>
         <TYPE>Token</TYPE>
         <FEATURE>string</FEATURE>
         <POSITION>0</POSITION>
       </ATTRIBUTE>
       </FEATURES-ARG1>
       <FEATURES-ARG2>
        <ARG>
          <NAME>ARG2</NAME>
          <SEMTYPE>NOMINAL</SEMTYPE>
          <TYPE>ACEEntity</TYPE>
          <FEATURE>MENTION_ID</FEATURE>
        </ARG>
        <ATTRIBUTE>
          <NAME>Form</NAME>
          <SEMTYPE>NOMINAL</SEMTYPE>
          <TYPE>Token</TYPE>
          <FEATURE>string</FEATURE>
          <POSITION>0</POSITION>
        </ATTRIBUTE>
```

```
      </FEATURES-ARG2>
      <ATTRIBUTE_REL>
        <NAME>EntityCom1</NAME>
        <SEMTYPE>NOMINAL</SEMTYPE>
        <TYPE>RE_INS</TYPE>
        <ARG1>arg1</ARG1>
        <ARG2>arg2</ARG2>
        <FEATURE>t12</FEATURE>
     </ATTRIBUTE_REL>
     <ATTRIBUTE_REL>
       <NAME>Class</NAME>
       <SEMTYPE>NOMINAL</SEMTYPE>
       <TYPE>ACERelation</TYPE>
       <ARG1>MENTION_ARG1</ARG1>
       <ARG2>MENTION_ARG2</ARG2>
       <FEATURE>Relation_type</FEATURE>
       <CLASS/>
     </ATTRIBUTE_REL>
 </DATASET>
</ML-CONFIG>
```

11.4 How to use the ML API

The ML API implements the procedure of using supervised machine learning for NLP, which generally has two steps, training and application. The training step learns some models from labelled data. The application step applies the learned models to the unlabelled data to add labels. Therefore, in order to use supervised ML for NLP, one should have some labelled data, which can be obtained either by manually annotating or from other resources. One also needs determine which linguistic features are used in the training (and the same features should be used in the application as well). Note that in the implementation all those features are based some GATE annotation type and one annotation feature of that type. Finally one should determine which learning algorithm will be used.

Based on the general procedure, we explain how to use the ML API step by step in the following.

1. Annotate some documents with labels that you want ML API to learn. The labels should be represented by the values of some feature of one GATE annotation type.
2. Determine the linguistic features that you want the ML API to use for learning.
3. Use the PRs of the GATE to obtain those linguistic features so that the documents for training or application indeed contain the features. By using the ANNIE you can have many useful features. The other PRs such as GATE morphological analyser and

the parsers may produce useful features as well. Sometimes you may need to write some Jape scripts to produce the features you want. Note that all the features are represented as values of the features of some GATE's annotation types, which should be specified in the *DATASET* element of the configuration file.

4. Create an XML configuration file for your learning problem. The file should contain one *DATASET* element specifying the NLP features used, one *ENGINE* element specifying the learning algorithm, and some optional settings as necessary. (**Tip**: it may be easier of copying one of the configuration files presented above and modifying them for your problem than writing a configuration file from scratch.)

5. Load the training documents containing the required annotations (in one annotation set) representing the linguistic feautures and the label, and put them into a corpus.

6. Load the ML API into GATE GUI. First you need load the plugin with name "learning" into GATE using the tool *Manage CREOLE Plugins*, if the GATE you are using haven't done it yet. Then you can create a PR for the plugin from the "Batch Learning PR" in the existing PR list by providing a configuration file for the ML plugin. After that you can put the PR into a *Corpus Pipeline* application to use it. Add the corpus containing training documents into the application too. Set the inputASName as the annotation set containing the annotations for linguistic features and labels.

7. Select the run-time parameter *learningMode* as "TRAINING" to learn a model from the training data. Or select the *learningMode* as "EVALUATION" to do evaluation on the training data and get some results. Note that when using the "EVALUATION" mode, please make sure that the *outputASName* is set with the same annotation set as the *inputASName*. (**Tip**: it may save your time if you first try the "EVALUATION" mode on a small number of documents to make sure that the ML API works well on your problem and outputs some reasonable results before training on the large data.)

8. If you want to apply the learned model to the new documents, load those new document into GATE, and pre-process them using exactly the same PRs as those used for pre-processing the training documents. Then load the ML API with the same configuration file as for training into the GATE, select the *learningMode* as "APPLICATION", and run the ML API on the corpus containing the new documents. The application results, namely the new annotations containing the labels, will be added into the annotation set specified by the *outputASName*.

9. If you just want the feature files produced by the system and do not want to do any learning or application, select the learning mode "ProduceFeatureFilesOnly".

11.5 The outputs of the ML API

There are several different types of the outputs of the ML API. First the ML API outputs some information about the learning settings. Those information will be printed in the

Messages Window of the GATE GUI and also into the log file "logFileForNLPLearning.save". The amount of the information displayed can be set via the *VERBORSITY* parameter in the XML configuration file. The main output of the learning system are the results of the ML API, which are different for different usage modes. In *training* mode the system produce the learned models. In *applicaiton* mode it annotates the documents using the learned model. In the *evaluation* mode it displays the evaluation results. Finally in *ProduceFeatureFilesOnly* mode it produces several feature files for the current corpus. In the following we explain the outputs for different learning modes, respectively.

Note that all the files produced by the ML API, including the log file, are in the sub-directory "savedFiles" of the working directory where the XML configuration file is in.

11.5.1 Training results

When the ML API is in the *training* mode, its main output is the learned model stored in one file named "learnedModels.save". For the SVM, the learned model file is a text file. For the learning algorithms implemented in Weka, the model file is a binary file. The output also includes the feature files described in subsection 11.5.4.

11.5.2 Application results

The main application result is the annotations added into the documents. Those annotations are the results of applying the ML model to the documents. The type of the annotations is specified in the class element in the configuration file. The class label of one annotation is the value of the annotation feature also specified in the class element. The annotation has another feature with name as "prob" which presents the confidence level of the learned model on this annotation.

11.5.3 Evaluation results

The ML API outputs the evaluation results for each run of the evaluation and also the averaged results over all the runs. For each run, it first prints the message about the names of documents in corpus for training and testing, respectively. Then it displays the evaluation results of this run — first the results for each class label and then the micro-averaged results over all labels. For each label, it presents the name of label, the number of instances belonging to the label in the training data, and the F-measure results on the testing data — the numbers of correct, partial correct, spurious and missing instances in the testing data, and the two types of F-measures (Precision, Recall and F1): the strict and the lenient. The F-measure results are obtained by using the *AnnotationDiff Tool* which is described in Chapter 13. Finally the system presents the means of the results of all runs for each label and the micro-averaged results, respectively.

11.5.4 Feature files

The ML API produces several feature files from the documents with the GATE annotations. These feature files could be used by user for evaluating the learning algorithms not implemented in this plug-in. In the following we describe the formats of those feature files. Note that all the data files described in the following can be obtained by selecting the run time parameter *learningMode* as "ProduceFeatureFilesOnly". But you may get some of data files when you use other learning mode.

NLP feature file. This file, named as *NLPFeatureData.save*, contains the NLP features of the instances, which are defined in the configuration file. The first few lines of an NLP feature file for information extraction are as

```
Class(es) Form(-1) Form(0) Form(1) Ortho(-1) Ortho(0) Ortho(1)
0 ft-airlines-27-jul-2001.xml 512
1 Number_BB _NA[-1] _Form_Seven _Form_UK[1] _NA[-1] _Ortho_upperInitial _Ortho_allCaps[1]
1 Country_BB _Form_Seven[-1] _Form_UK _Form_airlines[1] _Ortho_upperInitial[-1]
           _Ortho_allCaps _Ortho_lowercase[1]
0 _Form_UK[-1] _Form_airlines _Form_including[1] _Ortho_allCaps[-1] _Ortho_lowercase
           _Ortho_lowercase[1]
0 _Form_airlines[-1] _Form_including _Form_British[1] _Ortho_lowercase[-1] _Ortho_lowercas
           _Ortho_upperInitial[1]
1 Airline_BB _Form_including[-1] _Form_British _Form_Airways[1] _Ortho_lowercase[-1]
           _Ortho_upperInitial _Ortho_upperInitial[1]
1 Airline _Form_British[-1] _Form_Airways _Form_[1], _Ortho_upperInitial[-1]
           _Ortho_upperInitial _NA[1]
0 _Form_Airways[-1] _Form_, _Form_Virgin[1] _Ortho_upperInitial[-1] _NA
           _Ortho_upperInitial[1]
```

The first line of the NLP feature file lists the names of all features used. The number in the parenthesis following a feature name indicates the position of the feature. For example, "Form(-1)" means the form of token which is immediately before the current token, and "Form(0)" means the form of the current token. The NLP features for all instances are listed for one document following another. For one document, the first line shows the index of the document, the document's name and the number of instances in the document, as shown in the second line above. Then each of the following lines is for each of the instances in the document, with the order of their occurrences in the document. For each line for one instance, the first item is a number n, representing the number of class labels of the instance. Then the following n items are the labels. If the current instance is the first instance of an entity, its corresponding label has a suffix "_BB". The other items following the label item(s) are the NLP features of the instance, in the order as listed in the first line of the file. Each NLP feature contains the feature's name and value, separated by "_". At the end of one NLP feature, there may be an integer in the parenthesis "[]", which represents the position of the feature relative to the current instance. If there is no such an integer in "[]" at the end of one NLP feature, then the feature is at the position 0.

Feature vector file. It has the file name *featureVectorsData.save* and stores the feature vector in the sparse format for each instance. The first few lines of the feature vector file corresponding to the NLP feature file shown above are as

```
0 512 ft-airlines-27-jul-2001.xml
1 2 1 2 439:1.0 761:1.0 100300:1.0 100763:1.0
2 2 3 4 300:1.0 763:1.0 50439:1.0 50761:1.0 100440:1.0 100762:1.0
3 0 440:1.0 762:1.0 50300:1.0 50763:1.0 100441:1.0 100762:1.0
4 0 441:1.0 762:1.0 50440:1.0 50762:1.0 100020:1.0 100761:1.0
5 1 5 20:1.0 761:1.0 50441:1.0 50762:1.0 100442:1.0 100761:1.0
6 1 6 442:1.0 761:1.0 50020:1.0 50761:1.0 100066:1.0
7 0 66:1.0 50442:1.0 50761:1.0 100443:1.0 100761:1.0
```

The feature vectors are also listed for one document after another. For one document, the first line shows the index of the document, the number of instances in the document, and the document's name. Each of the following lines is for each of the instances in the document. The first item in the line of one instance is the index of the instance in the document. The second item is a number n, representing the number of labels the instance has. The following n items are the labels' indexes of the instance.

For the text classification and relation learning, the label's index comes directly from the *label list file* described below. For the chunk learning, the label's index presented in the feature vector file is a bit complicated, as explained in the following. If an instance (e.g. token) is the first one of a chunk with a label k, then the instance has the label's index as $2 * k - 1$, as shown in the fifth instance. If it is the last instance of the chunk, it has the label's index as $2 * k$, as shown in the sixth instance. If the instance is both the first one and the last one of the chunk (namely the chunk consists of one instance), it has two label indexes, $2 * k - 1$ and $2 * k$, as shown in the first (and second) instance.

The items following the label item are the non-zero components of the feature vector. Each component is represented by two numbers separated by ":". The first number is the dimension of the component in the feature vector, and the second one is the value of the component.

Label list file. It has the name *LabelsList.save* and stores a list of labels and their indexes. The following is a part of label list. Each line shows one label name and its index in the label list.

```
Airline 3
Bank 13
CalendarMonth 11
CalendarYear 10
Company 6
Continent 8
```

```
Country 2
CountryCapital 15
Date 21
DayOfWeek 4
```

NLP feature list. It has the name *NLPFeaturesList.save* and contains a list of NLP features and their indexes in the list. The following are the first few lines of an NLP feature list file.

```
totalNumDocs=14915
_EntityType_Date 13 1731
_EntityType_Location 170 1081
_EntityType_Money 523 3774
_EntityType_Organization 12 2387
_EntityType_Person 191 421
_EntityType_Unknown 76 218
_Form_'' 112 775
_Form_\$ 527 74
_Form_' 508 37
_Form_'s 63 731
_Form_( 526 111
```

The first line of the file shows the number of instances from which the NLP features were collected. The number of instances will be used for the computation of the so-called *idf* in document or sentence classification. The following lines are for the NLP features. Each line is for one unique feature. The first item in the line represents the NLP feature, which is a combination of the feature's name defined in the configuration file and the value of the feature. The second item is a positive integer, representing the index of the feature in the list. The last item is the number of instances that the feature occurs, which is needed for computing the *idf*.

N-grams (or language model) file. The file has the name **NgramList.save**, which only can be produced by selecting the learning mode as "ProduceFeatureFilesOnly". In order to produce the n-gram data, user may just use a very simple configuration file, e.g. it could only contain the *DATASET* element, and the data element could only contain an *NGRAM* element to specify the type of n-gram and the *INSTANCE-TYPE* element to define the part of document from which the n-gram data are created. The NGRAM element in configuration file specifies what type of n-grams the ML PR produces (see Section 11.2.2 for the explanation of the n-gram definition). For example, if you specify a bi-gram based on the string form of *Token*, you will obtain a list of bi-gram from the corpus you used. The following are the first lines of a bi-gram list based on the token's form and was obtained from 3 documents.

```
## The following 2-gram were obtained from 3 documents or examples
```

```
Aug<>, 3
Female<>; 3
Human<>; 3
2004<>Aug 3
;<>Female 3
.<>The 3
of<>a 3
)<>: 3
,<>and 3
to<>be 3
;<>Human 3
```

The two terms in one bi-gram are separated by "<>". The number following one n-gram is the number of occurrences of that n-gram in the corpus. The n-gram list is ordered according to the numbers of occurrence of n-gram terms. The most frequent terms in the corpus are in the beginning of the list.

The n-gram data produced can be based on any features of annotations available in the documents. Hence it can not only produce the conventional n-gram data based on the token's form or lemma, but also the n-gram based on e.g. token's POS, or combination of token's POS and form, or any feature of sentence annotation (see Section 11.2.2 for how to define different types of n-gram). Therefore one can regard the n-gram defined and produced by the ML PR as some kind of generalised n-gram.

Document-term matrix file. The file has the name **documentByTermMatrix.save**, which only can be produced by selecting the learning mode as "ProduceFeatureFilesOnly". Document-term matrix presents the weights of terms appearing in each of documents (see Section 9.19 for more explanations). Currently three types of weight are implemented, the binary, term frequency (tf), and tf-idf. The binary weight is the simple one – it is 1 for one term appearing in document and 0 if one term is not in document. *tf* refers to the number of occurrences of one term in document. *tf-idf* is popular in information retrieval and text mining. It is a multiplication of tf and idf. *idf* refers to inverse document frequency, defining as

$$idf_i = log\frac{|D|}{|\{d_j : t_i \in d_j\}|}$$

where $|D|$ is the total number of documents in the corpus, and $|\{d_j : t_i \in d_j\}|$ is the number of documents in which the term t_i appears. The type of weight is specified by the sub-element *ValueTypeNgram* in the *DATASET* element in configuration file (see Section 11.2.2).

Like the n-gram data, in order to produce the document-term matrix, user may just use a very simple configuration file, e.g. it could only contain the *DATASET* element, and the data element could only contain two elements, the *INSTANCE-TYPE* element to define the part of document from which the terms are counted, and a *NGRAM* element to specify the type of n-gram. As said above, the element *ValueTypeNgram* specifies the type of value used in the matrix. If it is not presented, the default type tf-idf will be used. The conventional document-term matrix can be produced by the uni-gram based on token's form or lemma and the instance-type covering the whole document. Any other choices for the n-gram and the instance type will result in some generalised document-term matrix.

The following was extracted from the beginning of a document-term matrix file, produced by uni-gram based on token's form. It presents a part of terms and their *tf* values in the document named "27.xml". One term and its tf are separated by ":". The terms are ranked in alphabetic order.

```
0 Documentname="27.xml", has 1 parts:
":2 (:6 ):6 ,:14 -:1 .:16 /:1 124:1 2004:1 22:1 29:1 330:1 54:1 8:2 ::5
;:11 Abstract:1 Adaptation:1 Adult:1 Atopic:2 Attachment:3 Aug:1
Bindungssicherheit:1 Cross-:1 Dermatitis:2 English:1 F-SOZU:1 Female:1
Human:1 In:1 Index:1 Insecure:1 Interpersonal:1 Irrespective:1 It:1 K-:1
Lebensqualitat:1 Life:1 Male:1 NSI:2 Neurodermitis:2 OT:1 Original:1
Patients:1 Psychological:1 Psychologie:1 Psychosomatik:1 Psychotherapie:1
Quality:1 Questionnaire:1 RSQ:1 Relations:1 Relationship:1 SCORAD:1 Scales:1
Sectional:1 Securely:1 Severity:2 Skindex-:1 Social:1 Studies:1 Suffering:1
Support:1 The:1 Title:1 We:3 [:1 ]:1 a:4 absence:1 affection:1 along:2
amount:1 an:1 and:9 as:1 assessed:1 association:2 atopic:5 attached:7
```

A list of names of documents processed. The file has the name **docsName.save**, which only can be produced by selecting the learning mode as "ProduceFeatureFilesOnly". It contains the names of all the documents processed. The first line shows the number of documents in the list. Then each line list one document's name. The first lines of one file are showed in the following.

```
##totalDocs=3
ft-bank-of-england-02-aug-2001.xml
ft-airtours-08-aug-2001.xml
ft-airlines-27-jul-2001.xml
```

A list of names of the selected documents for active learning purpose. The file has the name **ALSelectedDocs.save**. It is pure text file. It will be produced by selecting the learning mode as "ProduceFeatureFilesOnly". The file contain the names of documents which have been selected for annotating and training in the active learning process. It is used by the "RankingDocsForAL" learning mode to exclude those selected documents from the ranked documents for active learning purpose. When one or more documents are selected for annotating and training, their names should be put into this file, one line per document's name.

A list of names of ranked documents for active learning purpose. The file has the name **ALRankedDocs.save**. It can only be produced by selecting the learning mode as

"RankingDocsForAL". The file contains the list of names of the documents ranked for active learnig, according to their usefulness for learning. Those in the front of the list are the most useful documents for learning. The first line in the file shows the total number of documents in the list. Each of other lines in the fils lists one document and the averaged confidence score for classifiying the documents. An example of the file are showed in the following.

```
##numDocsRanked=3
ft-airlines-27-jul-2001.xml_000201 8.61744
ft-bank-of-england-02-aug-2001.xml_000221 8.672693
ft-airtours-08-aug-2001.xml_000211 9.82562
```

Chapter 12

Tools for Alignment Tasks

12.1 Introduction

This chapter introduces a new plug-in that allows users to create new tools for text alignment and cross-document processing.

Text alignment can be achieved at a document, section, paragraph, sentence and a word level. Given two parallel corpora, where the first corpus contains documents in a source language and the other in a target language, the first task is to find out the parallel documents and align them at the document level. Cross-document processing is where multiple documents need to be looked up in order to achieve some tasks, for example cross-document co-reference resolution.

For these tasks one would need to refer to more than one document at the same time. Hence, a need arises for Processing Resources (PRs) which can accept more than one document as parameters. For example given two documents, a source and a target, a Sentence Alignment PR would need to refer to both of them to identify which sentence of the source document aligns with which sentence of the target document. Similarly for a cross-document co-reference resolution, the respective PR would need to access both the documents simultaneously.

GATE framework deals with one document at a time. A GATE document does not have any dependency on any of the other resources in GATE. It means that it is an independent object which can be added or removed from a corpus or datastore. The standard behaviour of the GATE PRs contradicts the above mentioned requirements. GATE PRs accept one document at a time. Corpus pipeline which accepts a corpus as input, considers only one document at a time. Having said this it is not impossible to make PRs accepting more than one document but this would require a lot of re-engineering.

12.2 Tools for Alignment Tasks

We have introduced a few new resources in GATE to address these issues. These include CompoundDocument, CompositeDocument, and a new AlignmentEditor to name a few. Below we describe these components and how to use them.

12.2.1 Compound Document

A new Language Resource (LR), called CompoundDocument, has been introduced which is a collection of documents and allow various documents to be grouped together under a single document. The CompoundDocument allows adding more documents to it and removing them if required. It implements the gate.Document interface allowing users to carry out all operations that can be done on a normal gate document. A PR wishing to access multiple documents can group them under a composite document which internally allows accessing its members.

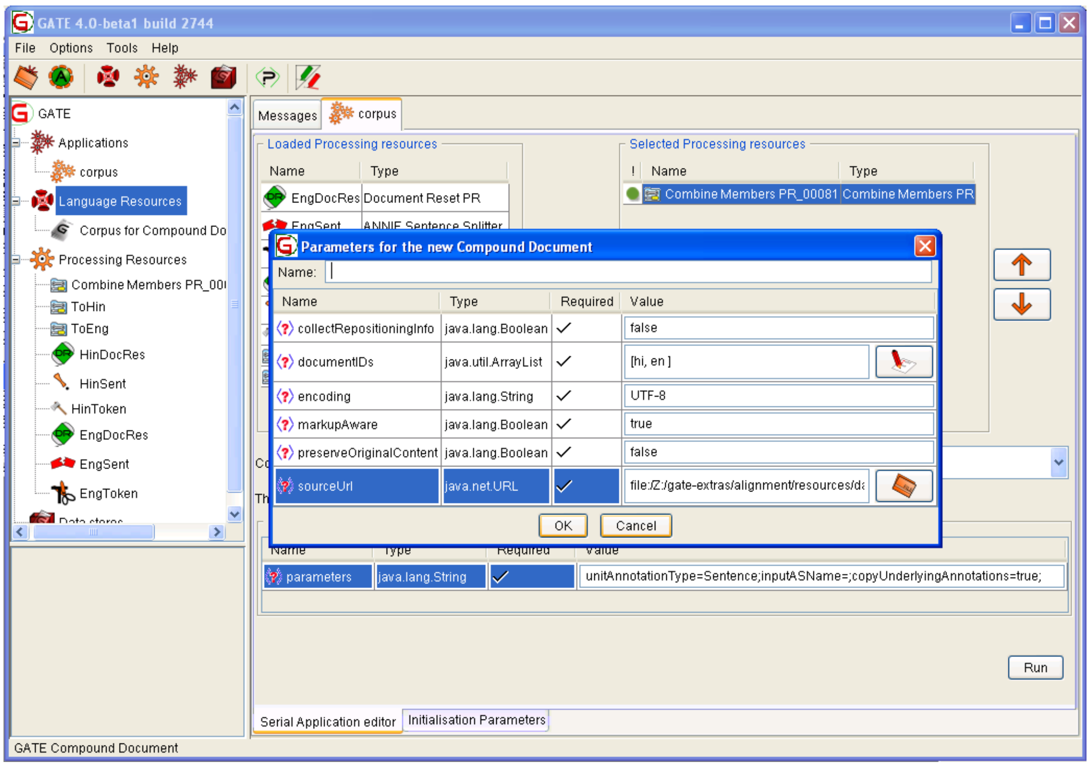
To instantiate CompoundDocument user needs to provide the following parameters.

- encoding - encoding of the member documents. All document members must have the same encoding (e.g. Unicode, UTF-8, UTF-16).
- collectRepositioningInfo - this parameter indicates whether the underlying documents should collect the repositioning information in case the contents of these documents change.
- preserveOriginalContent - if the original content of the underlying documents should be preserved.
- documentIDs - users need to provide a unique ID for each document member. These ids are used to locate the appropriate documents.
- sourceUrl - given a URL of one of the member documents, the instance of Compound-Document searches for other members in the same folder based on the ids provided in the documentIDs parameter. Following document name conventions are followed to search other member documents:
  - FileName.id.extension (filename followed by id followed the extension and all of these separated by a . (dot)).
  - For example if user provides three document IDs (e.g. “en”, “hi” and “gu”) and selects a file with name “File.en.xml”, the CompoundDocument will search for rest of the documents (i.e. “File.hi.xml” and “File.gu.xml”). The file name (i.e. “File”) and the extension (i.e. “xml”) remain common for all three members of the compound document.

To summarize, the following parameters are required to instantiate the CompoundDocument.

1. encoding (default = UTF-8, java.lang.String, required = true)
2. collectRepositioningInformation (default = false, java.lang.Boolean, required=true)
3. preserveOriginalContent (default = false, java.lang.Boolean, required = true)
4. documentIDs (default = empty, java.util.ArrayList, required = true)
5. sourceUrl (default = empty, java.net.Url, required = true)

Figure 12.1 shows a snapshot for instantiating a compound document from the GATE GUI.



Name	Type
EngDocRes	Document Reset PR
EngSent	ANNIE Sentence Splitter

!	Name	Type
●	Combine Members PR_00081	Combine Members PR

Name:

Name	Type	Required	Value
collectRepositioningInfo	java.lang.Boolean	✓	false
documentIDs	java.util.ArrayList	✓	[hi, en]
encoding	java.lang.String	✓	UTF-8
markupAware	java.lang.Boolean	✓	true
preserveOriginalContent	java.lang.Boolean	✓	false
sourceUrl	java.net.URL	✓	file:/Z:/gate-extras/alignment/resources/d

Name	Type	Required	Value
parameters	java.lang.String	✓	unitAnnotationType=Sentence;inputASName=;copyUnderlyingAnnotations=true;

Figure 12.1: Compound Document

Compound document provides various methods that help in accessing their individual members.

```
public Document getDocument(String docid);
```

The following method returns a map of documents where the key is a document ID and the value is its respective document.

```
public Map getDocuments();
```

Please note that at a given time only one document member has a focus set on it. All the standard document methods of gate.Document interface apply to this set document. For example there are two document, “hi” and “en” and the focus is set on the document “hi” then the getAnnotations() method will return a default annotation set of the “hi” document. One can use the following method to switch the focus of a compound document to a different document.

```
public void setCurrentDocument(String documentID);
public Document getCurrentDocument();
```

As explained above new documents can be added to or removed from the compound document using the following method.

```
public void removeDocument(String documentID);
public void addDocument(String documentID, Document document);
```

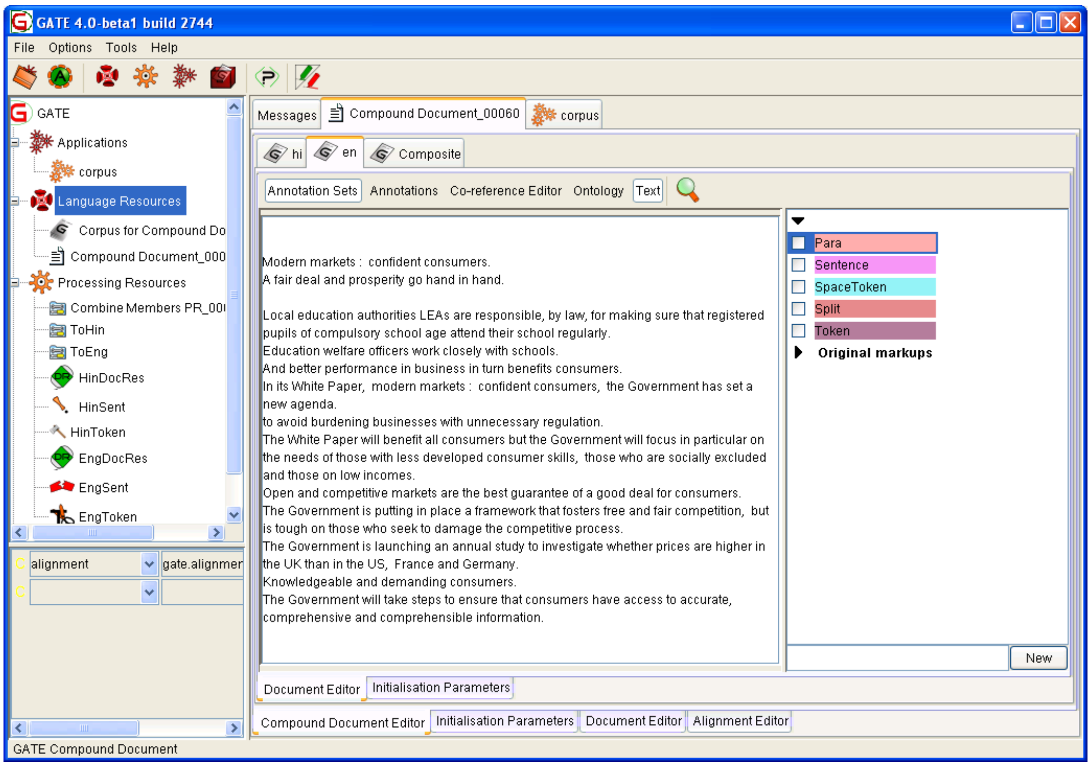
The following code snippet demonstrates how to create a new compound document using GATE API.

```
// step 1: initialize GATE
Gate.init();

// step 2: load the Alignment plugin
File alignmentHome = new File(Gate.getPluginsHome(),‘‘Alignment’’);
Gate.getCreoleRegister().addDirectory(ontoHome.toURL());

// step 3: set the parameters
FeatureMap fm = Factory.newFeatureMap();

// for example you want to create a compound document for
// File.id1.xml and File.id2.xml
List docIDs = new ArrayList();
docIDs.add(‘‘id1’’);
doicIDs.add(‘‘id2’’);
fm.put(‘‘documentIDs’’, docIDs);
fm.ptu(‘‘sourceURL’’, new URL(‘‘file://z:/data/File.id1.xml’’));

// step 4: finally create an instance of compound document
Document aDocument = (gate.compound.CompoundDocument)
 Factory.createResource(‘‘gate.compound.impl.CompoundDocumentImpl’’, fm);
```

12.2.2 Compound Document Editor

Compound document editor is a visual resource (VR) associated with the compound document. The VR contains several tabs - each representing a different member of the compound document. All standard functionalities such as GATE document editor with all its add-on plug-ins such as AnnotationSetView, AnnotationsList, coreference editor etc. are available to be used with each individual member.

Figure 12.2 shows a compound document editor with enlgish and hindi documents being a member of the compound document.



Modern markets : confident consumers.
A fair deal and prosperity go hand in hand.

Local education authorities LEAs are responsible, by law, for making sure that registered pupils of compulsory school age attend their school regularly.
Education welfare officers work closely with schools.
And better performance in business in turn benefits consumers.
In its White Paper, modern markets : confident consumers, the Government has set a new agenda.
to avoid burdening businesses with unnecessary regulation.
The White Paper will benefit all consumers but the Government will focus in particular on the needs of those with less developed consumer skills, those who are socially excluded and those on low incomes.
Open and competitive markets are the best guarantee of a good deal for consumers.
The Government is putting in place a framework that fosters free and fair competition, but is tough on those who seek to damage the competitive process.
The Government is launching an annual study to investigate whether prices are higher in the UK than in the US, France and Germany.
Knowledgeable and demanding consumers.
The Government will take steps to ensure that consumers have access to accurate, comprehensive and comprehensible information.

Figure 12.2: Compound Document Editor

12.2.3 Composite Document

Composite document allows users to merge the texts of members of a compound document and yet keep the merged text linked with their respective member documents. In other words, if users make any change to the composite document (e.g. add new annotations or remove any existing annotations), the relevant effect is made to their respective documents.

A PR called CombineMembersPR allows creating a new composite document. It asks for a class name that implements the CombiningMethod interface. The CombiningMethod tells the CombineMembersPR how to combine texts and create a new composite document.

For example, a default implementation of the CombiningMethod, called DefaultCombiningMethod, takes the following parameters and put the text of the compound document's members into a new composite document.

```
unitAnnotationType=Sentence
inputASName=Key
copyUnderlyingAnnotations=true;
```

The first parameter tells the combining method that it is the "Sentence" annotation type whose text needs to be merged and it should be taken from the "Key" annotation set (second parameter) and finally all the underlying annotations of every Sentence annotation must be copied in the composite document.

If there are two members of a compound document (e.g. hi and en), given the above parameters, the combining method finds out all the annotations of type Sentence from each document, sort them in ascending order, and one annotation from each document is put one after another in a composite document. This operation continues until all the annotations have been traversed.

```
Document en     Document hi
Sen1            Shi1
Sen2            Shi2
Sen3            Shi3

Document Composite
Sen1
Shi1
Sen2
Shi2
Sen3
Shi3
```

The composite document also maintains mapping of text offsets such that if someone adds a new annotation to or removes any annotation from the composite document, they are added to or removed from their respective documents. Finally the newly created composite document becomes a member of the same compound document.

12.2.4 DeleteMembersPR

This PR allows deleting a specific member of the compound document. It takes a parameter called "documentID" and deletes a document with this name.

12.2.5 SwitchMembersPR

As described above, only one member of the compound document can have a focus set on it. PRs trying to use the getDocument() method gets a pointer to the compound document however all the other methods of the compound document gives access to the information of the set document member. So if user wants to process a particular member of the compound document with some PRs, S/he should use the SwitchMembersPR that takes one parameter called documentID and set the focus on the document with that specific id.

12.2.6 Saving as XML

Calling toXml() method on a compound document returns the XML representation of the member which has a focus set on it. However, GATE GUI provides an option to save all member documents in different files. This option appears in the option menu when a user right clicks on the compound document. User is asked to provide a name for directory in which all the members of the compound document are saved in separate files.

It is also possible to save all members of the compound document in a single XML file. The option, "Save in a single XML Document", also appears in the option menu. After saving it in a single XML document, user can use the option "Compound Document from XML" to load the document back in GATE.

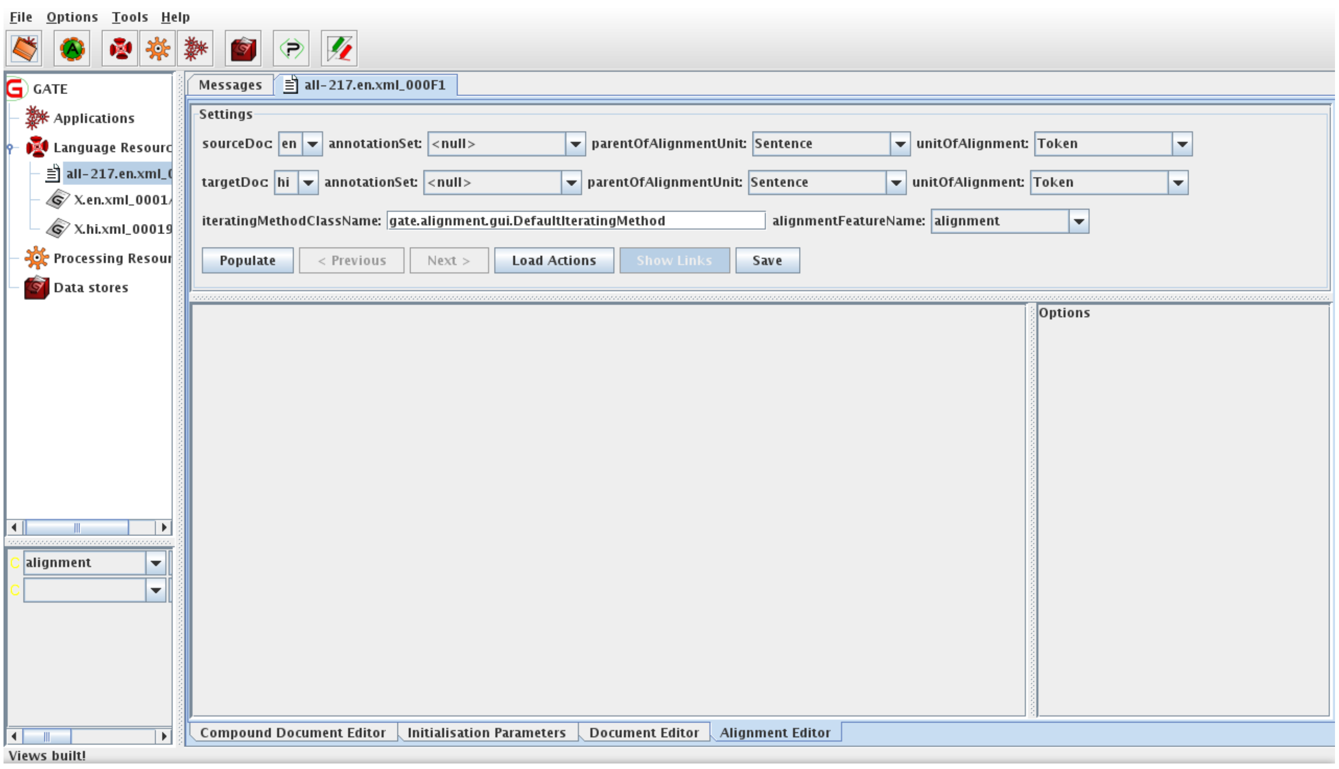
12.2.7 Alignment Editor

A new Visual Resource (VR) called AlignmentEditor has been implemented (Figure 12.3 and 12.4) and is attached with every compound document. As the name suggest, the purpose of the AlignmentEditor is to allow users to align texts from different members of the compound document at a section, paragraph, sentence and a word level. It provides a very user friendly interface to perform manual text alignment.

User is asked to provide certain parameters based on what a new alignment task is set up. Once the task has been setup user is shown some text and is asked to align the text. An instance of the gate.alignment.Alignment class is created and stored as a document feature on the compound document. This object is then used for storing all the alignment information such as which annotation is aligned with which annotations of what document and so on.

The parameters needed for setting up a new alignment task are as follows:

- Source and Target Documents: User is asked to choose one of the members of compound document as a source document and an another one as a target document.
- Annotation Sets: Users are also asked to choose relevant annotation sets in both the source and the target documents that need to be aligned.



sourceDoc: en annotationSet: <null> parentOfAlignmentUnit: Sentence unitOfAlignment: Token

targetDoc: hi annotationSet: <null> parentOfAlignmentUnit: Sentence unitOfAlignment: Token

iteratingMethodClassName: gate.alignment.gui.DefaultIteratingMethod alignmentFeatureName: alignment

alignment

Figure 12.3: Alignment Editor

- Unit Of Alignment: This is the annotation type that users want to perform alignment at. For example, if users want to align the text at a word level, they will need to process their documents with some tokenizer (e.g. ANNIE English Tokenizer) to generate tokens and provide Token as a unit of alignment.

- Parent Of Unit Of Alignment: Generally, if performing a word alignment task, people consider a pair of aligned sentences, one in a source language and the other one in a target language. Thus, the “Sentence” is a parent of unit of alignment. In other words, users should also process their documents with some sentence splitter (e.g. ANNIE Sentence Splitter) that identifies boundaries of the sentences and creates an annotation for each sentence in the text.

- Iterating Method: If the parent of unit of alignment is Sentence, it is essential to know the order in which sentences from the source and the target documents should be paired together. For example, one could simply specify to pair one sentence from the source document with one sentence from the target document (in the order they appear in their documents). However, it is possible that the sentences in both documents are not in the correct order or one sentence from the source document refers to more than one sentence in the target document or vice-verse. To make it customizable, this parameter allows users to specify a class that implements the gate.alignment.gui.IteratingMethod interface. The implementing class is seen as an iterator with a next() method that

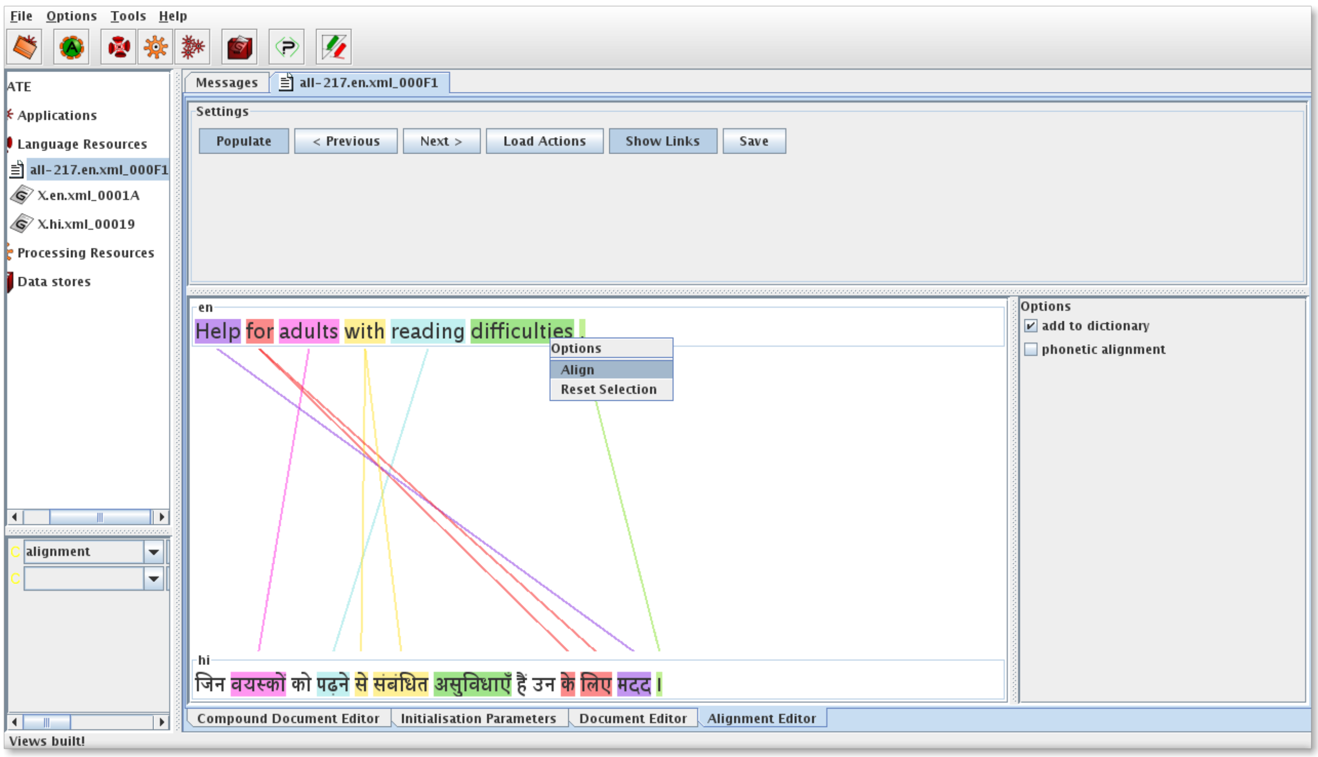


Figure 12.4: Alignment Editor with an example Pair

returns an object of gate.alignment.gui.Pair, one at a time. One such default implementation, gate.alignment.gui.DefaultIteratingMethod, is provided that takes two annotations (of type parent of unit of alignment), one from each, the source and the target documents, in their order of appearance, and form a pair.

- Alignment Feature Name: Information about the alignment (i.e. which annotation is aligned with what annotation) is stored as a document feature. Using this parameter, user can specify the name of the feature that should be used to store the alignment information.

```
Document en     Document hi
Sen1            Shi1
Sen2            Shi2
Sen3            Shi3
```

Given a compound document with two members (en and hi) as shown above, if user selects "en" as a source document, "hi" as a target document, "Key" as an input annotation set, "Sentence" as a parent of unit of alignment, "Token" as a value for unit of alignment and "gate.alignment.gui.DefaultIteratingMethod" as an iterating method, pairs will be created in the following manner.

```
Pair1 Sen1 Shi1
Pair2 Sen2 Shi2
Pair3 Sen3 Sen3
```

Each of these pairs is shown one at a time. If a user clicks on the next button, the next pair of sentences is shown. Similarly clicking on the previous button brings up the previous pair. In each of these sentences, the individual tokens are highlighted with a default colour (to mark the boundary for each unit of alignment). In order to align one or more units in the source language with one or more units in the target language, the user needs to select them by clicking on them individually. Clicking on units highlights them with an identical colour. Right clicking on any of the selected units brings up a menu with "Align" and "Reset Selection" options. The user can select "Align" to align the selected units or can select the "Reset Selection" option to reset the selection. If the annotations are unaligned, they are highlighted with the same color and a link (a line with the same color) between them is shown. In order to unalign them, the user needs to right click on the aligned annotation and click on the "Remove Alignment" option. If the annotation is part of an one-to-one alignment, both the annotations (i.e. the source and the target annotations) are unaligned. However, if there is another annotation in the same pair and the same document that is aligned with the same annotations in the target document, the annotation on which the user right clicks is taken out of the alignment leaving rest of the annotations still aligned.

Currently there is no implementation provided to export this alignment information, but one could easily write a PR that reads this information and export it to his/her desired format.

Advanced Features

The editor also allows adding more actions to the editor. There are in total three different types of actions:

- PreDisplayAction
- AlignmentAction
- FinishAlignmentAction

When users click on the next or the previous button, the editor obtains a pair that needs to be shown in the editor. Before it is displayed, the editor calls the registered instances of the PreDisplayAction and passes them the pair object. This could be helpful to preprocess a pair before it is displayed in the editor. For example, a wrapper could be written for a word alignment algorithm that identifies word alignments in the given sentence pair. More information on the methods of the PreDisplayAction interface could be found in the javadoc.

In case of the word alignment scenario, when a sentence pair is displayed, users can align new words and delete existing ones if needed. This could be achieved by clicking on the relevant buttons in the options menu. All buttons that appear in the option menu are instances of the AlignmentAction. As explained earlier, “Align”, “Reset Selection” and “Remove Alignments” are the three default buttons that are available to the users. The editor also has an “options tab” where users are allowed to add new actions. Users wishing to add new options to this tab or to the “options menu” need to provide their own implementations of the AlignmentAction interface. Below we list some of the methods of the AlignmentAction interface.

- public boolean invokeForAlignedAnnotation()
- public boolean invokeForHighlightedUnalignedAnnotation()
- public boolean invokeForUnhighlightedUnalignedAnnotation()
- public boolean invokeWithAlignAction()
- public boolean invokeWithRemoveAction()
- getCaption()

Users might want to restrict showing these buttons based on some conditions. For example, the “Align” button appears only when users select the unaligned units. Similarly the “Remove Alignments” button appears only when users right click on any of the aligned units. This can be controlled with the help of first three methods as specified above. For example the method “invokeForAlignedAnnotation()” indicates that the button should only appear when users right click on any of the unaligned units.

It is also possible that users might want to perform additional tasks when they click on any of the “Align” or the “Remove Alignment” buttons. For example, users can build a dictionary with new entries while aligning word pairs. In this case, an additional task of adding new entries to the dictionary can be performed when the “Align” button is clicked. On the other hand, not all entries that users align should be included in the dictionary. For the ones which aligners think should go in dictionary, they might want to ask the editor to add them explicitly. All these issues can be controlled by returning appropriate values for the last two methods of the AlignmentAction interface (i.e. invokeWithAlignAction() and invokeWithRemoveAction()).

It is important to note that the new option is added either to the options tab or to the options menu. Users wishing to add it as a button to the options menu must return “false” for the invokeWithAlignAction() and invokeWithRemoveAction() methods. Users wishing to add it to the options tab, must return true for at least one of these two methods. In case of the latter, the getCaption() method is used for obtaining a string that is used for creating a checkbox which is then added to the options tab. When users click on the “Align” or “Remove Alignment” button, the editor also calls the respective actions for the checked checkboxes.

Last of the three types of actions is FinishedAlignmentAction. Before users click on the next button, they are asked if the pair they were aligning has been aligned completely. In other words, if there is any alignment unit left that still needs to be aligned. If the alignment is complete, the registered instances of the FinishedAlignmentAction interface are called. This could be helpful to write an alignment exporter that takes an aligned pair as input and exports it in an appropriate format.

How to register actions? Having implemented various actions, users need to register them with the alignment editor. In order to do so, users can click on the "Load Actions" button. It brings a window and user is asked to provide a configuration file. A configuration file is a simple text file with fully-qualified class names specified in it. After the class name, users can specify any necessary parameters (delimited by a comma sign) that they wish to pass to respective actions classes when they are initialized. Below we give an example of such an entry in the actions configuration file.

```
#use the class DictionaryBuilder and pass the "/user-home/dictionary.txt" and
"root" as two parameters to the init method of the class.
gate.alignment.actions.DictionaryBuilder,/user-home/dictionary.txt,root
```

Chapter 13

Performance Evaluation of Language Analysers

When you can measure what you are speaking about, and express it in numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind: it may be the beginning of knowledge, but you have scarcely in your thoughts advanced to the stage of science. (Kelvin)

Not everything that counts can be counted, and not everything that can be counted counts. (Einstein)

GATE provides two useful tools for automatic evaluation: the AnnotationDiff tool and the Benchmarking Tool. These are particularly useful not just as a final measure of performance, but as a tool to aid system development by tracking progress and evaluating the impact of changes as they are made. The evaluation tool (AnnotationDiff) enables automated performance measurement and visualisation of the results, while the benchmarking tool enables the tracking of a system's progress and regression testing.

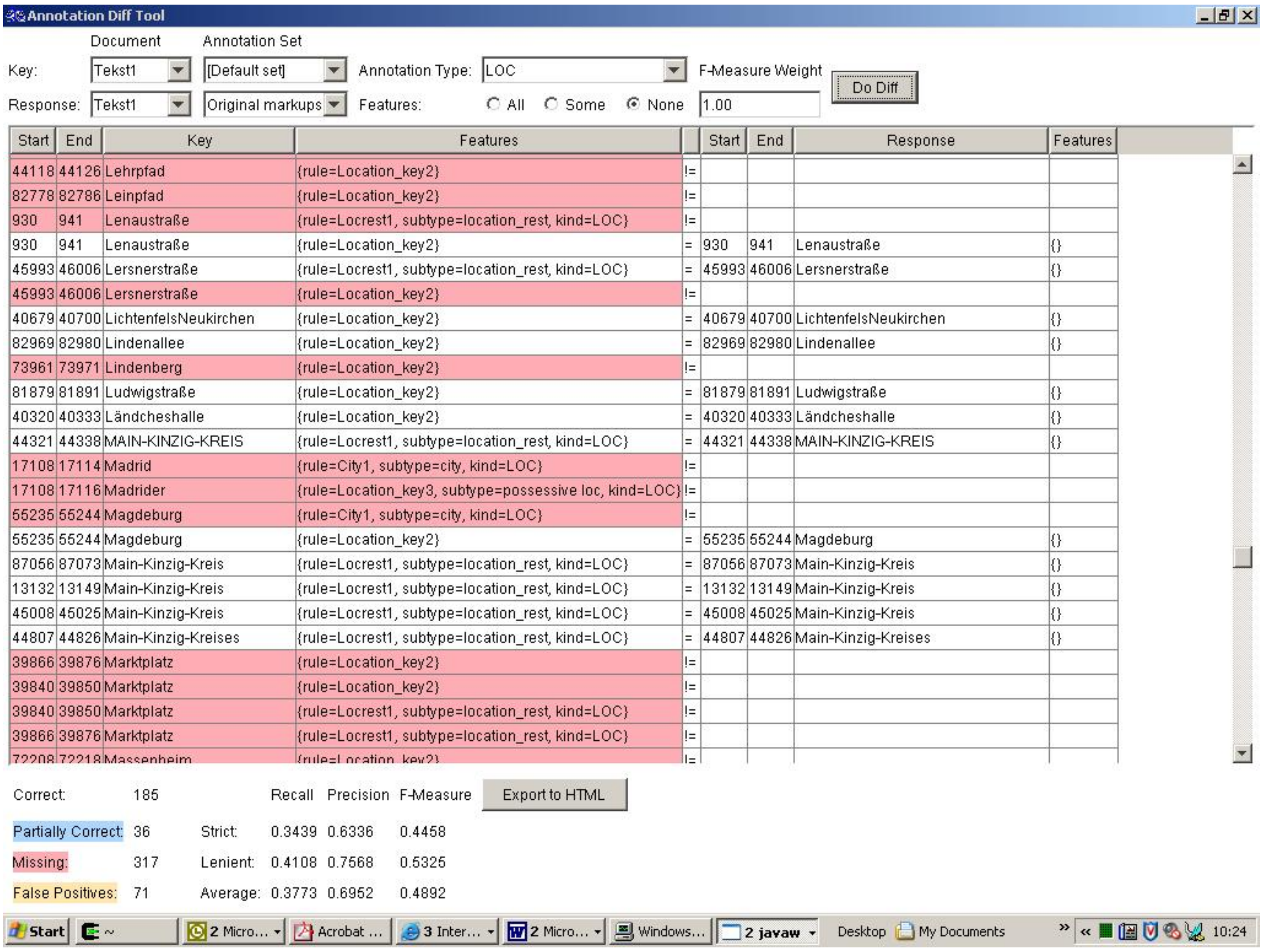
13.1 The AnnotationDiff Tool

The AnnotationDiff tool enables two sets of annotations on a document to be compared, in order either to compare a system-annotated text with a reference (hand-annotated) text, or to compare the output of two different versions of the system (or two different systems). For each annotation type, figures are generated for precision, recall, F-measure and false positives. Each of these can be calculated according to 3 different criteria - strict, lenient and average. The reason for this is to deal with partially correct responses in different ways.

- The Strict measure considers all partially correct responses as incorrect (spurious).

- The Lenient measure considers all partially correct responses as correct.
- The Average measure allocates a half weight to partially correct responses (i.e. it takes the average of strict and lenient).

It can be accessed both from GUI or from the API. Annotation Diff compares sets of annotations with the same type. When performing the diff, the annotation offsets and their features will be taken into consideration. and after that, the diff process is triggered. Figure 13.1 shows a part of the AnnotationDiff viewer.



Key: Document Tekst1, Annotation Set [Default set], Annotation Type: LOC, F-Measure Weight 1.00
Response: Document Tekst1, Annotation Set Original markups, Features: All Some [✓ None]

Start	End	Key	Features		Start	End	Response	Features
44118	44126	Lehrpfad	{rule=Location_key2}	!=				
82778	82786	Leinpfad	{rule=Location_key2}	!=				
930	941	Lenaustraße	{rule=Locrest1, subtype=location_rest, kind=LOC}	!=				
930	941	Lenaustraße	{rule=Location_key2}	=	930	941	Lenaustraße	{}
45993	46006	Lersnerstraße	{rule=Locrest1, subtype=location_rest, kind=LOC}	=	45993	46006	Lersnerstraße	{}
45993	46006	Lersnerstraße	{rule=Location_key2}	!=				
40679	40700	LichtenfelsNeukirchen	{rule=Location_key2}	=	40679	40700	LichtenfelsNeukirchen	{}
82969	82980	Lindenallee	{rule=Location_key2}	=	82969	82980	Lindenallee	{}
73961	73971	Lindenberg	{rule=Location_key2}	!=				
81879	81891	Ludwigstraße	{rule=Location_key2}	=	81879	81891	Ludwigstraße	{}
40320	40333	Ländcheshalle	{rule=Location_key2}	=	40320	40333	Ländcheshalle	{}
44321	44338	MAIN-KINZIG-KREIS	{rule=Locrest1, subtype=location_rest, kind=LOC}	=	44321	44338	MAIN-KINZIG-KREIS	{}
17108	17114	Madrid	{rule=City1, subtype=city, kind=LOC}	!=				
17108	17116	Madrider	{rule=Location_key3, subtype=possessive loc, kind=LOC}	!=				
55235	55244	Magdeburg	{rule=City1, subtype=city, kind=LOC}	!=				
55235	55244	Magdeburg	{rule=Location_key2}	=	55235	55244	Magdeburg	{}
87056	87073	Main-Kinzig-Kreis	{rule=Locrest1, subtype=location_rest, kind=LOC}	=	87056	87073	Main-Kinzig-Kreis	{}
13132	13149	Main-Kinzig-Kreis	{rule=Locrest1, subtype=location_rest, kind=LOC}	=	13132	13149	Main-Kinzig-Kreis	{}
45008	45025	Main-Kinzig-Kreis	{rule=Locrest1, subtype=location_rest, kind=LOC}	=	45008	45025	Main-Kinzig-Kreis	{}
44807	44826	Main-Kinzig-Kreises	{rule=Locrest1, subtype=location_rest, kind=LOC}	=	44807	44826	Main-Kinzig-Kreises	{}
39866	39876	Marktplatz	{rule=Location_key2}	!=				
39840	39850	Marktplatz	{rule=Location_key2}	!=				
39840	39850	Marktplatz	{rule=Locrest1, subtype=location_rest, kind=LOC}	!=				
39866	39876	Marktplatz	{rule=Locrest1, subtype=location_rest, kind=LOC}	!=				
72208	72218	Massenheim	{rule=Location_key2}	!=				

			Recall	Precision	F-Measure
Correct:	185				
Partially Correct:	36	Strict:	0.3439	0.6336	0.4458
Missing:	317	Lenient:	0.4108	0.7568	0.5325
False Positives:	71	Average:	0.3773	0.6952	0.4892

Start | 2 Micro... | Acrobat ... | 3 Inter... | 2 Micro... | Windows... | 2 javaw | Desktop | My Documents | 10:24

Figure 13.1: Part of the AnnotationDiff viewer

All annotations from the key set are compared with the ones from the response set, and those found to have the same start and end offsets are displayed on the same line in the table. Next, Annotation Diff evaluates if the features of each annotation from the response set subsume those features from the key set, as specified by the keyFeatureNamesSet parameter.

To understand this in more detail, see section 3.25, which describes the Annotation Diff parameters.

13.2 The six annotation relations explained

Coextensive

Two annotations are coextensive if they hit the same span of text in a document. Basically, both their start and end offsets are equal.

Overlaps

Two annotations overlap if they share a common span of text.

Compatible

Two annotations are compatible if they are coextensive and if the features of one (usually the ones from the key) are included in the features of the other (usually the response).

Partially Compatible

Two annotations are partially compatible if they overlap and if the features of one (usually the ones from the key) are included in the features of the other (response).

Missing This applies only to the key annotations. A key annotation is missing if either it is not coextensive or overlapping, orif one or more features are not included in the response annotation.

Spurious

This applies only to the response annotations. A response annotation is spurious if either it is not coextensive or overlapping, or if one or more features from the key are not included in the response annotation.

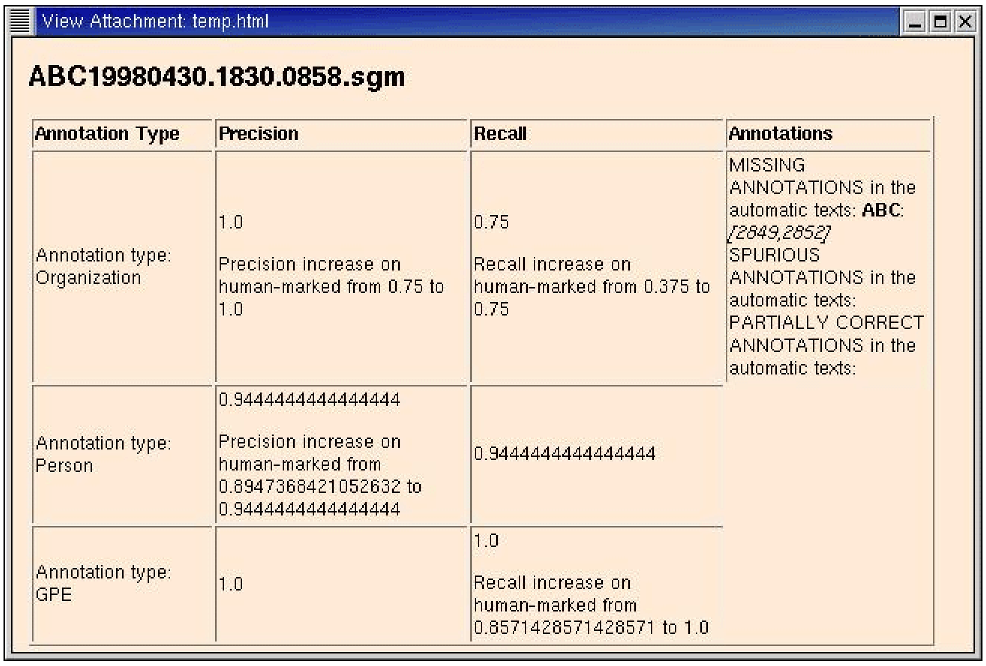
13.3 Benchmarking tool

The benchmarking tool differs from the AnnotationDiff in that it enables evaluation to be carried out over a whole corpus rather than a single document. It also enables tracking of the system's performance over time. The tool can be run in either GUI mode or standalone mode. For more information on how to run the tool, see 3.26.

The tool requires a clean version of a corpus (with no annotations) and an annotated reference corpus. First of all, the tool is run in generation mode to produce a set of texts annotated by the system. These texts are stored for future use. The tool can then be run in three ways:

1. comparing the stored processed set with the human-annotated set;
2. comparing the current processed set with the human-annotated set;
3. (default mode) comparing the stored processed set with the current processed set and the human-annotated set.

In each case, performance statistics will be output for each text in the set, and overall statistics for the entire set. In the default mode, information is also provided about whether the figures have increased or decreased in comparison with the annotated set. The processed set can be updated at any time by rerunning the tool in generation mode with the latest version of the system resources. Furthermore, the system can be run in verbose mode, where for each P and R figure below a certain threshold (set by the user), the non-coextensive annotations (and their corresponding text) will be displayed. The output of the tool is written to an HTML file in tabular form, for easy viewing of the results (see Figure 13.2).



ABC19980430.1830.0858.sgm

Annotation Type	Precision	Recall	Annotations
Annotation type: Organization	1.0 Precision increase on human-marked from 0.75 to 1.0	0.75 Recall increase on human-marked from 0.375 to 0.75	MISSING ANNOTATIONS in the automatic texts: **ABC**: *[2849,2852]* SPURIOUS ANNOTATIONS in the automatic texts: PARTIALLY CORRECT ANNOTATIONS in the automatic texts:
Annotation type: Person	0.9444444444444444 Precision increase on human-marked from 0.8947368421052632 to 0.9444444444444444	0.9444444444444444	
Annotation type: GPE	1.0	1.0 Recall increase on human-marked from 0.8571428571428571 to 1.0	

Figure 13.2: Fragment of results from benchmark tool

13.4 Metrics for Evaluation in Information Extraction

Much of the research in IE in the last decade has been connected with the MUC competitions, and so it is unsurprising that the MUC evaluation metrics of precision, recall and F-measure [Chinchor 92] also tend to be used, along with slight variations. These metrics have a very long-standing tradition in the field of IR [van Rijsbergen 79] (see also [Manning & Schütze 99, Frakes & Baeza-Yates 92]).

Precision measures the number of correctly identified items as a percentage of the number of items identified. In other words, it measures how many of the items that the system identified were actually correct, regardless of whether it also failed to retrieve correct items. The higher the precision, the better the system is at ensuring that what is identified is correct.

Error rate is the inverse of precision, and measures the number of incorrectly identified items as a percentage of the items identified. It is sometimes used as an alternative to precision.

Recall measures the number of correctly identified items as a percentage of the total number of correct items. In other words, it measures how many of the items that should have been identified actually were identified, regardless of how many spurious identifications were made. The higher the recall rate, the better the system is at not missing correct items.

Clearly, there must be a tradeoff between precision and recall, for a system can easily be made to achieve 100% precision by identifying nothing (and so making no mistakes in what it identifies), or 100% recall by identifying everything (and so not missing anything). The **F-measure** [van Rijsbergen 79] is often used in conjunction with Precision and Recall, as a weighted average of the two. **False positives** are a useful metric when dealing with a wide variety of text types, because it is not dependent on *relative document richness* in the same way that precision is. By this we mean the relative number of entities of each type to be found in a set of documents.

When comparing different systems on the same document set, relative document richness is unimportant, because it is equal for all systems. When comparing a single system's performance on different documents, however, it is much more crucial, because if a particular document type has a significantly different number of any type of entity, the results for that entity type can become skewed. Compare the impact on precision of one error where the total number of correct entities = 1, and one error where the total = 100. Assuming the document length is the same, then the false positive score for each text, on the other hand, should be identical.

Common metrics for evaluation of IE systems are defined as follows:

$$Precision = \frac{Correct + 1/2Partial}{Correct + Spurious + 1/2Partial} \tag{13.1}$$

$$Recall = \frac{Correct + 1/2Partial}{Correct + Missing + 1/2Partial} \tag{13.2}$$

$$F - measure = \frac{(\beta^2 + 1)P * R}{(\beta^2 R) + P} \tag{13.3}$$

where β reflects the weighting of P vs. R. If β is set to 1, the two are weighted equally.

$$FalsePositive = \frac{Spurious}{c} \tag{13.4}$$

where c is some constant independent from document richness, e.g. the number of tokens or sentences in the document.

Note that we consider annotations to be partially correct if the entity type is correct and the spans are overlapping but not identical. Partially correct responses are normally allocated a half weight.

13.5 Metrics for Evaluation of Inter-Annotator Agreement

When we evaluate the performance of a processing resource such as tokeniser, POS tagger, or a whole application, we usually have a human-authored "gold standard" against which to compare our software. However, it is not always easy or obvious what this gold standard should be, as different people may have different opinions about what is correct. Typically, we solve this problem by using more than one human annotator, and comparing their annotations. We do this by calculating inter-annotator agreement (IAA), also known as inter-rater reliability.

IAA can be used to assess how difficult a task is. This is based on the argument that if two humans cannot come to agreement on some annotation, it is unlikely that a computer could ever do the same annotation "correctly". Thus, IAA can be used to find the ceiling for computer performance.

There are many possible metrics for reporting IAA, such as Cohen's Kappa, prevalence, and bias [Eugenio & Glass 04]. Kappa is the best metric for IAA when all the annotators have identical exhaustive sets of questions on which they might agree or disagree. This could be a task like "read over this text and mark up all telephone numbers". However, sometimes there is disagreement about the set of questions, e.g. when the annotators themselves determine which text spans they ought to annotate. That could be a task like "read over this text and mark up all references to politics". When annotators determine their own sets of questions, it is appropriate to use precision, recall, and F-measure to report IAA. The following demonstrates best practices for calculating IAA in this way.

Let's assume we have two annotators, Ann1 and Ann2. We want to measure how well Ann1 annotates compared with Ann2, and vice versa. Note that P(Ann1 vs Ann2) == R(Ann2 vs Ann1), and, similarly, P(Ann2 vs Ann1) == R(Ann1 vs Ann2).

This means that we can simply run an Annotation Diff with Ann1 as the key, and Ann2 as the response, and then do the reverse: Ann1 as the response, and Ann2 as the key.

We then report Precision and F-measure from both runs, as well as the average of precision from both runs, i.e., [Prec(Ann1 vs Ann2) + Prec(Ann2 vs Ann1)] / 2. This latter number is the average precision of your annotators.

13.6 A Plugin Computing Inter-Annotator Agreement (IAA)

The IAA plugin computes different IAA measures for different tasks. For named entity annotations, it computes the F-measures, namely Precision, Recall, and F1 from two or more annotation sets. For text classification tasks, it computes the Cohen's kappa and some other IAA measures which are more suitable than the F-measures for the task. In the following subsections we will describe those measures and the output results from the plugin. But first we explain how to load the plugin, and the input to and the parameters of the plugin.

First you need to load the plugin named "iaaPlugin" into GATE using the tool *Manage CREOLE Plugins*, if it is not already loaded. Then you can create a PR for the plugin from the "IAA Computation" in the existing PR list. After that you can put the PR into a *Corpus Pipeline* to use it.

The corpus pipeline needs a corpus containing the documents, each of which should have two or more annotation sets for computing the IAA measures. One requirement of the plugin is that each document has two or more annotation sets, which may be produced by two or more annotators making the annotation for the same type, or may correspond to one gold standard set and one set from system's output respectively. The annotation set produced by one annotator should have the same name in all the documents. And one annotation type in different annotation sets should have the same name too. For example, suppose that we ask three annotators to annotate person names in two documents *Doc1* and *Doc2*. Then the *Doc1* should have three annotation sets, each of which contains the annotations from one annotator, e.g. the annotation sets *Ann1*, *Ann2* and *Ann3*, and each of which contains an annotation type *Per* for the person name annotations. The *Doc2* should have the three annotation sets with the same names and the same annotation types. Then one can compute the IAA measures for the three annotation sets on the two documents by specifying the runtime parameters for the IAA plugin, as explained next.

The IAA plugin has two runtime parameters **annSetsForIaa** and **annTypesAndFeats** for specifying the annotation sets and the annotation types and features, respectively. For the above example, you can set the value of *annSetsForIaa* as "Ann1;Ann2;Ann3" and the value of *annTypesAndFeats* as "Per" to compute the IAA for the three annotation sets on the annotation type *Per*. Note that the names of annotation sets are separated by ";". You can also specify more than one annotation type and separate them by ";" too, and optionally specify one annotation feature for one type by attaching a "->" followed by feature name to the end of the annotation name. For example, "Per->label;Org" specifies two annotation types *Per* and *Org* and also a feature name *label* for the type *Per*. If you specify one annotation feature for one annotation type, then two annotations of the same type from two different annotation sets in the same document will be regarded as being different if they have different values of that feature, even if the two annotations occupy exactly the same position in the document. On the other hand, if you do not specify any annotation feature for one annotation type, then the two annotations of the type will be regarded as the same if they occupy the same position in the document.

The plugin has another parameter **measureType** specifying the type of measure computed. There are two measure types, the *F-measure* (i.e. Precision, Recall and F1), and the *observed agreement and Cohen's Kappa*. For the classification task such as document or sentence classification, the observement agreement and Cohen's Kappa is often used, though the F-measure is applicable too. However, for the named entity recognition task, only the F-measure is applicable. See the following subsections for more discussions. The parameter has two values, *FMEASURE* and *AGREEMENTANDKAPPA*. The default value of the parameter is *FMEASURE*.

Another parameter **verbosity** specifies the verbosity level of the plugin's output. Level 2 displays the most detailed output, including the IAA measures on each document and the macro-averaged results over all documents. Level 1 only displays the IAA measures averaged over all documents. Level 0 does not have any output. The default value of the parameter is 1. In the following we will explain the outputs in detail.

Yet another runtime parameter **bdmScoreFile** specifies the URL for a file containing the BDM scores used for the BDM based IAA computation. The BDM score file should be produced by the BDM computation plugin, which is described in Section 13.7. The BDM-based IAA computation will be explained below. If the parameter is not assigned any value, or is assigned a file which is not a BDM score file, it will not compute the BDM based IAA.

13.6.1 IAA for Classification Task

IAA has been used mainly in the classification tasks, where two or more annotators are given a set of instances and are asked to classify those instances into some pre-defined categories. IAA measures the agreements among the annotators on the class labels assigned to the instances by the annotators. Text classification tasks include document classification, sentence classification(e.g. opinionated sentence recognition), and token classification (e.g. POS tagging). The important property of evaluationg a classification task is that the evaluation set and gold standard set have exactly the same instances but some instances in the two sets have different class labels.

The three commonly used IAA measures are *observed agreement*, *specific agreement*, and

Kappa (κ) [Hripcsak & Heitjan 02]. See the Appendix G for the detailed explanations of those measures. If you select the value of the run time parameter *measureType* as *AGREEMENTANDKAPPA*, the IAA plugin will compute and display those IAA measures for your classification task. In the following we will explain the output of the plugin for the agreement and Kappa measures.

At the verbosity level 2, the output of the plugin is the most detailed. It first prints out a list of the names of the annotation sets used for IAA computation. In the rest part of the results, the first annotation set is denoted as annotator 0, and the second annotation set is denoted as annotator 1, etc. Then the plugin outputs the IAA results for each document in the corpus.

For one document, it displays one annotation type and optionally an annotation feature if specified, and then the results for that type and that feature. Note that the IAA computations are based on the pairwise of annotators. In another word, we compute the IAA for each pair of annotators. The first results for one document and one annotation type are the macro-averaged ones over all pairs of annotators, which have three numbers for the three types of IAA measures, namely *Observed agreement*, *Cohen's kappa* and *Scott's pi*, respectively. Then for each pair of annotators, it outputs the three types of measures, a confusion matrix (or contingency table), and the specific agreements for each label. The labels are obtained from the annotations of that particular type. For one annotation type, if one feature is specified, then the labels are the values of the feature in the annotations. Please note that two specific terms may be added to the label list: one is the empty one obtained from those annotations which have the annotation feature but do not have a value for the feature; another one is the "Non-cat" corresponding to those annotations not having the feature at all. If no feature is specified, then two labels are used: "Anns" corresponding to the annotations of that type, and "Non-cat" corresponding to those annotations which are annotated by one annotator but are not annotated by another annotator.

After displaying the results for each document, the plugin prints out the macro-averaged results over all documents. First for each annotation type, it prints out the results for each pair of annotators, and the macro-averaged results over all pairs of annotators. Finally it prints out the macro-averaged results over all pair of annotators, all types and all documents.

Please note that the classification problem can be evaluated by the F-measure too. If you want to evaluate a classification problem using the F-measure, you just need to select the run time parameter *measureType* with the value *FMEASURE*.

13.6.2 IAA For Named Entity Annotation

The commonly used IAA measures such as Kappa and other statistical measures have not been used in the text mark-up tasks such as named entity recognition and information extraction, due to the reason explained in Section 13.5 (also see [Hripcsak & Rothschild 05]). Instead, the F-measures such as Precision, Recall, and F1 have been widely used in information extraction evaluations such as MUC, ACE and TERN for measuring IAA. This is because the computation of the F-measures does not need the number of non-entity examples. Another reason is that F-measures are commonly used for evaluating information extraction systems. Hence IAA F-measures can be directly compared with system's results.

For computing F-measure between two annotation sets, one can use one annotation set as gold standard and another set as system's output and compute the F-measures such as Precision, Recall and F1. One can switch the roles of the two annotation sets. The Precision and Recall in the former case become Recall and Precision in the latter, respectively. But the F1 remains the same in both cases. For more than two annotators, we first compute F-measures between any two annotators and use the mean of the pair-wise F-measures as an overall measure. The computation of the F-measures (e.g. Precision, Recall and F1) are shown in Section 13.5. As noted in [Hripcsak & Rothschild 05], the F1 computed for two annotators for one specific category is equivalent to the positive specific agreement of the category.

The outputs of the IAA plugins for named entity annotation are similar to those for classification. But the outputs are the F-measures such as Precision, Recall and F1, instead of the agreements and Kappas. It first prints out the results for each document. For one document, it prints out the results for each annotation type, macro-averaged over all pairs of annotators, then the results for each pair of annotators. In the last part, the macro-averaged results over all documents are displayed. Note that the results are reported in both the strict measure and the lenient measure, as defined in Section 13.1.

Please note that, for computing the F-measures for the named entity annotations, the IAA plugin carries out the same computation as the *Benchmarking tool.* The IAA plugin is simpler than the Benchmarking tool in the sense that the former needs only one set of documents with two or more annotation sets, whereas the latter needs three sets of the same documents, one without any annotation, another with one annotation set, and the third one with another annotation set. Additionally, the IAA plugin can deal with more than two annotation sets but the Benchmarking tool can only deal with two annotation sets.

13.6.3 The BDM Based IAA Scores

For a named entity recognition system, if the named entity's class labels are the names of concepts in some ontology (e.g. in the ontology-based information extraction), the system can be evaluated using the IAA measures based on the BDM scores. The BDM measures the closeness of two concepts in an ontology. If an entity is identified but is assigned a label which is close to but not the same as the true label, the system should obtain some credit for it, which the BDM-based metric can do. In contrast, the conventional named entity recognition measure does not take into account the closeness of two labels and does not give any credit to one identified entity with a wrong label, regardless of how close the assigned label is to the true label. For more explanation about BDM see Section 13.7.

In order to compute the BDM-based IAA, one has to assign the plugin's runtime parameter **bdmScoreFile** to the URL of a file containing the BDM scores. The file should be obtained by using the BDM computation plugin, which is described in Section 13.7. Currently the BDM-based IAA is only used for computing the F-measures for e.g. the entity recognition problem. Please note that the F-measures can also be used for evaluation of classification problem. The BDM is not used for computing other measures such as the *observed agreement* and *Kappa*, though it is possible to implement it. Therefore currently one has to select *FMEASURE* for the run time parameter *measureType* in order to use the BDM based IAA computation.

13.7 A Plugin Computing the BDM Scores for an Ontology

The BDM (balanced distance metric) measures the closeness of two concepts in an ontology or taxonomy [Maynard 05, Maynard *et al.* 06]. It is a real number between 0 and 1. The closer the two concepts are in an ontology, the greater their BDM score is. For detailed explanation about the BDM, see the papers [Maynard 05, Maynard *et al.* 06]. The BDM can be seen as an improved version of the learning accuracy [Cimiano *et al.* 03]. It is dependent on the length of the shortest path connecting the two concepts and also the deepness of the two concepts in ontology. It is also normalised with the size of ontology and also takes into account the concept density of the area containing the two involved concepts.

The BDM has been used to evaluate the ontology based information extraction (OBIE) system [Maynard *et al.* 06]. The OBIE identifies the instances for the concepts of an ontology. It's possible that an OBIE system identifies an instance successfully but does not assign it the correct concept. Instead it assigns the instance a concept being close to the correct one. For example, the entity "London" is an instance of the concept *Capital*, and an OBIE system assigns it the concept *City* which is close to the concept *Capital* in some ontology. In that case the OBIE should obtain some credit according to the closeness of the two concepts. That is where the BDM can be used. The BDM has also been used to evaluate the hierarchical classification system [Li *et al.* 07a]. It can also be used for ontology learning and alignment.

The BDM computation plugin computes BDM score for each pair of concepts in an ontology. It has two run time parameters:

- **ontologyURL** – its value should be the URL of the ontology that one wants to compute the BDM scores for.
- **outputBDMFile** – its value is the URL of a file which will store the BDM scores computed.

The plugin has the name *bdmComputation* and the corresponding processing resource's name is *BDM Computation PR*. The PR can be put into a Pipeline. If it is put into a Corpus Pipeline, the corpus used should contain at least one document.

The BDM computation used the formula given in [Maynard *et al.* 06]. The resulting file specified by the runtime parameter *outputBDMFile* contains the BDM scores. It is a text file. The first line of the file gives some meta information such as the name of ontology used for BDM computation. From the second line of the file, each line corresponds to one pair of concepts. One line is like

key=Service, response=Object, bdm=0.6617647, msca=Object, cp=1, dpk=1, dpr=0, n0=2.0, n1=2.0, n2=2.8333333, bran=1.9565217

It first shows the names of the two concepts (one as *key* and another as *response*, and the BDM score, and then other parameters' values used for the computation. Note that, since the BDM is symmetric for the two concepts, the resulting file contains only one line for each pair. So if you want to look for the BDM score for one pair of concepts, you can choose one as key and another as response. If you cannot find the line for the pair, you have to change the order of two concepts and retrieve the file again.

Chapter 14

Users, Groups, and LR Access Rights

"Well," he said, "it's to do with the project which first made the software incarnation of the company profitable. It was called Reason, and in its own way it was sensational."

"What was it?"

"Well, it was a kind of back-to-front program. It's funny how many of the best ideas are just an old idea back-to-front. You see there have already been several programs written that help you to arrive at decisions by properly ordering and analysing all the relevant facts so that they then point naturally towards the right decision. The drawback with these is that the decision which all the properly ordered and analysed facts point to is not necessarily the one you want."

"Yeeeess ..." said Reg's voice from the kitchen.

"Well, Gordon's great insight was to design a program which allowed you to specify in advance what decision you wished it to reach, and only then to give it all the facts. The program's task, which it was able to accomplish with consummate ease, was simply to construct a plausible series of logical-sounding steps to connect the premises with the conclusion.

"And I have to say that it worked brilliantly. Gordon was able to buy himself a Porsche almost immediately despite being completely broke and a hopeless driver. Even his bank manager was unable to find fault with his reasoning. Even when Gordon wrote it off three weeks later."

"Heavens. And did the program sell very well?"

"No. We never sold a single copy."

"You astonish me. It sounds like a real winner to me."

"It was," said Richard hesitantly. "The entire project was bought up, lock, stock and barrel, by the Pentagon. The deal put WayForward on a very sound financial foundation. Its moral foundation, on the other hand, is not something I would want to trust my weight to. I've recently been analysing a lot of the arguments put forward in favour of the Star Wars project, and if you know what you're looking for, the pattern of the algorithms is very clear.

"So much so, in fact, that looking at Pentagon policies over the last couple of years I think I can be fairly sure that the US Navy is using version 2.00 of the program, while the Air Force for some reason only has the beta-test version of 1.5. Odd, that."

Dirk Gently's Holistic Detective Agency, Douglas Adams, 1987 (pp. 55-56).

This chapter describes the LR access mechanism which is implemented for persistent LRs. At present there are two LR persistency storage methods: Java serialisation and Oracle. Here we will describe their security features in turn.

14.1 Java serialisation and LR access rights

At present the security model is not implemented for Java serialization. One should rely on the security control offered by the OS in order to restrict access to certain persistent resources.

14.2 Oracle Datastore and LR access rights

Warning: These features will not work, unless you have an Oracle pre-installed at your site[1] and you, or an administrator at your site, has installed the GATE Oracle support (see http://gate.ac.uk/gate/doc/persistence.pdf).

Oracle datastores have advanced LR access rights based on users and groups, which are similar to those in an operating system such as Linux.

In order to be able to access an LR stored in an Oracle datastore, a user needs to supply a user name, password and a group. These credentials are used to determine which LRs are accessible to this user for reading and writing.

14.2.1 Users, Groups, Sessions and Access Modes

The security model provides primitives such as users, groups, permissions and sessions similar to the ones provided by the operating systems:

[1]Oracle installation is not provided with GATE. You need to purchase this product separately from Oracle Corp. (see http://www.oracle.com).

- **users** - they are identified by login name and password (each limited to 16 symbols). A user may be member of one or more groups.
- **groups** - identified by name (up to 128 symbols).
- **session** - each user must log into the datastore (by providing name, password and group) in order to use its resources. A session is opened when the user logs in. The default inactivity period after which the session expires and the user should log into the datastore again is 4 hours.
- **access modes** - there are four access modes in the present implementation. The access (Read/Write) to a resource according to its owner and access mode is shown in Table 14.1.

Mode	Owner (R/W)	Owner's group (R/W)	Other users (R/W)
World Read/ Group Write	+/+	+/+	+/-
Group Read/ Group Write	+/+	+/+	-/-
Group Read/ Owner Write	+/+	+/-	-/-
Owner Read/ Owner Write	+/+	-/-	-/-

Table 14.1: Access Modes

When GATE is configured for use with Oracle, a superuser and group are created:

- super user - ADMIN, password 'sesame'.
- administrative group - ADMINS.

The superuser is similar to the root user in Unix and has access to any resource despite its access mode.This user can also create or remove other users We recommend that you change the password of the superuser immediately after you have installed the Oracle support for GATE.

14.2.2 User/Group Administration

Running the administration tool

When GATE Oracle tables are first created with the database install scripts, they only contain the `ADMIN` user which is the only user who can create and modify users and groups.[2] We do not recommend using the `ADMIN` user to store/access LRs in GATE.

[2]This user is similar to the `root` user in Unix operating systems.

Instead, immediately after installing Oracle support for GATE datastores, some users and groups must be created by running the `UserGroupEditor` tool. Before running this tool, the URL to the Oracle database needs to be specified in gate.xml (either the user's own or the site-wide gate.xml). An example entry is:

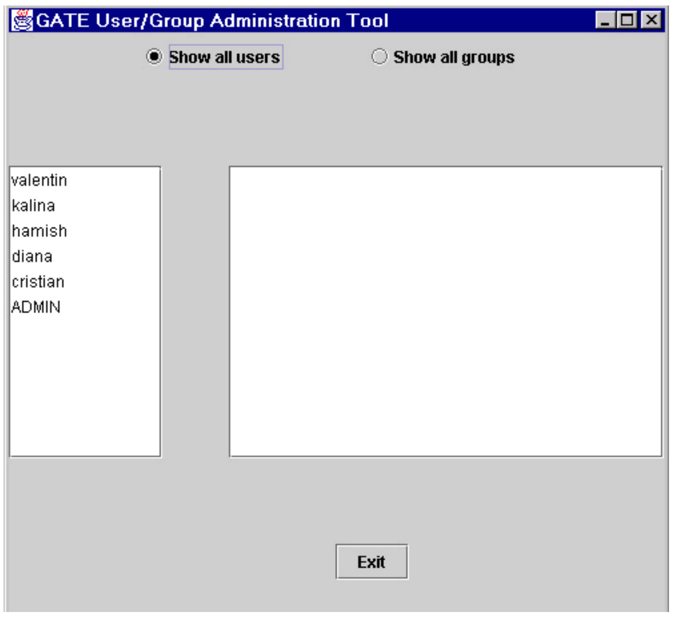
¡DBCONFIG url=”jdbc:oracle:thin:GATEUSER/gate@example.dcs.shef.ac.uk:1521:gate101”
url1=”jdbc:oracle:thin:GATEUSER/gate@testdb.dcs.shef.ac.uk:1521:gate02” /¿

The example entry shows that there are two databases configured for this site, one at each URL. There is no limit to the number of Oracle databases one can have, but they all need to have an attribute starting with ”url”, e.g., url1, url2.

To run the tool, call the gate script with the `-a` parameter.

When the tool starts up, it first asks you to select which Oracle database you wish to administer. All databases defined in the ¡DBCONFIG¿ section of gate.xml will be shown in a listbox. Once the database is chosen, a login dialog is shown, asking for the user name, password and group of the `ADMIN` user. The initial password of the `ADMIN` user is `sesame` and the group is `ADMINS`. We advise that these are changed, the first time this tool is run.

If all login credentials are provided correctly, the graphical tool starts up:



valentin
kalina
hamish
diana
cristian
ADMIN

Figure 14.1: The User/Group Administration Tool

Viewing user and group information

As shown in Figure 14.1, the user/group administration tool (called the UG tool for the rest of this section) consist of two parallel lists. By default, the left one shows a list of all users in the database and the right one is empty.

To view the groups to which a particular user belongs, you need to select that user in the list. Then the right list displays this user's groups. If the list remains empty, then it means that this user does not belong to any group.

In order to view all groups which are available, you need to switch the tool to a `Users for groups` mode, by clicking on the corresponding radio button. This will switch the tool to showing the list of all groups in the left panel. When you select a given group, then the right panel shows all users who belong to that group (see Figure 14.2).

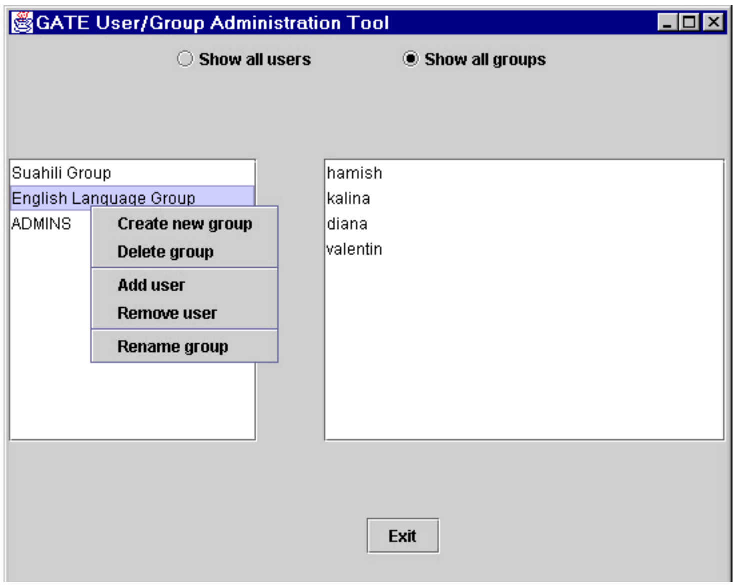


Figure 14.2: The tool in a group administration mode

User manipulation

Users are manipulated by selecting a user in the list of users and right-clicking on it to see the user manipulation menu. This menu allows the following actions:

Create new user: shows a dialog where the user name and password of the new user must be specified.

Delete user: delete the currently selected user.

Add to group: shows a dialog displaying all available groups. Select one to add the user to it.

Remove from group: in the given dialog, choose the group from which the user is to be removed.

Change password: shows a dialog where the new password can be specified;

Rename user: choose another name for the selected user.

All changes are automatically written to the Oracle database.

Group manipulation

Groups are manipulated by selecting a group in the list of groups and right-clicking on it to see the group manipulation menu. This menu allows the following actions:

Create new group: shows a dialog where the name of the new group must be specified.

Delete group: delete the currently selected group.

Add user: shows a dialog displaying all available users. Select one to add to the group.

Remove user: in the given dialog, choose the user to be removed.

Rename group: choose another name for the selected group.

All changes are automatically written to the Oracle database.

14.2.3 The API

In order to work with users and groups[3] programmatically, you need to use an access controller, which is the class that provides the connection to the Oracle database. The access controller needs to be closed before application exit.

Once the connection is established, you need to create a session by proving the login details of the user (user name, password and group). Any user who can login, can use the accessor methods for users/groups, but only the `ADMIN` user has priviliges to modify the data. The way to check whether the logged in user has the right to modify data, is to use the `isPriviligedSession()` method (see below). If a mutator method is used with a non-priviliged session, a `SecurityException` is thrown. All security-related classes and all their methods are documented in the GATE JavaDoc documentation, `java.security` package.

[3]See the latest API documentation online at: http://gate.ac.uk/gate/doc/javadoc/index.html. User and group API is located in the `gate.security` package.

```
AccessController ac = new AccessControllerImpl();
ac.open("jdbc:oracle:thin:GATEUSER/gate@machine.ac.uk:1521:GateDB");

Session mySession = null;
try {
  mySession = ac.login("myUser", "myPass",ac.findGroup("myGroup").getID());
} catch (gate.security.SecurityException ex) {
  ac.close();
  <print some error and exit>
}

//first check whether we have a valid session
if (! ac.isValidSession(mySession)){
  ac.close();
  <print some error and exit>
}

//then check that it is an administrative session
if (!mySession.isPrivilegedSession()) {
  ac.close();
  <print some error and exit>
}

User myUser = ac.findUser("myUser");
String myName = myUser.getName()
List myGroups = myUser.getGroups();
...
<more code to access/modify groups and users here>

//we're done now, just close the access controller connection
ac.close();
```

If you'd like to use a dialog, where the user can type those details, the session can be obtained by using the `login(AccessController ac, Component parent)` static method in the `UserGroupEditor` class. The login code would then look as follows:

```
mySession = UserGroupDialog.login(ac, someParentWindow);
```

For a full example of code using the security API, see `TestSecurity.java` and `UserGroupEditor.java`.

Chapter 15

Developing GATE

This chapter describes the protocols to follow and other information for those involved in developing GATE.

15.1 Creating new plugins

GATE provides a flexible structure where new resources can be plugged in very easily. There are three types of resources: Language Resource (LR), Processing Resource (PR) and Visual Resource (VR). In the following subsections we describe the necessary steps to write new PRs and VRs, and to add plugins to the nightly build. The guide on writing new LRs will be available soon.

15.1.1 Where to keep plugins in the GATE hierarchy

Each new resource added as a plugin should contain its own subfolder under the %GATE-HOME%/plugins folder. A plugin can have one or more resources declared in its creole.xml file. Creole.xml specifies one or more resources and required parameters for each such resources. The file should reside under the subfolder created for the plugin. More information on creole.xml and how to declare parameters and attributes is explained later.

15.1.2 Writing a new PR

Class Definition

Below we show a template class definition, which can be used in order to write a new Processing Resource.

```
package example;

/**
 * Processing Resource
*/
public class NewPlugin extends AbstractProcessingResource implements
ProcessingResource {

     /*
      * this method gets called whenever an object of this
      * class is created either from GATE GUI or if
      * initiated using Factory.createResource() method.
      */
     public Resource init() throws ResourceInstantiationException {
          // here initialize all required variables, and may
          // be throw an exception if the value for any of the
          // mandatory parameters is not provided

          if(this.rulesURL == null)
              throw new ResourceInstantiationException("rules URL is null");

          return this;
     }


     /*
      * this method should provide the actual functionality of the PR
      * (from where the main execution begins). This method
      * gets called when user click on the RUN button in the
      * GATE GUIs application window.
      */
     public void execute() throws ExecutionException {
          // write code here
     }

     /* this method is called to reinitialize the resource */
```

```
    public void reInit() throws ResourceInstantiationException {
        // reinitialization code
    }

    /*
     * There are two types of parameters
     * 1. Init time parameters  values for these parameters need to be
     * provided at the time of initializing a new resource and these values are
     * not supposed to be changed.
     * 2. Runtime parameters - values for these parameters are provided at the time
     * of executing the PR. These are runtime parameters and can be
     * changed before starting the execution
     * (i.e. before you click on the "RUN" button in the GATE GUI)
     * It is must to provide setter and getter methods for every such
     * parameter declared in the creole.xml.
     *
     * for example to set a value for outputAnnotationSetName
     */
    String outputAnnotationSetName;

    //getter and setter methods

    /* get<parameter name with first letter Capital>  */
    public String getOutputAnnotationSetName() {
         return outputAnnotationSetName;
    }

    public void setOuputAnnotationSetName(String setName) {
         this.outputAnnotationSetName = setName;
    }

    /** Init-time parameter */
    URL rulesURL;

    // getter and setter methods
    public URL getRulesURL() {
       return rulesFile;
    }

    public void setRulesURL(URL rulesURL) {
       this.rulesURL = rulesURL;
    }
}
```

PR Creole Entry

When writing a new resource entry in the creole.xml file, the user should provide details of the class that implements the new PR and its runtime and inittime parameters. The user can also specify other things such as the icon to be used in the resource tree, along with the resource name and the jar it belongs to.

```
<?xml version="1.0"?>
<CREOLE-DIRECTORY>
<CREOLE>
   <RESOURCE>
      <!-- Name of the PR that appears in GATE PR List -->
      <NAME>An Example Plugin</NAME>

      <!-- Jar where to look for the resource -->
       <JAR>newplugin.jar</JAR>

      <!-- Underlying class that implements the New Plugin -->
      <CLASS>example.NewPlugin</CLASS>

      <!-- Comment that appears when mouse hovers over the PR Name -->
      <COMMENT>An example plugin that demonstrates how to write a new
                          PR</COMMENT>

      <!-- Declaring various parameters-->
      <!-- PR need a document, which should be a runtime parameter -->
      <!-- Unless specified pa[sec:misc-creole:miniparrameters are manadatory -->
      <PARAMETER NAME="document"
         COMMENT="The document to be processed"
         RUNTIME="true">gate.Document</PARAMETER>

      <PARAMETER NAME="rulesURL"
         COMMENT="example of an inittime parameter"
         DEFAULT="resources/morph/default.rul" RUNTIME="false">
         java.net.URL</PARAMETER>

      <PARAMETER NAME="outputAnnotationSetName"
         COMMENT="name of the annotationSet used for output"
         RUNTIME="true"
         OPTIONAL="true">java.lang.String</PARAMETER>
   </RESOURCE>
</CREOLE>
</CREOLE-DIRECTORY>
```

Option Menu

Each resource (LR,PR) has some predefined actions associated with it. These actions appear in an options menu that appears in the GATE GUI when the user right clicks on any of the resources. For example if the selected resource is a Processing Resource, there will be at least four actions available in its options menu: 1. Close 2. Hide this view 3. Rename and 4. Reinitialize. New actions in addition to the predefined actions can be added by implementing the *gate.gui.ActionsPublisher* interface. Then the user has to implement the following method.

```
public List getActions() {
     return actions;
}
```

Here the variable *actions* should contain a list of instances of type *javax.swing.AbstractAction*. A string passed in the constructor of an AbstractAction object appears in the Options Menu. Adding a *null* element adds a separator in the menu.

Listeners

There are at least four important listeners which should be implemented in order to listen to the various relevant events happening in the background. These include:

- CreoleListener

  Creole-register keeps information about instances of various resources and refreshes itself on new additions and deletions. In order to listen to these events, a class should implement the *gate.event.CreoleListener*. Implenting CreoleListener requires users to implement the following methods:

  – public void resourceLoaded(CreoleEvent creoleEvent);
  – public void resourceUnloaded(CreoleEvent creoleEvent);
  – public void resourceRenamed(Resource resource, String oldName, String newName);
  – public void datastoreOpened(CreoleEvent creoleEvent);
  – public void datastoreCreated(CreoleEvent creoleEvent);
  – public void datastoreClosed(CreoleEvent creoleEvent);

- DocumentListener

  A traditional GATE document contains text and a set of annotationSets. To get notified about changes in any of these resources, a class should implement the *gate.event.DocumentListener*. This requires users to implement the following methods:

  – public void contentEdited(DocumentEvent event);

  – public void annotationSetAdded(DocumentEvent event);

  – public void annotationSetRemoved(DocumentEvent event);

- AnnotationSetListener

  As the name suggests, Annewplugin.texnotationSet is a set of annotations. To listen to the addition and deletion of annotations, a class should implement the *gate.event.AnnotationSetListener* and therefore the following methods:

  – public void annotationAdded(AnnotationSetEvent event);

  – public void annotationRemoved(AnnotationSetEvent event);

- AnnotationListener

  Each annotation has a featureMap associated with it, which contains a set of feature names and their respective values. To listen to the changes in annotation, one needs to implement the *gate.event.AnnotationListener* and implement the following method:

  – public void annotationUpdated(AnnotationEvent event);

15.1.3 Writing a new VR

Each resource (PR and LR) can have its own associated visual resource. When double clicked, the resource's respective visual resource appears in the GATE GUI. The GATE GUI is divided into three visible parts (See Figure 15.1). One of them contains a tree that shows the loaded instances of resources. The one below this is used for various purposes - such as to display document features and that the execution is in progress. This part of the GUI is referred to as "small". The third and the largest part of the GUI is referred to as "large". One can specify which one of these two should be used for displaying a new visual resource in the creole.xml.

Class Definition

Below we show a template class definition, which can be used in order to write a new Visual Resource.

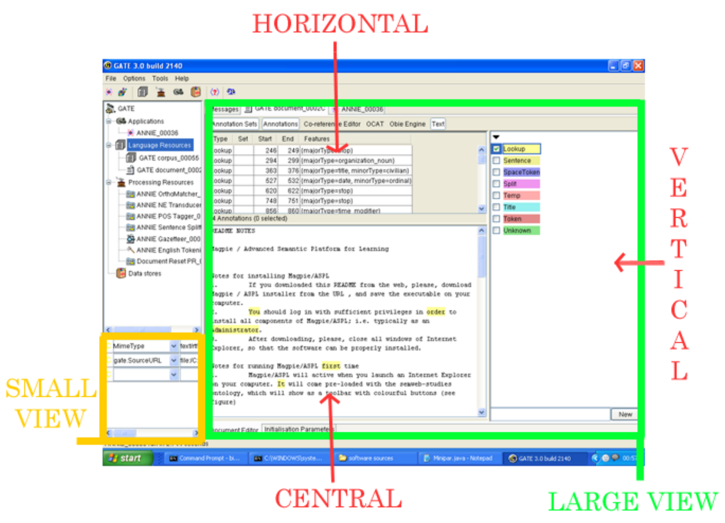


Figure 15.1: GATE GUI

```
package example.gui;

/*
* An example Visual Resource for the New Plugin
* Note that here we extends the AbstractVisualResource class
*/
public class NewPluginVR extends AbstractVisualResource {

               /*
                * An Init method called when the GUI is initialized for the first ti
                */
               public Resource init() {
                   // initialize GUI Components
                   return this;
               }

               /*
                * Here target is the PR class to which this Visual Resource Belongs
                * this method is called after the init() method
                */
               public void setTarget(Object target) {
                      // check if the target is an instance of what you expected
                      // and initialize local data structures if required
               }
}
```

Every document has its own document viewer associated with it. It comes with a single component that shows the text of the original document. GATE provides a way to attach new GUI plugins to the document viewer. For example AnnotationSet viewer, AnnotationList viewer and Co-Reference editor. These are the examples of DocumentViewer plug-ins shipped as part of the core GATE build. These plugins can be displayed either on the right or on top of the document viewer. They can also replace the text viewer in the center (See figure 15.1). A separate button is added at the top of the document viewer which can be pressed to display the GUI plug-in.

Below we show a template class definition, which can be used to develop a new DocumentViewer plugin.

```
/*
 * Note that the class needs to extends the AbstractDocumentView class
 */
public class DocumentViewerPlugin extends AbstractDocumentView {

      /* Implementers should override this method and use it for populating the GUI.
             public void initGUI() {
                // write code to initialize GUI
             }

             /* Returns the type of this view */
              public int getType() {
                    // it can be any of the following constants
                    // from the gate.gui.docview.DocumentView
                    // CENTRAL, VERTICAL, HORIZONTAL
              }

              /* Returns the actual UI component this view represents. */
              public Component getGUI() {
                     // return the top level GUI component
              }

              /* This method will be called whenever the view becomes active.*/
              public void registerHooks() {
                       // register listeners
              }

               /* This method will be called whenever this view becomes inactive. */
               public void unregisterHooks() {
                       // do nothing
               }
}
```

VR Creole Entry

As mentioned earlier, a VR needs to be associated with some PR or LR and therefore the creole entry for Visual Resource should specify the visual resource it belongs to. Below we extend the creole.xml explained in the previous section by adding an entry for the NewPluginVR.

```
<?xml version="1.0"?>
<CREOLE-DIRECTORY>
<CREOLE>
   <RESOURCE>
      <!-- Name of the PR that appears in GATE PR List -->
      <NAME>An Example Plugin</NAME>

      <!-- Jar where to look for the resource -->
       <JAR>newplugin.jar</JAR>

      <!-- Underlying class that implements the New Plugin -->
      <CLASS>example.NewPlugin</CLASS>

      <!-- Comment that appears when mouse hovers over the PR Name -->
      <COMMENT>An example plugin that demonstrates how to write a
                          new PR</COMMENT>

      <!-- Declaring various parameters-->
      <!-- PR need a document, which should be a runtime parameter -->
      <!-- Unless specified parameters are manadatory -->
      <PARAMETER NAME="document"
         COMMENT="The document to be processed"
         RUNTIME="true">gate.Document</PARAMETER>

      <PARAMETER NAME="rulesURL"
         COMMENT="example of an inittime parameter"
         DEFAULT="resources/morph/default.rul" RUNTIME="false">
         java.net.URL</PARAMETER>

      <PARAMETER NAME="outputAnnotationSetName"
         COMMENT="name of the annotationSet used for output"
         RUNTIME="true"
         OPTIONAL="true">java.lang.String</PARAMETER>
   </RESOURCE>
   <!-- New PluginVR entry -->
   <RESOURCE>
       <NAME>Visual Resource for New Plugin</NAME>

      <!-- Jar where to look for the resource -->
       <JAR>newplugin.jar</JAR>

      <!-- Class that implements the VR -->
      <CLASS>example.gui.NewPluginVR</CLASS>

      <!--  type values can be "large" or "small" -->
      <GUI TYPE="large">
                <MAIN_VIEWER />
               <!-- Target it belongs to (i.e. Name of the class
               this new VR associated with -->
               <RESOURCE_DISPLAYED>example.NewPlugin
       </RESOURCE_DISPLAYED>
       </GUI>
    </RESOURCE>
</CREOLE>
</CREOLE-DIRECTORY>
```

15.1.4 Adding plugins to the nightly build

As of November 2005, GATE plugins are now built every night as part of the nightly build process.

If you add a new plugin and want it to be part of the build process, you should create a build.xml file with targets ”build”, ”test”, ”distro.prepare” and ”clean”. The build target should build the JAR file, test should run any unit tests, distro.prepare should clean up any intermediate files (e.g. the classes/ directory) and leave just what's in Subversion, plus the compiled JAR file. The clean target should clean up everything, including the compiled JAR and any generated sources, etc. You should also add your plugin to ”plugins.to.build” in the top-level build.xml to include it in the build. This is by design - not all the plugins have build files, and of the ones that do, not all are suitable for inclusion in the nightly build (viz. SUPPLE, section 9.12).

Note that if you are currently building gate by doing ”ant jar”, be aware that this does not build the plugins. Running just ”ant” or ”ant all” will do so.

There are some changes you will notice as a result of all this:

1. You may suddenly find some plugins stop working when you update to the latest svn

revision, as their JAR files have been removed. Solution: update the top-level GATE build.xml file and then run "bin/ant plugins.build" in the GATE directory to rebuild the missing JARs.

2. If you have your own modified version of any of the affected plugins you will get a conflict for the JAR file when you update, saying something like "move away MiniparWrapper.jar, it is in the way". Solution: rename the offending JAR file, then svn update again and finally rename it back.

15.2 Updating this User Guide

The GATE User Guide is maintained in the GATE subversion repository at SourceForge. If you are a developer at Sheffield you do not need to check out the userguide explicitly, as it will appear under the `tao` directory when you check out `sale`. For others, you can check it out as follows:

```
svn checkout https://svn.sourceforge.net/svnroot/gate/userguide/trunk userguide
```

The user guide is written in LaTeX and translated to PDF using `pdflatex` and to HTML using `tex4ht`. The main file that ties it all together is `tao_main.tex`, which defines the various macros used in the rest of the guide and `\input`s the other `.tex` files, one per chapter.

15.2.1 Building the User Guide

You will need:

- A standard POSIX shell environment including GNU Make. On Windows this generally means Cygwin, on Mac OS X the XCode developer tools and on Unix the relevant packages from your distribution.
- A copy of the userguide sources (see above).
- A LaTeX installation, including pdflatex if you want to build the PDF version, and tex4ht if you want to build the HTML. MiKTeX should work for Windows, TeTeX (available in fink) for Mac OS X, or your choice of package for Unix.
- The BibTeX database `big.bib`. It must be located in the directory **above** where you have checked out the userguide, i.e. if the guide sources are in `/home/bob/svn/userguide` then `big.bib` needs to go in `/home/bib/svn`. Sheffield developers will find that it is already in the right place, under `sale`, others will need to download it from http://gate.ac.uk/sale/big.bib.
- A bit of luck.

Once these are all assembled it *should* be a case of running `make` to perform the actual build. To build just the PDF do `make tao.pdf`, for just the HTML do `make index.html`.

The PDF build generally works without problems, but the HTML build is known to hang on some machines for no apparent reason. If this happens to you try again on a different machine.

15.2.2 Making changes to the User Guide

To make changes to the guide simply edit the relevant `.tex` files, make sure the guide still builds (at least the PDF version), and check in your changes **to the source files only**. Please do not check in your own built copy of the guide, the official user guide builds are produced by a nightly cron job in Sheffield running on a known-good system.

If you add a section or subsection you should use the `\sect` or `\subsect` commands rather than the normal LaTeX `\section` or `\subsection`. These shorthand commands take an optional first parameter, which is the label to use for the section and should follow the pattern of existing labels. The label is also set as an anchor in the HTML version of the guide. For example a new section for the "Fish" plugin would go in `misc-creole.tex` with a heading of:

```
\sect[sec:misc-creole:fish]{The Fish Plugin}
```

and would have the persistent URL `http://gate.ac.uk/cgi-bin/userguide/sec:misc-creole:fis`

If your changes are to document a bug fix or a new (or removed) feature then you should also add an entry to the change log in `changes.tex`. You should include a reference to the full documentation for your change, in the same way as the existing changelog entries do. You should find yourself adding to the changelog every time except where you are just tidying up or rewording existing documentation.

Finally, you'll have to wait until the next morning (UK time) for your changes to be reflected in the online version of the guide.

Chapter 16

Combining GATE and UIMA

UIMA (Unstructured Information Management Architecture) is a platform for natural language processing developed by IBM. It has many similarities to the GATE architecture – it represents documents as text plus annotations, and allows users to define pipelines of *analysis engines* that manipulate the document (or *Common Analysis Structure* in UIMA terminology) in much the same way as processing resources do in GATE. IBM has released an implementation of the UIMA architecture, called the UIMA SDK, that provides support for building analysis components in Java and C++ and running them either locally on one machine, or deploying them as services that can be accessed remotely. The SDK is available for download from http://alphaworks.ibm.com/tech/uima/.

Clearly, it would be useful to be able to include UIMA components in GATE applications and vice-versa, letting GATE users take advantage of UIMA's flexible deployment options and UIMA users access JAPE and the many useful plugins already available in GATE. This chapter describes the interoperability layer provided as part of GATE to support this. The UIMA-GATE interoperability layer is based on the UIMA SDK version 1.2.3. Later versions of UIMA may also work but have not been tested.

The rest of this chapter assumes that you have at least a basic understanding of core UIMA concepts, such as *type systems*, *primitive* and *aggregate text analysis engines* (TAEs), *feature structures*, the format of AE XML descriptors, etc. It will probably be helpful to refer to the relevant sections of the UIMA SDK User's Guide and Reference (supplied with the SDK) alongside this document.

There are two main parts to the interoperability layer:

1. A wrapper to allow a UIMA Text Analysis Engine (TAE), whether primitive or aggregate, to be used within GATE as a Processing Resource (PR).
2. A wrapper to allow a GATE processing pipeline (specifically a `CorpusController`) to be used within UIMA as a TAE.

The two components operate in very similar ways. Given a document in the source form (either a GATE `Document` or a UIMA `CAS`), a document in the target form is created with a copy of the source document's text. Some of the annotations from the source are transferred to the target, according to a mapping defined by the user, and the target component is then run. Finally, some of the annotations on the updated target document are then transferred back to the source, according to the user-defined mapping.

The rest of this document describes this process in more detail. Section 16.1 describes the GATE TAE wrapper, and section 16.2 describes the UIMA CorpusController wrapper.

16.1 Embedding a UIMA TAE in GATE

Embedding a UIMA text analysis engine in a GATE application is a two step process. First, you must construct a *mapping descriptor* XML file to define how to map annotations between the UIMA CAS and the GATE Document. This mapping file, along with the analysis engine descriptor, is used to instantiate an *AnalysisEnginePR* which calls the analysis engine on an appropriately initialized CAS. Examples of all the XML files discussed in this section are available in `examples/conf` under the `uima` plugin directory.

16.1.1 Mapping File Format

Figure 16.1 shows the structure of a mapping descriptor. The `inputs` section defines how annotations on the GATE document are transferred to the UIMA CAS. The `outputs` section defines how annotations which have been added, updated and removed by the TAE are transferred back to the GATE document.

Input definitions

Each input definition takes the following form:

```
<uimaAnnotation type="uima.Type" gateType="GATEType" indexed="true|false">
  <feature name="..." kind="string|int|float|fs">
    <!-- element defining the feature value goes here -->
  </feature>
  ...
</uimaAnnotation>
```

When a document is processed, this will create one UIMA annotation of type `uima.Type` in the CAS for each GATE annotation of type `GATEType` in the input annotation set, covering

```
<uimaGateMapping>
  <inputs>
    <uimaAnnotation type="..." gateType="..." indexed="true|false">
      <feature name="..." kind="string|int|float|fs">
        <!-- element defining the feature value goes here -->
      </feature>
      ...
    </uimaAnnotation>
  </inputs>

  <outputs>
    <added>
      <gateAnnotation type="..." uimaType="...">
        <feature name="...">
          <!-- element defining the feature value goes here -->
        </feature>
        ...
      </gateAnnotation>
    </added>

    <updated>
      ...
    </updated>

    <removed>
      ...
    </removed>
  </outputs>
</uimaGateMapping>
```

Figure 16.1: Structure of a mapping descriptor for a TAE in GATE

the same offsets in the text. If `indexed` is `true`, GATE will keep a record of which GATE annotation gave rise to which UIMA annotation. If you wish to be able to track updates to this annotation's features and transfer the updated values back into GATE, you must specify `indexed="true"`. The `indexed` attribute defaults to `false` if omitted.

Each contained `feature` element will cause the corresponding feature to be set on the generated annotation. UIMA features can be string, integer or float valued, or can be a reference to another feature structure, and this must be specified in the `kind` attribute. The feature's value is specified using a nested element, but exactly how this value is handled is determined by the `kind`.

There are various options for setting feature values:

- `<string value="fixed string" />` The simplest case - a fixed Java String.
- `<docFeatureValue name="featureName" />` The value of the given named feature of the current GATE document.
- `<gateAnnotFeatureValue name="featureName" />` The value of a given feature on the current GATE annotation (i.e. the one on which the offsets of the UIMA annotation are based).
- `<featureStructure type="uima.fs.Type">...</featureStructure>` A feature structure of the given type. The `featureStructure` element can itself contain `feature` elements recursively.

The value is assigned to the feature according to the feature's `kind`:

string The value object's `toString()` method is called, and the resulting String is set as the string value of the feature.

int If the value object is a subclass of `java.lang.Number`, its `intValue()` method is called, and the result is set as the integer value of the feature. If the value object is not a `Number`, it is `toString()`ed, and the resulting String is parsed using `Integer.parseInt()`. If this succeeds, the integer result is used, if it fails the feature is set to zero.

float As for `int`, except that `Number`s are converted by calling `floatValue()`, and non-`Number`s are parsed using `Float.parseFloat()`.

fs The value object is assumed to be a `FeatureStructure`, and is used as-is. A `ClassCastException` will result if the value object is not a `FeatureStructure`.

In particular, `<featureStructure>` value elements should only be used with features of kind `fs`. While nothing will stop you using them with `string` features, the result will probably not be what you expected.

Output definitions

The output definitions take a similar form. There are three groups:

added Annotations which have been added by the TAE, and for which corresponding new annotations are to be created in the GATE document.

updated Annotations that were created by an input definition (with `indexed="true"`) whose feature values have been modified by the TAE, and these values are to be transferred back to the original GATE annotations.

removed Annotations that were created by an input definition (with `indexed="true"`) which have been removed from the CAS[1] and whose source annotations are to be removed from the GATE document.

The definition elements for these three types all take the same form:

```
<gateAnnotation type="GATEType" uimaType="uima.Type">
  <feature name="featureName">
    <!-- element defining the feature value goes here -->
  </feature>
  ...
</gateAnnotation>
```

For `added` annotations, this has the mirror-image effect to the input definition – for each UIMA annotation of the given type, create a GATE annotation at the same offsets and set its feature values as specified by `feature` elements. For a `gateAnnotation` the `feature` elements do not have a `kind`, as features in GATE can have arbitrary Objects as values. The possible feature value elements for a `gateAnnotation` are:

- `<string value="fixed string" />` A fixed string, as before.
- `<uimaFSFeatureValue name="uima.Type:FeatureName" kind="string|int|float" />` The value of the given feature of the current UIMA annotation. The feature name must be specified in fully-qualified form, including the type on which it is defined. The `kind` is used in a similar way as in input definitions:

  string The Java `String` object returned as the string value of the feature is used.

  int An `Integer` object is created from the integer value of the feature.

  float A `Float` object is created from the float value of the feature.

[1]Strictly speaking, removed from the annotation index, as feature structures cannot be removed from the CAS entirely.

fs The UIMA `FeatureStructure` object is returned. Since `FeatureStructure` objects are not guaranteed to be valid once the CAS has been cleared, a downstream GATE component must extract the relevant information from the feature structure before the next document is processed. You have been warned.

Feature names in `uimaFSFeatureValue` must be qualified with their type name, as the feature may have been defined on a supertype of the feature's own type, rather than the type itself. For example, consider the following:

```
<gateAnnotation type="Entity" uimaType="com.example.Entity">
  <feature name="type">
    <uimaFSFeatureValue name="com.example.Entity:Type" kind="string" />
  </feature>
  <feature name="startOffset">
    <uimaFSFeatureValue name="uima.tcas.Annotation:begin" kind="int" />
  </feature>
</gateAnnotation>
```

For `updated` annotations, there must have been an input definition with `indexed="true"` with the same GATE and UIMA types. In this case, for each GATE annotation of the appropriate type, the UIMA annotation that was created from it is found in the CAS. The feature definitions are then used as in the `added` case, but here, the feature values are set on the *original* GATE annotation, rather than on a newly created annotation.

For `removed` annotations, the feature definitions are ignored, and the annotation is removed from GATE if the UIMA annotation which it gave rise to has been removed from the UIMA annotation index.

A complete example

Figure 16.2 shows a complete example mapping descriptor for a simple UIMA TAE that takes tokens as input and adds a feature to each token giving the number of lower case letters in the token's string.[2] In this case the UIMA feature that holds the number of lower case letters is called `LowerCaseLetters`, but the GATE feature is called `numLower`. This demonstrates that the feature names do not need to agree, so long as a mapping between them can be defined.

[2]The Java code implementing this AE is in the `examples` directory of the `uima` plugin. The AE descriptor and mapping file are in `examples/conf`.

```
<uimaGateMapping>
  <inputs>
    <uimaAnnotation type="gate.uima.cas.Token" gateType="Token" indexed="true">
      <feature name="String" kind="string">
        <gateAnnotFeatureValue name="string" />
      </feature>
    </uimaAnnotation>
  </inputs>
  <outputs>
    <updated>
      <gateAnnotation type="Token" uimaType="gate.uima.cas.Token">
        <feature name="numLower">
          <uimaFSFeatureValue name="gate.uima.cas.Token:LowerCaseLetters"
                              kind="int" />
        </feature>
      </gateAnnotation>
    </updated>
  </outputs>
</uimaGateMapping>
```

Figure 16.2: An example mapping descriptor

16.1.2 The UIMA component descriptor

As well as the mapping file, you must provide the UIMA component descriptor that defines how to access the TAE that is to be called. This could be a primitive or aggregate analysis engine descriptor, or a URI specifier giving the location of a remote Vinci or SOAP service. It is up to the developer to ensure that the types and features used in the mapping descriptor are compatible with the type system and capabilities of the TAE, or a runtime error is likely to occur.

16.1.3 Using the `AnalysisEnginePR`

To use a UIMA TAE in GATE, load the `uima` plugin and create a "UIMA Analysis Engine" processing resource. If using the GATE framework rather than the GUI, the class name is `gate.uima.AnalysisEnginePR`. The processing resource expects two parameters:

analysisEngineDescriptor The URL of the UIMA analysis engine descriptor (or URI specifier, for a remote TAE service). This must be a `file:` URL, as UIMA needs a file path against which to resolve imports.

mappingDescriptor The URL of the mapping descriptor file. This may be any kind of

URL (`file:`, `http:`, `Class.getResource()`, `ServletContext.getResource()`, etc.)

Any errors processing either of the descriptor files will cause an exception to be thrown. Once instantiated, you can add the PR to a pipeline in the usual way. `AnalysisEnginePR` implements `LanguageAnalyser`, so can be used in any of the standard GATE pipeline types.

The PR takes the following runtime parameter (in addition to the `document` parameter which is set automatically by a `CorpusController`):

annotationSetName The annotation set to process. Any input mappings take annotations from this set, and any output mappings place their new annotations in this set (`added` outputs) or update the input annotations in this set (`updated` or `removed`). If not specified, the default (unnamed) annotation set is used.

The Annotator implementation must be available for GATE to load. For an annotator written in Java, this means that the JAR file containing the annotator class (and any other classes it depends on) must be present in the GATE classloader. The easiest way to achieve this is to put the JAR file or files in a new directory, and create a `creole.xml` file in the same directory to reference the JARs:

```
<CREOLE-DIRECTORY>
  <JAR>my-annotator.jar</JAR>
  <JAR>classes-it-uses.jar</JAR>
</CREOLE-DIRECTORY>
```

This directory should then be loaded in GATE as a CREOLE plugin. Note that, due to the complex mechanics of classloaders in Java, putting your JARs in GATE's `lib` directory will *not* work.

For annotators written in C++ you need to ensure that the C++ enabler libraries (available separately from http://alphaworks.ibm.com/tech/uima/) and the shared library containing your annotator are in a directory which is on the `PATH` (Windows) or `LD_LIBRARY_PATH` (Linux) when GATE is run.

16.1.4 Current limitations

If you are using Java 5.0 you may get a `NullPointerException` or `NoClassDefFoundError` from the UIMA XML parser when parsing the analysis engine descriptor. This can be fixed by copying `xml.jar` from the `lib` directory into *`GATE_HOME`*`/lib`.

16.2 Embedding a GATE `CorpusController` in UIMA

The process of embedding a GATE controller in a UIMA application is more or less the mirror image of the process detailed in the previous section. Again, the developer must supply a mapping descriptor defining how to map between UIMA and GATE annotations, and pass this, plus the GATE controller definition, to a TAE which performs the translation and calls the GATE controller.

16.2.1 Mapping file format

The mapping descriptor format is virtually identical to that described in section 16.1.1, except that the input definitions are `<gateAnnotation>` elements and the output definitions are `<uimaAnnotation>` elements. The input and output definition elements support an extra attribute, `annotationSetName`, which allows inputs to be taken from, and outputs to be placed in, different annotation sets. For example, the following hypothetical example maps `com.example.Person` annotations into the default set and `com.example.html.Anchor` annotations to "`a`" tags in the "Original markups" set.

```
<inputs>
  <gateAnnotation type="Person" uimaType="com.example.Person">
    <feature name="kind">
      <uimaFSFeatureValue name="com.example.Person:Kind" kind="string"/>
    </feature>
  </gateAnnotation>

  <gateAnnotation type="a" annotationSetName="Original markups"
                  uimaType="com.example.html.Anchor">
    <feature name="href">
      <uimaFSFeatureValue name="com.example.html.Anchor:hRef" kind="string" />
    </feature>
  </gateAnnotation>
</inputs>
```

Figure 16.3 shows a mapping descriptor for an application that takes tokens and sentences produced by some UIMA component and runs the GATE part of speech tagger to tag them with Penn TreeBank POS tags.[3] In the example, no features are copied from the UIMA tokens, but they are still `indexed="true"` as the POS feature must be copied back from GATE.

[3]The `.gapp` file implementing this example is in the `test/conf` directoriy under the `uima` plugin, along with the mapping file and the TAE descriptor that will run it.

```
<uimaGateMapping>
  <inputs>
    <gateAnnotation type="Token"
                    uimaType="com.ibm.uima.examples.tokenizer.Token"
                    indexed="true" />
    <gateAnnotation type="Sentence"
                    uimaType="com.ibm.uima.examples.tokenizer.Sentence" />
  </inputs>
  <outputs>
    <updated>
      <uimaAnnotation type="com.ibm.uima.examples.tokenizer.Token"
                      gateType="Token">
        <feature name="POS" kind="string">
          <gateAnnotFeatureValue name="category" />
        </feature>
      </uimaAnnotation>
    </updated>
  </outputs>
</uimaGateMapping>
```

Figure 16.3: An example mapping descriptor for the GATE POS tagger

16.2.2 The GATE application definition

The GATE application to embed is given as a standard "`.gapp` file", as produced by saving the state of an application in the GATE GUI. The `.gapp` file encodes the information necessary to load the correct plugins and create the various CREOLE components that make up the application. The `.gapp` file must be fully specified and able to be executed with no user intervention other than pressing the Go button. In particular, all runtime parameters must be set to their correct values before saving the application state. Also, since paths to things like CREOLE plugin directories, resource files, etc. are stored relative to the `.gapp` file's location, you must not move the `.gapp` file to a different directory unless you can keep all the CREOLE plugins it depends on at the same relative locations.

16.2.3 Configuring the `GATEApplicationAnnotator`

`GATEApplicationAnnotator` is the UIMA annotator that handles mapping the CAS into a GATE document and back again and calling the GATE controller. There is a template TAE descriptor XML file for the annotator provided in the `conf` directory. Most of the template file can be used unchanged, but you will need to modify the type system definition and input/output capabilities to match the types and features used in your mapping descriptor. If the mapping descriptor references a type or feature that is not defined in the type system, a runtime error will occur.

The annotator requires two external resources:

GateApplication The `.gapp` file containing the saved application state.

MappingDescriptor The mapping descriptor XML file.

These must be bound to suitable URLs, either by editing the `resourceManagerConfiguration` section of the primitive decriptor, or by supplying the binding in an aggregate descriptor that includes the `GATEApplicationAnnotator` as one of its delegates.

In addition, you may need to set the following Java system properties:

uima.gate.configdir The path to the GATE config directory. This defaults to `gate-config` in the same directory as `uima-gate.jar`.

uima.gate.siteconfig The location of the sitewide `gate.xml` configuration file. This defaults to *`gate.uima.configdir`*`/site-gate.xml`.

uima.gate.userconfig The location of the user-specific `gate.xml` configuration file. This defaults to *`gate.uima.configdir`*`/user-gate.xml`.

The default config files are deliberately simplified from the standard versions supplied with GATE, in particular they do not load any plugins automatically (not even ANNIE). All the plugins used by your application are specified in the `.gapp` file, and will be loaded when the application is loaded, so it is best to avoid loading any others from `gate.xml`, to avoid problems such as two different versions of the same plugin being loaded from different locations.

Classpath notes

In addition to the usual UIMA library JAR files, `GATEApplicationAnnotator` requires a number of JAR files from the GATE distribution in order to function. In the first instance, you should include `gate.jar` from GATE's `bin` directory, and also all the JAR files from GATE's `lib` directory on the classpath. If you use the supplied Ant build file, `ant documentanalyser` will run the document analyser with this classpath. Depending on exactly which GATE plugins your application uses, you may be able to exclude some of the `lib` JAR files (for example, you will not need Weka if you do not use the machine learning plugin), but it is safest to start with them all. GATE will load plugin JAR files through its own classloader, so these do not need to be on the classpath.

Note that the GATE `lib` directory includes a version of the Apache Xerces XML parser. UIMA also includes an XML parser in its `xml.jar`. If your program generates unexplained

XML parsing exceptions, try removing one or other of the XML parsers from the classpath to see if this solves the problem.

Appendices

Appendix A

Design Notes

> Why has the pleasure of slowness disappeared? Ah, where have they gone, the amblers of yesteryear? Where have they gone, those loafing heroes of folk song, those vagabonds who roam from one mill to another and bed down under the stars? Have they vanished along with footpaths, with grasslands and clearings, with nature? There is a Czech proverb that describes their easy indolence by a metaphor: 'they are gazing at God's windows.' A person gazing at God's windows is not bored; he is happy. In our world, indolence has turned into having nothing to do, which is a completely different thing: a person with nothing to do is frustrated, bored, is constantly searching for an activity he lacks.
>
> *Slowness*, Milan Kundera, 1995 (pp. 4-5).

GATE is a backplane into which specialised Java Beans plug. These beans are loose-coupled with respect to each other - they communicate entirely by means of the GATE framework. Inter-component communication is handled by model components - LanguageResources, and events.

Components are defined by conformance to various interfaces (e.g. LanguageResource), ensuring separation of interface and implementation.

The reason for adding to the normal bean initialisation mech is that LRs, PRs and VRs all have characteristic parameterisation phases; the GATE resources/components model makes explicit these phases.

A.1 Patterns

GATE is structured around a number of what we might call principles, or patterns, or alternatively, clever ideas stolen from better minds than mine. These patterns are:

- modelling most things as extensible sets of components (cf. Section A.1.1);
- separating components into model, view, or controller (cf. Section A.1.2) types;
- hiding implementation behind interfaces (cf. Section A.1.3).

Four interfaces in the top-level package describe the GATE view of components: Resource, ProcessingResource, LanguageResource and VisualResource.

A.1.1 Components

Architectural Principle

Wherever users of the architecture may wish to extend the set of a particular type of entity, those types should be expressed as components.

Another way to express this is to say that the architecture is based on *agents*. I've avoided this in the past because of an association between this term and the idea of bits of code moving around between machines of their own volition. I take this to be somewhat pointless, and probably the result of an anthropomorphic obsession with mobility as a correlate of intelligence. If we drop this connotation, however, we can say that GATE is an agent-based architecture. If we want to, that is.

Framework Expression

Many of the classes in the framework are components, by which we mean classes that conform to an interface with certain standard properties. In our case these properties are based on the Java Beans component architecture, with the addition of component metadata, automated loading and standardised storage, threading and distribution.

All components inherit from Resource, via one of:

- LanguageResource (LR) represents entities such as lexicons, corpora or ontologies;
- VisualResource (VR) represents visualisation and editing components that participate in GUIs;
- ProcessingResource (PR) represents entities that are primarily algorithmic, such as parsers, generators or ngram modellers.

A.1.2 Model, view, controller

According to Buschmann et al (Pattern-Oriented Software Architecture, 1996), the Model-View-Controller (MVC) pattern

> ...divides an interactive application into three components. The model contains the core functionality and data. Views display information to the user. Controllers handle user input. Views and controllers together comprise the user interface. A change-propagation mechanism ensures consistency between the user interface and the model. [p.125]

A variant of MVC, the Document-View pattern,

> ...relaxes the separation of view and controller... The View component of Document-View combines the responsibilities of controller and view in MVC, and implements the user interface of the system.

A benefit of both arrangements is that

> ...loose coupling of the document and view components enables multiple simultaneous synchronized but different views of the same document.

Geary (Graphic Java 2, 3rd Edtn., 1999) gives a slightly different view:

> MVC separates applications into three types of objects:
>
> - **Models:** Maintain data and provide data accessor methods
> - **Views:** Paint a visual representation of some or all of a model's data
> - **Controllers:** Handle events ... By encapsulating what other architectures intertwine, MVC applications are much more flexible and reusable than their traditional counterparts.
>
> [pp. 71, 75]

Swing, the Java user interface framework, uses

> a specialised version of the classic MVC meant to support pluggable look and feel instead of applications in general. [p. 75]

GATE may be regarded as an MVC architecture in two ways:

- directly, because we use the Swing toolkit for the GUIs;
- by analogy, where LRs are models, VRs are views and PRs are controllers. Of these, the latter sits least easily with the MVC scheme, as PRs may indeed be controllers but may also not be.

A.1.3 Interfaces

Architectural Principle

The implementation of types should generally be hidden from the clients of the architecture.

Framework Expression

With a few exceptions (such as for utility classes), clients of the framework work with the `gate.*` package. This package is mostly composed of interface definitions. Instantiations of these interfaces are obtained via the `Factory` class.

The subsidiary packages of GATE provide the implementations of the `gate.*` interfaces that are accessed via the factory. They themselves avoid directly constructing classes from other packages (with a few exceptions, such as JAPE's need for unattached annotation sets). Instead they use the factory.

A.2 Exception Handling

When and how to use exceptions? Borrowing from Bill Venners, here are some **guidelines** (with examples):

1. Exceptions exist to refer problem conditions up the call stack to a level at which they may be dealt with. "If your method encounters an abnormal condition *that it can't handle*, it should throw an exception." If the method can handle the problem rationally, it should catch the exception and deal with it.

   Example:
   If the creation of a resource such as a document requires a URL as a parameter, the method that does the creation needs to construct the URL and read from it. If there is an exception during this process, the GATE method should abort by throwing its own exception. The exception will be dealt with higher up the food chain, e.g. by asking the user to input another URL, or by aborting a batch script.

2. All GATE exceptions should inherit from gate.util.GateException (a descendant of java.lang.Exception, hence a checked exception) or gate.util.GateRuntimeException (a descendant of java.lang.RuntimeException, hence an unchecked exception). This rule means that clients of GATE code can catch all sorts of exceptions thrown by the system with only two catch statements. (This rule may be broken by methods that are not public, so long as their callers catch the non-GATE exceptions and deal with them or convert them to GateException/GateRuntimeException.) Almost **all** exceptions thrown by GATE should be checked exceptions: the point of an exception is that clients of your code get to know about it, so use a checked exception to make the compiler force them to deal with it. Except:

   Example:
   With reference to the previous example, a problem using the URL will be signalled by something like an UnknownHostException or an IOException. These should be caught and re-thrown as descendants of GateException.

3. In a situation where an exceptional condition is an indication of a bug in the GATE library, or in the implementation of some other library, then it is permissible to throw an unchecked exception.

   Example:
   If a method is creating annotations on a document, and before creating the annotations it checks that their start and end points are valid ranges in relation to the content of the document (i.e. they fall within the offset space of the document, and the end is after the start), then if the method receives an InvalidOffsetException from the AnnotationSet.add call, something is seriously wrong. In such cases it may be best to throw a GateRuntimeException.

4. Where you are inheriting from a non-GATE class and therefore have the exception signatures fixed for you, you may add a new exception deriving from a non-GATE class.

   Example:
   The SAX XML parser API uses SaxException. Implementing a SAX parser for a document type involves overiding methods that throw this exception. Where you want to have a subtype for some problem which is specific to GATE processing, you could use GateSaxException which extends SaxException.

5. Test code is different: in the JUnit test cases it is fine just to declare that each method throws Exception and leave it at that. The JUnit test runner will pick up the exceptions and report them to you. Test methods should, however, try and ensure that the exceptions thrown are meaningful. For example, avoid null pointer exceptions in the test code itself, e.g. by using assertNonNull.

Example:

```
public void testComments() throws Exception {
  ResourceData docRd = (ResourceData) reg.get("gate.Document");
  assertNotNull("testComments: couldn't find document res data", docRd);
  String comment = docRd.getComment();
  assert(
    "testComments: incorrect or missing COMMENT on document",
    comment != null && comment.equals("GATE document")
  );
} // testComments()
```

See also the testing notes.

6. "Throw a different exception type for each abnormal condition." You can go too far on this one - a hundred exception types per package would certainly be too much - but in general you should create a new exception type for each different sort of problem you encounter.

   Example:
   The gate.creole package has a ResourceInstantiationException - this deals with all problems to do with creating resources. We could have had "ResourceUrlProblem" and "ResourceParameterProblem" but that would probably have ended up with too many. On the other hand, just throwing everything as GateException is too coarse (Hamish take note!).

7. Put exceptions in the package that they're thrown from (unless they're used in many packages, in which case they can go in gate.util). This makes it easier to find them in the documentation and prevents name clashes.

   Example:
   gate.jape.ParserException is correctly placed; if it was in gate.util it might clash with, for example, gate.xml.ParserException if there was such.

Appendix B

JAPE: Implementation

The annual Diagram prize for the oddest book title of the year has been awarded to Gerard Forlin's Butterworths Corporate Manslaughter Service, a hefty law tome providing guidance and analysis on corporate liability for deaths in the workplace.

The book, not published until January, was up against five other shortlisted titles: Fancy Coffins to Make Yourself; The Flat-Footed Flies of Europe; Lightweight Sandwich Construction; Tea Bag Folding; and The Art and Craft of Pounding Flowers: No Paint, No Ink, Just a Hammer! The shortlist was thrown open to readers of the literary trade magazine The Bookseller, who chose the winner by voting on the magazine's website. Butterworths Corporate Manslaughter Service, a snip at 375, emerged as the overall victor with 35

The Diagram prize has been a regular on the award circuit since 1978, when Proceedings of the Second International Workshop on Nude Mice carried off the inaugural award. Since then, titles such as American Bottom Archaeology and last year's winner, High-Performance Stiffened Structures (an engineering publication), have received unwonted publicity through the prize. This year's winner is perhaps most notable for its lack of entendre.

Manslaughter Service kills off competition in battle of strange titles, Emma Yates, The Guardian, November 30, 2001.

This chapter gives implementation details and formal definitions of the JAPE annotation patterns language. Section B.1 gives a more formal definition of the JAPE grammar, and some examples of its use. Section B.2 describes JAPE's relation to CPSL. The next 3 sections describe the algorithms used, label binding, and the classes used. Section B.6 gives an example of the implementation; and finally, section B.7 explains the compilation process.

B.1 Formal Description of the JAPE Grammar

JAPE is similar to CPSL (a Common Pattern Specification Language, developed in the TIPSTER programme by Doug Appelt and others), with a few exceptions. Figure B.1 gives a BNF (Backus-Naur Format) description of the grammar.

An example rule LHS:

```
Rule: KiloAmount
( ({Token.kind == "containsDigitAndComma"}):number
  {Token.string == "kilograms"} ):whole
```

A basic constraint specification appears between curly braces, and gives a conjunction of annotation/attribute/value specifiers which have to match at a particular point in the annotation graph. A complex constraint specification appears within round brackets, and may be bound to a label with the ":" operator; the label then becomes available in the RHS for access to the annotations matched by the complex constraint. Complex constraints can also have Kleene operators (*, +, ?) applied to them. A sequence of constraints represents a sequential conjunction; disjunction is represented by separating constraints with "`|`".

Converted to the format accepted by the JavaCC LL parser generator, the most significant fragment of the CPSL grammar (as described by Appelt, based on an original specification from a TIPSTER working group chaired by Boyan Onyshkevych) goes like this:

```
constraintGroup -->
    (patternElement)+ ("|" (patternElement)+ )*

patternElement -->
    "{" constraint ("," constraint)* "}"
|   "(" constraintGroup ")" (kleeneOp)? (binding)?
```

Here the first line of `patternElement` is a basic constraint, the second a complex one.

```
MultiPhaseTransducer ::=
   ( <multiphase> <ident> )?
   ( ( SinglePhaseTransducer )+ | ( <phases> ( <ident> )+ ) )
   <EOF>
SinglePhaseTransducer ::=
   <phase> <ident> ( <input> ( <ident> )* )?
   ( <option> ( <ident> <assign> <ident> )* )?
   ( ( Rule ) | MacroDef )*
Rule ::=
   <rule> <ident> ( <priority> <integer> )?
   LeftHandSide "-->" RightHandSide
MacroDef ::=
   <macro> <ident> ( PatternElement | Action )
LeftHandSide ::=
   ConstraintGroup
ConstraintGroup ::=
   ( PatternElement )+ ( <bar> ( PatternElement )+ )*
PatternElement ::=
   ( <ident> | BasicPatternElement | ComplexPatternElement )
BasicPatternElement ::=
   ( ( <leftBrace> Constraint ( <comma> Constraint )* <rightBrace> )
     | ( <string> ) )
ComplexPatternElement ::=
   <leftBracket> ConstraintGroup <rightBracket>
   ( <kleeneOp> )? ( <colon> ( <ident> | <integer> ) )?
Constraint ::=
   ( <pling> )? <ident> ( <period> <ident> <equals> AttrVal )?
AttrVal ::=
   ( <string> | <ident> | <integer> | <floatingPoint> | <bool> )
RightHandSide ::=
   Action ( <comma> Action )*
Action ::=
   ( NamedJavaBlock | AnonymousJavaBlock | AssignmentExpression | <ident> )
NamedJavaBlock ::=
   <colon> <ident> <leftBrace> ConsumeBlock
AnonymousJavaBlock ::=
   <leftBrace> ConsumeBlock
AssignmentExpression ::=
   ( <colon> | <colonplus> ) <ident> <period> <ident>
   <assign> <leftBrace> (
     <ident> <assign>
     ( AttrVal | ( <colon> <ident> <period> <ident> <period> <ident> ) )
     ( <comma> )?
   )* <rightBrace>
ConsumeBlock ::=
   Java code
```

Figure B.1: BNF of JAPE's grammar

An example of a complete rule:

```
Rule: NumbersAndUnit
( ( {Token.kind == "number"} )+:numbers {Token.kind == "unit"} )
-->
:numbers.Name = { rule = "NumbersAndUnit" }
```

This says 'match sequences of numbers followed by a unit; create a Name annotation across the span of the numbers, and attribute rule with value NumbersAndUnit'.

B.2 Relation to CPSL

We *differ from the CPSL spec* in various ways:

1. No pre- or post-fix context is allowed on the LHS.
2. No function calls on the LHS.
3. No string shorthand on the LHS.
4. We have two rule application algorithms (one like TextPro, one like Brill/Mitre). See section B.3.
5. Expressions relating to labels unbound on the LHS are not evaluated on the RHS. (In TextPro they evaluate to "false".) See the binding scheme description in section B.4.
6. JAPE allows arbitrary Java code on the RHS.
7. JAPE has a different macro syntax, and allows macros for both the RHS and LHS.
8. JAPE grammars are compiled and stored as serialised Java objects.

Apart from this, it is a full implementation of CPSL, and the formal power of the languages is the same (except that a JAPE RHS can delete annotations, which straight CPSL cannot). The rule LHS is a regular language over annotations; the rule RHS can perform arbitrary transformations on annotations, but the RHS is only fired *after* the LHS been evaluated, and the effects of a rule application can only be referenced after the phase in which it occurs, so the recognition power is no more than regular.

B.3 Algorithms for JAPE Rule Application

JAPE rules are applied in one of two ways: Brill-style, where each rule is applied at every point in the document at which it matches; Appelt-style, where only the longest matching rule is applied at any point where more than one might apply.

In the Appelt case, the rule set for a phase may be considered as a single disjunctive expression (and an efficient implementation would construct a single automaton to recognise the whole rule set). To solve this problem, we need to employ two algorithms:

- one that takes as input a CPSL representation and builds a machine capable of recognizing the situations that match the rules and makes the bindings that occur each time a rule is applied. This machine is a Finite State Machine (FSM), somewhat similar to a lexical analyser (a deterministic finite state automaton).
- another one that uses the FSM built by the above algorithm and traverses the annotation graph in order to find the situations that the FSM can recognise.

B.3.1 The first algorithm

The first step that needs to be taken in order to create the FSM is to read the CPSL description from the external file(s). This is already done in the old version of Jape.

The second step is to build a nondeterministic FSM from the java objects resulted from the parsing process. This FSM will have one initial state and a set of final states, each of them being associated to one rule (this way we know what RHS we have to execute in case of a match). The nondeterministic FSM will also have empty transitions (arcs labeled with **nil**). In order to build this FSM we will need to implement a version of the algorithm used to convert regular expressions in NFAs.

Finally, this nondeterministic FSM will have to be converted to a deterministic one. The deterministic FSM will have more states (in the worst case s! (where s is the number of states in the nondeterministic one); this case is very improbable) but will be more efficient because it will not have to backtrack.

Let **NFSM** be the nondeterministic FSM and **DFSM** the deterministic one.

The issues that have to be addressed are:

The NFSM will basically be a big OR. This means that it will have an initial state from which empty transitions will lead to the sub-FSMs associated to each rule (see Fig. B.2). When the NFSM is converted to a DFSM the initial state will be the set containing all the initial states of the FSMs associated to each rule. From that state we will have to compute the possible transitions. For this, the classical algorithm requires us to check for each possible

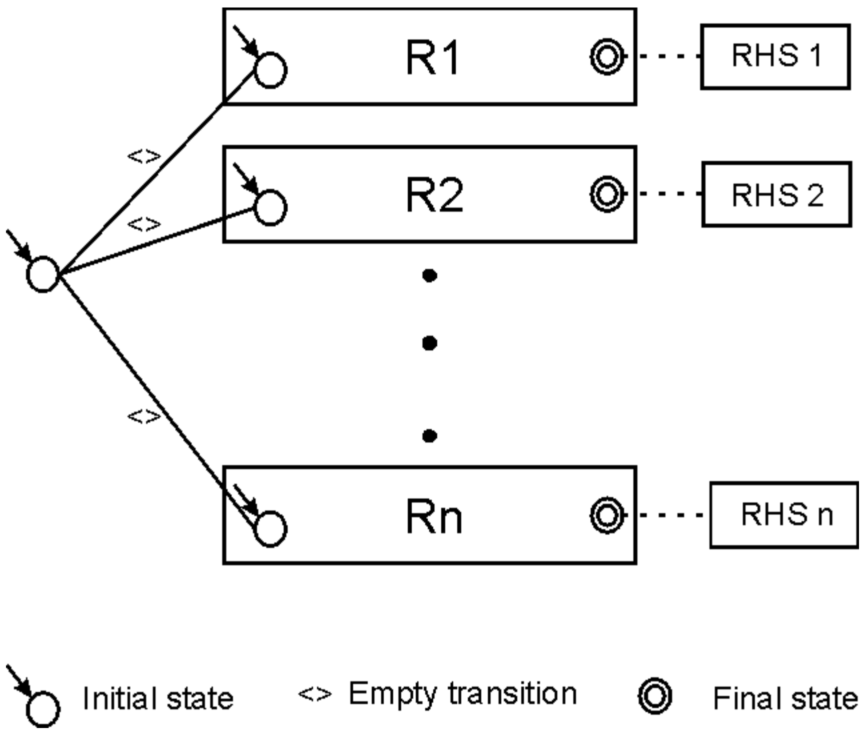


Figure B.2: A nondeterministic FSM

input symbol what is the set of reachable states. The problem is that our input symbols are actually sets of restrictions. This is similar to an automaton that has an infinite set of input symbols (although any given set of rules describes a finite set of constraints). This is not so bad, the real problem is that we have to check if there are transitions that have the same restrictions. We can safely consider that there are no two transitions with the same set of restrictions. This is safe because if this assumption is wrong, the result will be a state that has two transitions starting from it, transitions that consume the same symbol. This is not a problem because we have to check all outgoing transitions anyway; we will only check the same transition twice.

This leads to the next issue. Imagine the next part of the transition graph of a FSM (Fig. B.3):

The restrictions associated to a transition are depicted as graphical figures (the two coloured squares). Now imagine that the two sets of restrictions have a common part (the yellow triangle).

Let us assume that at one moment the current node in the FSM graph (for one of the active FSM instances) is state 1. We get from the annotation graph the set of annotations starting from the associated current node in the annotation graph and try to advance in the FSM transition graph. In order to do this we will have to find a subset of annotations that match the restrictions for moving to state 2 or state 3. In a classical algorithm what we would do

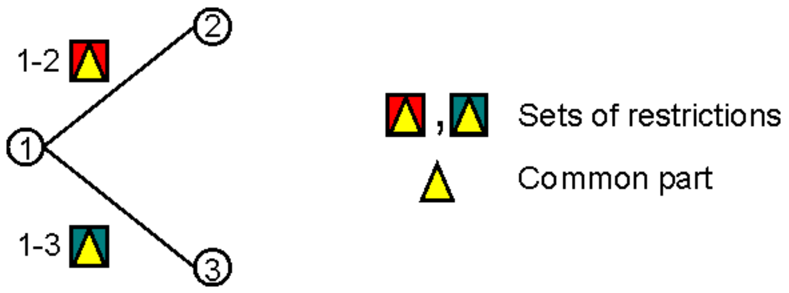


Figure B.3: Example of transitions

is to try to match the annotations against the restrictions "1-2" (this will return a boolean value and a set of bindings) and then we will try the matching against the restrictions "1-3" this means that we will try to match the restrictions in the common part **twice**. Because of the probable structure of the FSM transition graph there will be a lot of transitions starting from the same node which means that may be a lot of conditions checked more than one times.

What can we do to improve this?

We need a way to combine all the restrictions associated to all outgoing arcs of a state (see Fig. B.4).

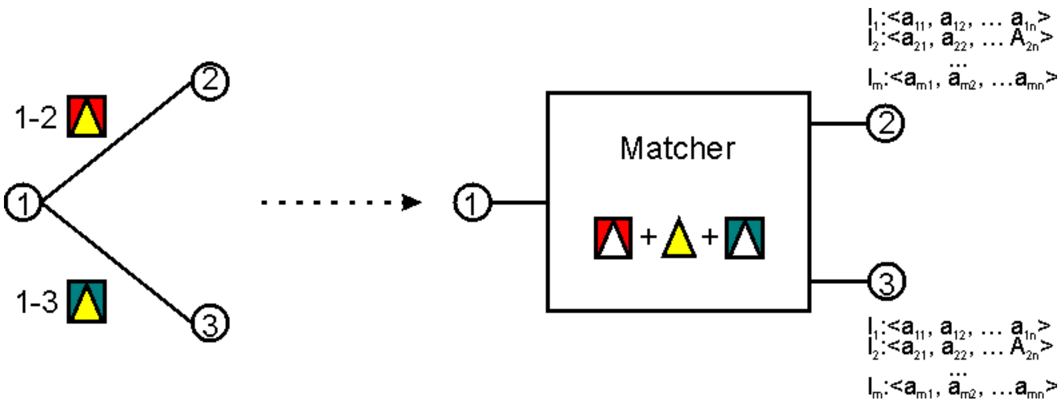


Figure B.4: A combined matching process

One way to do the (combined) matching is to pre-process the DFSM and to convert all transitions to matchers (as in Fig. B.4). This could be done using the following algorithm:

- **Input:** A DFSM;
- **Output:** A DFSM with compound restrictions checks.
- for each state s of the DFSM

1. collect all the restrictions in the labels of the outgoings arcs from s (in the DFSM transition graph)
   Note: these restrictions are either of form “Type == t_1” or of form “Type == t_1 && $Attr_i == Value_i$
2. Group all these restrictions by type and branch and create compound restrictions of form “[Type == t_1 && $Attr_1 == Value_1$ && $Attr_2 == Value_2$ && ... && $Attr_n == Value_n$]”

   The grouping has to be done with care so it doesn’t mix restrictions from different branches, creating unnecessary restrictive queries. These restrictions will be sent to the annotation graph which will do the matching for us. Note that we can only reuse previous queries if the restrictions are identical on two branches.[1]
3. Create the data structures necessary for linking the bindings to the results of the queries (see Fig B.5)

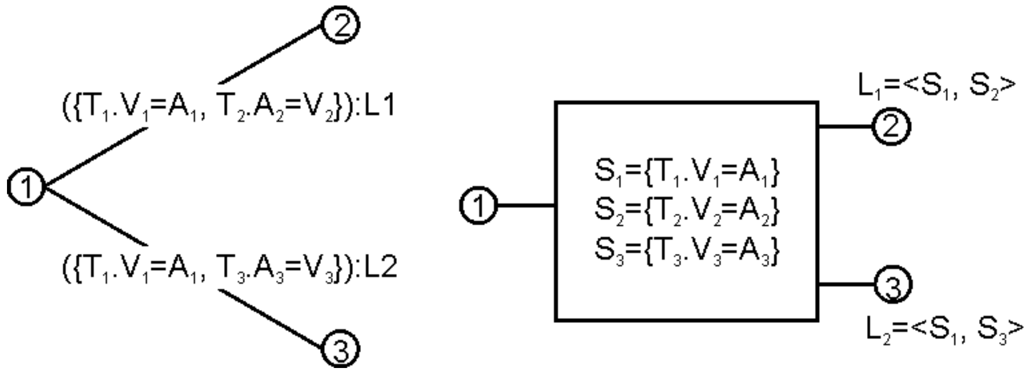


Figure B.5: Building a compound matcher

When this machine will be used for the actual matching the three queries will be run and the results will be stored in sets of annotations (S1..S3 in the picture) and...

- For each pair of annotations from (A_1, A_2) s.t. A_1 in S_1 & A_2 in S_2
  1. a new DFSM instance will be created;
  2. this instance will move to state 2;
  3. ¡A_1, A_2¿ will be bound to L_1
  4. the corresponding node in the annotation graph will become max(A_1 endNode(), A_2.endNode()).

[1]By this we mean restrictions referring to the same type of annotations. If for branches 1-2 and 1-3 the restrictions for the type T_1 are the same, the query for type T_1 will be run only once. Each of the two branches can also have restrictions for other types of annotations.

- Similarly, for each pair of annotations from (A_1, A_3) s.t. A_1 in S_1 & A_3 in S_3
  1. a new DFSM instance will be created;
  2. this instance will move to state 3;
  3. ¡A_1, A_3¿ will be bound to L_2
  4. the corresponding node in the annotation graph will become max(A_1.endNode(), A_3.endNode()).

While building the compound matcher it is possible to detect queries that depend one from another (e.g. if the expected results of a query are a subset of the results from another query). This kind of situations can be marked so when the queries are actually run some operations can be avoided (e.g. if the less restrictive search returned no results than the more restrictive one can be skipped, or if a search returns an AnnotationSet (an object that can be queried) than the more restrictive query can be.

B.3.2 Algorithm 2

Consider the following figure:

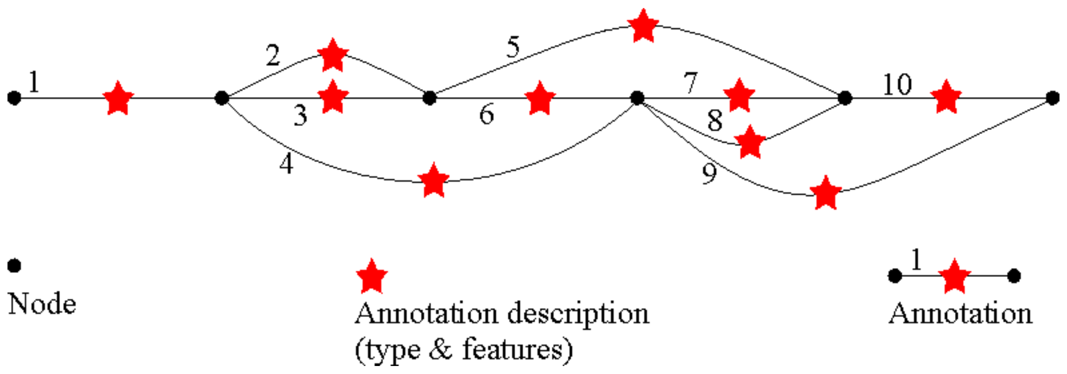


Figure B.6: An annotation graph

Basically, the algorithm has to traverse this graph starting from the leftmost node to the rightmost one. Each path found is a sequence of possible matches.

Because more than one annotation (all starting at the same point) can be matched at one step, a path is not viewed as a classical path in a graph, but a sequence of steps, each step being a set of annotations that start in the same node.
e.g. a path in the graph above can be: [1].[2,4].[7,8].[10];
Note that the next step continues from the rightmost node reached by the annotations in the current step.

The matchings are made by a Finite State Machine that resembles an clasical lexical analyser (*aka. scanner*). The main difference from a scanner is that there are no input symbols; the transition from one state to another is based on matching a set of objects (annotations) against a set of restrictions (the constraint group in the LHS of a CPSL rule).

The algorithm can be the following:

```
1. startNode = the leftmost node
2. create a first instance of the FSM and add it to the list of active instances;
3. for this FSM instance set current node as the leftmost node;
4. while(startNode != last node) do
     1 while (not over) do
         1 for each $Fi$ active instance of the FSM do
             1 if this instance is in a final state then save a clone of it in the set of
               accepting FSMs (instances of the FSM that have reached a final state);
             2 read all the annotations starting from the current node;
             3 select all sets of annotation that can be used to advance one step in the
               transition graph of the FSM;
             4 for each such set create a new instance of the FSM, put it in the active
               list and make it consume the corresponding set of annotations, making
               any necessary bindings in the process (this new instance will advance in
               the annotation graph to the rightmost node that is an end of a matched
               annotation);
             5 discard $F_i$;
         2 end for;
         3 if the set of active instances of FSM is empty * then over = true;
       end while;
     2 if the set of accepting FSMs is not empty
         1 from all accepting FSMs select ** the one that matched the longest path;if
           there are more than one for the same path length select the one with highest
           priority;
         2 execute the action associated to the final state of the selected FSM instance;
         3 startNode = selectedFSMInstance.getLastNode.getNextNode();
     3 else //the matching failed $\rightarrow$ start over from the next node // startNode = startN-
       ode.getNextNode();
5. end while;
```

**: the set of active FSM instances can decrease when an active instance cannot continue (there is no set of annotations starting from its current node that can be matched). In this case it will be removed from the set.*

***: if we do Brill style matching, we have to process each of the accepting instances.*

B.4 Label Binding Scheme

In TextPro, a ":" label binds to the last matched annotation in its scope. A "+:" label binds to all the annotations matched in the scope. In JAPE there is no "+:" label (though there is a ":+" – see below), due to the ambiguity with Kleene +. In CPSL a constraint group can be both labelled and have a Kleene operator. How can Kleene + followed by label : be distinguished from label +: ? E.g. given `(....)+:label` are the constraints within the brackets having Kleene + applied to them and being labelled, or is it a +: label?

Appelt's answer is that +: is always a label; to get the other interpretation use `((...)+):`. This may be difficult for rule developers to remember; JAPE disallows the "+:" label, and makes all matched annotations available from every label.

JAPE adds a ":+" label operator, which means that all the spans of any annotations matched are assigned to new annotations created on the RHS relative to that label. (With ordinary ":" labels, only the span of the outermost corners of the annotations matched is used.) (This operator disappears in GATE version 2, with the elimination of multi-span annotations.)

Another problem regards RHS interpretation of unbound labels. If we have something like

```
(
  ( {Word.string == "thing"} ):1
  |
  ( {Word.string == "otherthing"} ):2
)
```

on the LHS, and references to :1 and :2 on the RHS, only one of these will actually be bound to anything when the rule is fired. The expression containing the other should be ignored. In TextPro, an assignment on the RHS that references an unbound label is evaluated to the value "false". In JAPE, RHS expressions involving unbound operators are not evaluated.

B.5 Classes

The main external interfaces to JAPE are the classes `gate.jape.Batch` and `gate.jape.Compiler`. The CPSL Parser is implemented by `ParseCpsl.jj`, which is input to JavaCC (and JJDoc to produce grammar documentation) and finally Java itself. There are lots of other classes produced along the way by the compiler-compiler tools:

```
ASCII_CharStream.java JJTParseCpslState.java Node.java ParseCpsl.java
ParseCpslConstants.java ParseCpslTokenManager.java ParseCpslTreeConstants.java
ParseException.java SimpleNode.java TestJape.java Token.java TokenMgrError.java
```

These live in the parser subpackage, in the `gate/jape/parser` directory.

Each grammar results in an object of class `Transducer`, which has a set of `Rule`.

Constants are held in the interface `JapeConstants`. The test harness is in `TestJape`.

B.6 Implementation

B.6.1 A Walk-Through

The pattern application algorithm (which is either like Doug's, or like Brill's), makes a top-level call to something like

```
boolean matches(int position, Document doc,
                MutableInteger newPosition)
throws PostionOutOfRange
```

which is a method on each `Rule`. This is in turn deferred to the rule's `LeftHandSide`, and thence to the `ConstraintGroup` which each `LeftHandSide` contains. The `ConstraintGroup` iterates over its set of `PatternElementConjunctions`; when one succeeds, the matches call returns true; if none succeed, it returns false. The `Rules` also have

```
void transduce(Document doc) throws LhsNotMatched
```

methods, which may be called after a successful match, and result in the application of the `RightHandSide` of the `Rule` to the document.

`PatternElements` also implement the matches method. Whenever it succeeds, the annotations which were consumed during the match are available from that element, as are a composite span set, and a single span that covers the whole set. In general these will only be accessed via a `bindingName`, which is associated with `ComplexPatternElements`. The `LeftHandSide` maintains a mapping of `bindingNames` to `ComplexPatternElements` (which are accessed by array reference in `Rule RightHandSides`).

Although `PatternElements` give access to an annotation set, these are only built when they are asked for (caching ensures that they are only built once) to avoid storing annotations against every matched element. When asked for, the construction process is an iterative traversal of the elements contained within the element being asked for the annotations. This traversal always bottoms out into `BasicPatternElements`, which are the only ones that need to store annotations all the time.

In a `RightHandSide` application, then, a call to the `LeftHandSide's` binding environment will yield a `ComplexPatternElement` representing the bound object, from which annotations and spans can be retrieved as needed.

B.6.2 Example RHS code

Let's imagine we are writing an RHS for a rule which binds a set of annotations representing simple numbers to the label `:numbers`. We want to create a new annotation spanning all the ones matched, whose value is an Integer representing the sum of the individual numbers.

The RHS consists of a comma-separated list of blocks, which are either anonymous or labelled. (We also allow the CPSL-style shorthand notation as implemented in TextPro. This is more limiting than code, though, e.g. I don't know how you could do the summing operation below in CPSL.) Anonymous blocks will be evaluated within the same scope, which encloses that of all named blocks, and all blocks are evaluated in order, so declarations can be made in anonymous blocks and then referenced in subsequent blocks. Labelled blocks will only be evaluated when they were bound during LHS matching. The symbol `doc` is always scoped to the Document which the `Transducer` this rule belongs to is processing. For example:

```
// match a sequence of integers, and store their sum
Rule:   NumberSum

( {Token.kind == "otherNum"} )+ :numberList

-->

:numberList{
  // the running total
  int theSum = 0;

  // loop round all the annotations the LHS consumed
  for(int i = 0; i<numberListAnnots.length(); i++) {

    // get the number string for this annot
    String numberString = doc.spanStrings(numberListAnnots.nth(i));
    // parse the number string and add to running total
    try {
      theSum += Integer.parseInt(numberString);
    } catch(NumberFormatException e) {
      // ignore badly-formatted numbers
    }
  } // for each number annot

  doc.addAnnotation(
    "number",
    numberListAnnots.getLeftmostStart(),
    numberListAnnots.getRightmostEnd(),
    "sum",
    new Integer(theSum)
  );

} // :numberList
```

This stuff then gets converted into code (that is used to form the class we create for RHSs) looking like this:

```
package japeactionclasses;

import gate.*; import java.io.*; import gate.jape.*;
import gate.util.*; import gate.creole.*;

public class Test2NumberSumActionClass
implements java.io.Serializable, RhsAction {

  public void doit(Document doc, LeftHandSide lhs) {

    AnnotationSet numberListAnnots = lhs.getBoundAnnots("numberList");
    if(numberListAnnots.size() != 0) {
      int theSum = 0;

      for(int i = 0; i<numberListAnnots.length(); i++) {
        String numberString = doc.spanStrings(numberListAnnots.nth(i));

        try {
          theSum += Integer.parseInt(numberString);
        } catch(NumberFormatException e) {    }
      }
      doc.addAnnotation(
        "number",
        numberListAnnots.getLeftmostStart(),
        numberListAnnots.getRightmostEnd(),
        "sum",
        new Integer(theSum)
      );

    }
  }
}
```

B.7 Compilation

JAPE uses a compiler that translates CPSL grammars to Java objects that target the GATE API (and a regular expression library). It uses a compiler-compiler (JavaCC) to construct the parser for CPSL. Because CPSL is a transducer based on a regular language (in effect an FST) it deploys similar techniques to those used in the lexical analysers of parser generators (e.g. lex, flex, JavaCC tokenisation rules).

In other words, the JAPE compiler is a compiler generated with the help of a compiler-compiler which uses back-end code similar to that used in compiler-compilers. Confused? If not, welcome to the domain of the nerds, which is where you belong; I'm sure you'll be happy here.

B.8 Using a Different Java Compiler

GATE allows you to choose which Java compiler is used to compile the action classes generated from JAPE rules. The preferred compiler is specified by the `Compiler_type` option in `gate.xml`. At present the supported values are:

Sun The Java compiler supplied with the JDK. Although the option is called `Sun`, it supports any JDK that supplies `com.sun.tools.javac.Main` in a standard location, including the IBM JDK (all platforms) and the Apple JDK for Mac OS X.

Eclipse The Eclipse compiler, from the Java Development Tools of the Eclipse project[2]. Currently we use the compiler from Eclipse 3.2, which supports Java 5.0.

By default, the Eclipse compiler is used. It compiles faster than the Sun compiler, and loads dependencies via the GATE ClassLoader, which means that Java code on the right hand

[2]http://www.eclipse.org/jdt

side of JAPE rules can refer to classes that were loaded from a plugin JAR file. The Sun compiler can only load classes from the system classpath, so it will not work if GATE is loaded from a subsidiary classloader, e.g. a Tomcat web application. You should generally use the Eclipse compiler unless you have a compelling reason not to.

Support for other compilers can be added, but this is not documented here - if you're in a position to do this, you won't mind reading the source code...

Appendix C

Ant Tasks for GATE

This chapter describes the Ant tasks provided by GATE that you can use in your own build files. The tasks require Ant 1.7 or later.

C.1 Declaring the Tasks

To use the GATE Ant tasks in your build file you must include the following `<typedef>` (where `${gate.home}` is the location of your GATE installation):

```
<typedef resource="gate/util/ant/antlib.xml">
  <classpath>
    <pathelement location="${gate.home}/bin/gate.jar" />
    <fileset dir="${gate.home}/lib" includes="*.jar" />
  </classpath>
</typedef>
```

If you have problems with library conflicts you should be able to reduce the JAR files included from the lib directory to just jdom, xstream and jaxen (plus stax-api and wstx-lgpl if you are running on Java 5, but these are not required on Java 6).

C.2 The packagegapp task - bundling an application with its dependencies

C.2.1 Introduction

GATE saved application states (GAPP files) are an XML representation of the state of a GATE application. One of the features of a GAPP file is that it holds references to the external resource files used by the application as paths relative to the location of the GAPP file itself. This is useful in many cases but if you want to package up a copy of an application to send to a third party or to use in a web application, etc., then you need to be very careful to save the file in a directory above *all* its resources, and package the resources up with the GAPP file at the same relative paths. If the application refers to resources outside its own file tree (i.e. with relative paths that include `..`) then you must either maintain this structure or manually edit the XML to move the resource references around and copy the files to the right places to match. This can be quite tedious and error-prone...

The `packagegapp` Ant task aims to automate this process. It extracts all the relative paths from a GAPP file, writes a modified version of the file with these paths rewritten to point to locations below the new GAPP file location (i.e. with no `..` path segments) and copies the referenced files to their rewritten locations. The result is a directory structure that can be easily packaged into a zip file or similar and moved around as a self-contained unit.

This Ant task is the underlying driver for the "Export for Teamware" option described in section 3.24. Export for teamware does the equivalent of:

```
<packagegapp src="sourceFile.gapp"
             destfile="{tempdir}/application.xgapp"
             copyPlugins="yes"
             copyResourceDirs="yes"
             onUnresolved="recover" />
```

followed by packaging the temporary directory into a zip file. These options are explained in detail below.

The `packagegapp` task requires Ant 1.7 or later.

C.2.2 Basic Usage

In many cases, the following simple invocation will do what you want:

```
<packagegapp src="original.xgapp"
             destfile="package/target.xgapp" />
```

Note that the parent directory of the `destfile` (in this case `package`) must already exist. It will not be created automatically.

This will perform the following steps:

1. Read in the `original.xgapp` file and extract all the relative paths it contains.
2. For each plugin referred to by a relative path, `foo/bar/MyPluigin`, rewrite the plugin location to be `plugins/MyPlugin` (relative to the location of the `destfile`).
3. For each resource file referred to by the gapp, see if it lives under the original location of one of the plugins moved in the previous step. If so, rewrite its location relative to the new location of the plugin.
4. If there are any relative resource paths that are not accounted for by the above rule (i.e. they do not live inside a referenced plugin), the build fails (see section C.2.3 for how to change this behaviour).
5. Write out the modified GAPP to the `destfile`.
6. Recursively copy the whole content of each of the plugins from step 2 to their new locations[1].

This means that the all the relative paths in the new GAPP file (`package/target.xgapp`) will point to `plugins/Something`. You can now bundle up the whole `package` directory and take it elsewhere.

C.2.3 Handling non-plugin resources

By default, the task only handles relative resource paths that point within one of the plugins that the GAPP refers to. However, many applications refer to resources that live outside the plugin directories, for example custom JAPE grammars, gazetteer lists, etc. The task provides two approaches to support this: it can handle the unresolved references automatically, or you can provide your own "hints" to augment the default plugin-based ones.

Resolving unresolved resources

By default, the build will fail if there are any relative paths that cannot be accounted for by the plugins (or the explicit hints, see section C.2.3). However, this is configurable using the `onUnresolved` attribute, which can take the following values:

fail (default) the build fails if an unresolved relative path is found.

[1]This is done with an Ant copy task and so is subject to the normal defaultexcludes

absolute unresolved relative paths are left pointing to the same location as in the original file, but as an *absolute* rather than a relative URL. The same file will be used even if you move the GAPP file to a different directory. This option is useful if the resource in question is visible at the same absolute location on the machine where you will be putting the packaged file (for example a very large dictionary or ontology held on a network share).

recover attempt to recover gracefully (see below).

With `onUnresolved="recover"`, unresolved resources are relocated to a directory named `application-resources` under the target GAPP file location. Resources in the same original directory are copied to the same subdirectory of `application-resources`, files from different original directories are copied to different subdirectories. Typically, for a resource whose original location was `.../myresources/grammar/clever.jape` the target location would be `application-resources/grammar/clever.jape` but if the application also referred to (say) `.../otherresources/grammar/clean.jape` then this would be mapped into `application-resources/grammar-1` to avoid a name clash.

Example:

```
<packagegapp src="original.xgapp" destfile="package/target.xgapp"
             onUnresolved="recover" />
```

Providing mapping hints

By default, the task knows how to handle resources that live inside plugins. You can think of this as a "hint" `/foo/bar/MyPlugin -> plugins/MyPlugin`, saying that whenever the mapper finds a resource path of the form `/foo/bar/MyPlugin/X`, it will relocate it to `plugins/MyPlugin/X` relative to the output GAPP file. You can specify your own hints which will be used the same way.

```
<packagegapp src="original.xgapp" destfile="package/target.xgapp">
  <hint from="${user.home}/my-app-v1" to="resources/my-app" />
  <hint from="/share/data/bigfiles" absolute="yes" />
</packagegapp>
```

In this example, `~/my-app-v1/grammar/main.jape` would be mapped to `resources/my-app/grammar` (as always, relative to the output GAPP file). You can also hint that certain resources should be converted to absolute paths rather than being packaged with the application, using `absolute="yes"`. The `from` and `to` values refer to directories - you cannot hint a single file, nor put two files from the same original directory into different directories in the packaged GAPP.

Explicit hints override the default plugin-based hints. For example given the hint `from="${gate.home}/plugins/ANNIE/resources" to="resources/ANNIE"`, resources within the ANNIE plugin would be mapped into `resources/ANNIE`, but the plugin `creole.xml` itself would still be mapped into `plugins/ANNIE`.

As well as providing the hints inline in the build file you can also read them from a file in the normal Java Properties format[2], using

```
<hint file="hints.properties" />
```

The keys in the property file are the `from` paths (in this case, relative paths are resolved against the project base directory, as with the `location` attribute of a `property` task) and the values are the `to` paths relative to the output file location.

C.2.4 Streamlining your plugins

By default, the task will recursively copy the whole content of every plugin into the target directory. In most cases this is OK but it may be the case that your plugins contain many extraneous resources that are not used by your application. In this case you can specify `copyPlugins="no"`:

```
<packagegapp src="original.xgapp" destfile="package/target.xgapp"
             copyPlugins="no" />
```

In this mode, the packager task will copy only the following files from each plugin:

- `creole.xml`
- any JAR files referenced from `<JAR>` elements in `creole.xml`

In addition it will of course copy any files *directly* referenced by the GAPP, but not files referenced indirectly (the classic examples being `.lst` files used by a gazetteer `.def`, or the individual phases of a multiphase JAPE grammar) or files that are referenced by the `creole.xml` itself as `AUTOINSTANCE` parameters (e.g. the annotation schemas in ANNIE). You will need to name these extra files explicitly as extra resources (see the next section).

[2]the hint tag supports all the attributes of the standard Ant property tag so can load the hints from a file on disk or from a resource in a JAR file

C.2.5 Bundling extra resources

Apart from plugins (when you don't use `copyPlugins="no"`), the only files copied into the target directory are those that are referenced directly from the GAPP file. This is often but not always sufficient, for example if your application contains a multiphase JAPE transducer then `packagegapp` will include the main JAPE file but not the individual phase files. The task provides two ways to include extra files in the package:

- If you set the attribute `copyResourceDirs="yes"` on the `packagegapp` task then whenever the task packages a referenced resource file it will also recursively include the whole contents of the directory containing that file in the output package. You probably don't want to use this option if you have resource files in a directory shared with other files (e.g. your home directory...).
- To include specific extra resources you can use an `<extraresourcespath>` (see below).

The `<extraresourcespath>` allows you to specify specific extra files that should be included in the package:

```
<packagegapp src="original.xgapp" destfile="package/target.xgapp">
  <extraresourcespath>
    <pathelement location="${user.home}/common-files/README" />
    <fileset dir="${user.home}/my-app-v1" includes="grammar/*.jape" />
  </extraresourcespath>
</packagegapp>
```

As the name suggests, this is a path-like structure and supports all the usual elements and attributes of an Ant `<path>`, including multiple nested `fileset`, `filelist`, `pathelement` and other `path` elements. For specific types of indirect references, there are helper elements that can be included under `extraresourcespath`. Currently the only one of these is `gazetteerlists`, which takes the path to a gazetteer definition file and returns the set of `.lst` files the definition uses:

```
<gazetteerlists definition="my/resources/lists.def" encoding="UTF-8" />
```

Other helpers (e.g. for multiphase JAPE) may be implemented in future.

You can also refer to a path defined elsewhere in the usual way:

```
<path id="extra.files">
  ...
```

```
</path>

<packagegapp ...>
  <extraresourcespath refid="extra.files" />
</packagegapp>
```

Resources declared in the `extraresourcespath` and directories included using `copyResourceDirs` are treated exactly the same as resources that are referenced by the GAPP file - their target locations in the package are determined by the mapping hints, default plugin-based hints, and the `onUnresolved` setting as above. If you want to put extra resource files at specific locations in the package tree, independent of the mapping hints mechanism, you should do this with a separate `<copy>` task after the `<packagegapp>` task has done its work.

C.3 The `expandcreoles` task - merging annotation-driven config into creole.xml

The `expandcreoles` task processes a number of creole.xml files from plugins, processes any `@CreoleResource` and `@CreoleParameter` annotations on the declared resource classes, and merges this configuration with the original XML configuration into a new copy of the creole.xml. It is not necessary to do this in the normal use of GATE, and this task is documented here simply for completeness. It is intended simply for use with non-GATE tools that can process the creole.xml file format to extract information about plugins (the prime use case for this is to generate the GATE plugins information page automatically from the plugin definitions).

The typical usage of this task (taken from the GATE build.xml) is:

```
<expandcreoles todir="build/plugins" gatehome="${basedir}">
  <fileset dir="plugins" includes="*/creole.xml" />
</expandcreoles>
```

This will initialise GATE with the given GATE_HOME directory, then read each file from the nested fileset, parse it as a creole.xml, expand it from any annotation configuration, and write it out to a file under `build/plugins`. Each output file will be generated at the same location relative to the `todir` as the original file was relative to the `dir` of its `fileset`.

Appendix D

Named-Entity State Machine Patterns

There are, it seems to me, two basic reasons why minds aren't computers... The first... is that human beings are organisms. Because of this we have all sorts of needs - for food, shelter, clothing, sex etc - and capacities - for locomotion, manipulation, articulate speech etc, and so on - to which there are no real analogies in computers. These needs and capacities underlie and interact with our mental activities. This is important, not simply because we can't understand how humans behave except in the light of these needs and capacities, but because any historical explanation of how human mental life developed can only do so by looking at how this process interacted with the evolution of these needs and capacities in successive species of hominids.

...

The second reason... is that... brains don't work like computers.

Minds, Machines and Evolution, Alex Callinicos, 1997 (ISJ 74, p.103).

This chapter describes the individual grammars used in GATE for Named Entity Recognition, and how they are combined together. It relates to the default NE grammar for ANNIE, but should also provide guidelines for those adapting or creating new grammars. For documentation about specific grammars other than this core set, use this document in combination with the comments in the relevant grammar files. chapter 7 also provides information about designing new grammar rules and tips for ensuring maximum processnig speed.

D.1 Main.jape

This file contains a list of the grammars to be used, in the correct processing order. The ordering of the grammars is crucial, because they are processed in series, and later grammars may depend on annotations produced by earlier grammars.

The default grammar consists of the following phases:

- first.jape
- firstname.jape
- name.jape
- name_post.jape
- date_pre.jape
- date.jape
- reldate.jape
- number.jape
- address.jape
- url.jape
- identifier.jape
- jobtitle.jape
- final.jape
- unknown.jape
- name_context.jape
- org_context.jape
- loc_context.jape
- clean.jape

D.2 first.jape

This grammar must always be processed first. It can contain any general macros needed for the whole grammar set. This should consist of a macro defining how space and control characters are to be processed (and may consequently be different for each grammar set, depending on the text type). Because this is defined first of all, it is not necessary to restate this in later grammars. This has a big advantage – it means that default grammars can be used for specialised grammar sets, without having to be adapted to deal with e.g. different treatment of spaces and control characters. In this way, only the first.jape file needs to be changed for each grammar set, rather than every individual grammar.

The first.jape grammar also has a dummy rule in. This is never intended to fire – it is simply added because every grammar set must contain rules, but there are no specific rules we wish to add here. Even if the rule were to match the pattern defined, it is designed not to produce any output (due to the empty RHS).

D.3 firstname.jape

This grammar contains rules to identify first names and titles via the gazetteer lists. It adds a gender feature where appropriate from the gazeteer list. This gender feature is used later in order to improve co-reference between names and pronouns. The grammar creates separate annotations of type FirstPerson and Title.

D.4 name.jape

This grammar contains initial rules for organization, location and person entities. These rules all create temporary annotations, some of which will be discarded later, but the majority of which will be converted into final annotations in later grammars. Rules beginning with ”Not” are negative rules – this means that we detect something and give it a special annotation (or no annotation at all) in order to prevent it being recognised as a name. This is because we have no negative operator (we have ”=” but not ”!=”).

D.4.1 Person

We first define macros for initials, first names, surnames, and endings. We then use these to recognise combinations of first names from the previous phase, and surnames from their POS tags or case information. Persons get marked with the annotation ”TempPerson”. We also percolate feature information about the gender from the previous annotations if known.

D.4.2 Location

The rules for Location are fairly straightforward, but we define them in this grammar so that any ambiguity can be resolved at the top level. Locations are often combined with other entity types, such as Organisations. This is dealt with by annotating the two entity types separately, and them combining them in a later phase. Locations are recognised mainly by gazetter lookup, using not only lists of known places, but also key words such as mountain, lake, river, city etc. Locations are annotated as TempLocation in this phase.

D.4.3 Organization

Organizations tend to be defined either by straight lookup from the gazetteer lists, or, for the majority, by a combination of POS or case information and key words such as "company", "bank", "Services" "Ltd." etc. Many organizations are also identified by contextual information in the later phase org_context.jape. In this phase, organizations are annotated as TempOrganization.

D.4.4 Ambiguities

Some ambiguities are resolved immediately in this grammar, while others are left until later phases. For example, a Christian name followed by a possible Location is resolved by default to a person rather than a Location (e.g. "Ken London"). On the other hand, a Christian name followed by a possible organisation ending is resolved to an Organisation (e.g. "Alexandra Pottery"), though this is a slightly less sure rule.

D.4.5 Contextual information

Although most of the rules involving contextual information are invoked in a much later phase, there are a few which are invoked here, such as "X joined Y" where X is annotated as a Person and Y as an Organization. This is so that both annotations types can be handled at once.

D.5 name_post.jape

This grammar runs after the name grammar to fix some erroneous annotations that may have been created. Of course, a more elegant solution would be not to create the problem in the first instance, but this is a workaround. For example, if the surname of a Person contains certain stop words, e.g. "Mary And" then only the first name should be recognised as a Person. However, it might be that the firstname is also an Organization (and has been tagged with TempOrganization already), e.g. "U.N." If this is the case, then the annotation is left untouched, because this is correct.

D.6 date_pre.jape

This grammar precedes the date phase, because it includes extra context to prevent dates being recognised erroneously in the middle of longer expressions. It mainly treats the case where an expression is already tagged as a Person, but could also be tagged as a date (e.g. 16th Jan).

D.7 date.jape

This grammar contains the base rules for recognising times and dates. Given the complexity of potential patterns representing such expressions, there are a large number of rules and macros.

Although times and dates can be mutually ambiguous, we try to distinguish between them as early as possible. Dates, times and years are generally tagged separately (as TempDate, TempTime and TempYear respectively) and then recombined to form a final Date annotation in a later phase. This is because dates, times and years can be combined together in many different ways, and also because there can be much ambiguity between the three. For example, 1312 could be a time or a year, while 9-10 could be a span of time or date, or a fixed time or date.

D.8 reldate.jape

This grammar handles relative rather than absolute date and time sequences, such as "yesterday morning", "2 hours ago", "the first 9 months of the financial year"etc. It uses mainly explicit key words such as "ago" and items from the gazetteer lists.

D.9 number.jape

This grammar covers rules concerning money and percentages. The rules are fairly straightforward, using keywords from the gazetteer lists, and there is little ambiguity here, except for example where "Pound" can be money or weight, or where there is no explicit currency denominator.

D.10 address.jape

Rules for Address cover ip addresses, phone and fax numbers, and postal addresses. In general, these are not highly ambiguous, and can be covered with simple pattern matching, although phone numbers can require use of contextual information. Currenly only UK formats are really handled, though handling of foreign zipcodes and phone number formats is envisaged in future. The annotations produced are of type Email, Phone etc. and are then replaced in a later phase with final Address annotations with “phone” etc. as features.

D.11 url.jape

Rules for email addresses and Urls are in a separate grammar from the other address types, for the simple reason that SpaceTokens need to be identified for these rles to operate, whereas this is not necessary for the other Address types. For speed of processing, we place them in separate grammars so that SpaceTokens can be eliminated from the Input when they are not required.

D.12 identifier.jape

This grammar identifies ”Identifiers” which basically means any combination of numbers and letters acting as an ID, reference number etc. not recognised as any other entity type.

D.13 jobtitle.jape

This grammar simply identifies Jobtitles from the gazetteer lists, and adds a JobTitle annotation, which is used in later phases to aid recognition of other entity types such as Person and Organization. It may then be discarded in the Clean phase if not required as a final annotation type.

D.14 final.jape

This grammar uses the temporary annotations previously assigned in the earlier phases, and converts them into final annotations. The reason for this is that we need to be able to resolve ambiguities between different entity types, so we need to have all the different entity types handled in a single grammar somewhere. Ambiguities can be resolved using prioritisation techniques. Also, we may need to combine previously annotated elements, such as dates and times, into a single entity.

The rules in this grammar use Java code on the RHS to remove the existing temporary annotations, and replace them with new annotations. This is because we want to retain the features associated with the temporary annotations. For example, we might need to keep track of whether a person is male or female, or whether a location is a city or country. It also enables us to keep track of which rules have been used, for debugging purposes.

For the sake of obfuscation, although this phase is called final, it is not the final phase!

D.15 unknown.jape

This short grammar finds proper nouns not previously recognised, and gives them an Unknown annotation. This is then used by the namematcher – if an Unknown annotation can be matched with a previously categorised entity, its annotation is changed to that of the matched entity. Any remaining Unknown annotations are useful for debugging purposes, and can also be used as input for additional grammars or processing resources.

D.16 name_context.jape

This grammar looks for Unknown annotations occurring in certain contexts which indicate they might belong to Person. This is a typical example of a grammar that would benefit from learning or automatic context generation, because useful contexts are (a) hard to find manually and may require large volumes of training data, and (b) often very domain–specific. In this core grammar, we confine the use of contexts to fairly general uses, since this grammar should not be domain–dependent.

D.17 org_context.jape

This grammar operates on a similar principle to name_context.jape. It is slightly oriented towards business texts, so does not quite fulfil the generality criteria of the previous grammar. It does, however, provide some insight into more detailed use of contexts.¡/p¿

D.18 loc_context.jape

This grammar also operates in a similar manner to the preceding two, using general context such as coordinated pairs of locations, and hyponymic types of information.

D.19 clean.jape

This grammar comes last of all, and simply aims to clean up (remove) some of the temporary annotations that may not have been deleted along the way.

Appendix E

Part-of-Speech Tags used in the Hepple Tagger

CC - coordinating conjunction: ”and”, ”but”, ”nor”, ”or”, ”yet”, plus, minus, less, times (multiplication), over (division). Also ”for” (because) and ”so” (i.e., ”so that”).
CD - cardinal number
DT - determiner: Articles including ”a”, ”an”, ”every”, ”no”, ”the”, ”another”, ”any”, ”some”, ”those”.
EX - existential there: Unstressed ”there” that triggers inversion of the inflected verb and the logical subject; ”There was a party in progress”.
FW - foreign word
IN - preposition or subordinating conjunction
JJ - adjective: Hyphenated compounds that are used as modifiers; happy-go-lucky.
JJR - adjective - comparative: Adjectives with the comparative ending ”-er” and a comparative meaning. Sometimes ”more” and ”less”.
JJS - adjective - superlative: Adjectives with the superlative ending ”-est” (and ”worst”). Sometimes ”most”and ”least”.
JJSS - -unknown-, but probably a variant of JJS
-LRB- - -unknown-
LS - list item marker: Numbers and letters used as identifiers of items in a list.
MD - modal: All verbs that don’t take an ”-s” ending in the third person singular present: ”can”, ”could”, ”dare”, ”may”, ”might”, ”must”, ”ought”, ”shall”, ”should”, ”will”, ”would”.
NN - noun - singular or mass
NNP - proper noun - singular: All words in names usually are capitalized but titles might not be.
NNPS - proper noun - plural: All words in names usually are capitalized but titles might not be.
NNS - noun - plural
NP - proper noun - singular

NPS - proper noun - plural
PDT - predeterminer: Determinerlike elements preceding an article or possessive pronoun; "all/PDT his marbles", "quite/PDT a mess".
POS - possesive ending: Nouns ending in "'s" or "'".
PP - personal pronoun
PRPR$ - unknown-, but probably possessive pronoun
PRP - unknown-, but probably possessive pronoun
PRP$ - unknown, but probably possessive pronoun,such as "my", "your", "his", "his", "its", "one's", "our", and "their".
RB - adverb: most words ending in "-ly". Also "quite", "too", "very", "enough", "indeed", "not", "-n't", and "never".
RBR - adverb - comparative: adverbs ending with "-er" with a comparative meaning.
RBS - adverb - superlative
RP - particle: Mostly monosyllabic words that also double as directional adverbs.
STAART - start state marker (used internally)
SYM - symbol: technical symbols or expressions that aren't English words.
TO - literal to
UH - interjection: Such as "my", "oh", "please", "uh", "well", "yes".
VBD - verb - past tense: includes conditional form of the verb "to be"; "If I were/VBD rich...".
VBG - verb - gerund or present participle
VBN - verb - past participle
VBP - verb - non-3rd person singular present
VB - verb - base form: subsumes imperatives, infinitives and subjunctives.
VBZ - verb - 3rd person singular present
WDT - wh-determiner
WP$ - possesive wh-pronoun: includes "whose"
WP - wh-pronoun: includes "what", "who", and "whom".
WRB - wh-adverb: includes "how", "where", "why". Includes "when" when used in a temporal sense.

:: - literal colon
, - literal comma
$ - literal dollar sign
- - literal double-dash
- literal double quotes
- literal grave
(- literal left parenthesis
. - literal period
# - literal pound sign
) - literal right parenthesis
- literal single quote or apostrophe

Appendix F

Sample ML Configuration File

```
<?xml version="1.0" encoding="UTF-8"?>
<ML-CONFIG>
  <DATASET>
  <!-- The type of annotation used as instance -->
  <INSTANCE-TYPE>Token</INSTANCE-TYPE>
  <ATTRIBUTE>
    <!-- The name given to the attribute -->
    <NAME>Lookup(0)</NAME>
    <!-- The type of annotation used as attribute -->
    <TYPE>Lookup</TYPE>
    <!-- The position relative to the instance annotation -->
    <POSITION>0</POSITION>
  </ATTRIBUTE>


  <ATTRIBUTE>
    <!-- The name given to the attribute -->
    <NAME>Lookup_MT(-1)</NAME>
    <!-- The type of annotation used as attribute -->
    <TYPE>Lookup</TYPE>
    <!-- Optional: the feature name for the feature used to extract values
    for the attribute -->
    <FEATURE>majorType</FEATURE>

    <!-- The position relative to the instance annotation -->
    <POSITION>-1</POSITION>
    <!-- The list of permitted values.
    if present, marks a nominal attribute;
    if absent, the attribute is numeric (double)        -->
    <VALUES>
      <!-- One permitted value -->
      <VALUE>address</VALUE>
      <VALUE>cdg</VALUE>
      <VALUE>country_adj</VALUE>
      <VALUE>currency_unit</VALUE>
      <VALUE>date</VALUE>
      <VALUE>date_key</VALUE>
      <VALUE>date_unit</VALUE>
      <VALUE>facility</VALUE>
      <VALUE>facility_key</VALUE>
      <VALUE>facility_key_ext</VALUE>
      <VALUE>govern_key</VALUE>
      <VALUE>greeting</VALUE>
      <VALUE>ident_key</VALUE>
      <VALUE>jobtitle</VALUE>
      <VALUE>loc_general_key</VALUE>
      <VALUE>loc_key</VALUE>
      <VALUE>location</VALUE>
      <VALUE>number</VALUE>
      <VALUE>org_base</VALUE>
      <VALUE>org_ending</VALUE>
      <VALUE>org_key</VALUE>
      <VALUE>org_pre</VALUE>
      <VALUE>organization</VALUE>
      <VALUE>organization_noun</VALUE>
      <VALUE>person_ending</VALUE>
      <VALUE>person_first</VALUE>
      <VALUE>person_full</VALUE>
      <VALUE>phone_prefix</VALUE>
      <VALUE>sport</VALUE>
      <VALUE>spur</VALUE>
      <VALUE>spur_ident</VALUE>
      <VALUE>stop</VALUE>
      <VALUE>surname</VALUE>
      <VALUE>time</VALUE>
      <VALUE>time_modifier</VALUE>
      <VALUE>time_unit</VALUE>
      <VALUE>title</VALUE>
      <VALUE>year</VALUE>
    </VALUES>
    <!-- Optional: if present marks the attribute used as CLASS
    Only one attribute can be marked as class -->
  </ATTRIBUTE>
```

```
<ATTRIBUTE>
  <!-- The name given to the attribute -->
  <NAME>Lookup_MT(0)</NAME>
  <!-- The type of annotation used as attribute -->
  <TYPE>Lookup</TYPE>
  <!-- Optional: the feature name for the feature used to extract values
  for the attribute -->
  <FEATURE>majorType</FEATURE>

  <!-- The position relative to the instance annotation -->
  <POSITION>0</POSITION>
  <!-- The list of permitted values.
  if present, marks a nominal attribute;
  if absent, the attribute is numeric (double)        -->
  <VALUES>
    <!-- One permitted value -->
        <VALUE>address</VALUE>
    <VALUE>cdg</VALUE>
    <VALUE>country_adj</VALUE>
    <VALUE>currency_unit</VALUE>
    <VALUE>date</VALUE>
    <VALUE>date_key</VALUE>
    <VALUE>date_unit</VALUE>
    <VALUE>facility</VALUE>
    <VALUE>facility_key</VALUE>
    <VALUE>facility_key_ext</VALUE>
    <VALUE>govern_key</VALUE>
    <VALUE>greeting</VALUE>
    <VALUE>ident_key</VALUE>
    <VALUE>jobtitle</VALUE>
    <VALUE>loc_general_key</VALUE>
    <VALUE>loc_key</VALUE>
    <VALUE>location</VALUE>
    <VALUE>number</VALUE>
    <VALUE>org_base</VALUE>
    <VALUE>org_ending</VALUE>
    <VALUE>org_key</VALUE>
    <VALUE>org_pre</VALUE>
    <VALUE>organization</VALUE>
    <VALUE>organization_noun</VALUE>
    <VALUE>person_ending</VALUE>
    <VALUE>person_first</VALUE>
    <VALUE>person_full</VALUE>
```

```
      <VALUE>phone_prefix</VALUE>
      <VALUE>sport</VALUE>
      <VALUE>spur</VALUE>
      <VALUE>spur_ident</VALUE>
      <VALUE>stop</VALUE>
      <VALUE>surname</VALUE>
      <VALUE>time</VALUE>
      <VALUE>time_modifier</VALUE>
      <VALUE>time_unit</VALUE>
      <VALUE>title</VALUE>
      <VALUE>year</VALUE>
    </VALUES>
    <!-- Optional: if present marks the attribute used as CLASS
    Only one attribute can be marked as class -->
  </ATTRIBUTE>

  <ATTRIBUTE>
    <!-- The name given to the attribute -->
    <NAME>Lookup_MT(1)</NAME>
    <!-- The type of annotation used as attribute -->
    <TYPE>Lookup</TYPE>
    <!-- Optional: the feature name for the feature used to extract values
    for the attribute -->
    <FEATURE>majorType</FEATURE>

    <!-- The position relative to the instance annotation -->
    <POSITION>1</POSITION>

    <!-- The list of permitted values.
    if present, marks a nominal attribute;
    if absent, the attribute is numeric (double)        -->
    <VALUES>
      <!-- One permitted value -->
      <VALUE>address</VALUE>
      <VALUE>cdg</VALUE>
      <VALUE>country_adj</VALUE>
      <VALUE>currency_unit</VALUE>
      <VALUE>date</VALUE>
      <VALUE>date_key</VALUE>
      <VALUE>date_unit</VALUE>
      <VALUE>facility</VALUE>
      <VALUE>facility_key</VALUE>
      <VALUE>facility_key_ext</VALUE>
      <VALUE>govern_key</VALUE>
      <VALUE>greeting</VALUE>
      <VALUE>ident_key</VALUE>
      <VALUE>jobtitle</VALUE>
      <VALUE>loc_general_key</VALUE>
      <VALUE>loc_key</VALUE>
      <VALUE>location</VALUE>
      <VALUE>number</VALUE>
      <VALUE>org_base</VALUE>
      <VALUE>org_ending</VALUE>
      <VALUE>org_key</VALUE>
      <VALUE>org_pre</VALUE>
      <VALUE>organization</VALUE>
      <VALUE>organization_noun</VALUE>
      <VALUE>person_ending</VALUE>
      <VALUE>person_first</VALUE>
      <VALUE>person_full</VALUE>
      <VALUE>phone_prefix</VALUE>
      <VALUE>sport</VALUE>
      <VALUE>spur</VALUE>
      <VALUE>spur_ident</VALUE>
      <VALUE>stop</VALUE>
      <VALUE>surname</VALUE>
      <VALUE>time</VALUE>
      <VALUE>time_modifier</VALUE>
      <VALUE>time_unit</VALUE>
      <VALUE>title</VALUE>
      <VALUE>year</VALUE>
    </VALUES>
    <!-- Optional: if present marks the attribute used as CLASS
    Only one attribute can be marked as class -->
  </ATTRIBUTE>

  <ATTRIBUTE>
    <!-- The name given to the attribute -->
    <NAME>POS_category(-1)</NAME>
    <!-- The type of annotation used as attribute -->
    <TYPE>Token</TYPE>
    <!-- Optional: the feature name for the feature used to extract values
    for the attribute -->
    <FEATURE>category</FEATURE>

    <!-- The position relative to the instance annotation -->
    <POSITION>-1</POSITION>
```

```
<!-- The list of permitted values.
if present, marks a nominal attribute;
if absent, the attribute is numeric (double)        -->
<VALUES>
  <!-- One permitted value -->
    <VALUE>NN</VALUE>
    <VALUE>NNP</VALUE>
    <VALUE>NNPS</VALUE>
    <VALUE>NNS</VALUE>
    <VALUE>NP</VALUE>
    <VALUE>NPS</VALUE>
    <VALUE>JJ</VALUE>
    <VALUE>JJR</VALUE>
    <VALUE>JJS</VALUE>
    <VALUE>JJSS</VALUE>
    <VALUE>RB</VALUE>
    <VALUE>RBR</VALUE>
    <VALUE>RBS</VALUE>
    <VALUE>VB</VALUE>
    <VALUE>VBD</VALUE>
    <VALUE>VBG</VALUE>
    <VALUE>VBN</VALUE>
    <VALUE>VBP</VALUE>
    <VALUE>VBZ</VALUE>
    <VALUE>FW</VALUE>
    <VALUE>CD</VALUE>
    <VALUE>CC</VALUE>
    <VALUE>DT</VALUE>
    <VALUE>EX</VALUE>
    <VALUE>IN</VALUE>
    <VALUE>LS</VALUE>
    <VALUE>MD</VALUE>
    <VALUE>PDT</VALUE>
    <VALUE>POS</VALUE>
    <VALUE>PP</VALUE>
    <VALUE>PRP</VALUE>
    <VALUE>PRP$</VALUE>
    <VALUE>PRPR$</VALUE>
    <VALUE>RP</VALUE>
    <VALUE>TO</VALUE>
    <VALUE>UH</VALUE>
    <VALUE>WDT</VALUE>
    <VALUE>WP</VALUE>
    <VALUE>WP$</VALUE>
```

```
        <VALUE>WRB</VALUE>
        <VALUE>SYM</VALUE>
        <VALUE>\"</VALUE>
        <VALUE>#</VALUE>
        <VALUE>$</VALUE>
        <VALUE>'</VALUE>
        <VALUE>(</VALUE>
        <VALUE>)</VALUE>
        <VALUE>,</VALUE>
        <VALUE>--</VALUE>
        <VALUE>-LRB-</VALUE>
        <VALUE>.</VALUE>
        <VALUE>''</VALUE>
        <VALUE>:</VALUE>
        <VALUE>::</VALUE>
        <VALUE>`</VALUE>
    </VALUES>
    <!-- Optional: if present marks the attribute used as CLASS
    Only one attribute can be marked as class -->
  </ATTRIBUTE>

  <ATTRIBUTE>
    <!-- The name given to the attribute -->
    <NAME>POS_category(0)</NAME>
    <!-- The type of annotation used as attribute -->
    <TYPE>Token</TYPE>
    <!-- Optional: the feature name for the feature used to extract values
    for the attribute -->
    <FEATURE>category</FEATURE>

    <!-- The position relative to the instance annotation -->
    <POSITION>0</POSITION>

    <!-- The list of permitted values.
    if present, marks a nominal attribute;
    if absent, the attribute is numeric (double)        -->
    <VALUES>
      <!-- One permitted value -->
        <VALUE>NN</VALUE>
        <VALUE>NNP</VALUE>
        <VALUE>NNPS</VALUE>
        <VALUE>NNS</VALUE>
        <VALUE>NP</VALUE>
        <VALUE>NPS</VALUE>
        <VALUE>JJ</VALUE>
        <VALUE>JJR</VALUE>
        <VALUE>JJS</VALUE>
        <VALUE>JJSS</VALUE>
        <VALUE>RB</VALUE>
        <VALUE>RBR</VALUE>
        <VALUE>RBS</VALUE>
        <VALUE>VB</VALUE>
        <VALUE>VBD</VALUE>
        <VALUE>VBG</VALUE>
        <VALUE>VBN</VALUE>
        <VALUE>VBP</VALUE>
        <VALUE>VBZ</VALUE>
        <VALUE>FW</VALUE>
        <VALUE>CD</VALUE>
        <VALUE>CC</VALUE>
        <VALUE>DT</VALUE>
        <VALUE>EX</VALUE>
        <VALUE>IN</VALUE>
        <VALUE>LS</VALUE>
        <VALUE>MD</VALUE>
        <VALUE>PDT</VALUE>
        <VALUE>POS</VALUE>
        <VALUE>PP</VALUE>
        <VALUE>PRP</VALUE>
        <VALUE>PRP$</VALUE>
        <VALUE>PRPR$</VALUE>
        <VALUE>RP</VALUE>
        <VALUE>TO</VALUE>
        <VALUE>UH</VALUE>
        <VALUE>WDT</VALUE>
        <VALUE>WP</VALUE>
        <VALUE>WP$</VALUE>
        <VALUE>WRB</VALUE>
        <VALUE>SYM</VALUE>
        <VALUE>\"</VALUE>
        <VALUE>#</VALUE>
        <VALUE>$</VALUE>
        <VALUE>'</VALUE>
        <VALUE>(</VALUE>
        <VALUE>)</VALUE>
        <VALUE>,</VALUE>
        <VALUE>--</VALUE>
        <VALUE>-LRB-</VALUE>
        <VALUE>.</VALUE>
        <VALUE>''</VALUE>
        <VALUE>:</VALUE>
        <VALUE>::</VALUE>
        <VALUE>`</VALUE>
    </VALUES>
    <!-- Optional: if present marks the attribute used as CLASS
    Only one attribute can be marked as class -->
  </ATTRIBUTE>

  <ATTRIBUTE>
    <!-- The name given to the attribute -->
    <NAME>POS_category(1)</NAME>
    <!-- The type of annotation used as attribute -->
    <TYPE>Token</TYPE>
    <!-- Optional: the feature name for the feature used to extract values
    for the attribute -->
    <FEATURE>category</FEATURE>

    <!-- The position relative to the instance annotation -->
    <POSITION>1</POSITION>

    <!-- The list of permitted values.
    if present, marks a nominal attribute;
    if absent, the attribute is numeric (double)        -->
    <VALUES>
      <!-- One permitted value -->
        <VALUE>NN</VALUE>
        <VALUE>NNP</VALUE>
        <VALUE>NNPS</VALUE>
        <VALUE>NNS</VALUE>
        <VALUE>NP</VALUE>
        <VALUE>NPS</VALUE>
        <VALUE>JJ</VALUE>
        <VALUE>JJR</VALUE>
        <VALUE>JJS</VALUE>
        <VALUE>JJSS</VALUE>
        <VALUE>RB</VALUE>
        <VALUE>RBR</VALUE>
        <VALUE>RBS</VALUE>
        <VALUE>VB</VALUE>
        <VALUE>VBD</VALUE>
        <VALUE>VBG</VALUE>
        <VALUE>VBN</VALUE>
        <VALUE>VBP</VALUE>
        <VALUE>VBZ</VALUE>
        <VALUE>FW</VALUE>
        <VALUE>CD</VALUE>
        <VALUE>CC</VALUE>
        <VALUE>DT</VALUE>
        <VALUE>EX</VALUE>
        <VALUE>IN</VALUE>
        <VALUE>LS</VALUE>
        <VALUE>MD</VALUE>
        <VALUE>PDT</VALUE>
        <VALUE>POS</VALUE>
        <VALUE>PP</VALUE>
        <VALUE>PRP</VALUE>
        <VALUE>PRP$</VALUE>
        <VALUE>PRPR$</VALUE>
        <VALUE>RP</VALUE>
        <VALUE>TO</VALUE>
        <VALUE>UH</VALUE>
        <VALUE>WDT</VALUE>
        <VALUE>WP</VALUE>
        <VALUE>WP$</VALUE>
        <VALUE>WRB</VALUE>
        <VALUE>SYM</VALUE>
        <VALUE>\"</VALUE>
        <VALUE>#</VALUE>
        <VALUE>$</VALUE>
        <VALUE>'</VALUE>
        <VALUE>(</VALUE>
        <VALUE>)</VALUE>
        <VALUE>,</VALUE>
        <VALUE>--</VALUE>
        <VALUE>-LRB-</VALUE>
        <VALUE>.</VALUE>
        <VALUE>''</VALUE>
        <VALUE>:</VALUE>
        <VALUE>::</VALUE>
        <VALUE>`</VALUE>
    </VALUES>
    <!-- Optional: if present marks the attribute used as CLASS
    Only one attribute can be marked as class -->
  </ATTRIBUTE>

  <ATTRIBUTE>
```

```
    <!-- The name given to the attribute -->
    <NAME>Entity(0)</NAME>
    <!-- The type of annotation used as attribute -->
    <TYPE>Entity</TYPE>
    <!-- The position relative to the instance annotation -->
    <POSITION>0</POSITION>

    <CLASS/>
    <!-- Optional: if present marks the attribute used as CLASS
    Only one attribute can be marked as class -->
  </ATTRIBUTE>


  </DATASET>

  <ENGINE>
    <WRAPPER>gate.creole.ml.weka.Wrapper</WRAPPER>
    <OPTIONS>
        <CLASSIFIER OPTIONS="-S -C 0.25 -B -M 2">weka.classifiers.trees.J48</CLASSIF
        <CONFIDENCE-THRESHOLD>0.85</CONFIDENCE-THRESHOLD>
    </OPTIONS>
  </ENGINE>
</ML-CONFIG>
```

Appendix G

IAA Measures for Classification Tasks

IAA has been used mainly in the classification tasks, where two or more annotators are given a set of instances and are asked to classify those instances into some pre-defined categories. IAA measures the agreements among the annotators on the class labels assigned to the instances by the annotators. Text classification tasks include document classification, sentence classification(e.g. opinionated sentence recognition), and token classification (e.g. POS tagging).

The three commonly used IAA measures are *observed agreement*, *specific agreement*, and *Kappa (κ)* [Hripcsak & Heitjan 02]. Those measures can be calculated from a contingency table, which lists the numbers of instances of agreement and disagreement between two annotators on each category. To explain the IAA measures, a general contingency table for two categories *cat1* and *cat2* is shown in Table G.1.

Table G.1: Contingency table for two-category problem

	Annotator-2		
Annotator-1	cat1	cat2	marginal sum
cat1	a	b	a+b
cat2	c	d	c+d
marginal sum	a+c	b+d	a+b+c+d

Observed agreement is the portion of the instances on which the annotators agree. For the two annotators and two categories as shown in Table G.1, it is defined as

$$A_o = \frac{a+d}{a+b+c+d} \tag{G.1}$$

The extension of the above formula to more than two categories is straightforward. The extension to more than two annotators is usually taken as the mean of the pair-wise agreements [Fleiss 75], which is the average agreement across all possible pairs of annotators. An alternative compares each annotator with the majority opinion of the others [Fleiss 75].

However, the observed agreement has two shortcomings. One is that a certain amount of agreement is expected by chance. The Kappa measure is a chance-corrected agreement. Another is that it sums up the agreement on all the categories, but the agreements on each category may differ. Hence the category specific agreement is needed.

Specific agreement quantifies the degree of agreement for each of the categories separately. For example, the specific agreement for the two categories list in Table G.1 is the following, respectively,

$$A_{cat1} = \frac{2a}{2a+b+c}; \quad A_{cat2} = \frac{2d}{b+c+2d} \tag{G.2}$$

Kappa is defined as the observed agreements A_o minus the agreement expected by chance A_e and is normalized as a number between -1 and 1.

$$\kappa = \frac{A_o - A_e}{1 - A_e} \tag{G.3}$$

$\kappa = 1$ means perfect agreements, $\kappa = 0$ means the agreement is equal to chance, $\kappa = -1$ means "perfect" disagreement.

There are two different ways of computing the chance agreement A_e (for a detailed explanations about it see [Eugenio & Glass 04]). The Cohen's Kappa is based on the individual distribution of each annotator, while the Siegel & Castellan's Kappa is based on the assumption that all the annotators have the same distribution. The former is more informative than the latter and has been used widely.

The Kappa suffers from the prevalence problem which arises because imbalanced distribution of categories in the data increases A_e. The prevalence problem can be alleviated by reporting the positive and negative specified agreement on each category besides the Kappa [Hripcsak & Heitjan 02, Eugenio & Glass 04]. In addition, the so-called bias problem affects the Cohen's Kappa, but not S&C's. The bias problem arises as one annotator prefers one particular category more than another annotator. [Eugenio & Glass 04] advised to compute the S&C's Kappa and the specific agreements along with the Cohen's Kappa in order to handle these problems.

Despite the problem mentioned above, the Cohen's Kappa remains a popular IAA measure. Kappa can be used for more than two annotators based on pair-wise figures, e.g. the mean of all the pair-wise Kappa as an overall Kappa measure. The Cohen's Kappa can also be extended to the case of more than two annotators by using the following single formula [Davies & Fleiss 82]

$$\kappa = 1 - \frac{IJ^2 - \sum_i \sum_c Y_{ic}^2}{I(J(J-1)\sum_c (p_c(1-p_c)) + \sum_c \sum_j (p_{cj} - p_c)^2)} \tag{G.4}$$

Where I and J are the number of instances and annotators, respectively; Y_{ic} is the number of annotators who assigns the category c to the instance I; p_{cj} is the probability of the annotator j assigning category c; p_c is the probability of assigning category by all annotators (i.e. averaging p_{cj} over all annotators).

S&C's Kappa is applicable for any number of annotators. S&C's Kappa for two annotators is also known as Scott's Pi (see [Lombard *et al.* 02]). The Krippendorff's alpha, another variant of Kappa, differs only slightly from the S&C's Kappa on nominal category problem (see [Carletta 96, Eugenio & Glass 04]).

However, note that the Kappa (and the observed agreement) is not applicable to some tasks. Named entity annotation is one such task [Hripcsak & Rothschild 05]. In the named entity annotation task, annotators are given some text and are asked to annotate some named entities (and possibly their categories) in the text. Different annotators may annotate different instances of the named entity. So, if one annotator annotates one named entity in the text but another annotator does not annotate it, then that named entity is a non-entity for the latter. However, generally the non-entity in the text is not a well-defined term, e.g. we don't know how many words should be contained in the non-entity. On the other hand, if we want to compute Kappa for named entity annotation, we need the non-entities. This is why people don't compute Kappa for the named entity task.

Appendix H

Keyboard shortcuts for GATE

This chapter describes the keyboard shortcuts used in GATE core components.

General (section 3.6)	
F1	Display an help page for the selected component
Alt+F4	Exit the application without confirmation
Tab	Put the focus on the next component or frame
Shift+Tab	Put the focus on the previous component or frame
F6	Put the focus on the next frame
Shift+F6	Put the focus on the previous frame
Alt+F	Show the File menu
Alt+O	Show the Options menu
Alt+T	Show the Tools menu
Alt+H	Show the Help menu
F10	Show the first menu

Resources tree (section 3.6)	
Enter	Show the selected resources
Ctrl+H	Hide the selected resource
Ctrl+Shift+H	Hide all the resources
F2	Rename the selected resource
Ctrl+F4	Close the selected resource

Document editor (section 5.3)	
Ctrl+F	Show the search dialog for the document
Ctrl+S	Save the document in a file
F3	Show/Hide the annotation sets
Shift+F3	Show the annotation sets with preselection
F4	Show/Hide the annotations list
F5	Show/Hide the coreference editor
F7	Show/Hide the text

Annotation editor	
Right/Left	Grow/Shrink the annotation span at its start
Alt+Right/Alt+Left	Grow/Shrink the annotation span at its end
+Shift/+Ctrl+Shift	Use a span increment of 5/10 characters
Alt+Delete	Delete the currently edited annotation

Annic/Lucene datastore (section 9.29)	
Alt+Enter	Search the expression in the datastore
Alt+Backspace	Delete the search expression
Alt+Right	Display the next page of results
Alt+Left	Display the row manager
Alt+E	Export the results to a file

Search expression text field	
Ctrl+Enter	Insert a new line
Enter	Search the expression
Alt+Top	Select the previous result
Alt+Bottom	Select the next result

Table H.1: GATE keyboard shortcuts

References

[Appelt 99]
D. E. Appelt. An Introduction to Information Extraction. *Artificial Intelligence Communications*, 12(3):161–172, 1999.

[Aswani *et al.* 05]
N. Aswani, V. Tablan, K. Bontcheva, and H. Cunningham. Indexing and Querying Linguistic Metadata and Document Content. In *Proceedings of Fifth International Conference on Recent Advances in Natural Language Processing (RANLP2005)*, Borovets, Bulgaria, 2005.

[Aswani *et al.* 06]
N. Aswani, K. Bontcheva, and H. Cunningham. Mining information for instance unification. In *5th International Semantic Web Conference (ISWC2006)*, Athens, Georgia, USA, 2006.

[Azar 89]
S. Azar. *Understanding and Using English Grammar.* Prentice Hall Regents, 1989.

[Baker *et al.* 02]
P. Baker, A. Hardie, T. McEnery, H. Cunningham, and R. Gaizauskas. EMILLE, A 67-Million Word Corpus of Indic Languages: Data Collection, Mark-up and Harmonisation. In *Proceedings of 3rd Language Resources and Evaluation Conference (LREC'2002)*, pages 819–825, 2002.

[Bird & Liberman 99]
S. Bird and M. Liberman. A Formal Framework for Linguistic Annotation. Technical Report MS-CIS-99-01, Department of Computer and Information Science, University of Pennsylvania, 1999. `http://xxx.lanl.gov/abs/cs.CL/9903003`.

[Bontcheva 04]
K. Bontcheva. Open-source Tools for Creation, Maintenance, and Storage of Lexical Resources for Language Generation from Ontologies. In *Proceedings of 4th Language Resources and Evaluation Conference (LREC'04)*, 2004.

[Bontcheva *et al.* 00]
K. Bontcheva, H. Brugman, A. Russel, P. Wittenburg, and H. Cunningham. An

Experiment in Unifying Audio-Visual and Textual Infrastructures for Language Processing R&D. In *Proceedings of the Workshop on Using Toolsets and Architectures To Build NLP Systems at COLING-2000*, Luxembourg, 2000. `http://gate.ac.uk/`.

[Bontcheva *et al.* 02a]
K. Bontcheva, H. Cunningham, V. Tablan, D. Maynard, and O. Hamza. Using GATE as an Environment for Teaching NLP. In *Proceedings of the ACL Workshop on Effective Tools and Methodologies in Teaching NLP*, 2002. `http://gate.ac.uk/sale/acl02/gate4teaching.pdf`.

[Bontcheva *et al.* 02b]
K. Bontcheva, H. Cunningham, V. Tablan, D. Maynard, and H. Saggion. Developing Reusable and Robust Language Processing Components for Information Systems using GATE. In *Proceedings of the 3rd International Workshop on Natural Language and Information Systems (NLIS'2002)*, Aix-en-Provence, France, 2002. IEEE Computer Society Press. `http://gate.ac.uk/sale/nlis/nlis.ps`.

[Bontcheva *et al.* 02c]
K. Bontcheva, M. Dimitrov, D. Maynard, V. Tablan, and H. Cunningham. Shallow Methods for Named Entity Coreference Resolution. In *Chaînes de références et résolveurs d'anaphores, workshop TALN 2002*, Nancy, France, 2002. `http://gate.ac.uk/sale/taln02/taln-ws-coref.pdf`.

[Bontcheva *et al.* 03]
K. Bontcheva, A. Kiryakov, H. Cunningham, B. Popov, and M. Dimitrov. Semantic web enabled, open source language technology. In *EACL workshop on Language Technology and the Semantic Web: NLP and XML*, Budapest, Hungary, 2003. `http://gate.ac.uk/sale/eacl03-semweb/bontcheva-etal-final.pdf`.

[Bontcheva *et al.* 04]
K. Bontcheva, V. Tablan, D. Maynard, and H. Cunningham. Evolving GATE to Meet New Challenges in Language Engineering. *Natural Language Engineering*, 10(3/4):349—373, 2004.

[Booch 94]
G. Booch. *Object-Oriented Analysis and Design 2nd Edn.* Benjamin/Cummings, 1994.

[Brugman *et al.* 99]
H. Brugman, K. Bontcheva, P. Wittenburg, and H. Cunningham. Integrating Multimedia and Textual Software Architectures for Language Technology. Technical report MPI-TG-99-1, Max-Planck Institute for Psycholinguistics, Nijmegen, Netherlands, 1999.

[Campione *et al.* 98]
M. Campione, K. Walrath, A. Huml, and the Tutuorial Team. *The Java Tutorial Continued: The Rest of the JDK.* Addison-Wesley, Reading, MA, 1998.

[Carletta 96]
J. Carletta. Assessing agreement on classification tasks: the Kappa statistic. *Computational Linguistics*, 22(2):249–254, 1996.

[CC001]
LIBSVM: a library for support vector machines, 2001. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

[Chinchor 92]
N. Chinchor. Muc-4 evaluation metrics. In *Proceedings of the Fourth Message Understanding Conference*, pages 22–29, 1992.

[Cimiano *et al.* 03]
P. Cimiano, S.Staab, and J. Tane. Automatic Acquisition of Taxonomies from Text: FCA meets NLP. In *Proceedings of the ECML/PKDD Workshop on Adaptive Text Extraction and Mining*, pages 10–17, Cavtat-Dubrovnik, Croatia, 2003.

[Cobuild 99]
C. Cobuild, editor. *English Grammar*. Harper Collins, 1999.

[Cowie & Lehnert 96]
J. Cowie and W. Lehnert. Information Extraction. *Communications of the ACM*, 39(1):80–91, 1996.

[Cunningham & Bontcheva 05]
H. Cunningham and K. Bontcheva. Computational Language Systems, Architectures. *Encyclopedia of Language and Linguistics, 2nd Edition*, pages 733–752, 2005.

[Cunningham & Scott 04a]
H. Cunningham and D. Scott. Introduction to the Special Issue on Software Architecture for Language Engineering. *Natural Language Engineering*, 2004. `http://gate.ac.uk/sale/jnle-sale/intro/intro-main.pdf`.

[Cunningham & Scott 04b]
H. Cunningham and D. Scott, editors. *Special Issue of Natural Language Engineering on Software Architecture for Language Engineering*. Cambridge University Press, 2004.

[Cunningham 94]
H. Cunningham. Support Software for Language Engineering Research. Technical Report 94/05, Centre for Computational Linguistics, UMIST, Manchester, 1994.

[Cunningham 99a]
H. Cunningham. A Definition and Short History of Language Engineering. *Journal of Natural Language Engineering*, 5(1):1–16, 1999.

[Cunningham 99b]
H. Cunningham. Information Extraction: a User Guide (revised version). Research Memorandum CS–99–07, Department of Computer Science, University of Sheffield, May 1999.

[Cunningham 99c]
H. Cunningham. JAPE: a Java Annotation Patterns Engine. Research Memorandum CS–99–06, Department of Computer Science, University of Sheffield, May 1999.

[Cunningham 00]
H. Cunningham. *Software Architecture for Language Engineering.* Unpublished PhD thesis, University of Sheffield, 2000. `http://gate.ac.uk/sale/thesis/`.

[Cunningham 02]
H. Cunningham. GATE, a General Architecture for Text Engineering. *Computers and the Humanities*, 36:223–254, 2002.

[Cunningham 05]
H. Cunningham. Information Extraction, Automatic. *Encyclopedia of Language and Linguistics, 2nd Edition*, pages 665–677, 2005.

[Cunningham *et al.* 94]
H. Cunningham, M. Freeman, and W. Black. Software Reuse, Object-Oriented Frameworks and Natural Language Processing. In *New Methods in Language Processing (NeMLaP-1), September 1994*, Manchester, 1994. (Re-published in book form 1997 by UCL Press).

[Cunningham *et al.* 95]
H. Cunningham, R. Gaizauskas, and Y. Wilks. A General Architecture for Text Engineering (GATE) – a new approach to Language Engineering R&D. Technical Report CS–95–21, Department of Computer Science, University of Sheffield, 1995. `http://xxx.lanl.gov/abs/cs.CL/9601009`.

[Cunningham *et al.* 96a]
H. Cunningham, K. Humphreys, R. Gaizauskas, and M. Stower. CREOLE Developer's Manual. Technical report, Department of Computer Science, University of Sheffield, 1996. `http://www.dcs.shef.ac.uk/nlp/gate`.

[Cunningham *et al.* 96b]
H. Cunningham, K. Humphreys, R. Gaizauskas, and Y. Wilks. TIPSTER-Compatible Projects at Sheffield. In *Advances in Text Processing, TIPSTER Program Phase II.* DARPA, Morgan Kaufmann, California, 1996.

[Cunningham *et al.* 96c]
H. Cunningham, Y. Wilks, and R. Gaizauskas. GATE – a General Architecture for Text Engineering. In *Proceedings of the 16th Conference on Computational Linguistics (COLING-96)*, Copenhagen, August 1996. `ftp://ftp.dcs.shef.ac.uk/home/hamish/auto_papers/Cun96b.ps`.

[Cunningham *et al.* 96d]
H. Cunningham, Y. Wilks, and R. Gaizauskas. Software Infrastructure for Language Engineering. In *Proceedings of the AISB Workshop on Language Engineering for Document Analysis and Recognition*, Brighton, U.K., April 1996.

[Cunningham *et al.* 96e]
H. Cunningham, Y. Wilks, and R. Gaizauskas. New Methods, Current Trends and Software Infrastructure for NLP. In *Proceedings of the Conference on New Methods in Natural Language Processing (NeMLaP-2)*, Bilkent University, Turkey, September 1996. `ftp://ftp.dcs.shef.ac.uk/home/hamish/auto_papers/Cun96c.ps`.

[Cunningham *et al.* 97a]
H. Cunningham, K. Humphreys, R. Gaizauskas, and Y. Wilks. GATE – a TIPSTER-based General Architecture for Text Engineering. In *Proceedings of the TIPSTER Text Program (Phase III) 6 Month Workshop*. DARPA, Morgan Kaufmann, California, May 1997. `ftp://ftp.dcs.shef.ac.uk/home/hamish/auto_papers/Cun97e.ps`.

[Cunningham *et al.* 97b]
H. Cunningham, K. Humphreys, R. Gaizauskas, and Y. Wilks. Software Infrastructure for Natural Language Processing. In *Proceedings of the 5th Conference on Applied Natural Language Processing (ANLP-97)*, March 1997. `ftp://ftp.dcs.shef.ac.uk/home/hamish/auto_papers/Cun97a.ps.gz`.

[Cunningham *et al.* 98a]
H. Cunningham, W. Peters, C. McCauley, K. Bontcheva, and Y. Wilks. A Level Playing Field for Language Resource Evaluation. In *Workshop on Distributing and Accessing Lexical Resources at Conference on Language Resources Evaluation, Granada, Spain*, 1998. `http://www.dcs.shef.ac.uk/ hamish/dalr`.

[Cunningham *et al.* 98b]
H. Cunningham, M. Stevenson, and Y. Wilks. Implementing a Sense Tagger within a General Architecture for Language Engineering. In *Proceedings of the Third Conference on New Methods in Language Engineering (NeMLaP-3)*, pages 59–72, Sydney, Australia, 1998.

[Cunningham *et al.* 99]
H. Cunningham, R. Gaizauskas, K. Humphreys, and Y. Wilks. Experience with a Language Engineering Architecture: Three Years of GATE. In *Proceedings of the AISB'99 Workshop on Reference Architectures and Data Standards for NLP*, Edinburgh, April 1999. The Society for the Study of Artificial Intelligence and Simulation of Behaviour. `http://www.dcs.shef.ac.uk/ hamish/GateAisb99.html`.

[Cunningham *et al.* 00a]
H. Cunningham, K. Bontcheva, W. Peters, and Y. Wilks. Uniform language resource access and distribution in the context of a General Architecture for Text Engineering (GATE). In *Proceedings of the Workshop on Ontolo-*

gies and Language Resources (OntoLex'2000), Sozopol, Bulgaria, September 2000. http://gate.ac.uk/sale/ontolex/ontolex.ps.

[Cunningham *et al.* 00b]
H. Cunningham, K. Bontcheva, V. Tablan, and Y. Wilks. Software Infrastructure for Language Resources: a Taxonomy of Previous Work and a Requirements Analysis. In *Proceedings of the 2nd International Conference on Language Resources and Evaluation (LREC-2)*, Athens, 2000. http://gate.ac.uk/.

[Cunningham *et al.* 00c]
H. Cunningham, D. Maynard, K. Bontcheva, V. Tablan, and Y. Wilks. Experience of using GATE for NLP R&D. In *Proceedings of the Workshop on Using Toolsets and Architectures To Build NLP Systems at COLING-2000*, Luxembourg, 2000. http://gate.ac.uk/.

[Cunningham *et al.* 00d]
H. Cunningham, D. Maynard, and V. Tablan. JAPE: a Java Annotation Patterns Engine (Second Edition). Research Memorandum CS–00–10, Department of Computer Science, University of Sheffield, November 2000.

[Cunningham *et al.* 02]
H. Cunningham, D. Maynard, K. Bontcheva, and V. Tablan. GATE: A Framework and Graphical Development Environment for Robust NLP Tools and Applications. In *Proceedings of the 40th Anniversary Meeting of the Association for Computational Linguistics (ACL'02)*, 2002.

[Cunningham *et al.* 03]
H. Cunningham, V. Tablan, K. Bontcheva, and M. Dimitrov. Language Engineering Tools for Collaborative Corpus Annotation. In *Proceedings of Corpus Linguistics 2003*, Lancaster, UK, 2003. http://gate.ac.uk/sale/cl03/distrib-ollie-cl03.doc.

[Davies & Fleiss 82]
M. Davies and J. Fleiss. Measuring Agreement for Multinomial Data. *Biometrics*, 38:1047–1051, 1982.

[Dean *et al.* 04]
M. Dean, G. Schreiber, S. Bechhofer, F. van Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Stein. OWL web ontology language reference. W3C recommendation, W3C, Feb 2004. http://www.w3.org/TR/owl-ref/.

[Dimitrov 02a]
M. Dimitrov. *A Light-weight Approach to Coreference Resolution for Named Entities in Text.* MSc Thesis, University of Sofia, Bulgaria, 2002. http://www.ontotext.com/ie/thesis-m.pdf.

[Dimitrov 02b]
M. Dimitrov. *A Light-weight Approach to Coreference Resolution for Named Entities in Text.* MSc Thesis, University of Sofia, Bulgaria, 2002. `http://www.ontotext.com/ie/thesis-m.pdf`.

[Dimitrov *et al.* 02]
M. Dimitrov, K. Bontcheva, H. Cunningham, and D. Maynard. A Light-weight Approach to Coreference Resolution for Named Entities in Text. In *Proceedings of the Fourth Discourse Anaphora and Anaphor Resolution Colloquium (DAARC)*, Lisbon, 2002.

[Dimitrov *et al.* 04]
M. Dimitrov, K. Bontcheva, H. Cunningham, and D. Maynard. A Light-weight Approach to Coreference Resolution for Named Entities in Text. In A. Branco, T. McEnery, and R. Mitkov, editors, *Anaphora Processing: Linguistic, Cognitive and Computational Modelling.* John Benjamins, 2004.

[Dowman *et al.* 05a]
M. Dowman, V. Tablan, H. Cunningham, and B. Popov. Content augmentation for mixed-mode news broadcasts. In *Proceedings of the 3rd European Conference on Interactive Television: User Centred ITV Systems, Programmes and Applications*, Aalborg University, Denmark, 2005. `http://gate.ac.uk/sale/euro-itv-2005/content-augmentation-for-mixed-mode-news-broad`

[Dowman *et al.* 05b]
M. Dowman, V. Tablan, H. Cunningham, and B. Popov. Web-assisted annotation, semantic indexing and search of television and radio news. In *Proceedings of the 14th International World Wide Web Conference*, Chiba, Japan, 2005. `http://gate.ac.uk/sale/www05/web-assisted-annotation.pdf`.

[Dowman *et al.* 05c]
M. Dowman, V. Tablan, H. Cunningham, C. Ursu, and B. Popov. Semantically enhanced television news through web and video integration. In *Second European Semantic Web Conference (ESWC'2005)*, 2005.

[Eugenio & Glass 04]
B. D. Eugenio and M. Glass. The kappa statistic: a second look. *Computational Linguistics*, 1(30), 2004. (squib).

[Fleiss 75]
J. L. Fleiss. Measuring agreement between two judges on the presence or absence of a trait. *Biometrics*, 31:651–659, 1975.

[Frakes & Baeza-Yates 92]
W. Frakes and R. Baeza-Yates, editors. *Information retrieval, data structures and algorithms.* Prentice Hall, New York, Englewood Cliffs, N.J., 1992.

[Gaizauskas & Wilks 98]
R. Gaizauskas and Y. Wilks. Information Extraction: Beyond Document Retrieval. *Journal of Documentation*, 54(1):70–105, 1998.

[Gaizauskas *et al.* 96a]
R. Gaizauskas, P. Rodgers, H. Cunningham, and K. Humphreys. GATE User Guide. `http://www.dcs.shef.ac.uk/nlp/gate`, 1996.

[Gaizauskas *et al.* 96b]
R. Gaizauskas, H. Cunningham, Y. Wilks, P. Rodgers, and K. Humphreys. GATE – an Environment to Support Research and Development in Natural Language Engineering. In *Proceedings of the 8th IEEE International Conference on Tools with Artificial Intelligence (ICTAI-96)*, Toulouse, France, October 1996. `ftp://ftp.dcs.shef.ac.uk/home/robertg/ictai96.ps`.

[Gambäck & Olsson 00]
B. Gambäck and F. Olsson. Experiences of Language Engineering Algorithm Reuse. In *Second International Conference on Language Resources and Evaluation (LREC)*, pages 155–160, Athens, Greece, 2000.

[Gazdar & Mellish 89]
G. Gazdar and C. Mellish. *Natural Language Processing in Prolog.* Addison-Wesley, Reading, MA, 1989.

[Grishman 97]
R. Grishman. TIPSTER Architecture Design Document Version 2.3. Technical report, DARPA, 1997. `http://www.itl.nist.gov/div894/894.02/related_projects/-tipster/`.

[Hepple 00]
M. Hepple. Independence and commitment: Assumptions for rapid training and execution of rule-based POS taggers. In *Proceedings of the 38th Annual Meeting of the Association for Computational Linguistics (ACL-2000)*, Hong Kong, October 2000.

[Horrocks & vanHarmelen 01]
I. Horrocks and F. van Harmelen. Reference Description of the DAML+OIL (March 2001) Ontology Markup Language. Technical report, 2001. `http://www.daml.org/2001/03/reference.html`.

[Hripcsak & Heitjan 02]
G. Hripcsak and D. Heitjan. Measuring agreement in medical informatics reliability studies. *Journal of Biomedical Informatics*, 35:99–110, 2002.

[Hripcsak & Rothschild 05]
G. Hripcsak and A. S. Rothschild. Agreement, the F-measure, and Reliability in Information Retrieval. *Journal of the American Medical Informatics Association*, 12(3):296–298, 2005.

[Humphreys *et al.* 96]
K. Humphreys, R. Gaizauskas, H. Cunningham, and S. Azzam. CREOLE Module Specifications. `http://www.dcs.shef.ac.uk/nlp/gate/`, 1996.

[Jackson 75]
M. Jackson. *Principles of Program Design.* Academic Press, London, 1975.

[Kiryakov 03]
A. Kiryakov. Ontology and Reasoning in MUMIS: Towards the Semantic Web. Technical Report CS–03–03, Department of Computer Science, University of Sheffield, 2003. `http://gate.ac.uk/gate/doc/papers.html`.

[Lal & Ruger 02]
P. Lal and S. Ruger. Extract-based summarization with simplification. In *Proceedings of the ACL 2002 Automatic Summarization / DUC 2002 Workshop*, 2002. `http://www.doc.ic.ac.uk/ srueger/pr-p.lal-2002/duc02-final.pdf`.

[Lal 02]
P. Lal. Text summarisation. Unpublished M.Sc. thesis, Imperial College, London, 2002.

[Lassila & Swick 99]
O. Lassila and R. Swick. Resource Description Framework (RDF) Model and Syntax Specification. Technical Report 19990222, W3C Consortium, `http://www.w3.org/-TR/REC-rdf-syntax/`, 1999.

[Li & Shawe-Taylor 03]
Y. Li and J. Shawe-Taylor. The SVM with Uneven Margins and Chinese Document Categorization. In *Proceedings of The 17th Pacific Asia Conference on Language, Information and Computation (PACLIC17)*, Singapore, Oct. 2003.

[Li *et al.* 02]
Y. Li, H. Zaragoza, R. Herbrich, J. Shawe-Taylor, and J. Kandola. The Perceptron Algorithm with Uneven Margins. In *Proceedings of the 9th International Conference on Machine Learning (ICML-2002)*, pages 379–386, 2002.

[Li *et al.* 04]
Y. Li, K. Bontcheva, and H. Cunningham. An SVM Based Learning Algorithm for Information Extraction. Machine Learning Workshop, Sheffield, 2004. `http://gate.ac.uk/sale/ml-ws04/mlw2004.pdf`.

[Li *et al.* 05a]
Y. Li, K. Bontcheva, and H. Cunningham. SVM Based Learning System For Information Extraction. In M. N. J. Winkler and N. Lawerence, editors, *Deterministic and Statistical Methods in Machine Learning*, LNAI 3635, pages 319–339. Springer Verlag, 2005.

[Li *et al.* 05b]
Y. Li, K. Bontcheva, and H. Cunningham. Using Uneven Margins SVM and Perceptron for Information Extraction. In *Proceedings of Ninth Conference on Computational Natural Language Learning (CoNLL-2005)*, 2005.

[Li *et al.* 05c]
Y. Li, C. Miao, K. Bontcheva, and H. Cunningham. Perceptron Learning for Chinese Word Segmentation. In *Proceedings of Fourth SIGHAN Workshop on Chinese Language processing (Sighan-05)*, pages 154–157, Korea, 2005.

[Li *et al.* 07a]
Y. Li, K. Bontcheva, and H. Cunningham. Cost Sensitive Evaluation Measures for F-term Patent Classification. In *The First International Workshop on Evaluating Information Access (EVIA 2007)*, pages 44–53, May 2007.

[Li *et al.* 07b]
Y. Li, K. Bontcheva, and H. Cunningham. Experiments of opinion analysis on the corpora MPQA and NTCIR-6. In *Proceedings of the Sixth NTCIR Workshop Meeting on Evaluation of Information Access Technologies: Information Retrieval, Question Answering and Cross-Lingual Information Access*, pages 323–329, May 2007.

[Li *et al.* 07c]
Y. Li, K. Bontcheva, and H. Cunningham. SVM Based Learning System for F-term Patent Classification. In *Proceedings of the Sixth NTCIR Workshop Meeting on Evaluation of Information Access Technologies: Information Retrieval, Question Answering and Cross-Lingual Information Access*, pages 396–402, May 2007.

[Lombard *et al.* 02]
M. Lombard, J. Snyder-Duch, and C. C. Bracken. Content analysis in mass communication: Assessment and reporting of intercoder reliability. *Human Communication Research*, 28:587–604, 2002.

[LREC-1 98]
Conference on Language Resources Evaluation (LREC-1), Granada, Spain, 1998.

[LREC-2 00]
Second Conference on Language Resources Evaluation (LREC-2), Athens, 2000.

[Manning & Schütze 99]
C. Manning and H. Schütze. *Foundations of Statistical Natural Language Processing*. MIT press, Cambridge, MA, 1999. Supporting materials available at `http://www.sultry.arts.usyd.edu.au/fsnlp/` .

[Manov *et al.* 03]
D. Manov, A. Kiryakov, B. Popov, K. Bontcheva, and D. Maynard. Experiments with geographic knowledge for information extraction. In *Workshop on Analysis of Geographic References, HLT/NAACL'03*, Edmonton, Canada, 2003. `http://gate.ac.uk/sale/hlt03/paper03.pdf`.

[Maynard 05]
D. Maynard. Benchmarking ontology-based annotation tools for the semantic web. In *UK e-Science Programme All Hands Meeting (AHM2005) Workshop "Text Mining, e-Research and Grid-enabled Language Technology"*, Nottingham, UK, 2005.

[Maynard *et al.*]
D. Maynard, K. Bontcheva, and H. Cunningham. From information extraction to content extraction. Submitted to EACL'2003.

[Maynard *et al.* 00]
D. Maynard, H. Cunningham, K. Bontcheva, R. Catizone, G. Demetriou, R. Gaizauskas, O. Hamza, M. Hepple, P. Herring, B. Mitchell, M. Oakes, W. Peters, A. Setzer, M. Stevenson, V. Tablan, C. Ursu, and Y. Wilks. A Survey of Uses of GATE. Technical Report CS–00–06, Department of Computer Science, University of Sheffield, 2000.

[Maynard *et al.* 01]
D. Maynard, V. Tablan, C. Ursu, H. Cunningham, and Y. Wilks. Named Entity Recognition from Diverse Text Types. In *Recent Advances in Natural Language Processing 2001 Conference*, pages 257–274, Tzigov Chark, Bulgaria, 2001.

[Maynard *et al.* 02a]
D. Maynard, K. Bontcheva, H. Saggion, H. Cunningham, and O. Hamza. Using a Text Engineering Framework to Build an Extendable and Portable IE-based Summarisation System. In *Proceedings of the ACL Workshop on Text Summarisation*, pages 19–26, Phildadelphia, Pennsylvania, 2002. ACM.

[Maynard *et al.* 02b]
D. Maynard, H. Cunningham, K. Bontcheva, and M. Dimitrov. Adapting A Robust Multi-Genre NE System for Automatic Content Extraction. In *Proceedings of the Tenth International Conference on Artificial Intelligence: Methodology, Systems, Applications (AIMSA 2002)*, 2002.

[Maynard *et al.* 02c]
D. Maynard, H. Cunningham, and R. Gaizauskas. Named entity recognition at sheffield university. In H. Holmboe, editor, *Nordic Language Technology – Arbog for Nordisk Sprogtechnologisk Forskningsprogram 2002-2004*, pages 141–145. Museum Tusculanums Forlag, 2002.

[Maynard *et al.* 02d]
D. Maynard, V. Tablan, H. Cunningham, C. Ursu, H. Saggion, K. Bontcheva, and Y. Wilks. Architectural Elements of Language Engineering Robustness. *Journal of Natural Language Engineering – Special Issue on Robust Methods in Analysis of Natural Language Data*, 8(2/3):257–274, 2002.

[Maynard *et al.* 03a]
D. Maynard, K. Bontcheva, and H. Cunningham. Towards a semantic extraction

of Named Entities. In *Recent Advances in Natural Language Processing*, Bulgaria, 2003.

[Maynard *et al.* 03b]
D. Maynard, V. Tablan, and H. Cunningham. NE recognition without training data on a language you don't speak. In *ACL Workshop on Multilingual and Mixed-language Named Entity Recognition: Combining Statistical and Symbolic Models*, Sapporo, Japan, 2003.

[Maynard *et al.* 04a]
D. Maynard, K. Bontcheva, and H. Cunningham. Automatic Language-Independent Induction of Gazetteer Lists. In *Proceedings of 4th Language Resources and Evaluation Conference (LREC'04)*, Lisbon, Portugal, 2004. ELRA.

[Maynard *et al.* 04b]
D. Maynard, H. Cunningham, A. Kourakis, and A. Kokossis. Ontology-Based Information Extraction in hTechSight. In *First European Semantic Web Symposium (ESWS 2004)*, Heraklion, Crete, 2004.

[Maynard *et al.* 04c]
D. Maynard, M. Yankova, N. Aswani, and H. Cunningham. Automatic Creation and Monitoring of Semantic Metadata in a Dynamic Knowledge Portal. In *Proceedings of the 11th International Conference on Artificial Intelligence: Methodology, Systems, Applications (AIMSA 2004)*, Varna, Bulgaria, 2004.

[Maynard *et al.* 06]
D. Maynard, W. Peters, and Y. Li. Metrics for evaluation of ontology-based information extraction. In *WWW 2006 Workshop on "Evaluation of Ontologies for the Web" (EON)*, Edinburgh, Scotland, 2006.

[McEnery *et al.* 00]
A. McEnery, P. Baker, R. Gaizauskas, and H. Cunningham. EMILLE: Building a Corpus of South Asian Languages. *Vivek, A Quarterly in Artificial Intelligence*, 13(3):23–32, 2000.

[Pastra *et al.* 02]
K. Pastra, D. Maynard, H. Cunningham, O. Hamza, and Y. Wilks. How feasible is the reuse of grammars for named entity recognition? In *Proceedings of the 3rd Language Resources and Evaluation Conference*, 2002. `http://gate.ac.uk/sale/lrec2002/reusability.ps`.

[Peters *et al.* 98]
W. Peters, H. Cunningham, C. McCauley, K. Bontcheva, and Y. Wilks. Uniform Language Resource Access and Distribution. In *Workshop on Distributing and Accessing Lexical Resources at Conference on Language Resources Evaluation*, Granada, Spain, 1998.

[Polajnar *et al.* 05]
T. Polajnar, V. Tablan, and H. Cunningham. User-friendly ontology authoring using a controlled language. Technical Report CS Report No. CS-05-10, University of Sheffield, Sheffield, UK, 2005.

[Porter 80]
M. Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.

[Ramshaw & Marcus 95]
L. Ramshaw and M. Marcus. Text Chunking Using Transformation-Based Learning. In *Proceedings of the Third ACL Workshop on Very Large Corpora*, 1995.

[Saggion *et al.* 02a]
H. Saggion, H. Cunningham, K. Bontcheva, D. Maynard, C. Ursu, O. Hamza, and Y. Wilks. Access to Multimedia Information through Multisource and Multilanguage Information Extraction. In *Proceedings of the 7th Workshop on Applications of Natural Language to Information Systems (NLDB 2002)*, Stockholm, Sweden, 2002.

[Saggion *et al.* 02b]
H. Saggion, H. Cunningham, D. Maynard, K. Bontcheva, O. Hamza, C. Ursu, and Y. Wilks. Extracting Information for Information Indexing of Multimedia Material. In *Proceedings of 3rd Language Resources and Evaluation Conference (LREC'2002)*, 2002. `http://gate.ac.uk/sale/lrec2002/mumis_lrec2002.ps`.

[Saggion *et al.* 03a]
H. Saggion, K. Bontcheva, and H. Cunningham. Robust Generic and Query-based Summarisation. In *Proceedings of the European Chapter of Computational Linguistics (EACL), Research Notes and Demos*, 2003.

[Saggion *et al.* 03b]
H. Saggion, H. Cunningham, K. Bontcheva, D. Maynard, O. Hamza, and Y. Wilks. Multimedia Indexing through Multisource and Multilingual Information Extraction; the MUMIS project. *Data and Knowledge Engineering*, 48:247–264, 2003.

[Saggion *et al.* 03c]
H. Saggion, J. Kuper, H. Cunningham, T. Declerck, P. Wittenburg, M. Puts, F. DeJong, and Y. Wilks. Event-coreference across Multiple, Multi-lingual Sources in the Mumis Project. In *Proceedings of the European Chapter of Computational Linguistics (EACL), Research Notes and Demos*, 2003.

[Shaw & Garlan 96]
M. Shaw and D. Garlan. *Software Architecture*. Prentice Hall, New York, 1996.

[Stevenson *et al.* 98]
M. Stevenson, H. Cunningham, and Y. Wilks. Sense tagging and language engineering. In *Proceedings of the 13th European Conference on Artificial Intelligence (ECAI-98)*, pages 185–189, Brighton, U.K., 1998.

[Tablan *et al.* 02]
V. Tablan, C. Ursu, K. Bontcheva, H. Cunningham, D. Maynard, O. Hamza, T. McEnery, P. Baker, and M. Leisher. A Unicode-based Environment for Creation and Use of Language Resources. In *3rd Language Resources and Evaluation Conference*, Las Palmas, Canary Islands – Spain, 2002. ELRA. `http://gate.ac.uk/sale/iesl03/iesl03.pdf`.

[Tablan *et al.* 03]
V. Tablan, K. Bontcheva, D. Maynard, and H. Cunningham. OLLIE: On-Line Learning for Information Extraction. In *Proceedings of the HLT-NAACL Workshop on Software Engineering and Architecture of Language Technology Systems*, Edmonton, Canada, 2003. `http://gate.ac.uk/sale/hlt03/ollie-sealts.pdf`.

[Unicode Consortium 96]
Unicode Consortium. *The Unicode Standard, Version 2.0.* Addison-Wesley, Reading, MA, 1996.

[Ursu *et al.* 05]
C. Ursu, T. Tablan, H. Cunningham, and B. Popav. Digital media preservation and access through semantically enhanced web-annotation. In *Proceedings of the 2nd European Workshop on the Integration of Knowledge, Semantic and Digital Media Technologies (EWIMT 2005)*, London, UK, December 01 2005.

[van Rijsbergen 79]
C. van Rijsbergen. *Information Retrieval.* Butterworths, London, 1979.

[Wang *et al.* 05]
T. Wang, D. Maynard, W. Peters, K. Bontcheva, and H. Cunningham. Extracting a domain ontology from linguistic resource based on relatedness measurements. In *Proceedings of the 2005 IEEE/WIC/ACM International Conference on Web Intelligence (WI 2005)*, pages 345–351, Compiegne, France, Septmeber 2005.

[Wang *et al.* 06]
T. Wang, Y. Li, K. Bontcheva, H. Cunningham, and J. Wang. Automatic Extraction of Hierarchical Relations from Text. In *Proceedings of the Third European Semantic Web Conference (ESWC 2006)*, Budva, Montenegro, 2006.

[Witten & Frank 99]
I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations.* Morgan Kaufmann, 1999.

[Wood *et al.* 03]
M. M. Wood, S. J. Lydon, V. Tablan, D. Maynard, and H. Cunningham. Using parallel texts to improve recall in IE. In *Recent Advances in Natural Language Processing*, Bulgaria, 2003.

[Wood *et al.* 04]
M. Wood, S. Lydon, V. Tablan, D. Maynard, and H. Cunningham. Populating a Database from Parallel Texts using Ontology-based Information Extraction. In *Proceedings of NLDB 2004*, 2004. `http://gate.ac.uk/sale/nldb2004/NLDB.pdf`.

[Yourdon 89]
E. Yourdon. *Modern Structured Analysis.* Prentice Hall, New York, 1989.

[Yourdon 96]
E. Yourdon. *The Rise and Resurrection of the American Programmer*. Prentice Hall, New York, 1996.

Colophon

Formal semantics (henceforth FS), at least as it relates to computational language understanding, is in one way rather like connectionism, though without the crucial prop Sejnowski's work (1986) is widely believed to give to the latter: both are old doctrines returned, like the Bourbons, having learned nothing and forgotten nothing. But FS has nothing to show as a showpiece of success after all the intellectual groaning and effort.

On Keeping Logic in its Place (in Theoretical Issues in Natural Language Processing, ed. Wilks), Yorick Wilks, 1989 (p.130).

We wanted to be modern, we wanted to make the XML people feel like progress is indeed happening, we wanted to update our CVs with the latest trick.... So we looked into using XML as source for this document, and using something like DocBook to translate it into the PDF and HTML versions that we wanted to provide for printing and web viewing. Nice ideas, but our conclusion was that they're not really ready right now. So in the end it was good old LaTeX and TeX4HT for the HTML production. Thank you Don Knuth, Leslie Lamport and Eitan Gurari.