Developing Language Processing Components with GATE Version 3 (a User Guide)

For GATE version 3.1 (March 2006)
(built 7th April 2006)

Hamish Cunningham
Diana Maynard
Kalina Bontcheva
Valentin Tablan
Cristian Ursu
Marin Dimitrov
Mike Dowman
Niraj Aswani
Ian Roberts

http://gate.ac.uk/

HTML version: http://gate.ac.uk/sale/tao/

The sources for this document are available at:
http://gate.ac.uk/sale/tao/tao-sources.zip

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, KnowledgeWeb, PrestoSpace, h-TechSight, enIRaF).

Brief Contents

1 Introduction **2**
- 1.1 How to Use This Text . . . 3
- 1.2 Context . . . 4
- 1.3 Overview . . . 5
- 1.4 Structure of the Book . . . 9
- 1.5 Further Reading . . . 10

2 Change Log **16**
- 2.1 Version 3.1 (April 2006) . . . 16
- 2.2 January 2005 . . . 19
- 2.3 December 2004 . . . 20
- 2.4 September 2004 . . . 20
- 2.5 Version 3 Beta 1 (August 2004) . . . 20
- 2.6 July 2004 . . . 21
- 2.7 June 2004 . . . 22
- 2.8 April 2004 . . . 22
- 2.9 March 2004 . . . 23
- 2.10 Version 2.2 – August 2003 . . . 23
- 2.11 Version 2.1 – February 2003 . . . 23
- 2.12 June 2002 . . . 24

3 How To... **25**
- 3.1 Download GATE . . . 25
- 3.2 Install and Run GATE . . . 25
- 3.3 [**D,F**] Use System Properties with GATE . . . 27
- 3.4 [**D,F**] Use (CREOLE) Plug-ins . . . 29
- 3.5 Troubleshooting . . . 30
- 3.6 [**D**] Get Started with the GUI . . . 30
- 3.7 [**D,F**] Configure GATE . . . 32
- 3.8 Build GATE . . . 33
- 3.9 [**D,F**] Create a New CREOLE Resource . . . 34
- 3.10 [**F**] Instantiate CREOLE Resources . . . 38
- 3.11 [**D**] Load CREOLE Resources . . . 40
- 3.12 [**D,F**] Configure CREOLE Resources . . . 42
- 3.13 [**D**] Create and Run an Application . . . 45
- 3.14 [**D**] Run PRs Conditionally on Document Features . . . 45
- 3.15 [**D**] View Annotations . . . 46
- 3.16 [**D**] Do Information Extraction with ANNIE . . . 47
- 3.17 [**D**] Modify ANNIE . . . 47
- 3.18 [**D**] Create and Edit Test Data . . . 47
- 3.19 [**D,F**] Create a New Annotation Schema . . . 48
- 3.20 [**D**] Save and Restore LRs in Data Stores . . . 50

3.21 [**D**] Save Resource Parameter State to File . . . 51
3.22 [**D,F**] Perform Evaluation with the AnnotationDiff tool . . . 51
3.23 [**D**] Use the Corpus Benchmark Evaluation tool . . . 53
3.24 [**D**] Write JAPE Grammars . . . 55
3.25 [**F**] Embed NLE in other Applications . . . 55
3.26 [**F**] Use GATE within a Tomcat Web Application . . . 56
3.27 [**F**] Use GATE in a Multithreaded Environment . . . 59
3.28 [**D,F**] Add support for a new document format . . . 60
3.29 [**D**] Dump Results to File . . . 61
3.30 [**D**] Stop GUI 'Freezing' on Linux . . . 62
3.31 [**D**] Stop GUI Crashing on Linux . . . 63
3.32 [**D**] Stop GATE Restoring GUI Sessions/Options . . . 63
3.33 Work with Unicode . . . 63
3.34 Work with Oracle and PostgreSQL . . . 64

4 CREOLE: the GATE Component Model 66
4.1 The Web and CREOLE . . . 67
4.2 Java Beans: a Simple Component Architecture . . . 68
4.3 The GATE Framework . . . 69
4.4 Language Resources and Processing Resources . . . 70
4.5 The Lifecycle of a CREOLE Resource . . . 71
4.6 Processing Resources and Applications . . . 72
4.7 Language Resources and Datastores . . . 72
4.8 Built-in CREOLE Resources . . . 73

5 Visual CREOLE 74
5.1 Gazetteer Visual Resource - GAZE . . . 74
5.2 Ontogazetteer . . . 76
5.3 Protégé in GATE . . . 78
5.4 The Co-reference Editor . . . 80

6 Language Resources: Corpora, Documents and Annotations 82
6.1 Features: Simple Attribute/Value Data . . . 82
6.2 Corpora: Sets of Documents plus Features . . . 83
6.3 Documents: Content plus Annotations plus Features . . . 83
6.4 Annotations: Directed Acyclic Graphs . . . 83
6.5 Document Formats . . . 88
6.6 XML Input/Output . . . 102

7 JAPE: Regular Expressions Over Annotations 103
7.1 Use of Context . . . 109
7.2 Use of Priority . . . 110
7.3 Useful tricks . . . 112
7.4 Ontology aware grammar transduction . . . 115
7.5 Using Java code in JAPE rules . . . 115

7.6 Optimising for speed 118
7.7 Serializing JAPE Transducer 118
7.8 The JAPE Debugger 119

8 ANNIE: a Nearly-New Information Extraction System 125
8.1 Tokeniser 126
8.2 Gazetteer 129
8.3 Sentence Splitter 130
8.4 Part of Speech Tagger 130
8.5 Semantic Tagger 131
8.6 Orthographic Coreference (OrthoMatcher) 131
8.7 Pronominal Coreference 132
8.8 A Walk-Through Example 138

9 (More CREOLE) Plugins 140
9.1 Document Reset 141
9.2 Verb Group Chunker 141
9.3 Noun Phrase Chunker 141
9.4 OntoText Gazetteer 142
9.5 Flexible Gazetteer 144
9.6 Gazetteer List Collector 145
9.7 Tree Tagger 146
9.8 Stemmer 148
9.9 GATE Morphological Analyzer 148
9.10 MiniPar Parser 151
9.11 RASP Parser 154
9.12 SUPPLE Parser (formerly BuChart) 155
9.13 Montreal Transducer 160
9.14 Language Plugins 161
9.15 Chemistry Tagger 161
9.16 Flexible Exporter 162
9.17 Annotation Set Transfer 163
9.18 Information Retrieval in GATE 164
9.19 Crawler 168
9.20 Google Plugin 170
9.21 WordNet in GATE 171
9.22 Machine Learning in GATE 172
9.23 Probability Finder 180
9.24 MinorThird 181
9.25 MIAKT NLG Lexicon 182
9.26 Kea - Automatic Keyphrase Detection 184
9.27 Ontotext JapeC Compiler 186

10 Working with Ontologies 201
10.1 Data Model for Ontologies 202

10.2 The Ontology Resource 204
10.3 GATE's Ontology Viewer 205
10.4 Ontology-Aware JAPE Transducer 206
10.5 Populating Ontologies 207

11 Performance Evaluation of Language Analysers 210
11.1 The AnnotationDiff Tool 210
11.2 The six annotation relations explained 212
11.3 Benchmarking tool 212
11.4 Metrics for Evaluation in Information Extraction 213
11.5 Metrics for Evaluation of Inter-Annotator Agreement 215

12 Users, Groups, and LR Access Rights 217
12.1 Java serialisation and LR access rights 218
12.2 Oracle Datastore and LR access rights 218

13 Developing GATE 224
13.1 Creating new plugins 224

14 Combining GATE and UIMA 235
14.1 Embedding a UIMA TAE in GATE 236
14.2 Embedding a GATE `CorpusController` in UIMA 243

Appendices 248

A Design Notes 248
A.1 Patterns 248
A.2 Exception Handling 251

B JAPE: Implementation 254
B.1 Formal Description of the JAPE Grammar 255
B.2 Relation to CPSL 257
B.3 Algorithms for JAPE Rule Application 258
B.4 Label Binding Scheme 264
B.5 Classes 264
B.6 Implementation 265
B.7 Compilation 268
B.8 Using a Different Java Compiler 268

C Named-Entity State Machine Patterns 270
C.1 Main.jape 270
C.2 first.jape 271
C.3 firstname.jape 272
C.4 name.jape 272
C.5 name_post.jape 273
C.6 date_pre.jape 274

C.7 date.jape . . . 274
C.8 reldate.jape . . . 274
C.9 number.jape . . . 274
C.10 address.jape . . . 275
C.11 url.jape . . . 275
C.12 identifier.jape . . . 275
C.13 jobtitle.jape . . . 275
C.14 final.jape . . . 275
C.15 unknown.jape . . . 276
C.16 name_context.jape . . . 276
C.17 org_context.jape . . . 276
C.18 loc_context.jape . . . 277
C.19 clean.jape . . . 277

D Part-of-Speech Tags used in the Hepple Tagger 278

E Sample ML Configuration File 280

References 290

Contents

1 Introduction **2**
1.1 How to Use This Text . . . 3
1.2 Context . . . 4
1.3 Overview . . . 5
1.3.1 Developing and Deploying Language Processing Facilities . . . 5
1.3.2 Built-in Components . . . 7
1.3.3 Additional Facilities . . . 7
1.3.4 An Example . . . 8
1.4 Structure of the Book . . . 9
1.5 Further Reading . . . 10

2 Change Log **16**
2.1 Version 3.1 (April 2006) . . . 16
2.1.1 Major new features . . . 16
2.1.2 Other new features and improvements . . . 17
2.1.3 Bug fixes . . . 18
2.2 January 2005 . . . 19
2.3 December 2004 . . . 20
2.4 September 2004 . . . 20
2.5 Version 3 Beta 1 (August 2004) . . . 20
2.6 July 2004 . . . 21
2.7 June 2004 . . . 22
2.8 April 2004 . . . 22
2.9 March 2004 . . . 23
2.10 Version 2.2 – August 2003 . . . 23
2.11 Version 2.1 – February 2003 . . . 23
2.12 June 2002 . . . 24

3 How To. . . **25**
3.1 Download GATE . . . 25
3.2 Install and Run GATE . . . 25
3.2.1 The Easy Way . . . 26
3.2.2 The Hard Way (1) . . . 26
3.2.3 The Hard Way (2): Anonymous CVS . . . 27

3.3 [**D,F**] Use System Properties with GATE . . . 27
3.4 [**D,F**] Use (CREOLE) Plug-ins . . . 29
3.5 Troubleshooting . . . 30
3.6 [**D**] Get Started with the GUI . . . 30
3.7 [**D,F**] Configure GATE . . . 32
3.7.1 [**F**] Save Config Data to gate.xml . . . 33
3.8 Build GATE . . . 33
3.9 [**D,F**] Create a New CREOLE Resource . . . 34
3.10 [**F**] Instantiate CREOLE Resources . . . 38
3.11 [**D**] Load CREOLE Resources . . . 40
3.11.1 Loading Language Resources . . . 40
3.11.2 Loading Processing Resources . . . 41
3.11.3 Loading and Processing Large Corpora . . . 41
3.12 [**D,F**] Configure CREOLE Resources . . . 42
3.13 [**D**] Create and Run an Application . . . 45
3.14 [**D**] Run PRs Conditionally on Document Features . . . 45
3.15 [**D**] View Annotations . . . 46
3.16 [**D**] Do Information Extraction with ANNIE . . . 47
3.17 [**D**] Modify ANNIE . . . 47
3.18 [**D**] Create and Edit Test Data . . . 47
3.18.1 Saving the test data . . . 48
3.19 [**D,F**] Create a New Annotation Schema . . . 48
3.20 [**D**] Save and Restore LRs in Data Stores . . . 50
3.21 [**D**] Save Resource Parameter State to File . . . 51
3.22 [**D,F**] Perform Evaluation with the AnnotationDiff tool . . . 51
3.22.1 GUI . . . 52
3.23 [**D**] Use the Corpus Benchmark Evaluation tool . . . 53
3.23.1 GUI mode . . . 53
3.23.2 Standalone mode . . . 54
3.23.3 How to define the properties of the benchmark tool . . . 54
3.24 [**D**] Write JAPE Grammars . . . 55
3.25 [**F**] Embed NLE in other Applications . . . 55
3.26 [**F**] Use GATE within a Tomcat Web Application . . . 56
3.26.1 Recommended Directory Structure . . . 57
3.26.2 Configuration files . . . 57
3.26.3 Initialization code . . . 58
3.27 [**F**] Use GATE in a Multithreaded Environment . . . 59
3.28 [**D,F**] Add support for a new document format . . . 60
3.29 [**D**] Dump Results to File . . . 61
3.30 [**D**] Stop GUI 'Freezing' on Linux . . . 62
3.31 [**D**] Stop GUI Crashing on Linux . . . 63
3.32 [**D**] Stop GATE Restoring GUI Sessions/Options . . . 63
3.33 Work with Unicode . . . 63
3.34 Work with Oracle and PostgreSQL . . . 64

4 CREOLE: the GATE Component Model **66**
4.1 The Web and CREOLE 67
4.2 Java Beans: a Simple Component Architecture 68
4.3 The GATE Framework 69
4.4 Language Resources and Processing Resources 70
4.5 The Lifecycle of a CREOLE Resource 71
4.6 Processing Resources and Applications 72
4.7 Language Resources and Datastores 72
4.8 Built-in CREOLE Resources 73

5 Visual CREOLE **74**
5.1 Gazetteer Visual Resource - GAZE 74
5.1.1 Running Modes 75
5.1.2 Loading a Gazetteer 75
5.1.3 Linear Definition Pane 75
5.1.4 Linear Definition Toolbar 75
5.1.5 Operations on Linear Definition Nodes 76
5.1.6 Gazetteer List Pane 76
5.1.7 Mapping Definition Pane 76
5.2 Ontogazetteer 76
5.2.1 Gazetteer Lists Editor and Mapper 77
5.2.2 Ontogazetteer Editor 77
5.3 Protégé in GATE 78
5.3.1 Opening Protégé projects and creating new ones 78
5.3.2 How to Import RDF files in Protégé project 78
5.3.3 How to Save a Protégé project as RDF files 78
5.3.4 How to Set the Protégé plugin directory parameter in GATE 79
5.3.5 How to save a Protégé ontology in Ontotext ontology file format . . . 79
5.3.6 Known problems and bugs 79
5.4 The Co-reference Editor 80

6 Language Resources: Corpora, Documents and Annotations **82**
6.1 Features: Simple Attribute/Value Data 82
6.2 Corpora: Sets of Documents plus Features 83
6.3 Documents: Content plus Annotations plus Features 83
6.4 Annotations: Directed Acyclic Graphs 83
6.4.1 Annotation Schemas 83
6.4.2 Examples of Annotated Documents 85
6.4.3 Creating, Viewing and Editing Diverse Annotation Types 87
6.5 Document Formats 88
6.5.1 Detecting the right reader 89
6.5.2 XML 91
6.5.3 HTML 97
6.5.4 SGML 98

6.5.5 Plain text 98
6.5.6 RTF 99
6.5.7 Email 100
6.6 XML Input/Output 101

7 JAPE: Regular Expressions Over Annotations 103
7.1 Use of Context 109
7.2 Use of Priority 110
7.3 Useful tricks 112
7.4 Ontology aware grammar transduction 115
7.5 Using Java code in JAPE rules 115
7.5.1 Adding a feature to the document 116
7.5.2 Using named blocks 117
7.6 Optimising for speed 118
7.7 Serializing JAPE Transducer 118
7.7.1 How to serialize? 119
7.7.2 How to use the serialized grammar file? 119
7.8 The JAPE Debugger 119
7.8.1 Debugger GUI 120
7.8.2 Using the Debugger 121
7.8.3 Known Bugs 122

8 ANNIE: a Nearly-New Information Extraction System 125
8.1 Tokeniser 126
8.1.1 Tokeniser Rules 127
8.1.2 Token Types 127
8.1.3 English Tokeniser 128
8.2 Gazetteer 129
8.3 Sentence Splitter 130
8.4 Part of Speech Tagger 130
8.5 Semantic Tagger 131
8.6 Orthographic Coreference (OrthoMatcher) 131
8.6.1 GATE Interface 132
8.6.2 Resources 132
8.6.3 Processing 132
8.7 Pronominal Coreference 132
8.7.1 Quoted Speech Submodule 133
8.7.2 Pleonastic It submodule 133
8.7.3 Pronominal Resolution Submodule 134
8.7.4 Detailed description of the algorithm 134
8.8 A Walk-Through Example 138
8.8.1 Step 1 - Tokenisation 139
8.8.2 Step 2 - List Lookup 139
8.8.3 Step 3 - Grammar Rules 139

9 (More CREOLE) Plugins 140
9.1 Document Reset . . . 141
9.2 Verb Group Chunker . . . 141
9.3 Noun Phrase Chunker . . . 141
9.3.1 Differences from the Original . . . 141
9.3.2 Using the Chunker . . . 142
9.4 OntoText Gazetteer . . . 142
9.4.1 Prerequisites . . . 143
9.4.2 Setup . . . 143
9.5 Flexible Gazetteer . . . 144
9.6 Gazetteer List Collector . . . 145
9.7 Tree Tagger . . . 146
9.7.1 POS tags . . . 148
9.8 Stemmer . . . 148
9.8.1 Algorithms . . . 148
9.9 GATE Morphological Analyzer . . . 148
9.9.1 Rule File . . . 149
9.10 MiniPar Parser . . . 151
9.10.1 Platform Supported . . . 152
9.10.2 Resources . . . 152
9.10.3 Parameters . . . 153
9.10.4 Prerequisites . . . 153
9.10.5 Grammatical Relationships . . . 153
9.11 RASP Parser . . . 154
9.12 SUPPLE Parser (formerly BuChart) . . . 155
9.12.1 Requirements . . . 155
9.12.2 Building SUPPLE . . . 156
9.12.3 Running the parser in GATE . . . 156
9.12.4 System properties . . . 157
9.12.5 Configuration files . . . 157
9.12.6 Parser and Grammar . . . 158
9.12.7 Mapping Named Entities . . . 159
9.12.8 Upgrading from BuChart to SUPPLE . . . 160
9.13 Montreal Transducer . . . 160
9.13.1 Main Improvements . . . 160
9.13.2 Main Bug fixes . . . 161
9.14 Language Plugins . . . 161
9.15 Chemistry Tagger . . . 162
9.15.1 Using the tagger . . . 162
9.16 Flexible Exporter . . . 162
9.17 Annotation Set Transfer . . . 163
9.18 Information Retrieval in GATE . . . 164
9.18.1 Using the IR functionality in GATE . . . 165
9.18.2 Using the IR API . . . 167

9.19 Crawler . . . 168
9.19.1 Using the Crawler PR . . . 169
9.20 Google Plugin . . . 170
9.20.1 Using the GooglePR . . . 170
9.21 WordNet in GATE . . . 171
9.21.1 The WordNet API . . . 172
9.22 Machine Learning in GATE . . . 172
9.22.1 ML Generalities . . . 172
9.22.2 The Machine Learning PR in GATE . . . 174
9.22.3 The WEKA Wrapper . . . 176
9.22.4 Training an ML model with the ML PR and WEKA wrapper . . . 176
9.22.5 Applying a learnt model . . . 176
9.22.6 The MAXENT Wrapper . . . 177
9.22.7 The SVM Light Wrapper . . . 178
9.23 Probability Finder . . . 180
9.24 MinorThird . . . 181
9.25 MIAKT NLG Lexicon . . . 182
9.25.1 Complexity and Generality . . . 183
9.26 Kea - Automatic Keyphrase Detection . . . 184
9.26.1 Using the "KEA Keyphrase Extractor" PR . . . 184
9.26.2 Using Kea corpora . . . 185
9.27 Ontotext JapeC Compiler . . . 186

10 Working with Ontologies 201
10.1 Data Model for Ontologies . . . 202
10.2 The Ontology Resource . . . 204
10.3 GATE's Ontology Viewer . . . 205
10.4 Ontology-Aware JAPE Transducer . . . 207
10.5 Populating Ontologies . . . 207

11 Performance Evaluation of Language Analysers 210
11.1 The AnnotationDiff Tool . . . 210
11.2 The six annotation relations explained . . . 212
11.3 Benchmarking tool . . . 212
11.4 Metrics for Evaluation in Information Extraction . . . 213
11.5 Metrics for Evaluation of Inter-Annotator Agreement . . . 215

12 Users, Groups, and LR Access Rights 217
12.1 Java serialisation and LR access rights . . . 218
12.2 Oracle Datastore and LR access rights . . . 218
12.2.1 Users, Groups, Sessions and Access Modes . . . 218
12.2.2 User/Group Administration . . . 219
12.2.3 The API . . . 222

13 Developing GATE 224

13.1 Creating new plugins 224
13.1.1 Where to keep plugins in the GATE hierarchy 224
13.1.2 Writing a new PR 225
13.1.3 Writing a new VR 229
13.1.4 Adding plugins to the nightly build 233

14 Combining GATE and UIMA **235**
14.1 Embedding a UIMA TAE in GATE 236
14.1.1 Mapping File Format 236
14.1.2 The UIMA component descriptor 241
14.1.3 Using the `AnalysisEnginePR` 241
14.1.4 Current limitations 242
14.2 Embedding a GATE `CorpusController` in UIMA 243
14.2.1 Mapping file format 243
14.2.2 The GATE application definition 244
14.2.3 Configuring the `GATEApplicationAnnotator` 244

Appendices **248**

A Design Notes **248**
A.1 Patterns 248
A.1.1 Components 249
A.1.2 Model, view, controller 250
A.1.3 Interfaces 251
A.2 Exception Handling 251

B JAPE: Implementation **254**
B.1 Formal Description of the JAPE Grammar 255
B.2 Relation to CPSL 257
B.3 Algorithms for JAPE Rule Application 258
B.3.1 The first algorithm 258
B.3.2 Algorithm 2 262
B.4 Label Binding Scheme 264
B.5 Classes 264
B.6 Implementation 265
B.6.1 A Walk-Through 265
B.6.2 Example RHS code 266
B.7 Compilation 268
B.8 Using a Different Java Compiler 268

C Named-Entity State Machine Patterns **270**
C.1 Main.jape 270
C.2 first.jape 271
C.3 firstname.jape 272
C.4 name.jape 272

C.4.1 Person . . . 272
C.4.2 Location . . . 272
C.4.3 Organization . . . 273
C.4.4 Ambiguities . . . 273
C.4.5 Contextual information . . . 273
C.5 name_post.jape . . . 273
C.6 date_pre.jape . . . 274
C.7 date.jape . . . 274
C.8 reldate.jape . . . 274
C.9 number.jape . . . 274
C.10 address.jape . . . 275
C.11 url.jape . . . 275
C.12 identifier.jape . . . 275
C.13 jobtitle.jape . . . 275
C.14 final.jape . . . 275
C.15 unknown.jape . . . 276
C.16 name_context.jape . . . 276
C.17 org_context.jape . . . 276
C.18 loc_context.jape . . . 277
C.19 clean.jape . . . 277

D Part-of-Speech Tags used in the Hepple Tagger 278

E Sample ML Configuration File 280

References 290

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

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

available as open-source free software under the GNU library licence[2]. GATE uses Unicode [Unicode Consortium 96] throughout, 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 and French. Version 3 of the system, a complete re-implementation and extension of the original, 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 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 11), 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

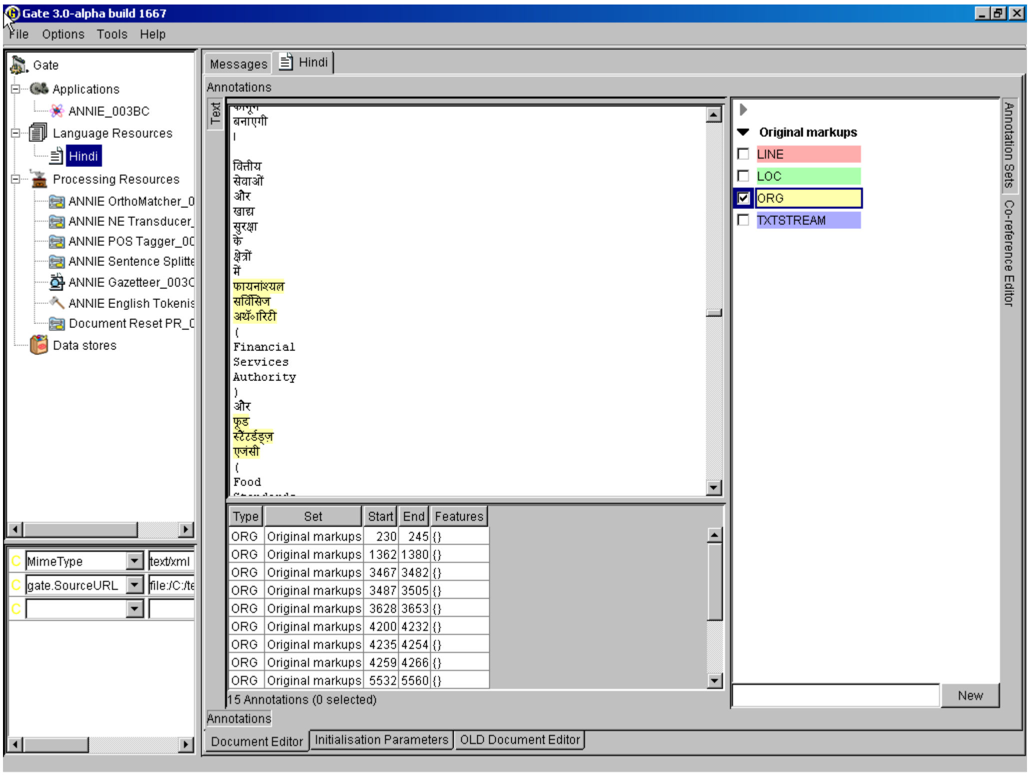


Figure 1.1: One of GATE's visual resources

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 11.
- 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 3.33 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 how to measure the performance of language analysis components.

Chapter 12 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 C 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);
- a guide to using GATE for manual annotation;
- 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 [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.* **05**] (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.

Never in the history of the Research Assessment Exercise has so much been owed by so many to so few exercises in copy-and-paste.

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 3.1 (April 2006)

2.1.1 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 14.

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 9.27.

2.1.2 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 7.2.
- 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 8.4.
- 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 9.15 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 7.7.
- 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.

2.1.3 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 9.20.
- 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...

2.2 January 2005

Release of version 3.

New plugins for processing in various languages (see 9.14). 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 9.15
- Montreal Transducer 9.13
- RASP Parser 9.11
- MiniPar 9.10
- Buchart Parser 9.12
- MinorThird 9.24
- NP Chunker 9.3
- Stemmer 9.8
- TreeTagger 9.7
- Probability Finder 9.23
- Crawler 9.19
- Google PR 9.20

Support for SVM Light, a support vector machine implementation, has been added to the machine learning plugin (see section 9.22.7).

2.3 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 3.26 for details.

2.4 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.

2.5 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 subsiduary 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 **??**.
- The NLG Lexical tools from the MIAKT project have now been released. See documentation in Section 9.25.
- The Features viewer/editor has been completely updated – see Sections 3.15 and 3.18 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 3.20 for more information.

2.6 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);
```

2.7 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 3.22.

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.

2.8 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 10.4.

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.

2.9 March 2004

Support was added for the MAXENT machine learning library. (See 9.22.6 for details.)

2.10 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 3.34 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).

2.11 Version 2.1 – February 2003

Integration of Machine Learning PR and WEKA wrapper (see Section 9.22).

Addition of DAML+OIL exporter.

Integration of WordNet in GATE (see Section 9.21).

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

2.12 June 2002

Conditional versions of the controllers are now available (see Section 3.14). 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 (see Section 5.3), so that people with developed Protégé ontologies can use them within GATE.

Addition of IR facilities in GATE (see Section 9.18).

Modification of the corpus benchmark tool (see Section 3.23), 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. 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:

- for the development enviroment, see the visual tutorials available on our 'movies' page;
- for the framework, see the example code at http://gate.ac.uk/GateExamples/doc/.

3.1 Download GATE

To download GATE point your web browser at http://gate.ac.uk/ and follow the download link. Fill in the form there, and you will be emailed an FTP address to download the system from.

3.2 Install and Run GATE

GATE will run anywhere that supports recent versions of Java (1.4.0 or above, 1.4.2 preferred), including Solaris, Linux and Windoze platforms. We don't run tests on other platforms, but have had reports of successfull installs elsewhere. We are also testing released installers on MacOS X.

3.2.1 The Easy Way

The easy way to install is to use one of the platform-specific installers (created using ZeroG's InstallAnywhere product). Download a 'platform-specific installer' and follow the instructions it gives you.

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 or above, available free from Sun Microsystems or from your UNIX supplier. (We test on various Sun 1.4 JDKs on Solaris, Linux, NT 4, Windows 2000 and 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.sh`) 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): Anonymous CVS

You can use anonymous CVS from `cvs.sourceforge.net` to check out the source code and then build it using `ant`:

host: `cvs.sourceforge.net`

access method: `pserver`

remote repository: `/cvsroot/gate`

username: `anonymous`

Here's a sample command line that would work in unix or cygwin:

```
cvs -d :pserver:anonymous@cvs.sourceforge.net:/cvsroot/gate login
cvs -d :pserver:anonymous@cvs.sourceforge.net:/cvsroot/gate checkout gate
```

No password is required, just press return at the password prompt. If you are behind a firewall that prevents you from connecting to the CVS server, you can use one of the following alternative values for `-d`, to use ports that are more likely to be open:
`:pserver:anonymous@cvs-pserver.sf.net:80/cvsroot/gate` or
`:pserver:anonymous@cvs-pserver.sf.net:443/cvsroot/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.

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.`

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()` and `setUserConfigFile()` 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
```

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
```

[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.

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. By default this only includes the ANNIE plugin (see Section 3.16).

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 <u>gate.Gate</u>

`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 animated demonstrations and tutorials on the 'movies' page. There is also a shorter manual aimed at those who just want to use GATE for annotating texts and viewing the results.

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

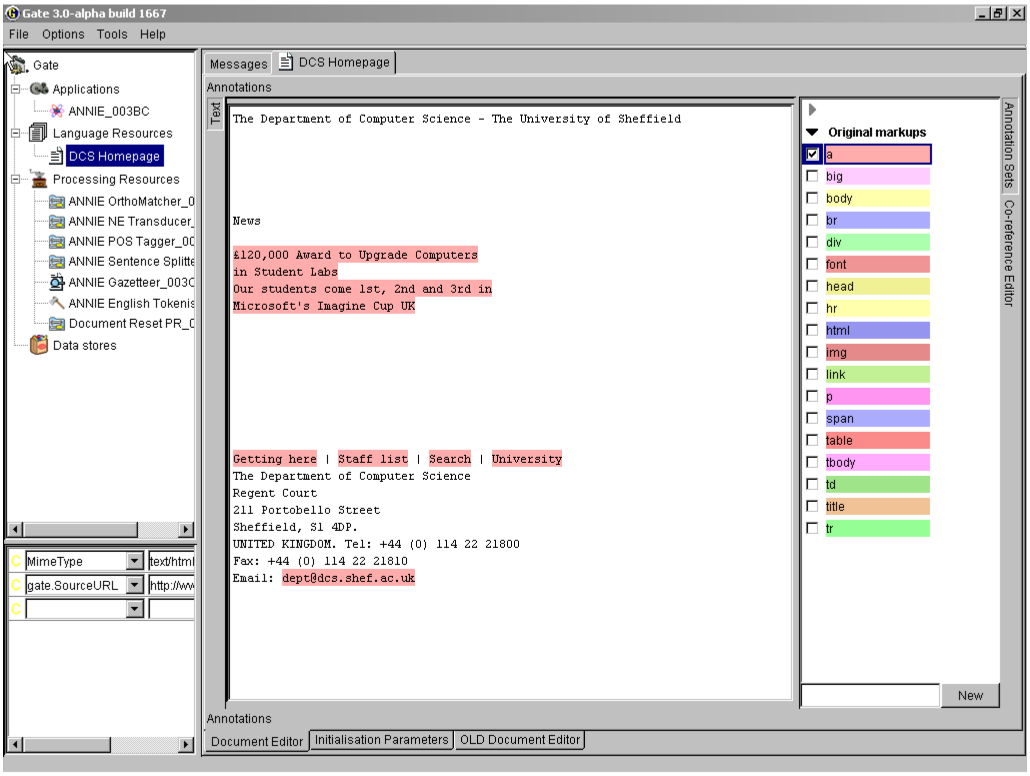


Figure 3.1: Main Window

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 black rectangle, which is the *small resource viewer*;
4. on the right of the main area, containing tabs with 'Messages' and 'GATE Document_0001F', the *main resource viewer*;
5. the *messages bar* along the bottom (where it says 'Finished dumping...').

The menu and the messages bars 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.

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 GATE CVS repository obtained through anonymous CVS (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,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

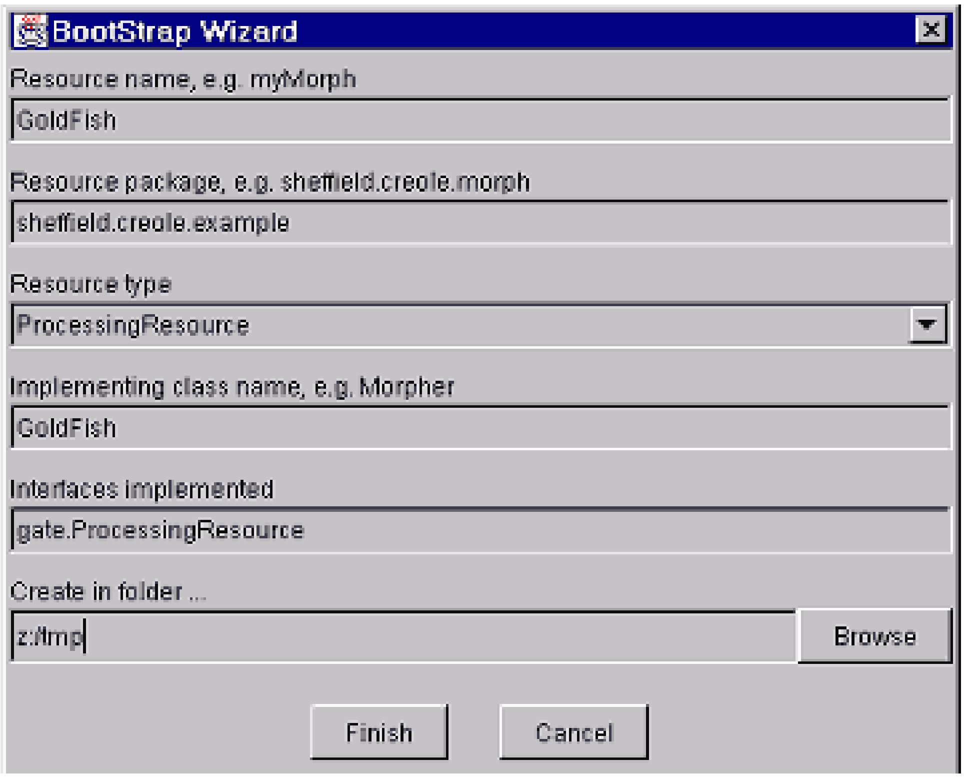


Figure 3.2: BootStrap Wizard Dialogue

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. `java.util.Set`) that you would like your new resource to implmenent. 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 create the JAR and XML files that allow GATE to load the new resource. (There's no reason not to use your own favourite alternative, e.g. ANT.) For the pre-requisites of the build process that we use (based on Makefiles, and the GNU shell tools) see section 3.8. When you have these pre-requisites available do the following from a command prompt (working from the `GoldFish/build` directory that the bootstrapper created for you):

```
./configure
make depend
make
make jar
```

This will create the two files that GATE needs to load your new resource: `GoldFish.jar` and `creole.xml`.

You can now load this resource into GATE; see

- section 3.10 for how to instantiate the resource from the framework;
- section 3.11 for how to load the resource in the development environment;
- section 3.12 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 1.118 2006/04/06 13:19:39 valyt 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.
 */
public class GoldFish extends AbstractProcessingResource
  implements ProcessingResource {


} // class GoldFish
```

The default XML configuration for GoldFish looks like this:

```
<!-- resource.xml GoldFish -->
<!--  hamish, 26/9/2001 -->
<!-- $Id: howto.tex,v 1.118 2006/04/06 13:19:39 valyt Exp $ -->

<CREOLE-DIRECTORY>

<CREOLE>
  <RESOURCE>
    <NAME>GoldFish</NAME>
    <JAR>GoldFish.jar</JAR>
    <CLASS>sheffield.creole.example.GoldFish</CLASS>
  </RESOURCE>
</CREOLE>

</CREOLE-DIRECTORY>
```

The directory structure containing these files is shown in figure 3.3. `GoldFish.java` lives in the `src/sheffield/creole/example` directory. `creole.xml` is generated in the `build` directory from a source file called `resource.xml` which lives in the `src` directory. (The `lib` directory is for libraries; the `classes` directory is where Java class files are placed; the `doc` directory is for documentation.)

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 `resource.xml`) and copy it into your existing one.

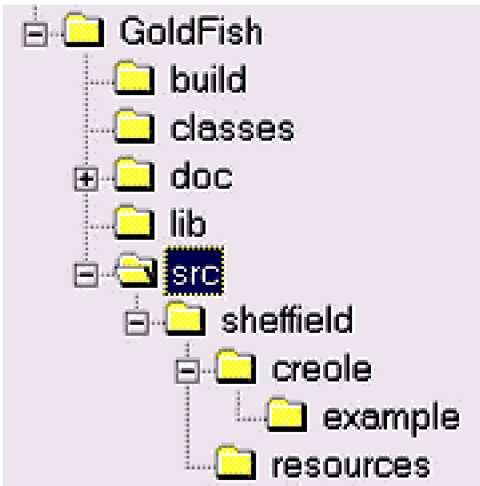


Figure 3.3: BootStrap directory tree

3.10 [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[2]. The Factory takes care of all this, makes sure that the GUI is told about what is happenning (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 – see section 3.12.

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:

```
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

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

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:

```
<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="gate:/creole/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).

When creating a Document, however, the URL of the source for the document must be provided[3]. 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

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

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 12.

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());
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);
```

3.11 [D] Load CREOLE Resources

3.11.1 Loading Language Resources

Load a language resource by right clicking on “Language Resources” and selecting a language resource type (document, corpus or annotation schema). Choose a name for the resource, and choose any parameters as necessary.

For a document, a file or url should be selected as the value of "sourceUrl" (double clicking in the "values" box brings up a tree structure to enable selection of documents). Other parameters can be selected or changed as necessary, such as the encoding of the document, and whether it should be markup aware.

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.

To add a new annotation schema, simply choose the name and the path or Url. For more information about schema, see 6.4.1.

3.11.2 Loading Processing Resources

This section describes how to load and run CREOLE resources not present in ANNIE. To load ANNIE, see Section 3.16. 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).

3.11.3 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 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.

3.12 [D,F] Configure CREOLE Resources

This section desscribes how to write entries in the `creole.xml` file that is used to describe resources to GATE. This data is used to tell GATE things like what parameters a resource has, how to display it if it has a visualisation, etc.

An example file:

```
<CREOLE-DIRECTORY>
  <CREOLE>
    <RESOURCE>
      <NAME>GATE XML Document Format</NAME>
      <CLASS>gate.corpora.XmlDocumentFormat</CLASS>
      <AUTOINSTANCE/>
      <PRIVATE/>
      <JAR>gate.jar</JAR>
    </RESOURCE>
  </CREOLE>
</CREOLE-DIRECTORY>
```

These files have as a root element `CREOLE-DIRECTORY`, and may contain any number of `CREOLE` elements, which in turn contain any number of `RESOURCE` elements[4].

Each resource must give a name, a Java class and the JAR file that it can be loaded from. The above example defines GATE's XML document format analyser resource. This resource has no parameters, is automatically loaded when the `creole.xml` data is loaded, is not displayed to the GUI user (it is used internally by the document creation code), and is loaded from `gate.jar`.

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

[4] The purpose of the `CREOLE` element is to allow files to be build up from the concatenation of multiple other files.

```
<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.

Each parameter has a type, which may be the type of another resource, or a Java built in. Attributes of parameters:

NAME: name of the property that the parameter refers to; if supplied it will change the name that the initialisation routines assume are available to get/set on the resource (which are normally based on the value of the parameter, i.e. on the type of the parameter). The name must be identical to the property of the resource that the parameter relates to.

DEFAULT: default value.

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

Visual Resources also have a `GUI` tag, which describes the resource (PR or LR) that it displays, whether it is the main viewer for that resource or not (main viewers are the first tab displayed for a resource) and whether the VR should go in the small viewers window or the large one. For example:

```
<RESOURCE>
  <NAME>Features Editor</NAME>
  <CLASS>gate.gui.FeaturesEditor</CLASS>
  <!-- type values can be "large" or "small"-->
  <GUI TYPE="large">
    <MAIN_VIEWER/>
    <RESOURCE_DISPLAYED>gate.util.FeatureBearer</RESOURCE_DISPLAYED>
  </GUI>
</RESOURCE>
```

More information:

- To collect PRs into an application and run them, see section 3.13.
- GATE's internal creole.xml file (note that there are no JAR entries there, as the file is bundled with GATE itself).

3.13 [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 paramters 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.

For how to use the *conditional* versions of the pipelines see section 3.14.

3.14 [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 3.13 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.

3.15 [D] View Annotations

To view a document, double click on the filename in the left hand pane. Note that it may take a few seconds for the text to be displayed if it is long.

To view the annotation sets, click on AnnotationSets on the right pane. 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.

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

To view the annotations and their features, click on Annotations at the top or bottom of the main window. The annotation 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. 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). 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.

3.16 [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 3.25.

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 3.11.1), 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.

3.17 [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 3.11.2.

3.18 [D] Create and Edit Test Data

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
- The most recent annotation type to have been used will be displayed in a popup box. If this is not the one you want, use the menu to change it. If it is correct, you need do nothing further. You can add or change features and their values using the menu in the box.
- To delete an annotation, click on the red X in the popup box.

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 3.19. 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 the Organization annotation for the string "University of Sheffield" (highlighted in grey) to the Default Annotation set.

To add a second annotation to a selected piece of text, or to add an overlapping annotatin 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.

3.18.1 Saving the test data

The data can either be dumped out as a file (see Section 3.29 or saved in a data store (see Section 3.20.

3.19 [D,F] Create a New Annotation Schema

GUI

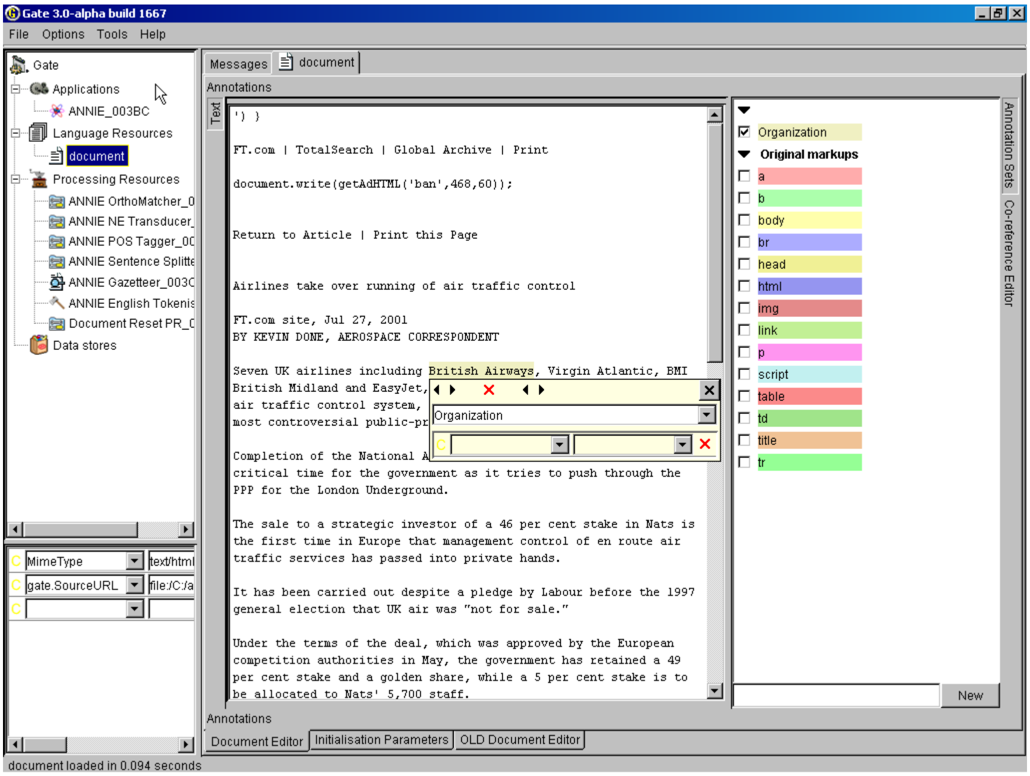


Figure 3.4: Adding an Organization annotation to the Default Annotation Set

An annotation schema file can be loaded or unloaded in GATE just like any other language resource. Once loaded into the system, the SchemaAnnotationEditor 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 creole.xml:

```
<!-- 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 ="gate:/creole/schema/DateSchema.xml"/></AUTOINSTANCE>
  <AUTOINSTANCE><PARAM NAME ="xmlFileUrl"
   VALUE ="gate:/creole/schema/FacilitySchema.xml"/></AUTOINSTANCE>
  <AUTOINSTANCE><PARAM NAME ="xmlFileUrl"
   VALUE ="gate:/creole/schema/TokenSchema.xml"/></AUTOINSTANCE>
  <AUTOINSTANCE><PARAM NAME ="xmlFileUrl"
   VALUE ="gate:/creole/schema/SyntaxTreeNodeSchema.xml"/></AUTOINSTANCE>
  <AUTOINSTANCE><PARAM NAME ="xmlFileUrl"
   VALUE ="gate:/creole/schema/CorefSchema.xml"/></AUTOINSTANCE>
</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.

3.20 [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.

3.21 [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 3.11). It is possible to save the data used to create a an application to file and re-load it later. To save the application to 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) then that file must still exist when you restore 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. The easiest way of deploying a portable GATE application is to store the application file and the application resources under the same top directory which will become the deployment unit.

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

Section 11 describes the theory behind this tool.

3.22.1 GUI

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 11.1 and 11.4 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.

3.23 [D] Use the Corpus Benchmark Evaluation tool

The Corpus Benchmark tool can be run in two ways: standalone and GUI mode. Section 11.3 describes the theory behind this tool.

3.23.1 GUI mode

To use the tool in GUI mode, first make sure the properties of the tool have been set correctly (see section 3.23.3 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 shuold 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.

3.23.2 Standalone mode

Alternatively, the tool can be run in standalone mode, using the following commands:

- To process the corpus, issue the command

  `gate -e -generate corpusname`

  (where 'corpusname' is the name of the corpus)

- To run in Stored mode, issue the command

  `gate -e [-verbose] -marked\_stored corpusname`

- To run in Current mode, issue the command

  `gate -e [-verbose] -marked\_clean corpusname`

- To run in Default mode, issue the command

  `gate -e [-verbose] corpusname`

The tool can be run in verbose mode for any of these options using the [-verbose] flag. The results can be piped to an html file and viewed with an internet browser.

3.23.3 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.

An example file is shown below:

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

3.24 [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 C describes the default Named Entity rules distributed with GATE.

3.25 [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:

```
public static void main(String args[]) throws GateException, IOException {
  // initialise the GATE library
  Gate.init();

  // initialise ANNIE
  // create a corpus pipeline controller to run ANNIE with
  annieController =
    (SerialAnalyserController) Factory.createResource(
    "gate.creole.SerialAnalyserController", Factory.newFeatureMap(),
    Factory.newFeatureMap(), "ANNIE_" + Gate.genSym()
  );

  // load each PR as defined in ANNIEConstants
  for(int i = 0; i < ANNIEConstants.PR_NAMES.length; i++) {
    FeatureMap params = Factory.newFeatureMap(); // use default parameters
    ProcessingResource pr = (ProcessingResource)
    Factory.createResource(ANNIEConstants.PR_NAMES[i], params);

    // add the PR to the pipeline controller
    annieController.add(pr);
  } // for each ANNIE PR

....
```

A longer example of embedding ANNIE is available at http://gate.ac.uk/GateExamples/doc/.

3.26 [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.

3.26.1 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.
```

3.26.2 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.

3.26.3 Initialization code

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

```
// imports
...
public class MyServlet extends GenericServlet {
  private static boolean gateInited = false;

  public void init() throws ServletException, IOException {
    if(!gateInited) {
      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;
    }
  }
}
```

Once initialized, you can create GATE resources using the Factory in the usual way (for example, see section 3.25 for an example of how to create an ANNIE application). You should also read section 3.27 for important notes on using GATE in a multithreaded application.

3.27 [F] Use GATE in a Multithreaded Environment

GATE can be used in multithreaded applications, so long as you observe a few restrictions. It is important to note that individual GATE processing resources, language resources and controllers are *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 PR it should be possible to use several different instances of the same PR at once, each in a different thread.

When writing a PR you should bear the following in mind, to ensure that your PR will be useable in this way.

- Avoid static data. Where possible, you should avoid using static fields in your PR 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 runtime parameters should be read at each `execute()`.
- Use temporary files correctly. If your PR 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 PR, make sure these objects are accessed either read-only, or in a thread-safe way. In particular you must be very careful if your PR can take other PR instances as init or runtime parameters (e.g. the Flexible Gazetteer, section 9.5).

Of course, if your PR is simply a wrapper around an external library that imposes these kinds of limitations there is only so much you can do. If your PR 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. 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.

3.28 [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.

3.29 [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[5], 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.

3.30 [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'.

This feature is available totally free of charge.

[5]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.

3.31 [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`.

3.32 [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

3.33 Work with Unicode

GATE provides various facilities for working with Unicode beyond those that come as default with Java[6]:

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.

1 using the editor:
In the GUI, select 'Unicode editor' from the 'Tools' menu. This will display an editor window,

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

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.)

3.34 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).

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.

4.1 The Web and CREOLE

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

Resource implementations are stored at a URL (when the resources are in the local file system this can be a `file:/` URL). When the URL is given to GATE the `creole.xml` component configuration file is sucked down the pipe and the resource information added to the CREOLE register. When a user requests an instantiation of a resource, the class files are sucked up too, and an object created in the local 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],

[1]The Java DataBase Connectivity layer.

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 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. (To be precise only their configuration data is loaded to begin with; the actual classes are loaded when the user requests the instantiation of a resource.)

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 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]

---

[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.

[4]This point is due to Wim Peters.

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 start this process – see section 3.9. 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 3.10 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 3.11.

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 3.12.

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 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 3.14 for how to use these.

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. Currently two such mechanisms are implemented:

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

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.

Contributions to further developments gratefully received (unmarked low-denomination notes preferred). Bugs to `santa@northpole.org`.

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 standrad 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 defintion 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 lsited 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 Protégé in GATE

Protégé is integrated in GATE so that people with developed Protégé ontologies can use them in GATE (for example in the hierarchical gazetteer), and also so that they can take advantage of being able to read different format ontology files in Protégé.

It is best to download and investigate Protégé (http://protege.stanford.edu/index.html) before trying to use it from GATE. In GATE you will have the same Protégé GUI as in the original application and it is the same application embedded with some restrictions - for example, there is no menu or toolbar. You have some of the Protégé menu items in GATE resource pop-up menu.

5.3.1 Opening Protégé projects and creating new ones

To open a Protégé project you have to create a new GATE LR (Language Resource) - Protégé Project. In the field projectName put the file name of Protégé project (it is best to give the full file name). The second parameter is the URL for the Ontotext format ontology file. You can use this parameter to save a Protégé ontology in Ontotext Ontology Editor format.

If you want to create a new project you have to leave the first parameter empty. In this case you will be asked for Protégé project data format during creation. The standard two are "Standard Text Files" and "JDBC Database". In the current installation of Protégé 2000 ver. 1.7 you have a standard plugin (included in the installation) for "RDF Schema" format.

After creation of a Protégé Project LR, you can open it by doubleclicking on resource in the GATE resource tree. You will see the Protégé GUI inside the GATE tabbed pane.

5.3.2 How to Import RDF files in Protégé project

Whenever you load a Protégé project or create an empty one, you can import another project, using the popup menu item "Import?". You will be asked if you want to save changes in the current project. Select the format of the project you have to import. Than you can select files for the import of the data. Protégé RDF format keeps the ontology data in two files - classes file *.RDFS and instances file *.RDF.

5.3.3 How to Save a Protégé project as RDF files

If you do not choose the RDF file format on creation of new ontology or you do not open an RDF project, you can use the popup menu item "Save In Format?" and select "RDF Schema" format.

5.3.4 How to Set the Protégé plugin directory parameter in GATE

The Protégé architecture allows integration of the plugin to extend the functionality of the application. The RDF file format support is integrated in Protégé as a plugin included in the standard installation. You can find all installed plugins in the subdirectory plugins of your Protégé installation. Here you can add any other Protégé plugin you may want to have (see http://protege.stanford.edu/plugins.html). If the Protégé plugins directory is accessible in GATE on creation of the Protégé project LR, you should see a similar output in the GATE Messages tab:

```
Plugin classpath:
file:/D:/projects/gate/plugins/
file:/D:/projects/gate/plugins/query_tab.jar
file:/D:/projects/gate/plugins/rdf-api.jar
file:/D:/projects/gate/plugins/rdf_backend.jar
file:/D:/projects/gate/plugins/standard_extensions.jar
file:/D:/projects/gate/plugins/xerces.jar
```

Otherwise you will see only "Plugin classpath:" without the list of assigned plugins and maybe some warnings because of some missing plugin.

There are two ways of providing access to this directory when you use Protégé embedded in GATE. The first is to copy the "plugins" directory in the current GATE directory. This is the simple, but not very "clear" way. In this case if you add some plugin in your Protégé installation you should copy this directory every time to your GATE directory to get it there too. The second way is to tell GATE where the Protégé installation directory ism so it can find the plugins subdirectory. You can do this by setting the Java VM property: -Dprotege.dir="protege_installation_directory" You can see more about this in Section 5.3.6 below.

5.3.5 How to save a Protégé ontology in Ontotext ontology file format

You can display a Prot?g? ontology using the Ontology Editor tab of the Prot?g? project LR GUI.

5.3.6 Known problems and bugs

- If you give only the filename on creation of Protégé project instead of full file name:

  There is a strange java.lang.NullPointerException when you try to load a Protégé project giving only a file name instead of the full file name. You will have the same exception in a Protégé application outside GATE, so this is not an integration-specific problem. It is best to give the full file name on creation or load of a Protégé project or on import of data files.

- Unable to find Protégé plugin directory:

  To gain some Protégé extra functionality, you should have a Protégé plugin directory somewhere and to give the location of this directory to your Protégé application. You can find the Protégé plugin directory as a subdirectory of your Protégé installation (http://protege.stanford.edu/download.html) named "plugins". You have to give full path to the Protégé installation. You should add Java VM property: -Dprotege.dir="D:/projects/Protege" where "D:/projects/Protege" is a directory with subdirectory "plugins" in it. Another way is to copy this "plugins" directory in your GATE application directory.

- Save Protégé project error when this project is not shown in the GATE tab pane:

  There is a Protégé exception in this case. The Protégé action for saving the project requires an active GUI. So, the simple solution is to doubleclick on resource to show the Protégé GUI in GATE. Then you can use the Protégé specific popup menu actions without problem.

5.4 The Co-reference Editor

The co-reference editor allows co-reference chains (see section 8.7) 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 3.19 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. (**NOTE:** the ruler doesn't scale very well in HTML; for a better picture see the original 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

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

represented by an annotation of type parse.

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

Text				
To: All Barnyard Animals				
|0...|5...|10..|15..|20				
From: Chicken Little				
|25...|30...|35..|40..|45				
Date: November 10,1194				
|50...|55...|60..|65..				
Subject: Descending Firmament				
|70...|75...|80..|85..|90..|95..				
Priority : Urgent.				
|100...|105...|110..|115..				
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

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 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).

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

To view and edit annotation types, see Section 3.15. To add annotations of a new type, see Section 3.18. To add a new annotation schema, see Section 3.19.

6.5 Document Formats

The following document formats are supported by GATE:

- Plain Text
- HTML
- SGML
- XML
- RTF
- Email

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

When opening a document in GATE, the file extension (e.g. `xml`) is important but if not present, GATE uses some other means to detect its type. In order to successfully apply the document creation algorithm described above, GATE needs to detect the proper reader to use for each document format. In order to do that, it takes 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. 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
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)]}
```

[1]It's not an XML entity but a information extraction named entity

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

The parser used to access HTML documents is the one provided by Java. 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 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 7.2)
- 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 7.3.

Input annotations must also be defined at the start of each grammar. If no annotations are defined, the default will be Token, SpaceToken and Lookup (i.e. only these annotations will be considered when attempting a match). See Section 7.6 for more details.

There are 3 main ways in which the pattern can be specified:

- specify a string of text, e.g. {Token.string == "of"}
- specify an annotation previously assigned from a gazetteer, tokeniser, or other module, e.g. {Lookup}
- specify the attributes (and values) of an annotation), e.g. {Token.kind == number}

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.

The same operators can be used as for the tokeniser rules, i.e.

```
|
*
?
+
```

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 annnotated 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 7.5).

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
```

[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

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
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 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

({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"}
```

7.2 Use of Priority

Each grammar has 4 possible control styles: “brill”, ”all”, ”first” 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 if fired; it does not attempt to get a longer match (as the other two styles do).

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.

7.3 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.

- 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. A negative operator cannot be specified as such. 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 7.5 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 7.5 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.

7.4 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 10.4.

7.5 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);
}
```

7.5.1 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");
}
```

7.5.2 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.

7.6 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 Tokens and SpaceTokens 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.

7.7 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.

7.7.1 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.

7.7.2 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.

7.8 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.

7.8.1 Debugger GUI

The layout of the JAPE-debugger user interface is shown in Figure 7.1.

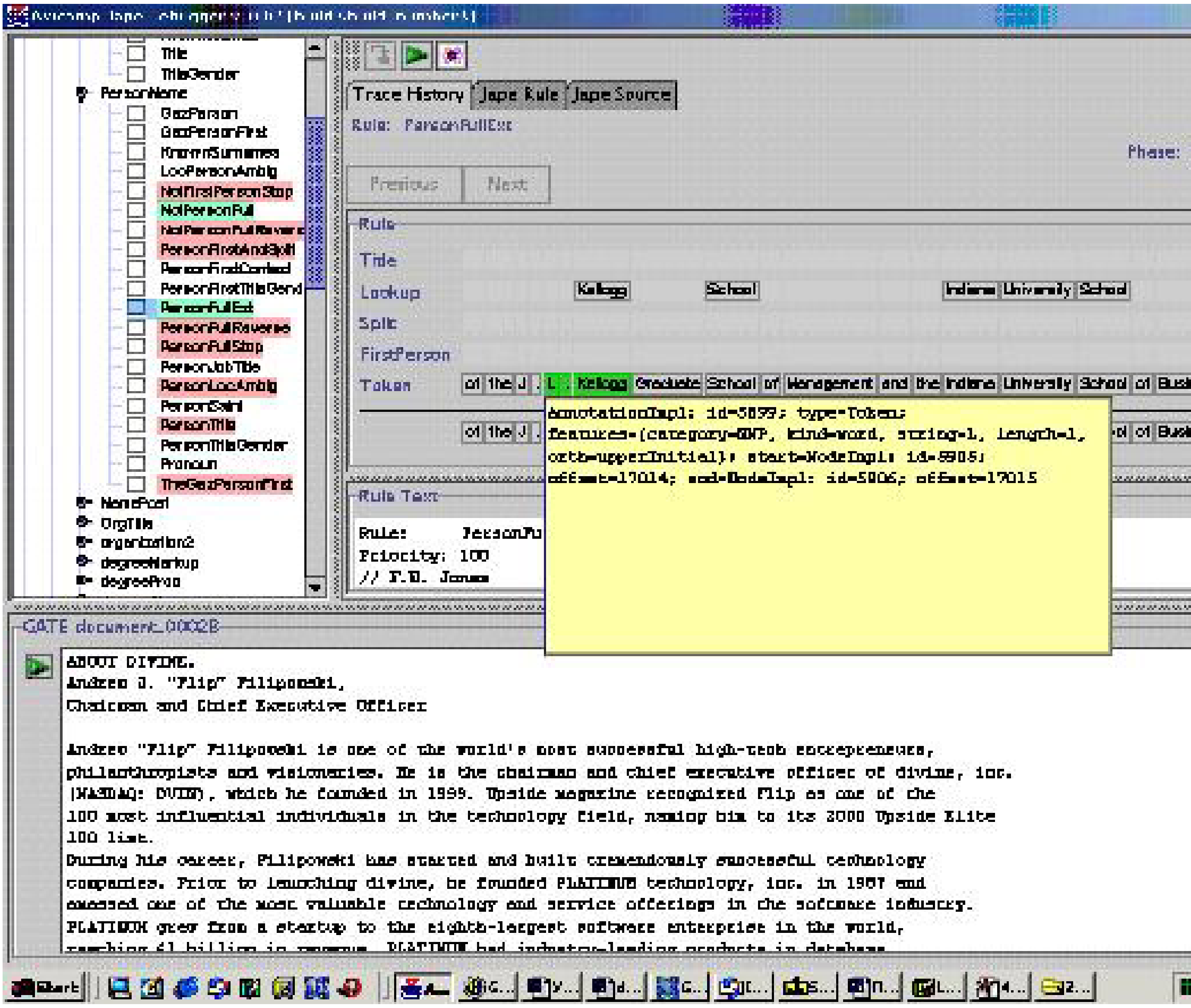


Figure 7.1: The JAPE Debugger User Interface

The debuggers 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).

7.8.2 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.
- 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 wouldnt be so easy to find the reason for this error, because the rule NotPersonFull doesnt 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.

7.8.3 Known Bugs

1. Debugger doesn't see processing resource reinitialization. A possible workaround is to close and open the resource again.

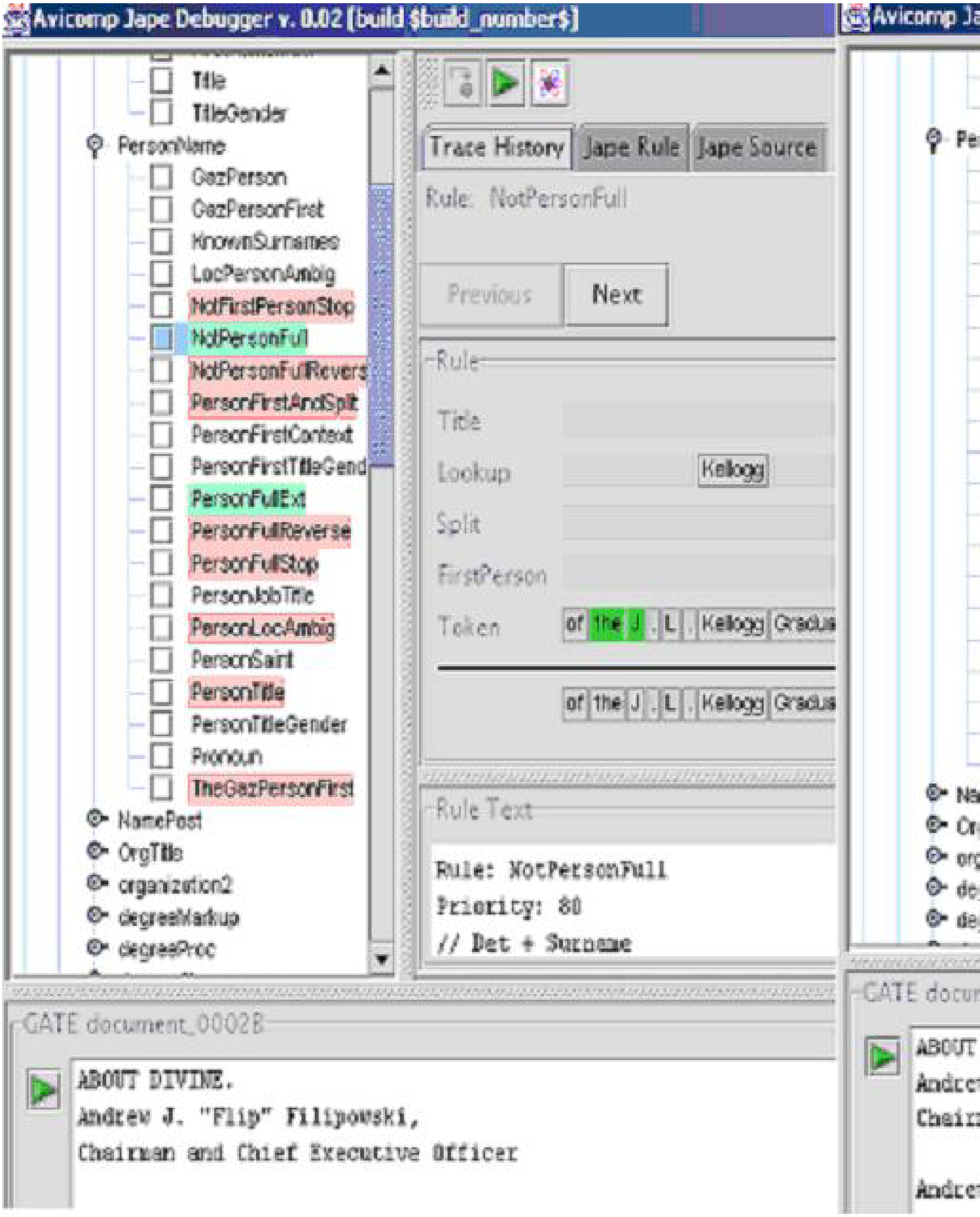


Figure 7.2: Finding Errors

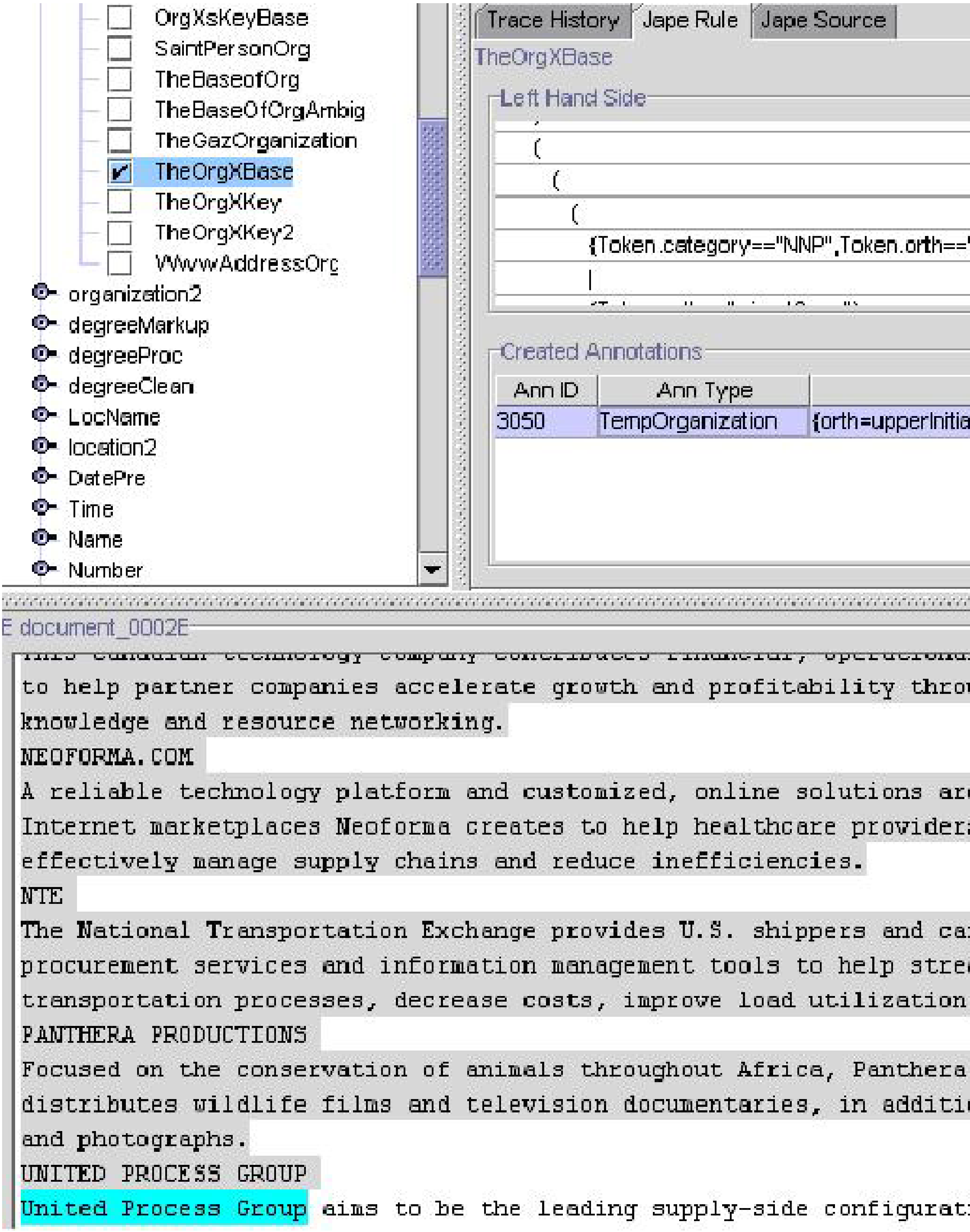


Figure 7.3: The Interface of the JAPE Debugger while Running in Breakpoint Mode

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

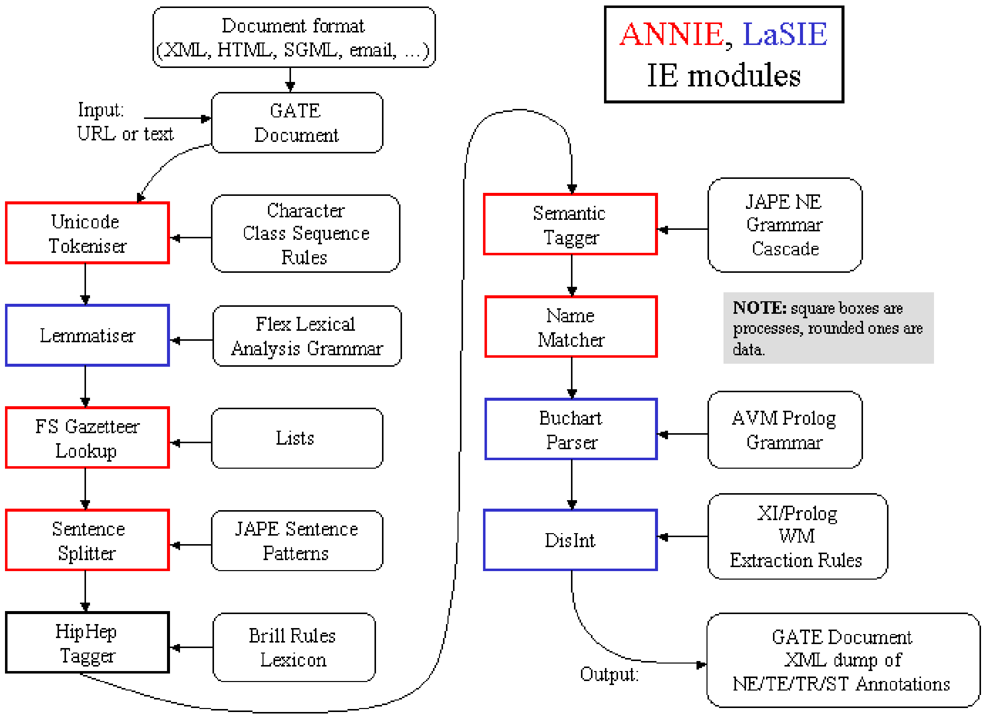


Figure 8.1: ANNIE and LaSIE

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.

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.

8.4 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 D. 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.

8.5 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.

8.6 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.

8.6.1 GATE Interface

Input – entity annotations, with an id attribute.

Output – matches attributes added to the existing entity annotations.

8.6.2 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.

8.6.3 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.

8.7 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.

8.7.1 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).

8.7.2 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.

8.7.3 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).

8.7.4 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.

8.8 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"}
```

8.8.1 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
```

8.8.2 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
```

8.8.3 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). 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 has 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 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).

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.

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

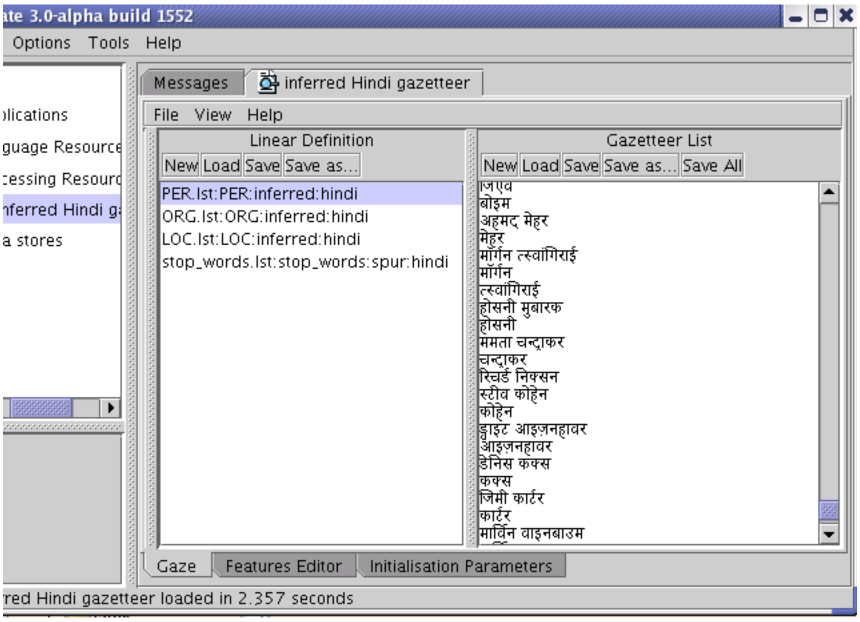


Figure 9.1: Lists collected automatically for Hindi

majorType whose value is the type of list, a minorType "inferred" (since the lists have been inferred from the text), and the language "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, Italian, Spansih and Bulgarian. 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/CLAC).. The

GATE wrapper calls TreeTagger as an external program, passing gate documents as input, and adding two new features to the existing Tokens, 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 the TreeTagger wrapper shell script. Currently, there are four shell scripts under the 'cmd' directory of the TreeTagger package, each of which corresponds to a specific language.
  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
```

- Platform dependability: TreeTagger is now supported for Windows as well as Linux and Mac OS X operating systems.

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.

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);

Figure 9.3 shows a MiniPar annotated document in GATE.

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. The GATE wrapper calls RASP as an external program, passing plain texts and shell script runall.sh as input. The Shell script runall.sh then invokes the tokeniser/tagger/morph/parser on raw texts. The output of RASP is saved as "RASPToken" annotations and grammatical relation annotations.

- Features of the RASPToken:
  i) id: id of the token;
  ii) POS: the part-of-speech tag of the token;
  iii) Morph: the base word of the token;
  iv) content: the string image of the token;
  v) length: the length of the token string.
- Grammatical relation annotations:
  i) type: grammatical relation type;
  ii) head: head token of the relationship;
  iii) dependant: dependant token of the relationship;
  iv) other parameter: other parameters of the relationship.
- Runtime parameters:
  i) document: the document to be processed
  ii) annotationSetName: the annotation set to be used for generated annotations
  iii) raspscript: a URL indicating the location of the RASP shell script runall.sh

- Platform dependability: 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.

- Requirements: RASP is available from http://www.informatics.susx.ac.uk/research/nlp/rasp/. 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 them to insert the appropriate value for the shell variable RASP, which should be the file system pathname where you have installed the RASP tools.

9.12 SUPPLE Parser (formerly BuChart)

The BuChart parser has been deprecated and replaced by SUPPLE: The Sheffield University Prolog Parser for Language Engineering. BuChart will be removed from future versions of GATE and so users are encouraged to upgrade applications 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: SyntaxTreeNodes which are used to display the parsing tree when the sentence is 'edited'; 'parse' annotations containing a bracketed representation of the parse; and 'semantics' annotations that contains the logical forms produced by the parser.

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), 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.SWIProlog or shef.nlp.supple.prolog.SWIJavaProlog).

  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 System properties

The `SICStusProlog` 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.

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.

9.12.5 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:....).
```

9.12.6 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.

9.12.7 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.

9.12.8 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 Montreal Transducer

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.

9.13.1 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}
- 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
- 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.

9.13.2 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
- The ∗ and + Kleene operators were not greedy when they occurred inside a rule. 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
```

9.14 Language Plugins

There are plugins available for processing the following languages: French, German, Spanish, 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.

We are grateful to Fabio Ciravegna and the Dot.KOM project for use of some of the components for the German plugin.

If you wish to use the applications as they are, you do not need to load the plugins as the relevant plugins will be automatically loaded. The language applications can be found in the resources directory of the relevant plugin (e.g. you can find the Chinese application in the Chinese plugin directory gate/plugins/chinese/resources/chinese.gapp). Note that application files conventionally have the suffix ".gapp" (which stands for Gate APPlication). There may be other applications also present in the resources directory, which you are welcome to use (though they may not necessarily be documented). For example, the Chinese plugin has applications to create lists of various patterns from the corpus, using the gazetteer lists collector PR.

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..

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 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).

9.15.1 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 three init parameters giving the locations of the two gazetteer list definitions 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).

9.16 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.

9.17 Annotation Set Transfer

The Annotation Set Transfer enables the parts of a document matching a particular annotation to be transferred into a new annotation set. 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 which is to be transferred. If nothing is specified, the Default annotation set will be used.
- outputASName - this defines the new annotation set which will contain the transferred annotations. The default for this is a set called Filtered.

- tagASName - this defines the annotation set which contains the annotation covering the relevant part of the document to be transferred.
- textTagName - this defines the name of the annotation covering the relevant part of the document to be transferred.

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

9.18 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:

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 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 11.4 for more details).

9.18.1 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.

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).

9.18.2 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());
}
```

9.19 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.

9.19.1 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. Once the crawler is initialized, the PR automatically creates a corpus named crawl where all the documents from the web crawl will be stored. 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).

- 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
- 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 corpus crawl. Figure 9.7 shows the crawled pages added to the corpus.

9.20 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.

9.20.1 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.

9.21 WordNet in GATE

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
                <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

9.21.1 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**

9.22 Machine Learning in GATE

9.22.1 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.

9.22.2 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 E.

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 <, &rt; and &. 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.
- **OPTIONS**: the contents of the OPTIONS element will be passed verbatim to the ML engine used.

9.22.3 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 classifications 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.

9.22.4 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.

9.22.5 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.

9.22.6 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 algorithm, 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.

9.22.7 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.

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 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.

9.23 Probability Finder

The probability finder PR is used to obtain the probabilities for the occurrences of each of the named entities found in a corpus.

The following describes the basic steps to use it

- Build the corpus either manually adding documents or by querying Google or crawling the WWW
- Once the corpus is built, load ANNIE

- Perform any processing required to find the Named Entities, using any JAPE Transducers, etc.
- Load the ProbabilityPR via the Manage Plugins menu
- Create a new pipeline or add the Probability PR to the existing pipeline.
- In the runtime parameters, you will see the following options:
  - the Entity Type to be used, e.g. Person, Location, Organization. (Note that you can use only entities in the Default Annotation Set, and that you can only use one entity type at a time)
  - the processed corpus to be used.
- Once run, the output in the messages tab will correspond to the basic probabilistic outcomes of occurrence for each entity. Please note that there is no co-reference resolution done.

9.24 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 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/.

9.25 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).

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.

9.25.1 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.

9.26 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).

9.26.1 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.

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.

9.26.2 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.

9.27 Ontotext JapeC Compiler

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[1] and associated libraries. The japec compiler can then be built by running:

```
../../bin/ant japec.clean japec
```

from the Jape_Compiler plugin directory.

[1] GHC version 6.4.1 was used to build the supplied binaries for Windows, Linux and Mac

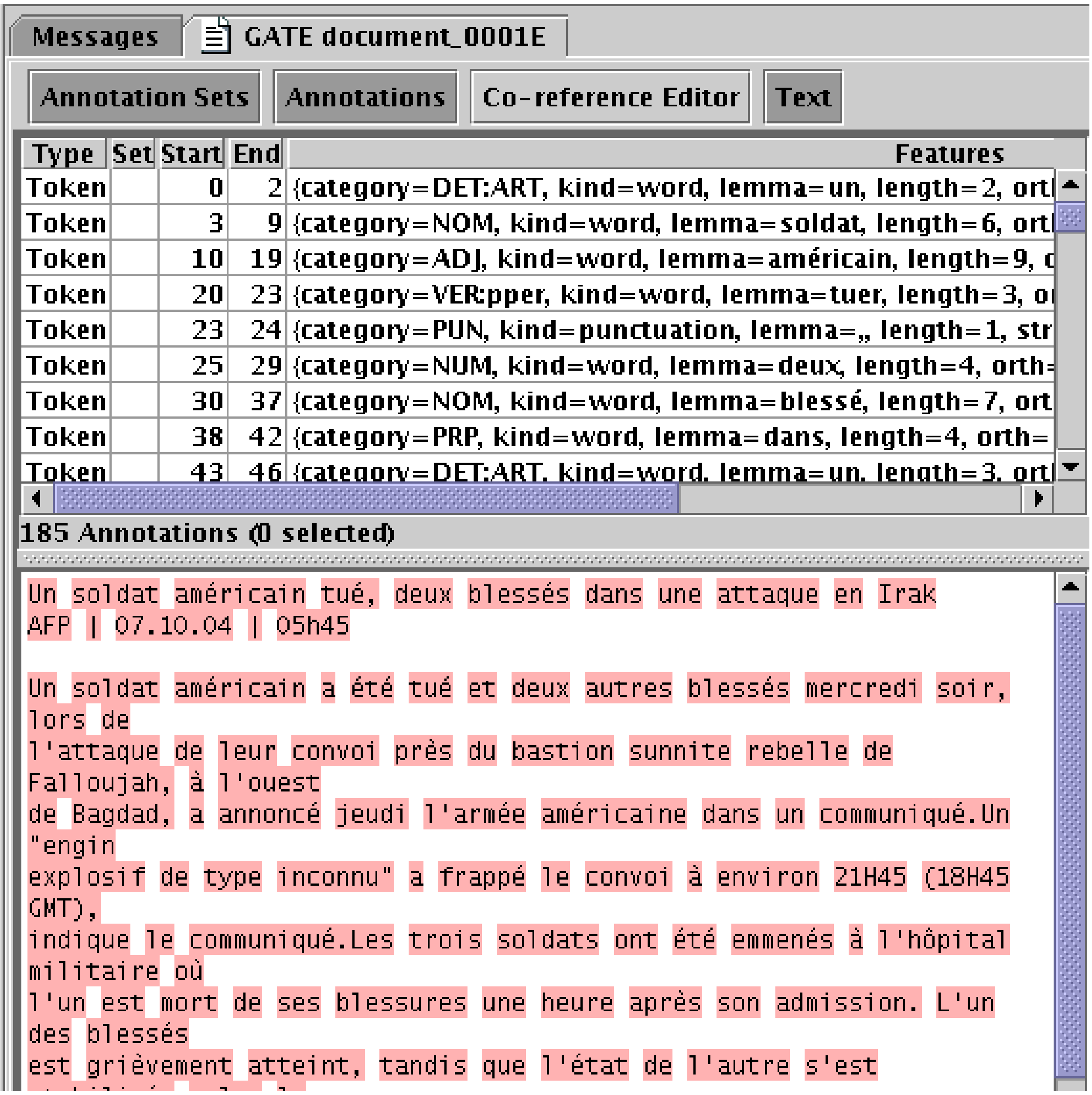


Figure 9.2: a TreeTagger processed document

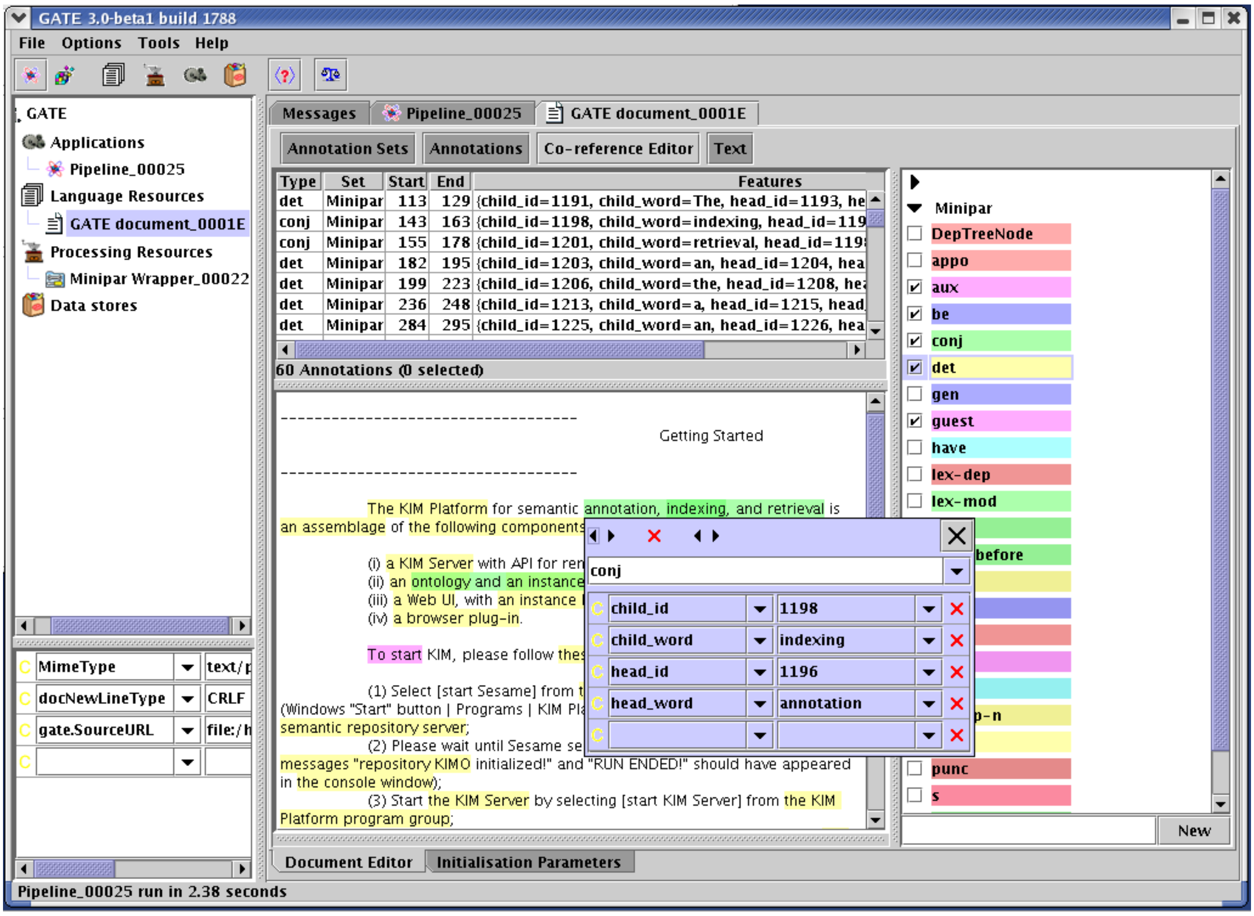


Figure 9.3: a MiniPar annotated document

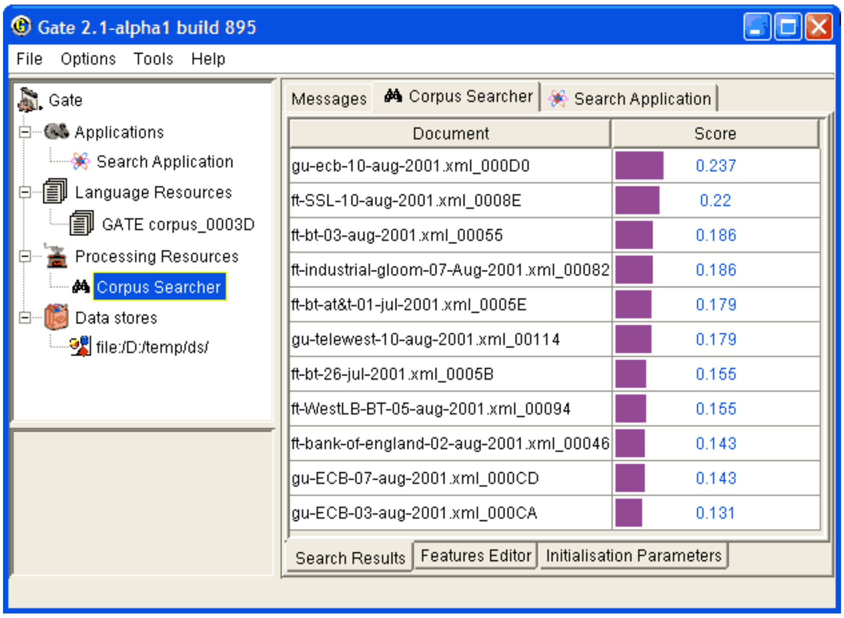


Figure 9.4: Documents with scores, returned from a search over a corpus

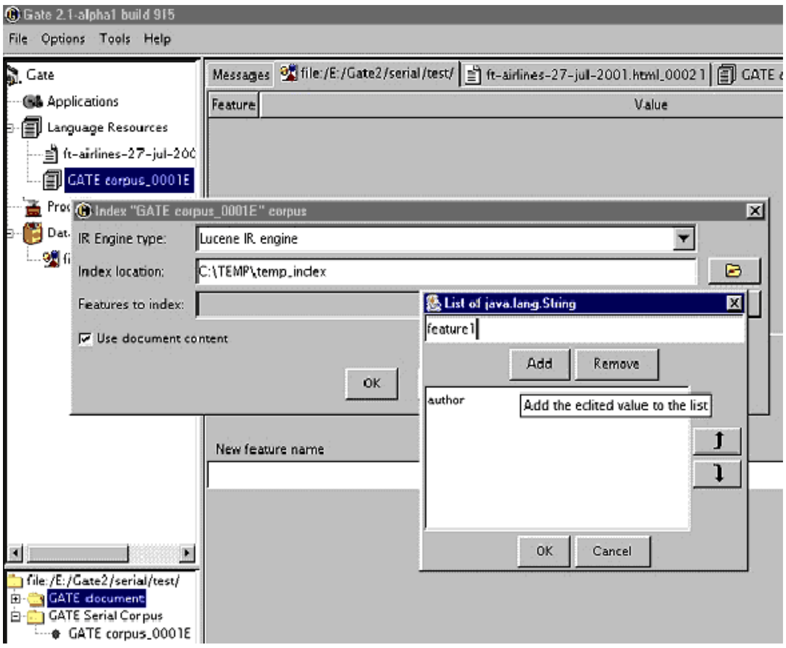


Figure 9.5: Indexing a corpus by specifying the index location and indexed features (and content)

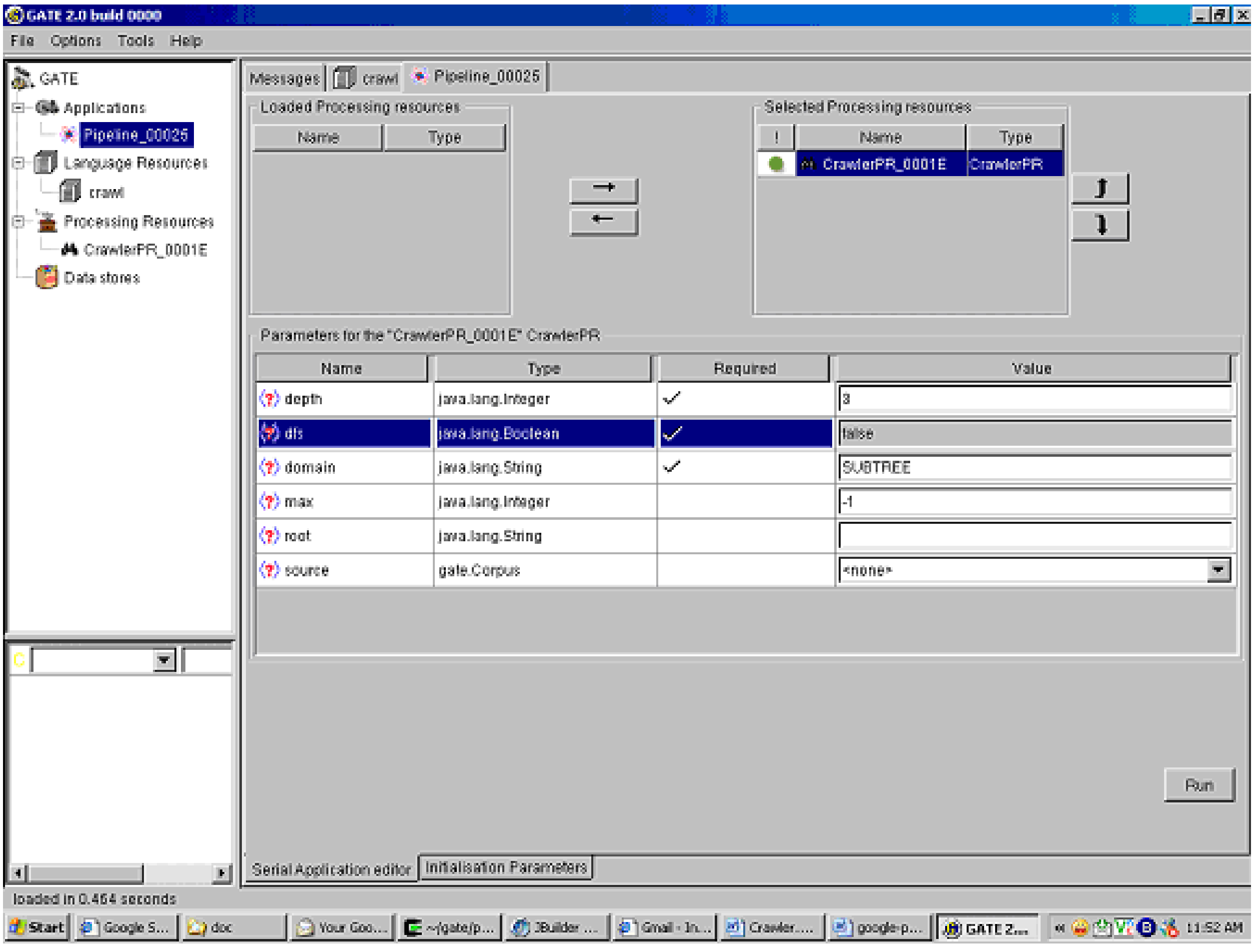


Figure 9.6: Crawler parameters

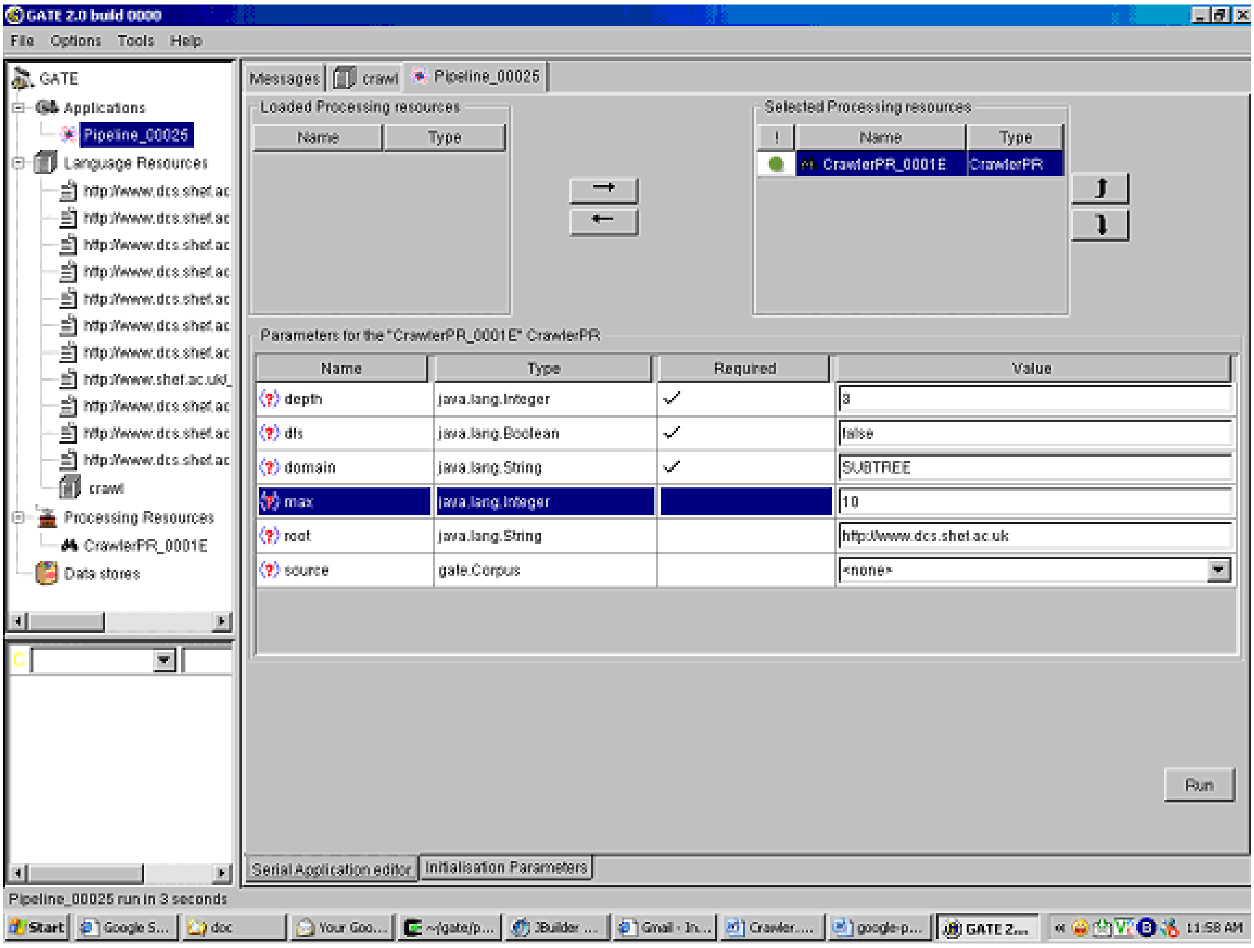


Figure 9.7: Crawled pages added to the corpus

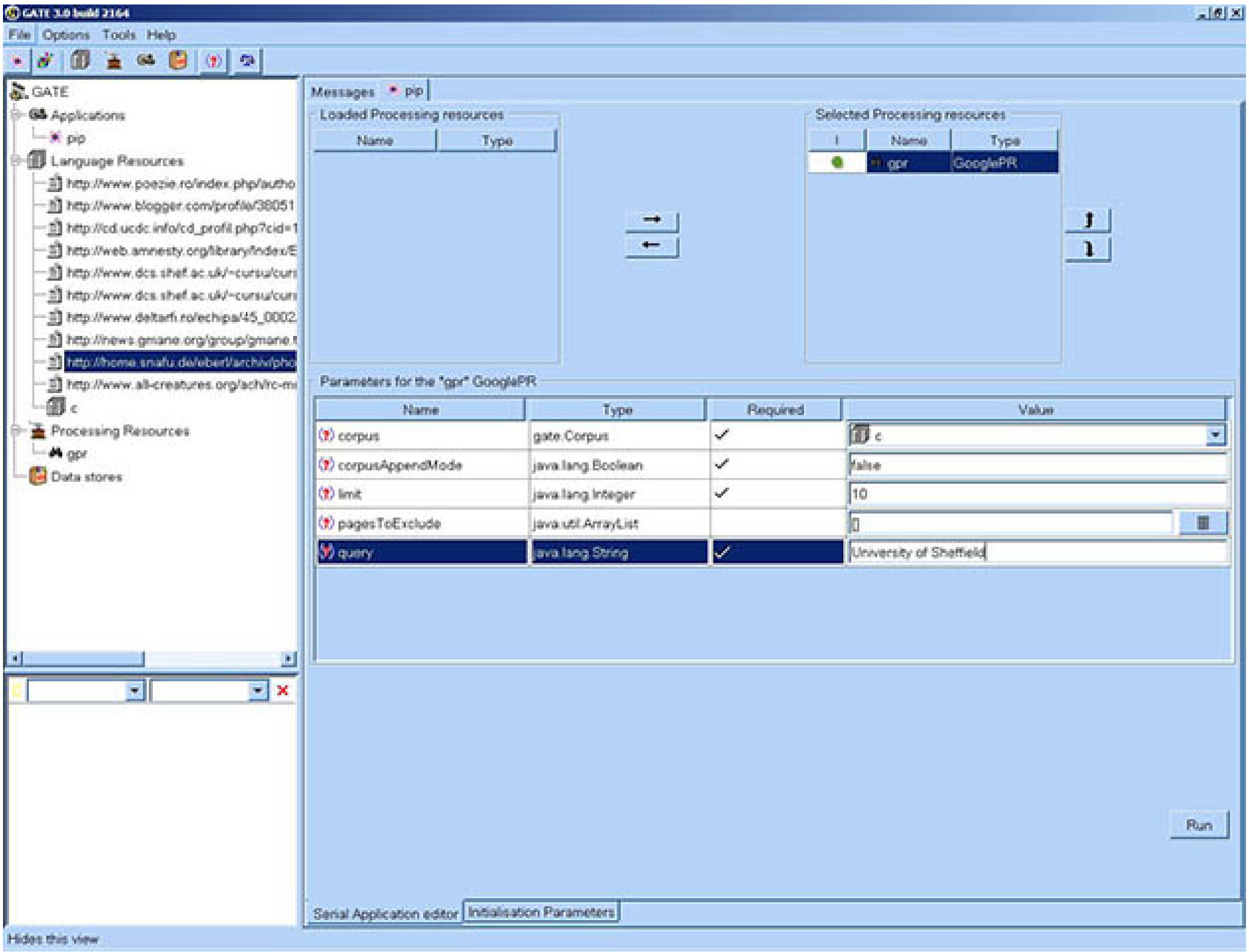


Figure 9.8: URLs added to the corpus

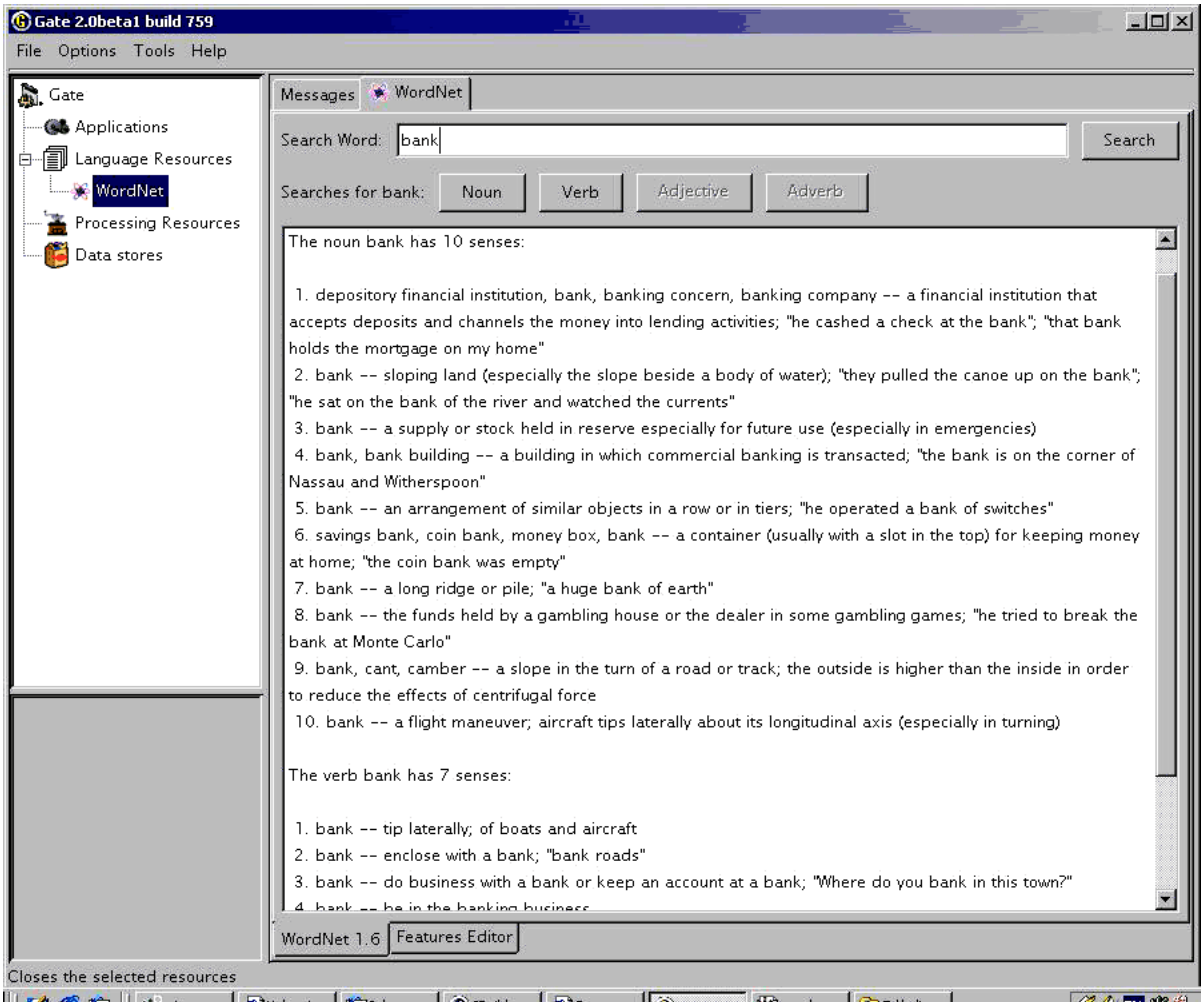


Figure 9.9: WordNet in GATE – results for "bank"

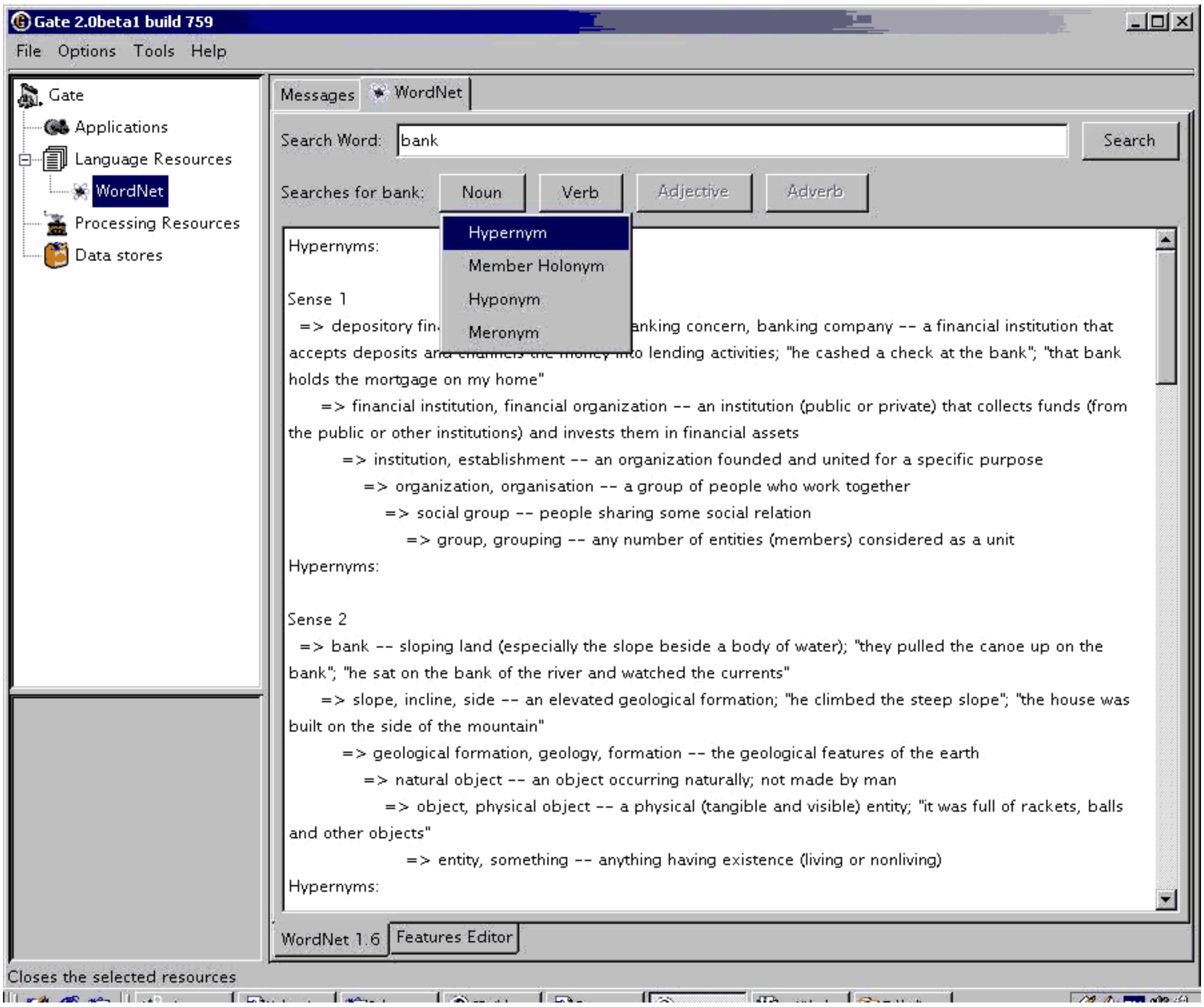


Figure 9.10: WordNet in GATE

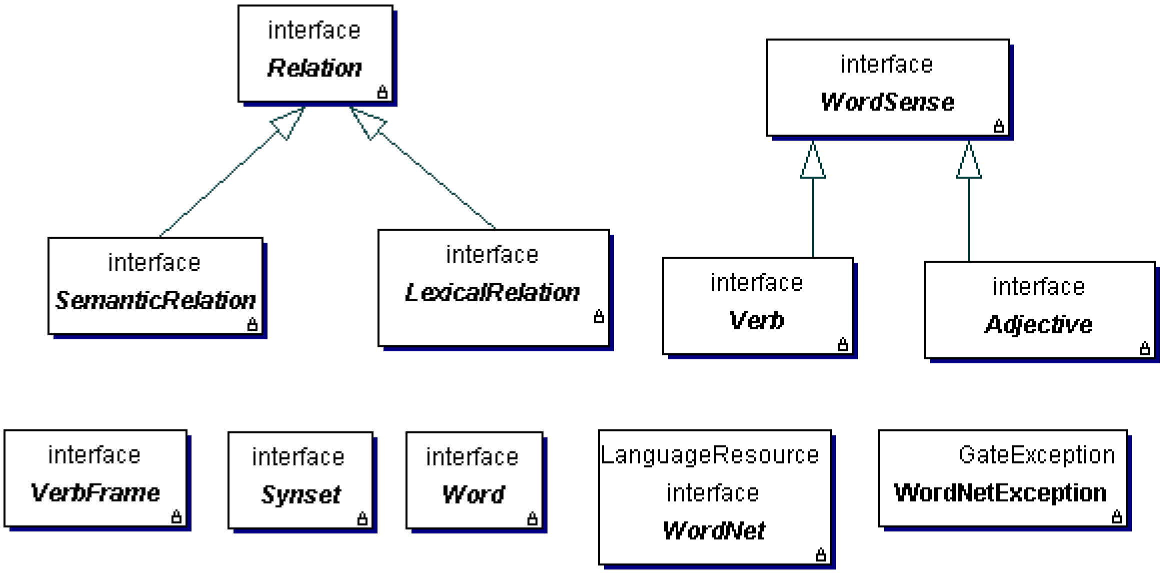


Figure 9.11: The Wordnet API

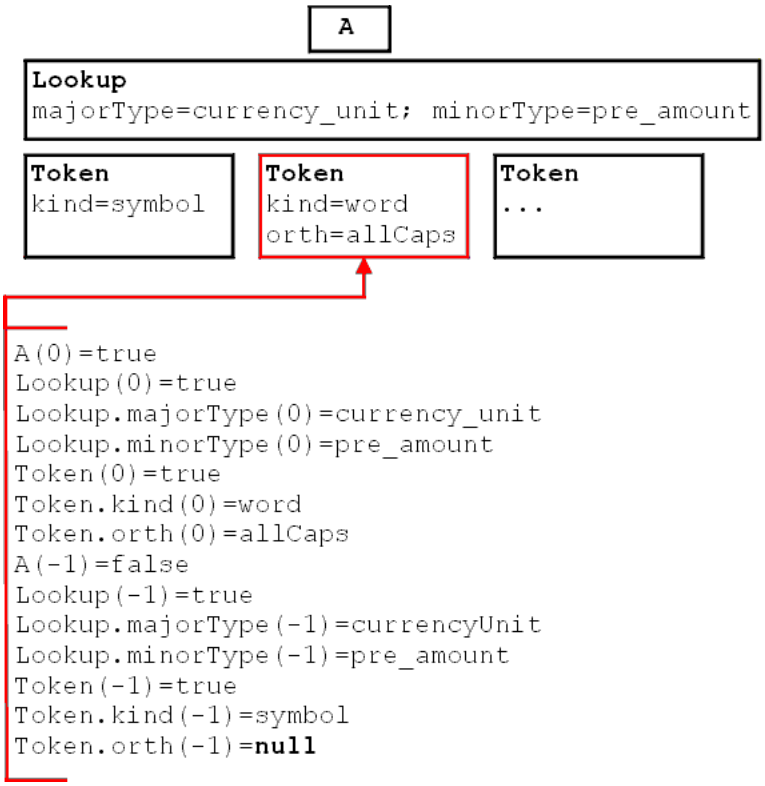


Figure 9.12: Sample attributes and their values

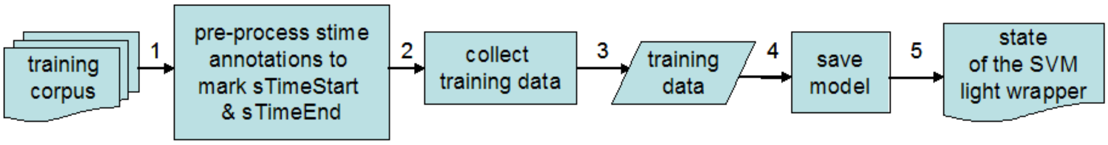


Figure 9.13: Flow diagram explaining the SVM training data collection

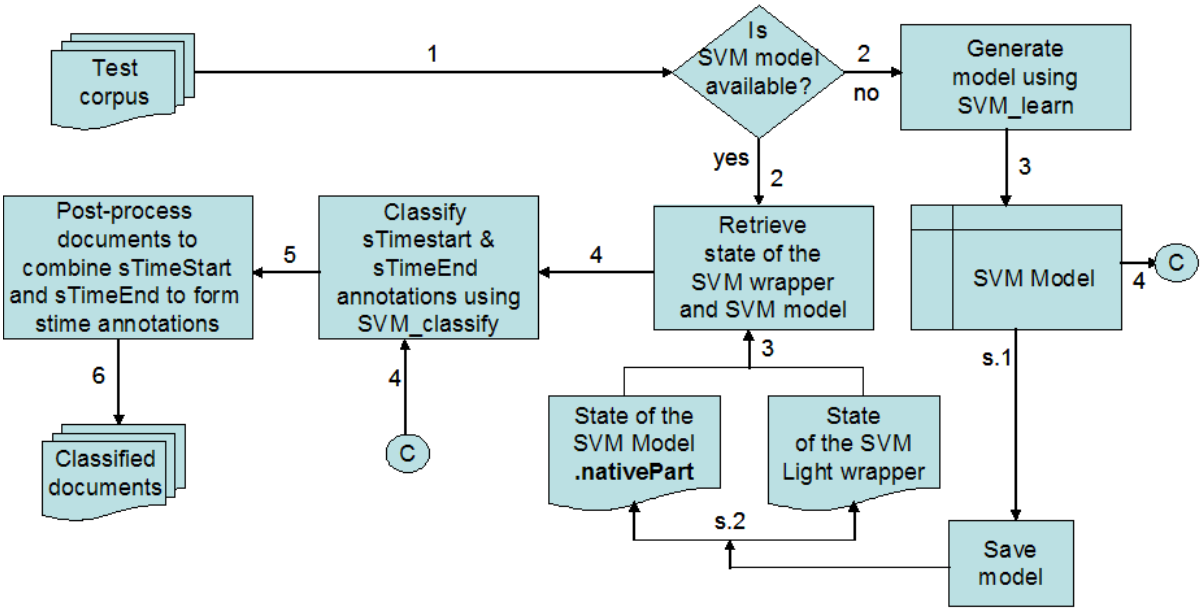


Figure 9.14: Flow diagram explaining document classifying process

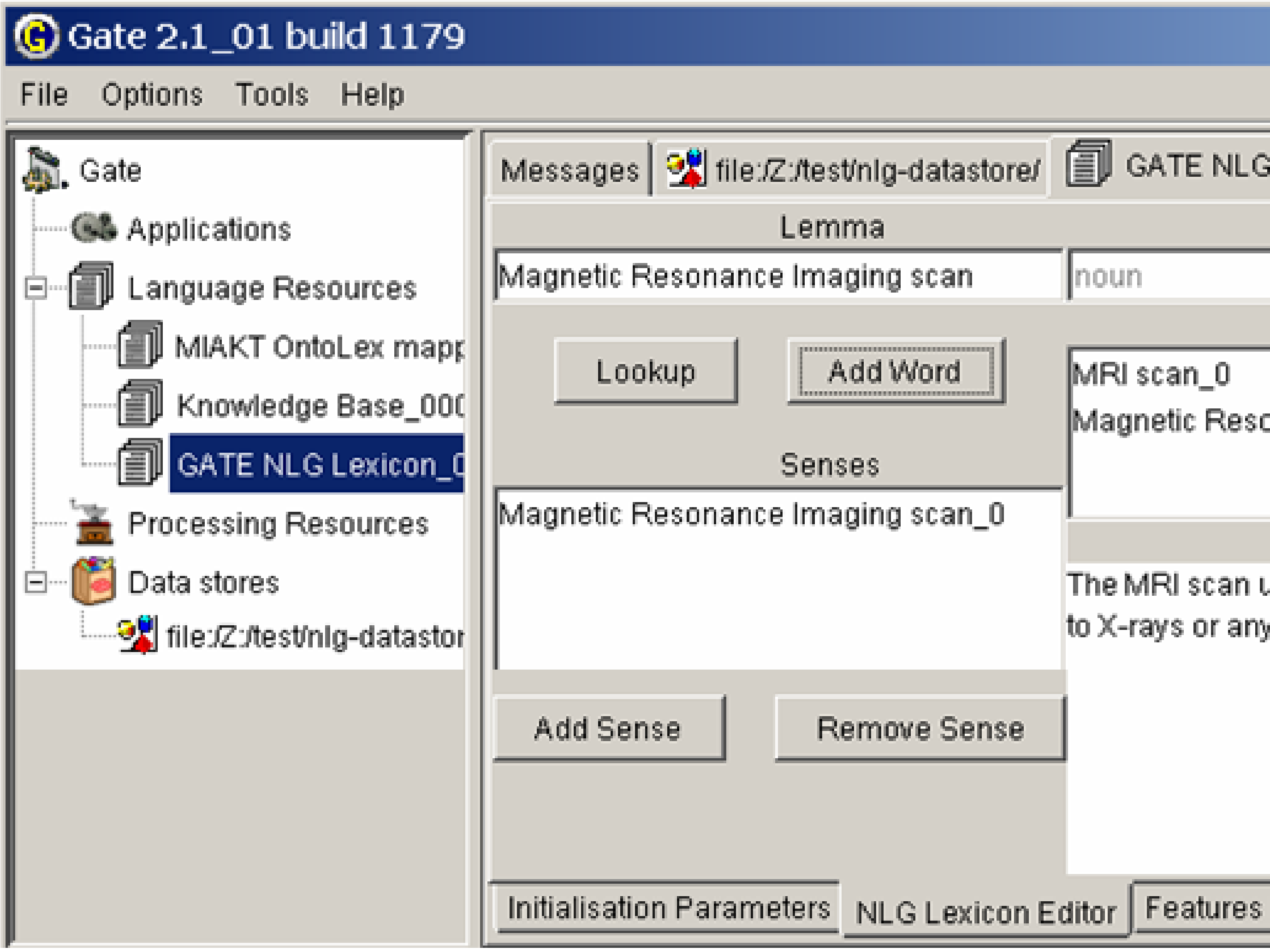


Figure 9.15: Example NLG Lexicon

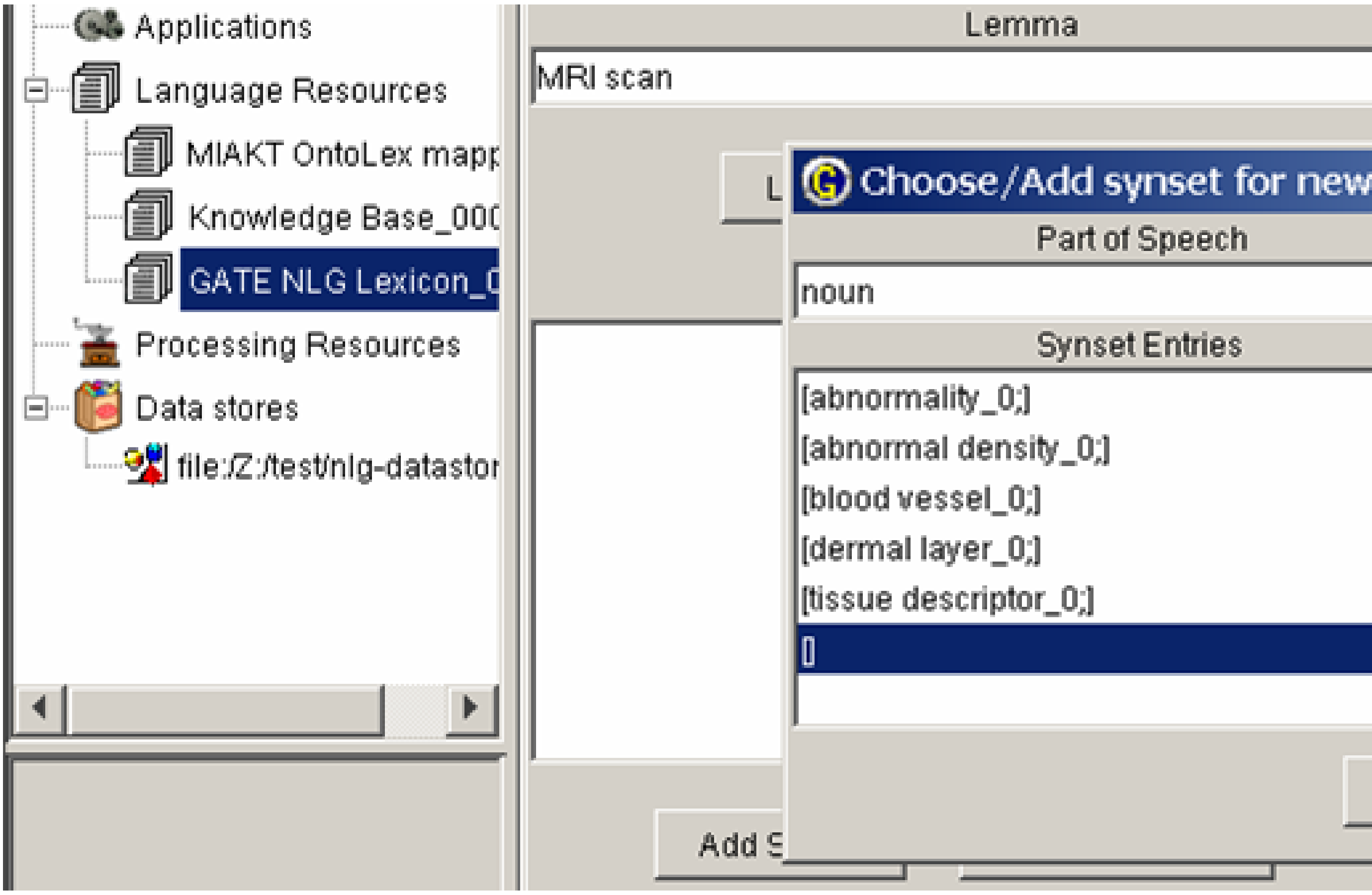


Figure 9.16: Editing synset information

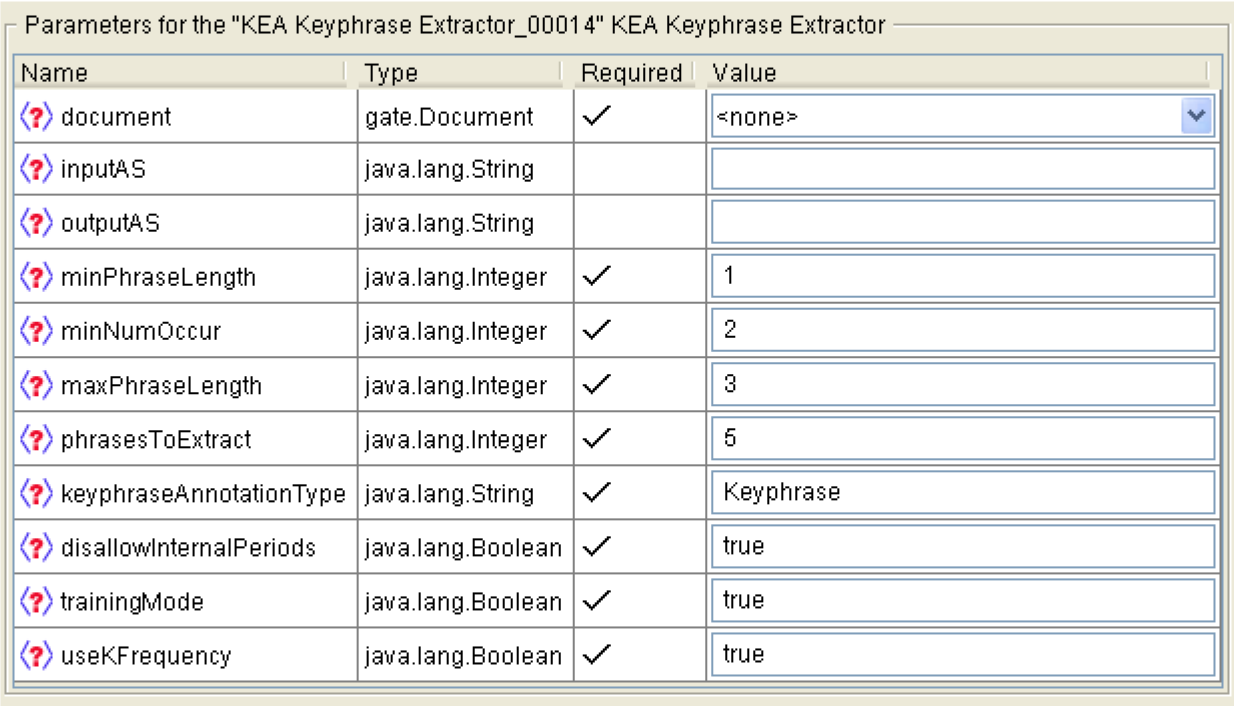


Figure 9.17: Parameters used by the Kea PR

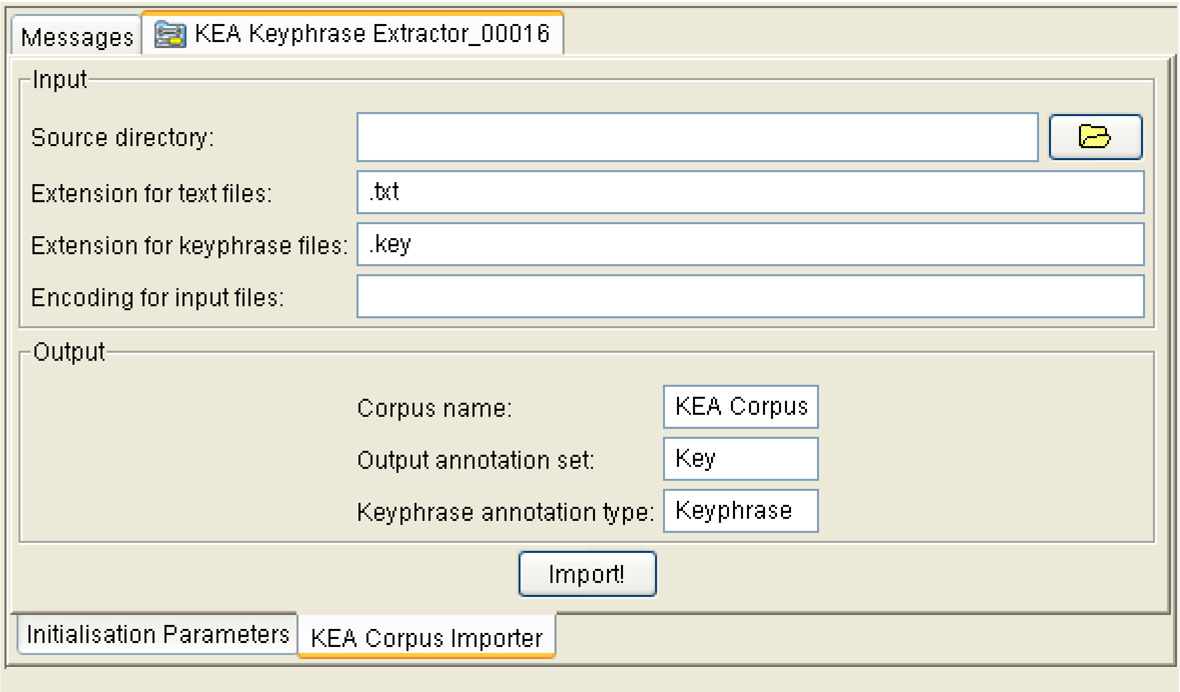


Figure 9.18: Options for the "KEA Corpus Importer"

Chapter 10

Working with Ontologies

WARNING: The ontologies support in GATE is currently under review and is likely to change before the next release. It should not be considered *stable*, it is only included with the distribution for convenience reasons. Another thing to note is that support for DAML has been removed as it has been superceded by the newer OWL support.

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 only supports those features that are 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, hierarchies of properties, and instances (also known as individuals).

Hierarchies of classes

The central role in the ontology data model is played by the class hierarchy, also known as a *taxonomy*. This consists of a set of classes linked by `subClassOf` and `superClassOf` relations. Classes have a `name` and a `URI`; in most cases the `name` is the local part of the `URI`, though this is not enforced. Class names within an ontology must be unique. Classes can also have `comments` which are used to explain their intended meaning.

All classes can have a set of super classes and a set of sub classes which 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 this relations given a class. The transitive closure for the set of super classes for a given class is a set containing all the super classes of that class, as well as all the super classes of its direct super classes, and so on until no more super classes are found. This calculation is finite, the upper bound being the set of all the classes in the ontology. A class that has no super classes 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 are probably denoting an error in the definition of the ontology. Care should be taken to avoid such situations.

Instances

Instances are objects that belong to classes. Like classes, they have a `name` and a `URI` and can have `comments`. One instance can belong to one or more classes and can have properties with values. There are API methods provided for getting all the instances that belong to a given class or the property values for a given instance.

Hierarchies of properties

The last part of the data model is made up of hierarchies of properties that can be associated to objects in the ontology. GATE defines three types of properties:

Object Properties are properties that are associated to an ontology instance and have an instance as value.

Datatype Properties are properties that are associated to an ontology instance and can have any Java object as value.

Generic Properties are associated to any ontology object, be it a `class`, an `instance`, or another `property`, and have any Java object as value.

From an OWL/RDF perspective, object and datatype properties are similar to the property types as defined by OWL variants while the generic properties are intended to model the properties defined by RDF.

Unlike some other representation models, in GATE, properties do not 'belong' to classes, 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`. A `domain` for a property is a set of ontology classes. The classes listed in the domain should be seen as a set of restrictions: for a property to be applicable to an instance, that instance needs to belong to all the classes in the property's domain. A property with a domain that is an empty set can apply to instances of any type (i.e. there are no restrictions given).

Similarly, the types of values that a property can take is restricted through the definition of a `range`. Object properties take instances as values so their ranges are also sets of classes; the manner in which the range restrictions are interpreted is similar to the case of the domain. Datatype and generic properties can have as value any Java object. Their ranges are sets of Java Classes (objects of type `java.lang.Class`) which means they can only have values that implement those classes. In most cases, such a range will either be empty (allowing any type of object) or contain a single Class value (the most specific Class from the class hierarchy that is appropriate). Having several classes that not in a hierarchical relation effectively blocks that property from accepting **any** value, while including several classes that hierarchically linked (derived from one another) is superfluous as only the most specific one is significant. Of course a range can contain a set of objects representing interfaces, in which case non-singleton sets make sense.

Although the data-model and API permit the use of any kind of values for datatype and generic properties, if input/output is required using XML serialisations (such as XML-based OWL or RDF), then the allowed values are restricted by the set of types representable in those serialisations (usually the ones defined by the XML-Schema specification).

All 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`. Additionally, object properties can also be `reflexive`, `symmetric`, and `transitive`. If one likens ontology properties with algebraic relations, the semantics of these become apparent.

Similar to classes, properties can also be organised in hierarchies by means of establishing `subPropertyOf` and `superPropertyOf` relations. Methods are provided for obtaining the set of super- and sub-properties as well as the transitive closures of these sets.

The API provides methods for obtaining the names of the properties that are set for a given instance and for obtaining the sets of values for a given property and a given instance.

10.2 The Ontology Resource

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. Once this is done, a new language resource called '`Jena Ontology`' becomes available.

A new ontology can be created using the usual mechanism by either right-clicking on the '`Language Resources`' sub-tree in the main resources tree and choosing '`Jena Ontology`' or by using the '`New Language Resource`' option in then main '`File`' menu. This will open a dialogue like the one shown in Figure 10.1.

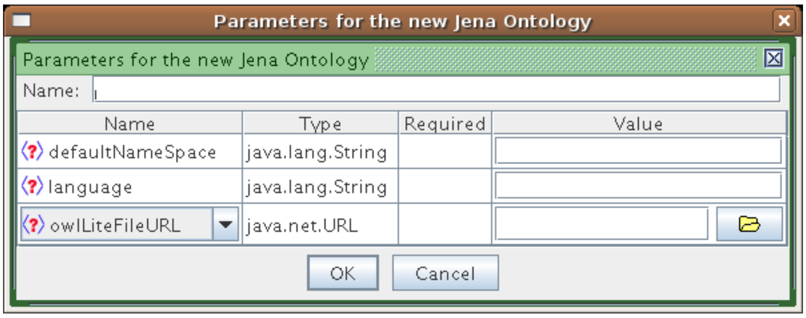


Figure 10.1: Parameters for a new ontology

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:

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 if the ontology being loaded will only be used in read-only fashion.

language the default language indicator (e.g. 'EN' for English) for the labels and comments used throughout the ontology.

... and one of the following, which can be chosen from the drop-down menu:

- **owlLiteFileURL** a URL pointing to a file containing OWL Lite data
- **owDlFileURL** a URL pointing to a file containing OWL DL data

owlFullFileURL a URL pointing to a file containing OWL Full data

rdfsFileURL a URL pointing to a file containing RDF(S) 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.

Another option that is available is cleaning the ontology – which will delete all the information from the ontology, and saving it to a file – which will save the data in the ontology in OWL Lite format.

10.3 GATE's Ontology Viewer

The ontology support in GATE also includes a simple viewer that can be used to navigate an ontology and quickly inspect the information relating to any of the objects defined in it – classes, instances and their properties.

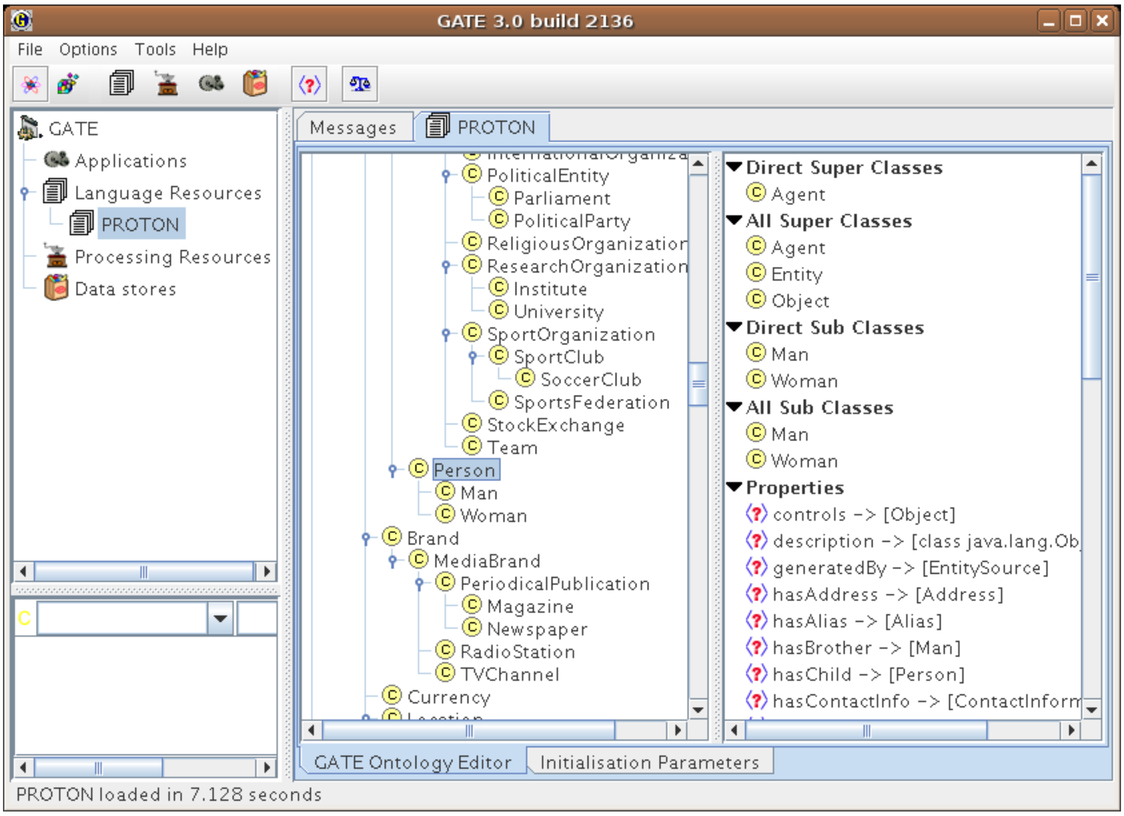


Figure 10.2: The GATE Ontology Viewer

Figure 10.2 shows the GATE ontology viewer displaying a segment of the PROTON[1] ontology and the data associated with the class `Person` which is currently selected. The viewer is divided into two areas, one on the left that displays the hierarchy of classes as well as any instances and another to the right that displays the details pertaining to the object currently selected in the hierarchical view.

The left-hand-side view displays a tree which shows all the classes defined in the ontology. The tree can have several root nodes - one for each top class in the ontology. The same tree is also used to show the instances for each of the classes. Instances that belong to several classes are shown as children of all the classes they belong to.

Whenever a item is selected in the tree view, the right-hand-side view is populated with the details that are appropriate for the selected object. If the object is an ontology class then the details include the set of direct super classes, the set of all the super classes using the transitive closure of the hierarchical relations, the set of direct sub classes, the set of all the sub classes, and the set of applicable properties.

When the selected object is an instance, the details displayed include the set of direct types (the list of classes this instance is known to belong to), the set of all types this instance belongs to (obtained through a transitive closure of the set of direct types), and the values for all the properties that are set.

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 of these lists 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.

[1]The PROTON ontology is available from `http://proton.semanticweb.org/`

10.4 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 the 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`.

10.5 Populating Ontologies

One 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:

```
1  Rule: FindEntities
2  ({Mention}):mention
3  -->
4  {
5    //find the annotation matched by LHS
6    //we know the annotation set returned
7    //will always contain a single annotation
8    Annotation mentionAnn = (Annotation)
9      ((AnnotationSet)bindings.get("mention")).
10     iterator().next();
11
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 = doc.getContent().
18     getContent(
19       mentionAnn.getStartNode().getOffset(),
20       mentionAnn.getEndNode().getOffset()).
21     toString();
22
23   //add the instance to the ontology
24   //first identify the class
25   TClass aClass = ontology.getClassByName(className);
26   if(aClass == null){
27     System.err.println("Error class \"" + className
28       + "\" does not exist!");
29   }
30   //now create the instance in the ontology
31   ontology.addInstance(mentionName, (OClass)aClass);
32 }
```

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 16–21). 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 24–29) and then a new instance for that class is created (lines 30–31).

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 when an entity mention is identified in the text, the application would have to check whether the entity mentioned is already known to the ontology and only add a new instance when it is not found. Also 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.

Chapter 11

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.

11.1 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 11.1 shows a part of the AnnotationDiff viewer.

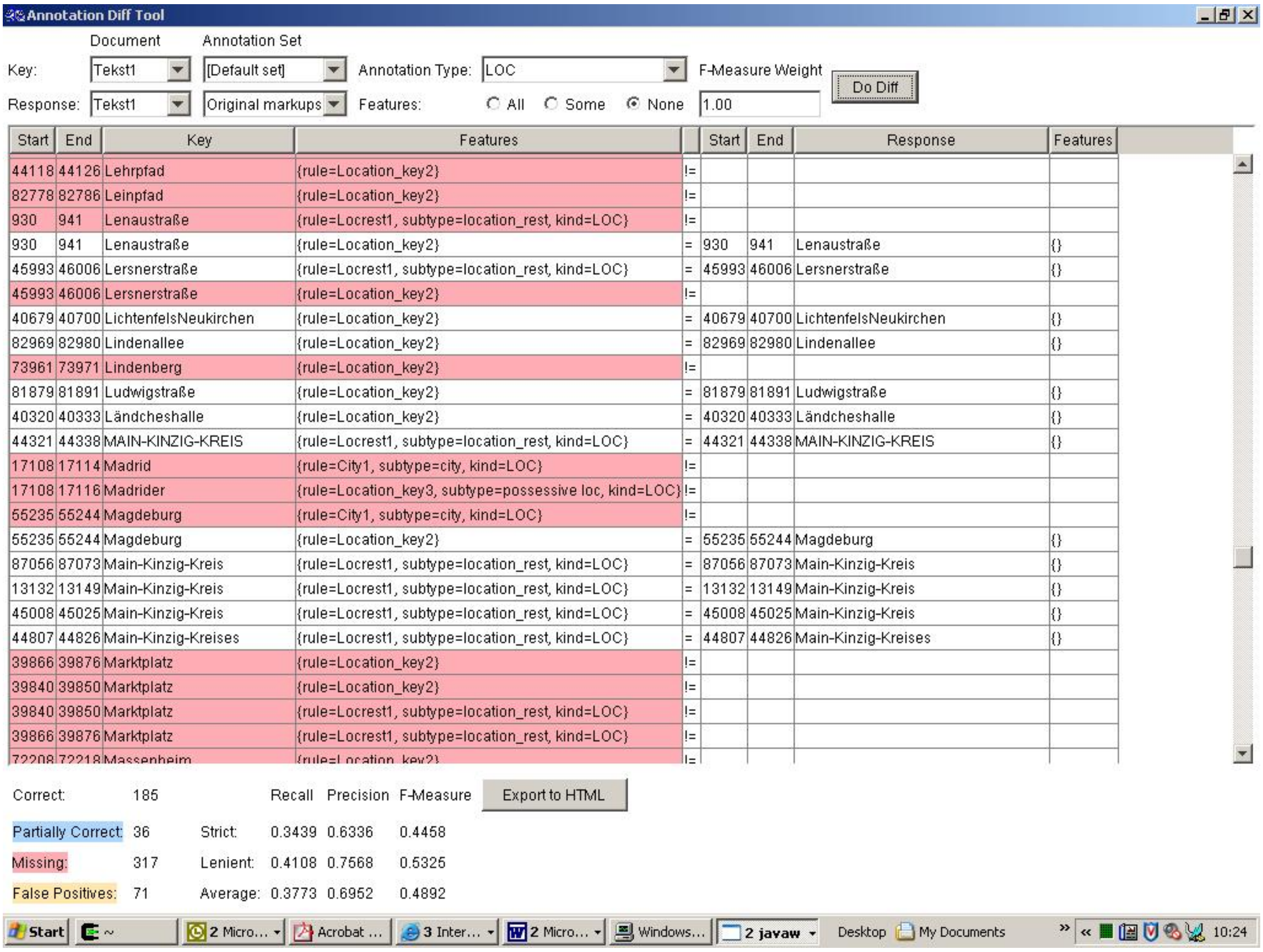


Figure 11.1: Part of the AnnotationDiff viewer

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 3.22, which describes the Annotation Diff parameters.

11.2 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.

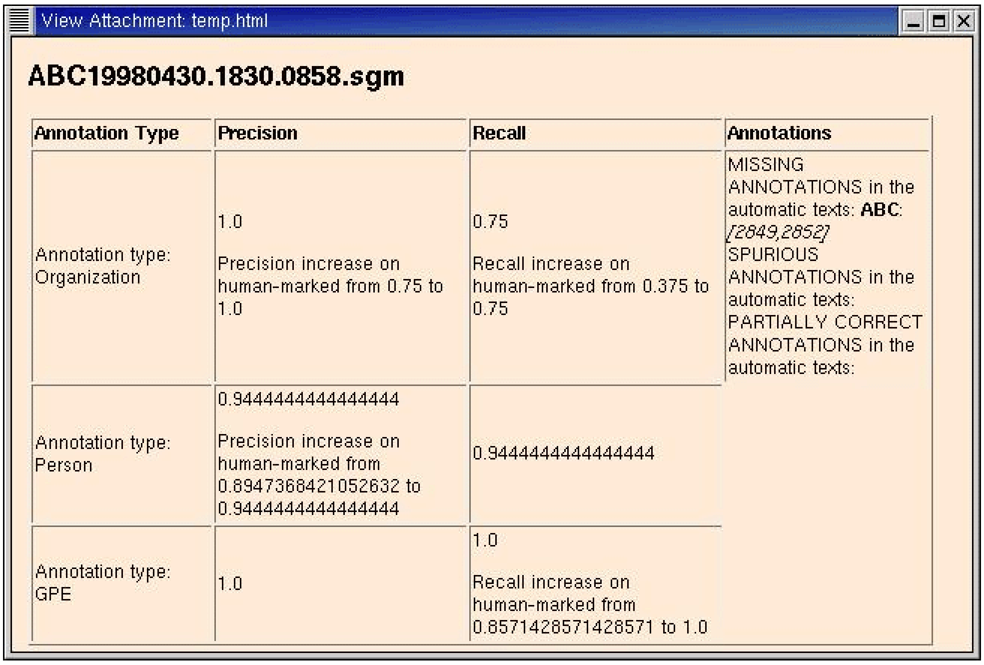
11.3 Benchmarking tool

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 3.23.

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 11.2).


View Attachment: temp.html

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 11.2: Fragment of results from benchmark tool

11.4 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 mnay 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:

$$Precision = \frac{Correct + 1/2 Partial}{Correct + Spurious + 1/2 Partial} \tag{11.1}$$

$$Recall = \frac{Correct + 1/2 Partial}{Correct + Missing + 1/2 Partial} \tag{11.2}$$

$$F - measure = \frac{(\beta^2 + 1)P * R}{(\beta^2 R) + P} \tag{11.3}$$

where β reflects the weighting of P vs. R. If β is set to 1, the two are weighted equally.

$$FalsePositive = \frac{Spurious}{c} \tag{11.4}$$

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.

11.5 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.

Chapter 12

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.

12.1 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.

12.2 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.

12.2.1 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 12.1.

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 12.1: 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.

12.2.2 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:

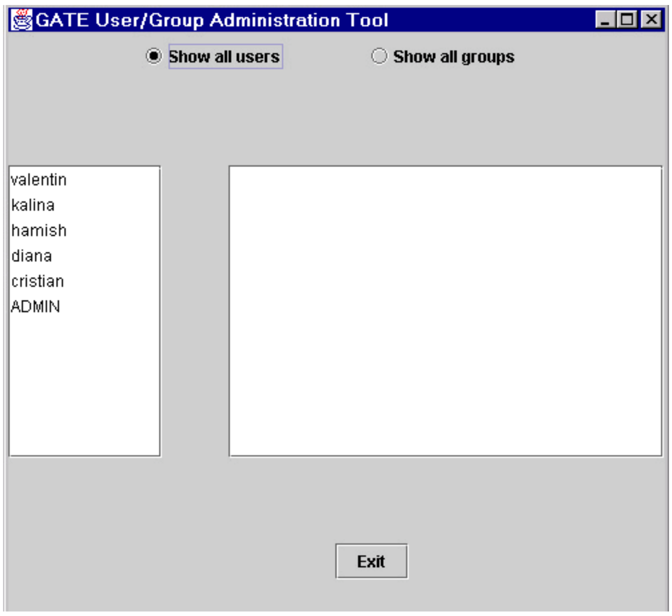


Figure 12.1: The User/Group Administration Tool

Viewing user and group information

As shown in Figure 12.1, 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 12.2).

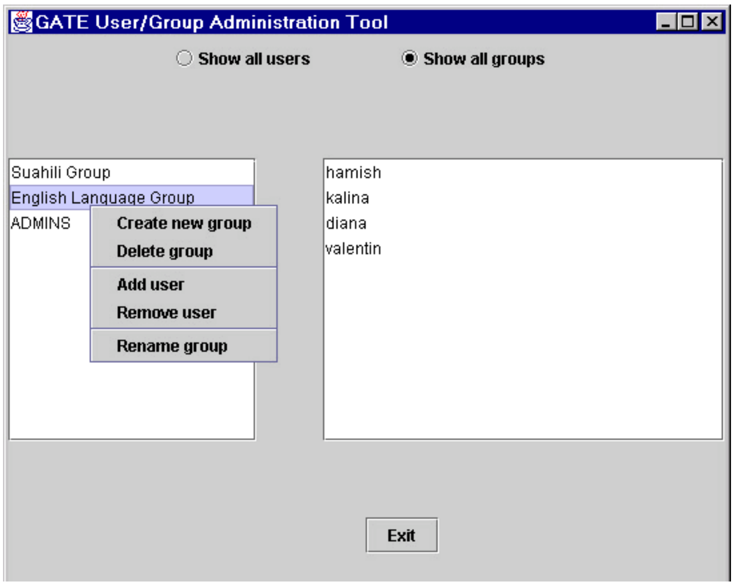


Figure 12.2: The tool in a group administration mode

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.

12.2.3 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 13

Developing GATE

This chapter describes the protocols to follow and other information for those involved in developing GATE.

13.1 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.

13.1.1 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.

13.1.2 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);

13.1.3 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 13.1). 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.

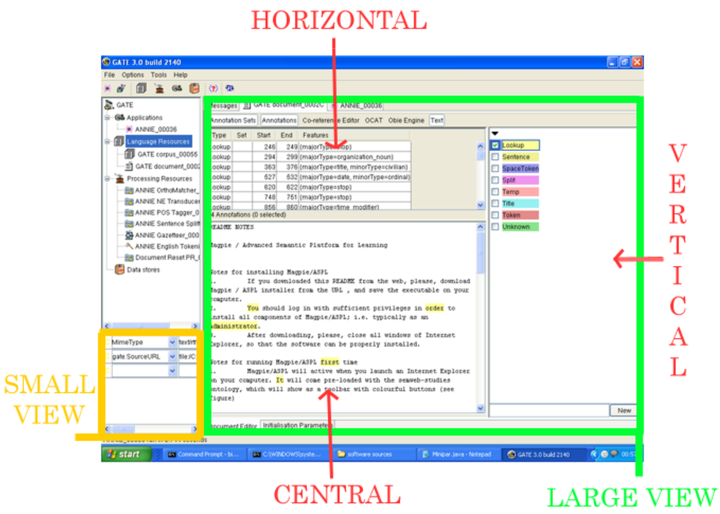


Figure 13.1: GATE GUI

```
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 13.1). 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>
```

13.1.4 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 CVS, 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. BuChart).

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 CVS,

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 cvs update again and finally rename it back.

Chapter 14

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 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 14.1 describes the GATE TAE wrapper, and section 14.2 describes the UIMA CorpusController wrapper.

14.1 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.

14.1.1 Mapping File Format

Figure 14.1 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 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

```
<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 14.1: Structure of a mapping descriptor for a TAE in GATE

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 14.2 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 14.2: An example mapping descriptor

14.1.2 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.

14.1.3 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.

14.1.4 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`.

14.2 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.

14.2.1 Mapping file format

The mapping descriptor format is virtually identical to that described in section 14.1.1, 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 14.3 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 14.3: An example mapping descriptor for the GATE POS tagger

14.2.2 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.

14.2.3 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

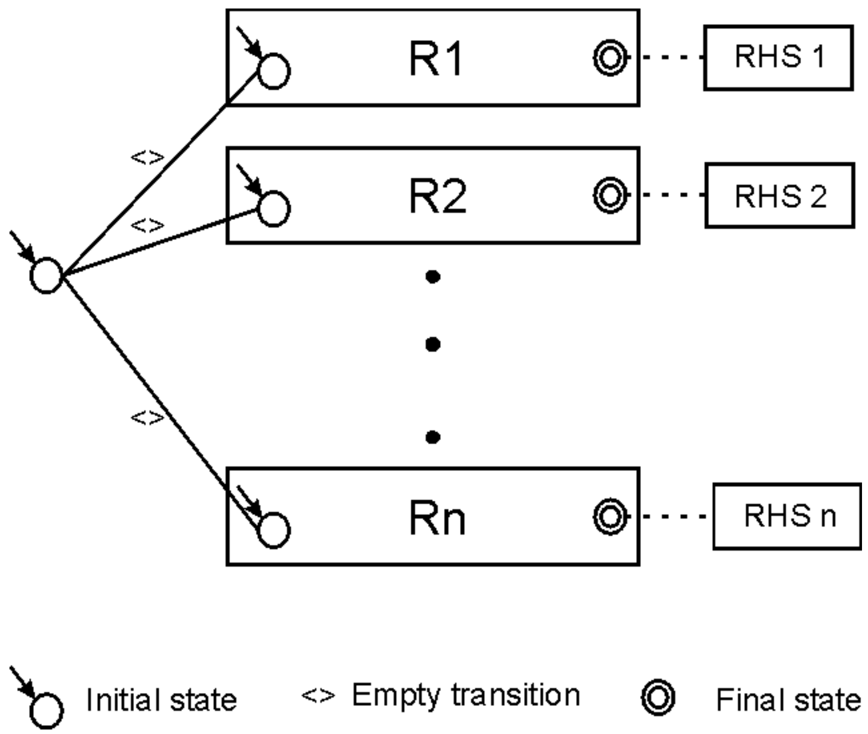


Figure B.2: A nondeterministic FSM

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

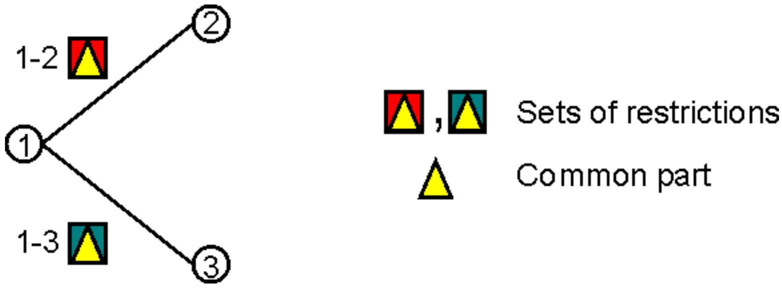


Figure B.3: Example of transitions

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).

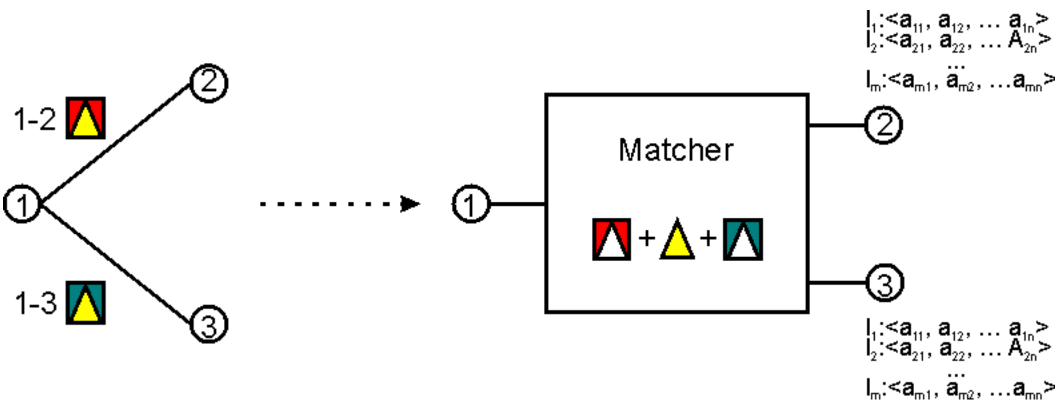


Figure B.4: A combined matching process

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)

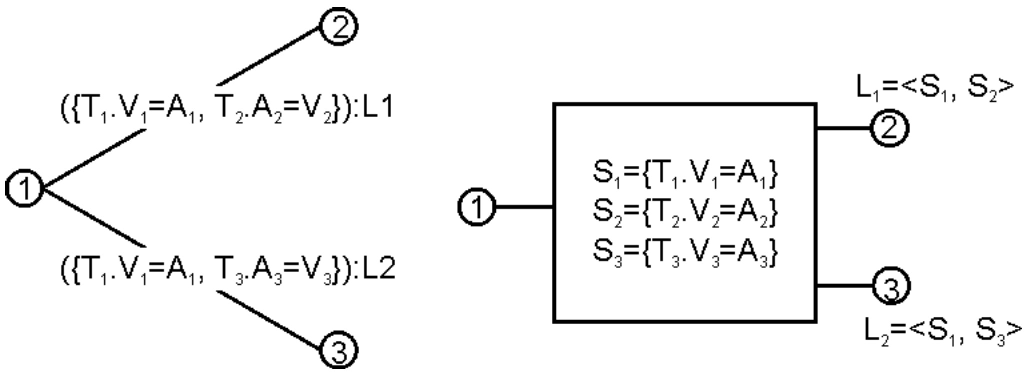


Figure B.5: Building a compound matcher

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:

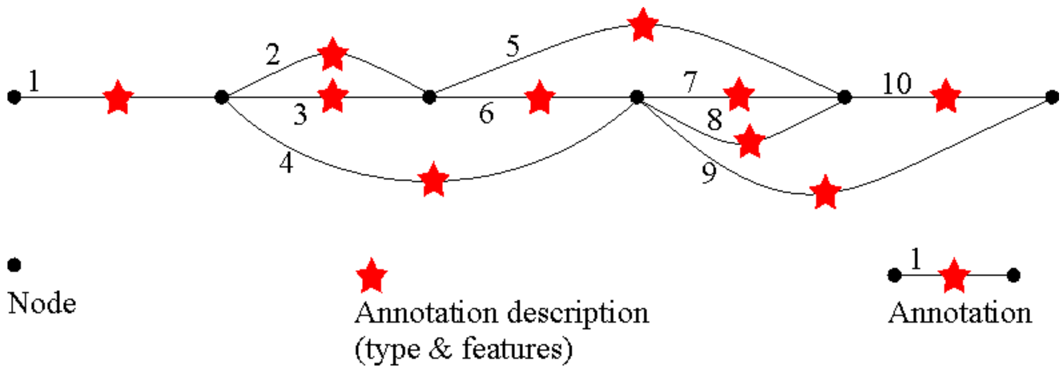


Figure B.6: An annotation graph

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 → start over from the next node // startNode = startNode.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].

By default, the Eclipse compiler is used. The two options have their pros and cons:

[2] http://www.eclipse.org/jdt

- Sun
  - Pros: supports all features of your environment, e.g. on JDK 5, the Sun compiler supports 1.5 language features such as generics.
  - Cons: requires that `gate.jar` be on the system classpath, so this compiler will not work if GATE is loaded from another class loader (e.g. in Tomcat).
- Eclipse
  - Pros: loads dependencies via the GATE class loader, meaning (a) it's faster, (b) gate.jar does not need to be on the classpath, and (c) Java code on the right hand side of JAPE rules can refer to classes that are part of a plugin.
  - Cons: Currently supports only Java 1.4 language features.

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

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.

C.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

C.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).

C.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.

C.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 "!=").

C.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.

C.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.

C.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.

C.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.

C.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.

C.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.

C.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).

C.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.

C.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.

C.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.

C.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.

C.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.

C.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.

C.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.

C.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!

C.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.

C.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.

C.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¿

C.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.

C.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 D

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 E

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>
```

References

[Appelt 99]
D. 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.

[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. http://www.gate.ac.uk/sale/jnle-sale/subs/BONTCHEVA--jnle-final.pdf.

[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.

[Chinchor 92]
N. Chinchor. Muc-4 evaluation metrics. In *Proceedings of the Fourth Message Understanding Conference*, pages 22–29, 1992.

[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*, 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*, 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 Ontologies 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`.

[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.* 05]
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, 2005.

[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).

[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. `ftp://ftp.dcs.shef.ac.uk/home/hamish/gamback-lrec2000.ps`.

[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`.

[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 *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 *Proceedings of Sheffield Machine Learning Workshop*, Lecture Notes in Computer Science. 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.

[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 *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. `http://gate.ac.uk/sale/ranlp2001/maynard-etal.pdf`.

[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*, 2002. `http://gate.ac.uk/sale/hsl/hsl.pdf`.

[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. `http://gate.ac.uk/sale/aimsa/aimsa.pdf`.

[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. `http://gate.ac.uk/sale/robust/robust.pdf`.

[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. `http://gate.ac.uk/sale/acl03/surprise.pdf`.

[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)*, 2004. `http://gate.ac.uk/sale/lrec2004/gazcollector.pdf`.

[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. `http://gate.ac.uk/sale/aimsa04/aimsa.pdf`.

[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*, 2002. `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.

[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.