Implementing a Sense Tagger in a General Architecture for Text Engineering

Hamish Cunningham, Mark Stevenson and Yorick Wilks
Department of Computer Science,
University of Sheffield,
Regent Court, 211 Portobello Street,
Sheffield S1 4DP, UK
{hamish, marks, yorick}@dcs.shef.ac.uk
http://www.dcs.shef.ac.uk/research/groups/nlp/gate/

Abstract

We describe two systems: GATE (General Architecture for Text Engineering), an architecture to aid in the production and delivery of language engineering systems which significantly reduces development time and ease of reuse in such systems. We also describe a sense tagger which we implemented within the GATE architecture, and which achieves high accuracy (92% of all words in text to a broad semantic level). We used the implementation of the sense tagger as a real-world task on which to evaluate the usefulness of the GATE architecture and identified strengths and weaknesses in the architecture.

1 Introduction

This paper is about two things: a novel hybrid sense tagger for unrestricted text (Wilks and Stevenson, 1997), and the experience of developing this system within GATE – a General Architecture for Text Engineering (Cunninham et al., 1997; Cunningham, Wilks, and Gaizauskas, 1996a).

We hope you can forgive this mild schizophrenia – we feel that both topics are relevant to the subject of new methods in NLP: the first because both the problem of arriving at methods for sense tagging and of tuning those methods to specific domains and lexical resources is an increasingly active topic in the field (Basili, Della Rocca, and Pazienza, 1997; Harley and Glennon, 1997); the second because work on NLP components shares a heap of problems with other language processing work to do with reusability, data visualisation, and software-level robustness and efficiency that, we feel, are best solved by the provision of a inclusive and general architecture and development environment for the field.

We begin with a review of the general concept behind GATE (section 2), then describe the practicalities of the system that are relevant to the sense tagging system we have developed (section 3). Next we discuss the sense tagging problem (section 4), and then our system (section 5). Finally we look at the experience of developing the tagger within the architecture (section 6), and draw out some lessons for the future (section 7).

2 GATE – the concept

GATE is an architecture and development environment for research and development workers in NLP and Language Engineering[1]. It is an architecture in the sense that it specifies a macro-level organisational pattern for the various components and data resources that make up a language processing (actually at present only *text* processing) system (Shaw and Garlan, 1996). It is also a development environment that adds a rich set of graphical tools to the architecture enabling the developer to easily integrate new processing componemts, to manage flow of control between components, to visualise the data produced by and passed between components, and evaluate the contribution of components to some externally defined and measured language processing task.

As we've noted elsewhere (Cunningham, Gaizauskas, and Wilks, 1995; Cunningham, Wilks, and Gaizauskas, 1996b), the motivating factors behind development of the architecture included the facilitation of reuse of components (which has previously been successful in NLP only

[1] The application of NLP and CL theory to the creation of practical applications software has recently become known as Language Engineering, or LE, or NLE, and has been defined in various ways in e.g. (Mitkov, 1996; Thompson, 1985; Boguraev, Garigliano, and Tait, 1995; Gazdar, 1996). Our gloss on these various definitions is that Language Engineering is 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 to a body of practise.

Hamish Cunningham, Mark Stevenson and Yorick Wilks (1998) **Implementing a Sense Tagger in a General Architecture for Text Engineering.** In D.M.W. Powers (ed.) *NeMLaP3/CoNLL98: New Methods in Language Processing and Computational Natural Language Learning,* ACL, pp 59-71.

for data resources (Cunningham, Freeman, and Black, 1994; Cunningham, 1994)), comparative and task-based evaluation, collaborative research, and software-level robustness, efficiency and portability. The design we arrived at in support of these aims is sketched in the rest of this section.

NLP systems produce information about texts (which may sometimes be the results of automatic speech recognition) and existing systems that aim to provide software infrastructure for NLP can be classified as belonging to one of three types according to the way in which they treat this information:

additive, or markup-based: information produced is added to the text in the form of markup, e.g. in SGML (Thompson and McKelvie, 1996);

referential, or annotation-based: information is stored separately with references back to the original text, e.g. in the TIPSTER architecture (Grishman, 1996);

abstraction-based: the original text is preserved in processing only as parts of an integrated data structure that represents information about the text in a uniform theoretically-motivated model, e.g. attribute-value structures in the ALEP system (Simkins, 1994).

A fourth category might be added to cater for those systems that provide communication and control infrastructure without addressing the text-specific needs of NLP (e.g. Verbmobil's ICE architecture (Amtrup, 1995)).

As noted at a previous conference in this series (Cunningham, Wilks, and Gaizauskas, 1996b), we believe that performance and other considerations favour the referential approach, but also that SGML is a key part of any general text processing strategy. The first design decision we made, then, was to base GATE on a referential core using the TIPSTER architecture, and to cater for SGML via I/O format conversion filters. This led to the development of one of three key pillars of the system: GDM, the GATE Document Manager. GDM and the TIPSTER API that it implements forms a buffer between processing modules in a GATE-based NLP system. Modules no longer talk to each other, with the coherence and coupling implications that direct unrestricted communication can imply, but to GDM via the TIPSTER API.

One of the key benefits of adopting an explicit architecture for data management is that it becomes possible to easily add a of layer graphical interface access to architecural services and data visualisation tools, and such a layer is our second pillar: GGI, the GATE graphical interface. GGI has functions for creating, viewing and editing the collections of documents which are managed by the GDM and that form the corpora which LE modules and systems in GATE use as input data. It also has facilities to display the results of module or system execution – new or changed annotations associated with the document. These annotations can be viewed either in raw form, using a generic annotation viewer, or in an annotation-specific way, if special annotation viewers are available. For example, named entity annotations which identify and classify proper names (e.g. organization names, person names, location names) are shown by colour-coded highlighting of relevant words; phrase structure annotations are shown by graphical presentation of parse trees. Note that the viewers are general for particular types of annotation, so, for example, the same procedure is used for any POS tag set, Named-Entity markup etc. Thus developers reuse GATE data visualisation code with negligible overhead.

Lastly, the third pillar of the system is the one that does all the real work of processing texts and discovering information about their content: CREOLE, a Collection of REusable Objects for Language Engineering. In a sense CREOLE isn't part of GATE at all, but is the set of resources currently integrated with the system, but we also use the term to refer to the mechanismss available for integrating modules into GATE. This process has been automated to a large degree and can be driven from the interface. The developer is required to produce some C++ or Tcl code that uses the GDM TIPSTER API to get information from the database and write back results. When the module pre-dates integration, this is called a *wrapper* as it encapsulates the module in a standard form that GATE expects. When modules are developed specifically for GATE they can embed TIPSTER calls throughout their code and dispense with the wrapper intermediary. The underlying module can be an external executable written in any language (the current CREOLE set includes Prolog, Lisp and Perl programs, for example).

CREOLE wrappers encapsulate information about the preconditions for a module to run (data that must be present in the GDM database) and post-conditions (data that will result). This information is needed by GGI, and is provided by the developer in a configuration file, which also details what sort of viewer to use for the module's results and any parameters that need passing to the module. These parameters can be changed from the interface at run-time, e.g. to tell a parser to use a different lexicon. Aside from the information needed for GGI to provide access to a module, GATE compatibility equals TIPSTER compatibility – i.e. there will be very little overhead in making any TIPSTER module run in GATE. Given an integrated module, all other interface functions happen automatically. For example, the module will appear in a graph of all modules available, with permissible links to other modules automatically displayed, having been derived from the module pre- and post-conditions. At any point the developer can create a new graph from a subset of available CREOLE modules to perform a task of specific interest.

The integration mechanisms also reduce the documentation load: users can reference the TIPSTER API to describe the interchange format of the data they produce and the GATE documentation for integration details. Of course GATE doesn't solve all the problems involved in plugging diverse LE modules together. There are three barriers to such integration:

- managing *storage and exchange* of information about texts;
- incompatibility of *representation* of information about texts;
- incompatibility of *type* of information used and produced by different modules.

GATE provides a solution to the first two of these, allowing the integrator to concentrate on the core issue of the meaningful content of the information exchanged.

3 GATE – practicalities

A main purpose of GGI is to allow execution of the modules within GATE and to provide a graphical access point to the results they produce. Section 3.1 describes the meaning of the primitives in the graph and how it is executed, section 3.2 describes the method used to autogenerate the graph, section 3.3 discusses the method of creating manageable subgraphs, and section 3.4 discusses results visualisation facilities.

3.1 Graph Syntax and Semantics

An example of a system graph is shown in figure 1[2]. A system graph is an executable graph, and is a simple data flow program. Modules are shown as nodes in the graph, with the data flow indicated by the arcs. Each incoming arc to a module indicates a dependency on results of previous processing. All modules at the source of arcs connecting to a dependent module must be run before the dependent module is executed, except where the incoming arcs are connected by lines, in which case the module requires the execution of only one of the modules at the other end of the arc (these arcs are then termed *or*-arcs). Thus, in the example graph of figure 1, the `buChart Parser` module may only be run if the results of the `Gazetteer Lookup` module and either the `Tagged Morph` module or the `Morph` module are available. They in turn have earlier dependencies. The `Tokenizer` module has no dependencies and so begins execution. There are two modules with no downstream children: `MUC-6 Results` and `MUC-6 NE Results`, so either of these must produce an end result. However, because results from modules in the middle of the graph may be of interest to a NLP researcher, any module can be chosen as the final one that will be executed.

At any point in time, the state of execution of the system, or, more accurately, the availability of data from various modules, is depicted through colour-coding of the module boxes. Figure 1 shows a system window. Light grey modules (green, in the real display) can be executed. Modules that require input from others not yet executed, and so cannot be executed yet, are shown with a white background (amber, in reality). The modules that have already been executed are shown in dark grey (red), at which point their results are available from a menu associated with each box (see below).

The system graph can either be run in batch mode or in an interactive manner. To run in batch mode, the user selects a path though the graph and clicks on the final module. The current state of the graph, and the document (or collection of documents) currently undergoing execution is shown. The system ensures that the path chosen by the user is valid by only allowing a module to be selected if all its inputs have already been selected. Selected modules are executed in a data driven manner, with modules being executed as soon as their input data is available.

The interactive mode is designed for module developers. The modules under development can be executed as with the batch mode then the module or modules to be retried (after the underlying code or resources have been changed) can be reset by a mouse click. This clears the database of the post-condition annotations and allows the modules to be rerun.

[2] These and other screen dumps below look better in colour! The description below will be a bit like the TV snooker commentator who said "For those of you watching in black and white, the pink is behind the blue".

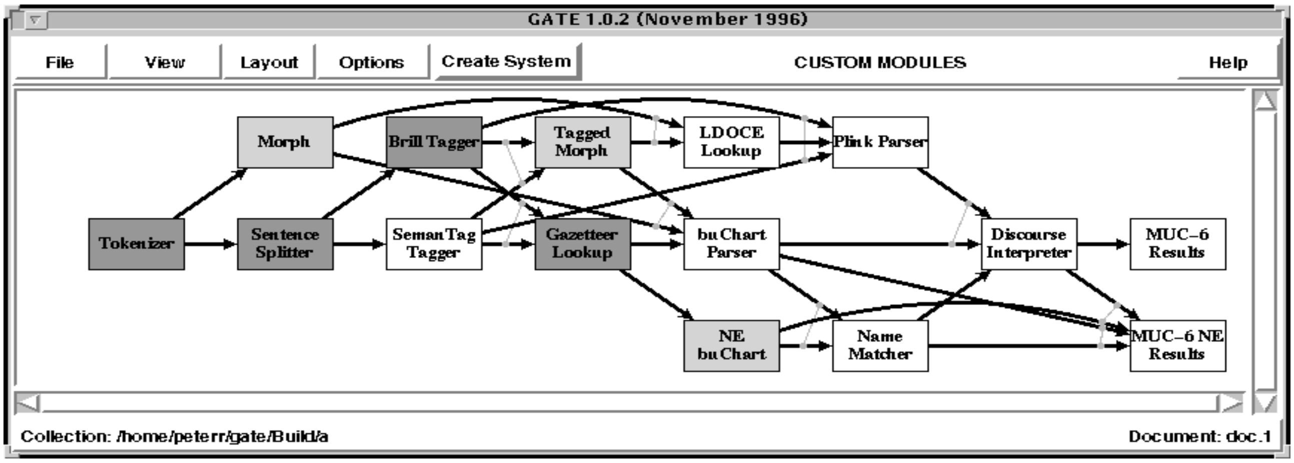

Figure 1: The GATE System Graph

The nature of the database (where each module produces a specific set of annotation types) means that it is possible to view partial results of execution without recourse to buffering intermediate data (Woodruff and Stonebreaker, 1995).

3.2 Autogeneration

The graph shown in figure 1 is in fact the *custom graph*. This is the system graph that shows all the modules in the particular GATE environment. The custom window is automatically generated from the configuration information that is associated with each module, e.g., for the `buChart` module:

```
set creole_config(buchart) {
    title {buChart Parser}
    pre_conditions {
        document_attributes {language_english}
        annotations {token sentence morph lookup}
    }
    post_conditions {
        document_attributes {language_english}
        annotations {name syntax semantics}
    }
    viewers {
        {name single_span}
        {syntax tree}
        {semantics raw}
    }
}
```

This data structure (actually a Tcl array (Ousterhout, 1994)) describes the TIPSTER objects that a module requires to run, the objects it produces, and the types of viewers to use for visualising its results. Along with code that uses the TIPSTER API to get information from the database and to store results back there, this configuration file is all that an integrator need produce to connect a module to GATE. Typically the biggest overhead here is converting pre-existing modules to preseve byte-offset information. See (Cunningham et al., 1996) for details.

The autogeneration algorithm creates data flow arcs from modules that have an annotation type in their postconditions to the other modules that have the same annotation type in their precondition. For example, `Gazetteer Lookup` has the annotation type `lookup` in its postconditions, so an arc connects it with `buChart Parser`, which has that annotation type in its preconditions. Arcs are not created between modules that operate on different languages, however in figure 1, all the modules operate on English language documents. When more than one module has the same annotation type in its postcondition then it is assumed that either module may produce the required result, and so the two arcs are *or*-arcs and are connected by a line (both `Morph` and `Tagged Morph` produce the same annotation and so have *or*-arcs into `buChart Parser`).

The most computationally expensive part of autogeneration goes into discarding redundant arcs. Redundant arcs are those that connect an upstream module to a downstream module where it can be deduced that the preconditions of modules between the two given modules cover the annotation types that the arc represents. For example, the `Tokenizer` produces annotation types required by `buChart Parser`, but there is no need for a data flow arc between these modules as modules between them also require these annotation types.

The autogeneration facility allows easy integration of new modules into GGI. Most NLP tasks can be expressed in the simple data flow techniques of this system, but it is currently not possible to integrate NLP tasks that require iteration.

Some modules have the same annotation type in both pre- and postconditions. These modify the result of previous computation and pass the data flow down stream. This kind of module, termed a *filter*, cannot be automatically positioned in the diagram, instead the user selects the position of filters from the arcs on which they may appear (arcs from modules that produce the annotation type the filter operates on). During execution filters are treated as normal modules.

3.3 Customising Graphs

The system graphs are displayed with a graph drawing tool which is is also used in tree based visualisation tools available for display of e.g. syntactic parse results. This tool allows commands to be associated with nodes, hence it can be used for data flow graphs. It has a layout algorithm based on the method used by daVinci (Fröhlich and Werner, 1995) to minimise arc crossing.

GGI suffers from the scaling problem (Burnett et al., 1987), as the size of the custom graph quickly becomes unmanageable. This can be alleviated by creating new system graphs from specified subgraphs of the custom graph. A later release will allow collapsing of graph sections.

It is possible to group these derived system graphs together so that the user may chose from a selection of tasks at the top level of GGI (not shown here for space reasons). Having chosen a task (e.g. parsing), an intermediate level display appears, presenting the user with a selection of icons, one for each of the one or more specific systems capable of performing the selected task (e.g. the buChart parser or the Plink parser). Once a particular system is selected, a final window appears displaying the appropriate system graph.

3.4 Visualising Results

NLP data is wide ranging in scope but has specific characteristics that mean the problems with visualising large amounts of data (Burnett et al., 1987) are less significant. This is because either the information can be visualised as coloured markup on the text (meaning that the text can be displayed using traditional textual techniques (Jonassen, 1982)), or the information is grouped over small segments of text, such as paragraphs or sentences.

GGI has several viewers for the display of TIPSTER annotations. The viewer for each postcondition annotation is specified by the module configuration file, an example of which is given in section 3.2. The viewers can be classified into those which display the text and overlay the annotations as colours or shades ('single span', 'multiple span', 'text-attribute'); and those that visualise a more complex relationship between annotations in an acyclic graph format ('tree'). Where no viewer is specified, a default annotation dump is displayed. The configuration file for the `buChart Parser` module in section 3.2 specifies that the 'name' annotation type is assigned the 'single span' viewer, 'syntax' the 'tree' viewer, and 'semantics' the 'raw' or annotation dump viewer. New viewers can be written where the default ones are not appropriate for new annotation types.

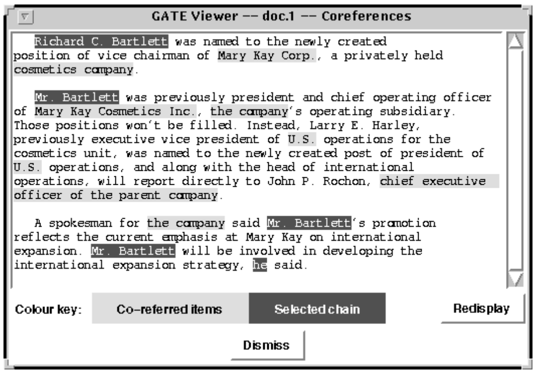

Figure 2: Multiple Span Viewer

The 'single span' and 'text-attribute' viewers are fairly simple, assigning different colours to each annotation. 'multiple span' is more complex, as it is designed to view annotation chains. An annotation chain is a list of annotations specified by annotation references. The user chooses a highlighted part of the text, and all the other highlights that are part of the same chain are displayed. Figure 2 shows this viewer displaying the results of a coreference task. Coreference identifies elements of the text that are interpreted as referring to the same real world entity. For example, a person and a pronoun might be coreferential. In figure 2 the user has chosen one of the highlights referring to 'Richard Bartlett'.

The 'tree' viewer containing 'syntax' annotations (produced by the `buChart Parser`) is shown in figure 3. The parse trees currently integrated into GATE span at most a sentence, so that the tree size is always manageable.

The viewers are activated by first clicking with the mouse on a module whose results are present (i.e. it has been executed and it's box has turned red) which reveals a menu of annotations; choosing an annotation brings up the appropriate viewer.

There is a certain amount of connectivity between these viewers, as it is possible to click on a node in the parse tree and have the area of text highlighted in a text display window, or it is possible to highlight areas of text and display the raw annotations that are contained within the highlighted span.

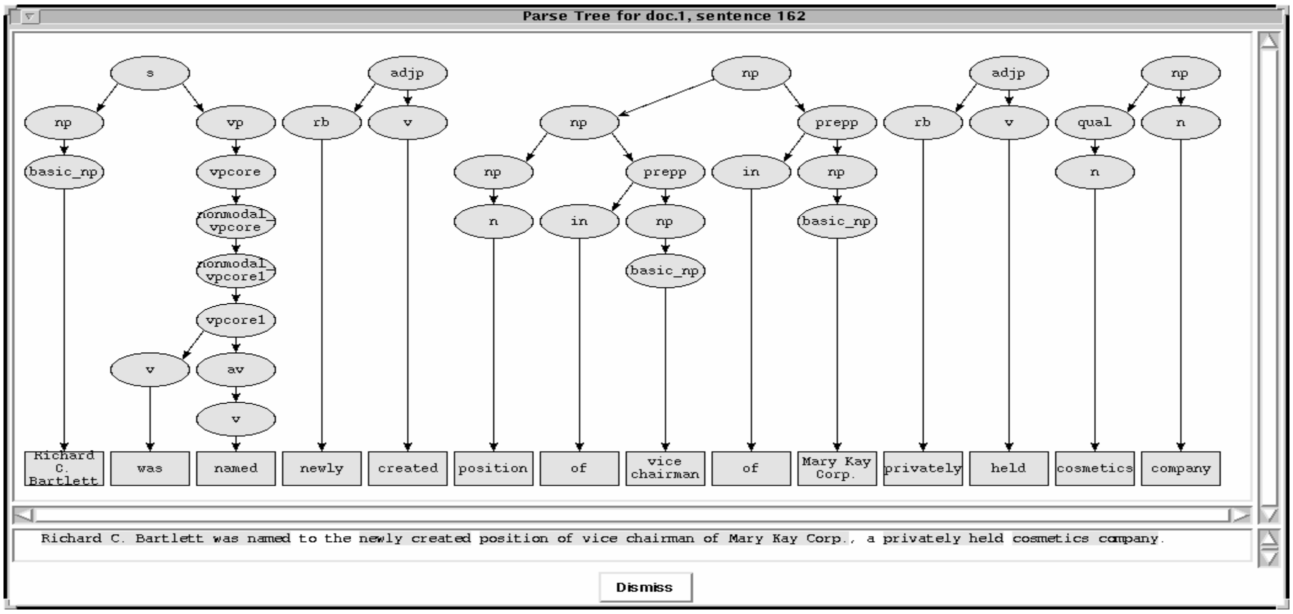

Figure 3: Tree Viewer

3.5 GATE Users

GATE version 1 was released in November 1996 and is in use for a number of projects around the World – see for example `http://www.sics.se/humle/projects/svensk/svensk.html`, who evaluated the system relative to ALEP, and `http://www.dcs.shef.ac.uk/research/groups/nlp/gate/users.html`. Figure 4 lists the sites that have licenced the system so far.

4 Word sense tagging

We have recently implemented a sense tagger within the GATE framework.

Sense tagging is the process of assigning the appropriate sense from some semantic lexicon to each word[3] in a text. This is similar to the more widely known technology of part-of-speech tagging, but the tags which are assigned in sense tagging are semantic tags from a dictionary rather than the grammatical tags assigned by a part-of-speech tagger.

Our sense tagger uses the machine readable version of *Longman Dictionary of Contemporary English* (LDOCE) (Procter, 1978) to provide the semantic tag set. LDOCE is a learners' dictionary – one designed not for native speakers of English but for those learning English as a second language and has been used extensively in machine readable dictionary research ((Ide and Veronis, 1994), (Cowie, Guthrie, and Guthrie, 1992), (Bruce and Wiebe, 1994)).

The clearest way to understand what a sense tagger does is to look at an example of the output we would like it to produce. Consider the sentence *"The interest on my bank account accrued over the years"*, our tagger should assign a single sense from LDOCE to each of the content words in the sentence. The choice of senses in the assignment should be the same as that a human would choose. An example of a desired assignment is shown in figure 5.

As can be seen from the senses assigned, each LDOCE sense has a homograph and sense number, these are used to identify different levels of semantic distinction between senses and act as identifying markers. Homograph distinctions signify broad semantic differences between senses (such as the 'edge of river' and 'financial institution' senses of *bank*) while sense distinctions signify differences between senses which are more related (such as the 'building' and 'company' senses of the word). These numbers are followed by the textual definition of the sense and, possibly, by an example sentence which is a particular use of the sense and is printed in *this type*[4].

The information provided by these tags is potentially valuable for downstream tasks in a language processing system. For example, the system could benefit from knowing that *"bank"* in this case means

[3] This is often loosened to each *content* word.

[4] LDOCE senses have additional information such as subject categories, subcategorisation information and selectional restrictions which we do not show here.

Microsoft Research Institute,	Australia
INRIA / Group ATOLL,	France
SPIRAL, Strasbourg,	France
Thompson-CSF,	France
Gesellschaft fur Multulinguale System,	Germany
University of Koblenz,	Germany
University of Stuttgart,	Germany
Esteam,	Greece
ILSP,	Greece
NCSR Demokritus,	Greece
Weizmann Institute of Science,	Israel
University of Ancona,	Italy
University of Roma Tor Vergata,	Italy
UPC Barcelona,	Spain
University of La Coruna,	Spain
University of Gothenburg,	Sweden
Institution for Linguistics,	Sweden
Linkoping University,	Sweden
Swedish Institute of Computer Science,	Sweden
Uppsala University,	Sweden
University of Fribourg,	Switzerland
LITH-DI, EPSFL,	Switzerland
University of Zurich,	Switzerland
IRSIT,	Tunisia
ITRI, Brighton,	UK
University of Durham,	UK
University of Edinburgh,	UK
University of Hertfordshire,	UK
University of Sussex,	UK
University of Lancaster,	UK
University of Luton,	UK
UMIST, University of Manchester,	UK
CRL, New Mexico,	USA
ISI, California	USA
University of Delaware Research Foundation,	USA
National Library of Medicine, Bethesda,	USA

Figure 4: GATE licence holders at 1st May 1997

a building rather than the edge of a river or a pile of something.

Sense tagging should not be confused with the more common language processing task of *word sense disambiguation* (see (Yarowsky, 1995), (Pedersen and Bruce, 1997), (Schütze, 1992) and (Ng and Lee, 1996)). There are two main differences between sense tagging and word sense disambiguation: word sense disambiguation algorithms tend to limit themselves to a small number of word types, for example Yarowskys unsupervised word sense disambiguation system disambiguated only 12 different words. Another difference is that the tags assigned in semantic tagging are taken from a standard lexicon in

- `interest`
  homograph 1 sense 6 — money paid for the use of money *"He lent me money at 6% interest."*
- `bank`
  homograph 4 sense 1 — a place where money is kept and paid out on demand, and where related activities go on
- `account`
  homograph 1 sense 5 — a sum of money kept in a bank which may be added to or taken from *"My account is empty."*
- `accrued`
  homograph 0 sense 0 — to become bigger by addition
- `years`
  homograph 0 sense 3 — a period of 365 days measured from any point *"I arrived here 2 years ago today."*

Figure 5: Desired sense assignment from LDOCE

which words may be many-way polysemous (for example some words have more than 20 different senses in LDOCE) while most word sense disambiguation algorithms assume words to have only two senses. Sense tagging can be considered as a true language engineering task since it is concerned with the robust processing of large amounts of text and aims to provide useful information for further langauge engineering processing while word sense disambiguation has more limited aims and is mainly experimental in nature.

5 Sheffield's Sense Tagger

An interesting fact about recent semantic disambiguation algorithms is that they have made use of different, orthogonal, sources of information: the information provided by each source seems independent of and has no bearing on any of the others. For example, Yarowsky used syntactic collocates (Yarowsky, 1993), Mahesh et. al. (Mahesh et al., 1997) used selectional retrcitions and Bruce and Guthrie used pragmatic codes from LDOCE (Bruce and Guthrie, 1992), in these different methods we can see that syntactic, semantic and pragmatic information has been used in word sense disambiguation. Each of these knowledge sources has enjoyed some degree of success which would suggest that they all have something to add to the problem of resolving senses in texts. A natural extension to this observation is to create a disambiguation system which makes use of several of these independent knowledge sources and combines their results in an intelligent way.

Our system is based on a set of *partial taggers*, each of which uses a different knowledge source, with their results being combined. Our system is in the tradition of McRoy (McRoy, 1992), who also made use of several knowledge sources for word sense disambiguation, although the information sources she used were not independent, making it difficult to evaluate the contribution of each component. Our system makes use of strictly independent knowledge sources and is implemented within GATE whose plug-and-play architecture makes the evaluation of individual components more straightforward.

At the moment the sense tagger consists of six stages (shown in figure 6), the first two preprocess the text which is to be disambiguated while the remaining four carry out the disambiguation.

Preprocessing

- Named-entity identification
- Dictionary look-up

Disambiguation

- Part-of-speech filtering
- Dictionary definition overlap
- Domain code overlap
- Scoring mechanism

Figure 6: Stages in the Sense Tagging process

1. The text is first processed by a named-entity identifier, which we developed as part of Sheffield's entry for MUC-6 (Wakao, Gaizauskas, and Humphries, 1996; Gaizauskas et al., 1996). This identifies certain forms of proper names in the text and classifies them as either place, person, organization or location. For details of the classification scheme see (Def, 1995). We make no use of these classifications at present, however, they are of potential use to a module carrying out disambiguation using selectional restrictions.

   The tagger does not attempt to disambiguate any words which are identified as part of a named-entity.

2. The remaining text is stemmed, leaving only morphological roots, and split into sentences. Then words belonging to a list of stop words[5] are removed. The words which have not been identified as part of a named entity or removed because it is a stop word are considered by the system to be *ambiguous words* and those are the words which are disambiguated.

   For each of the ambiguous words, its set of possible senses is extracted from LDOCE and stored. Each sense in LDOCE contains a short textual definition (such as those shown in figure 5) which, when extracted from the dictionary, is processed to remove stop words and stem the remaining words.

3. The text is tagged using the Brill tagger (Brill, 1992) and a translation is carried out using a manually defined mapping from the syntactic tags assigned by Brill (Penn Tree Bank tags (Marcus, Santorini, and Marcinkiewicz, 1993)) onto the simpler part-of-speech categories associated with LDOCE senses[6]. We then remove from consideration any of the senses whose part-of-speech is not consistent with the one assigned by the tagger, if none of the senses are consistent with the part-of-speech we assume the tagger has made an error and leave the set of senses for that word unaltered.

4. Our next module is based on a proposal by Lesk (Lesk, 1986) that words in a sentence could be disambiguated by choosing the the sense which produced the maximum overlap of the content words in the textual definitions of the word's senses. In practise this led to massive computations with as many as 10^{10} possible combinations of senses for a single sentence.

   Cowie et. al. (Cowie, Guthrie, and Guthrie, 1992) used simulated annealing (Kirkpatrick, Gelatt, and Vecci, 1983), a numerical optimisation algorithm, to make this process tractable.

[5] In our system a stop word is defined to be any word which is not a noun, verb, adjective or adverb. Prepositions are included in the list of stop words and are not disambiguated.

[6] The Brill tagger uses the tag set from the Penn Tree Bank which contains 48 tags (Marcus, Santorini, and Marcinkiewicz, 1993), LDOCE uses a set of 17, more general, tags.

The simulated annealing algorithm proceeds by disambiguating a sentence at a time. A random configuration of senses is chosen such that one sense is assigned to each ambiguous word in the sentence. A score is given to this configuration based on the number of content words which are shared between the textual definition in the senses. Other, random, configurations are then generated and the simulated annealing algorithm is used to optimise over them. When this process is complete the algorithm returns a configuration which assigns the optimal configuration of senses based on the overlap of words in the definition text.

This process identifies a single condidate LDOCE sense for each ambiguous word.

5. The text is then run through a module which optimises the overlap of domain codes for the senses of nouns in each paragraph of the text. The optimisation algorithm used is similar to simulated annealing (see section 4), although it has been modified in two ways. Firstly, we maximise the overlap of the pragmatic codes associated with the word senses rather than the content words in their definitions. Secondly, we optimise over entire paragraphs at a time rather than just sentences, this is done because there is good evidence (Gale, Church, and Yarowsky, 1992) that a wide context, of around 100 words, is optimal when disambiguating using domain codes. This process, like the previous module, identifies a single candidate sense for each ambiguous word.

6. The final stage is to combine the results of the preceding processes. This is done using a very simple mechanism which we plan to replace with an optimisation algorithm. We assign a score to each of the senses of the ambiguous words. These scores are initialised to `0` and `+1` is added to a sense's score for each of the simulated annealing or pragmatic code modules which select that sense. The sense with the highest score is chosen as the tag for each ambiguous word. If there is a tie (two senses with the same score, which will happen if the two partial taggers disagree) it is broken by choosing the first sense, as listed in the dictionary. This is a sensible tie-breaker since the senses are roughly ordered by frequency of occurrence in text[7]. After this process is completed every ambiguous word has exactly one sense from LDOCE associated with it, this sense is the tag which our system has assigned to that word.

We have conducted some preliminary testing of our tagger: our tests were run on 14 hand-disambiguated (by one of the authors) sentences from the Wall Street Journal, amounting to a 250 word corpus. We found that, of the tokens with more than 1 homograph in LDOCE, 92% were assigned the correct homograph and 75% the correct sense using our tagger. These figures should be compared to the 72% correct homograph assignment and 47% correct sense assignment reported by Cowie et. al. (Cowie, Guthrie, and Guthrie, 1992) using simmulated annealing alone on the same test set.

6 Developing the tagger with GATE

The sense tagger was implemented as a set of 11 CREOLE modules, 6 of which had been implemented as part of VIE and the remaining 5 were developed specifically for the sense tagger. These were implemented in a variety of programming languages: C[++], Perl and Prolog. These five modules are varied in their implementation methods. Two are written entirely in C++ and are linked with the GATE executable at runtime using GATE's dynamic loading facility (see (Cunningham et al., 1996)). Three are made up of a variety of Perl scripts, Prolog saved states or C executables, which are run as external processes via GATE's Tcl (Ousterhout, 1994) API. This is typical of systems we have seen built using GATE, and illustrates its flexibility with respect to implementation options.

The GATE graphical representation of the sense tagger is shown in figure 7.

A special viewer was implemented within GATE to display the results of the sense tagging process. After the final module in the tagger has been run it is possible to call a viewer which displays the text which has been processed with the ambiguous words highlighted (see figure 8). Clicking on one of these highlighted words causes another window to appear which contains the sense which has been assigned to that word by the tagger (see figure 9). Using this viewer we can quickly see that the tagger has assigned the 'chosen for job' sense of "appointment" in *"Kando, whose appointment takes effect from today ..."* which is the correct sense in this context.

[7]We are using the 1st Edition of LDOCE in which the publishers make no claim that the senses are ordered by frequency of occurrence in text (although they do in later editions). However, (Guo, 1989) has found evidence that there is a correspondence between the order in which sense are listed and the frequency of occurrence.

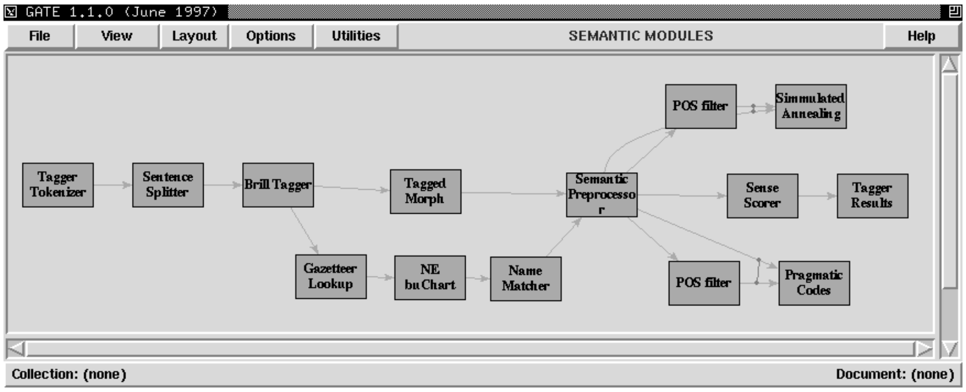

Figure 7: The sense tagger in GATE

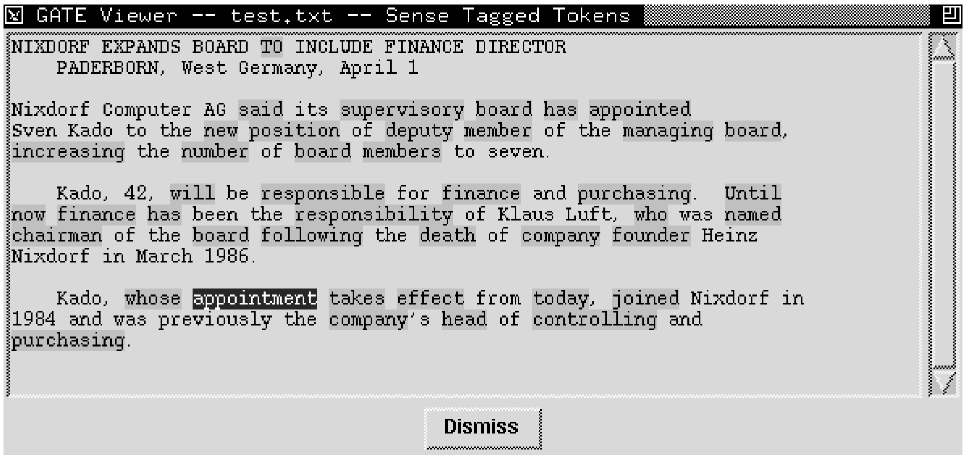

Figure 8: Words disambiguated by the tagger

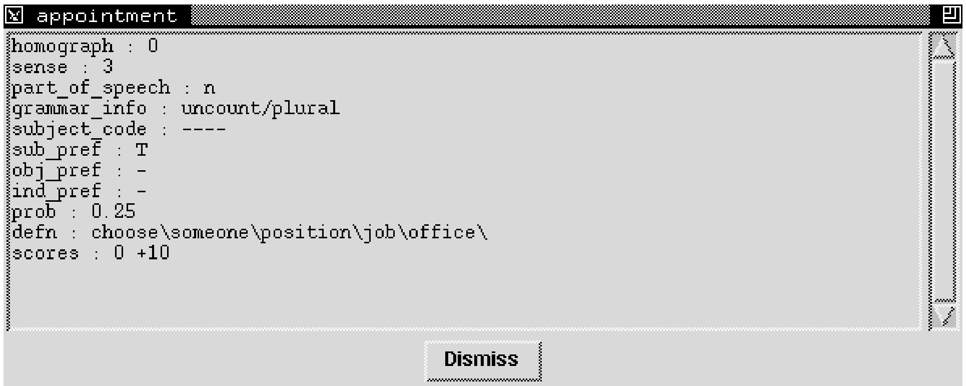

Figure 9: View of sense assigned to *"appointment"*

7 Lessons Learned

Our experience of implementing a novel system within GATE has given us insights into the usefulness of the architecture which we could not have gained otherwise, the overall result of the experience was very encouraging.

In the TIPSTER annotation scheme, GATE provides a well-defined standard for data-transfer between modules. This standard approach allows for the rapid re-use of existing modules and reduced the need to provide data-transfer routes between modules. Almost the entire preprocessing of the text was carried out using modules which had already been implemented within GATE: the tokeniser, sentence splitter, Brill part-of-speech tagger and the modules which made up the Named Entity identifier. This meant that we could quickly implementat the modules which carried out the disambiguation, and those were the modules in which we were most interested. The implementation was further speeded up by the use of results viewers which allowed us to examine the annotations in the TIPSTER DataBase after a module had been run, allowing us to discover bugs far more quickly than would have been possible in a system which is not as explicitly modular as GATE. One aspect of sense tagging in which we are interested is the effect of including and excluding different modules, and this could be easily carried out using GGI.

One particular limitation of the current GATE implementation became apparent during this work, viz. the necessity of cascading module reset in the presence of non-monotonic database updates. For example, the POS filter modules remove some of the sense definitions associated with words by the lexical preprocessing stages. When reseting these modules it is therefore necessary to reset the preprocessor stage in order that the database is returned to a consistent state (this is done automatically by GATE, which identifies cases where modules alter previously existing annotations by examination of the pre-/post-conditions of the module supplied by the developer as configuration information prior to loading). This leads to redundant processing, and in the case of slow modules (like our LDOCE lookup module) this can be an appreciable brake on the development cycle. The planned solution is to change the implmentation of the reset function. Currently this simply deletes the database objects created by a module. Given a database implementation that supports transactions we can use timestamp and rollback for a more intelligent reset, and avoid the redundant processing caused by reset cascading.

An additional, lesser problem, is the complexity of the generation algorithms for the task graphs, and the diffculty of managing these graphs as the number of modules in the system grows. The graphs currently make two main contributions to the system: they give a graphical representation of control flow, and allow the user to manipulate execution of modules; they give a graphical entry point to results visualisation. These benefits will have to be balanced against their disadvantages in future versions of the system. Another problem may arise when the architecture includes facilities for distributed processing (Zajac et al., 1997; Zajac, 1997), as it is not obvious how the linear model currently embodied in the graphs could be extended to support non-linear control strucures.

8 Conclusion

The previous section indicates that GATE version 1 goes a long way to meeting it's design goals (noted in section 2). The reuse of components we have experienced in the sense tagging project and a number of other local and collaborative projects is in itself justification of the development effort spent on the system, and, hopefully, these savings will be multiplied accross other users of the system. Future versions will address the problems we uncovered above.

Distribution

GATE and a MUC-6 (Grishman and Sundheim, 1996) style Information Extraction (Gaizauskas et al., 1996; Humphreys et al., 1996) system that comes with it is free for academic research – see `http://www.dcs.shef.ac.uk/research/groups/nlp/gate/` for details.

Acknowledgements

This work has been supported by the UK's EPSRC and the European Commission Language Engineering programme under grants GR/K25267 (GATE), LE1-2238 (AVENTINUS) and LE1-2110 (ECRAN).

References

Amtrup, J.W. 1995. ICE – INTARC Communication Environment User Guide and Reference Manual Version 1.4. Technical report, University of Hamburg.

Basili, R., M. Della Rocca, and M.T. Pazienza. 1997. Towards a bootstrapping framework for corpus semantic tagging. In *Proceedings of the SIGLEX Workshop "Tagging Text with Lexical Semantics: What, why and how?"*, Washington, D.C., April. ANLP.

Boguraev, B., R. Garigliano, and J. Tait. 1995. Editorial. *Natural Language Engineering.*, 1, Part 1.

Brill, E. 1992. A simple rule-based part of speech tagger. In *Proceedings of the DARPA Speech and Natural Language Workshop*. Harriman, NY.

Bruce, R. and L. Guthrie. 1992. Genus disambiguation: A study in weighted preference.

In *Proceesings of COLING-92*, pages 1187–1191, Nantes, France.

Bruce, R. and J. Wiebe. 1994. Word-sense disambiguation using decomposable models. In *Proceedings of the 32nd Annual Meeting of the Association for Computational Linguistics*, pages 139–145, Las Cruces, New Mexico.

Burnett, M., M.J. Baker, C. Bohus, P. Carlson, S. Yang, and van Zee P. 1987. Scaling Up Visual Languages. *IEEE Computer*, 28(3):45–54.

Cowie, J., L. Guthrie, and J. Guthrie. 1992. Lexical disambiguation using simulated annealing. In *Proceedings of the 14th International Conference on Computational Linguistics (COLING-92)*, pages 359–365, Nantes, France.

Cunningham, H. 1994. Support Software for Language Engineering Research. Technical Report 94/05, Centre for Computational Linguistics, UMIST, Manchester.

Cunningham, H., M. Freeman, and W.J. Black. 1994. Software Reuse, Object-Oriented Frameworks and Natural Language Processing. In *Proceedings of the conference on New Methods in Natural Language Processing (NeMLaP-1)*, Manchester.

Cunningham, H., R.G. Gaizauskas, and Y. Wilks. 1995. 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. Also available as `http://xxx.lanl.gov/ps/cmp-lg/9601009`.

Cunningham, H., K. Humphreys, R. Gaizauskas, and M. Stower, 1996. *CREOLE Developer's Manual.* Department of Computer Science, University of Sheffield. Available at `http://www.dcs.shef.ac.uk/research/groups/nlp/gate`.

Cunningham, H., Y. Wilks, and R. Gaizauskas. 1996a. GATE – a General Architecture for Text Engineering. In *Proceedings of the 16th Conference on Computational Linguistics (COLING-96)*, Copenhagen, August.

Cunningham, H., Y. Wilks, and R.J. Gaizauskas. 1996b. 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. Also available as `http://xxx.lanl.gov/ps/cmp-lg/9607025`.

Cunninham, H., K. Humphreys, R. Gaizauskas, and Y. Wilks. 1997. Software Infrastructure for Natural Language Processing. In *Proceedings of the Fifth Conference on Applied Natural Language Processing (ANLP-97)*, March. Available as `http://xxx.lanl.gov/ps/9702005`.

Defense Advanced Research Projects Agency. 1995. *Proceedings of the Sixth Message Understanding Conference (MUC-6)*. Morgan Kaufmann.

Fröhlich, M. and M. Werner. 1995. Demonstration of the Graph Visualization System daVinci. In *Proceedings of DIMACS Workshop on Graph Drawing '94, LNCS 894*. Springer-Verlag.

Gaizauskas, R.G., H. Cunningham, Y. Wilks, P. Rodgers, and K. Humphreys. 1996. 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.

Gale, W., K. Church, and D. Yarowsky. 1992. One sense per discourse. In *Proceedings of the DARPA Speech and Natural Language Workshop*, pages 233–237, Harriman, New York, February.

Gazdar, G. 1996. Paradigm merger in natural language processing. In R. Milner and I. Wand, editors, *Computing Tomorrow: Future Research Directions in Computer Science*. Cambridge University Press, pages 88–109.

Grishman, R. 1996. TIPSTER Architecture Design Document Version 2.2. Technical report, DARPA. Available at `http://www.tipster.org/`.

Grishman, R. and B. Sundheim. 1996. Message understanding conference - 6: A brief history. In *Proceedings of the 16th International Conference on Computational Linguistics*, Copenhagen, June.

Guo, C-M. 1989. Constructing a Machine Tractable Dictionary from Longman Dictionary of Contemporary English. Technical Report MCCS-89-156, Computing Research Laboratory, New Mexico State University.

Harley, A. and D. Glennon. 1997. Sense tagging in action: Combining different tests with additive weights. In *Proceedings of the SIGLEX Workshop "Tagging Text with Lexical Semantics"*. Association for Computational Linguistics, Washington, D.C., pages 74–78.

Humphreys, K., R. Gaizauskas, H. Cunningham, and S. Azzam, 1996. *CREOLE Module Specifications.* Department of Computer Science, University of Sheffield.

Ide, N. and J. Veronis. 1994. Have we wasted our time? In *Proceedings of the International Workshop on the Future of the Dictionary*, Grenoble.

Jonassen, D.H., editor. 1982. *The Technology of Text.* Educational Technology Publications.

Kirkpatrick, S., C. Gelatt, and M. Vecci. 1983. Optimisation by simulated annealing. *Science*, 220(4598):671–680.

Lesk, M. 1986. Automatic sense disambiguation using machine readable dictionaries: how to tell a pine cone from an ice cream cone. In *Proceedings of ACM SIGDOC Conference*, pages 24–26, Toronto, Ontario.

Mahesh, K., S. Nirenburg, S. Beale, E. Viegas, V. Raskin, and B. Onyshkeyvych. 1997. Word sense disambiguation: Why have statistics when we have these numbers. In *Proceedings of the 7th International Conference on Theoretical and Methodological Issues in Machine Translation*, pages 151–159, July.

Marcus, M., B. Santorini, and M. A. Marcinkiewicz. 1993. Building a large annotated corpus of English: The Penn Tree Bank. *Computational Linguistics*, 19(2):313–330.

McRoy, S. 1992. Using multiple knowledge sources for word sense disambiguation. *Computational Linguistics*, 18(1):1–30.

Mitkov, R. 1996. Language Engineering: towards a clearer picture. In *Proceedings of ICML.*

Ng, H. T. and H. B. Lee. 1996. Integrating multiple knowldge sources to disambiguate word sense: An exemplar-based approach. In *Proceedings of ACL96.*

Ousterhout, J.K. 1994. *Tcl and the Tk Toolkit.* Addison-Wesley.

Pedersen, T. and R. Bruce. 1997. Distinguishing word senses in untagged text. In *Proceedings of the Second Conference on Empirical Methods in Natural Language Processing*, Providence, RI, August.

Procter, P. 1978. *Longman Dictionary of Contemporary English.* Longman Group, Essex, England.

Schütze, H. 1992. Dimensions of meaning. In *Proceedings of Supercomputing '92*, pages 787–796, Minneapolis, MN.

Shaw, M. and D. Garlan. 1996. *Software Architecture.* Prentice Hall.

Simkins, N. K. 1994. An Open Architecture for Language Engineering. In *First Language Engineering Convention, Paris.*

Thompson, H. 1985. Natural language processing: a critical analysis of the structure of the field, with some implications for parsing. In K. Sparck Jones and Y. Wilks, editors, *Automatic Natural Language Parsing.* Ellis Horwood.

Thompson, H.S. and D. McKelvie. 1996. A Software Architecture for Simple, Efficient SGML Applications. In *Proceedings of SGML Europe '96, Munich.*

Wakao, T., R. Gaizauskas, and K. Humphries. 1996. Evaluation of an algorithm for the recognition and calssification of proper names. In *Proceedings of the 16th International Conference on Computational Linguistics (COLING96)*, pages 418–423, Copenhagen, Denmark.

Wilks, Y. and M. Stevenson. 1997. Sense tagging: Semantic tagging with a lexicon. In *Proceedings of the SIGLEX Workshop "Tagging Text with Lexical Semantics: What, why and how?"*. ANLP. Available as `http://xxx.lanl.gov/ps/cmp-lg/9705016`.

Woodruff, A. and M. Stonebreaker. 1995. Buffering of Intermediate Results in Dataflow Diagrams. In *Proceedings VL'95 11th International IEEE Symposium on Visual Languages, Darmstadt.* IEEE Computer Society Press.

Yarowsky, D. 1993. One sense per collocation. In *Proceedings ARPA Human Language Technology Workshop*, pages 266–271.

Yarowsky, D. 1995. Unsupervised word-sense disambiguation rivaling supervised methods. In *Proceedings of ACL95.*

Zajac, R. 1997. An Open Distributed Architecture for Reuse and Integration of Heterogenous NLP Components. In *Proceedings of the 5th conference on Applied Natural Language Processing (ANLP-97).*

Zajac, R., V. Mahesh, H. Pfeiffer, and M. Casper. 1997. The Corelli Document Processing architecture. Technical report, Computing Research Lab, New Mexico State University.