Software Architecture for Language Engineering

Hamish Cunningham

Doctor of Philosophy

Department of Computer Science

University of Sheffield

June 2000

Abstract

This thesis defines the boundaries of Software Architecture for Language Engineering (SALE), an area formed by the intersection of human language computation and software engineering. SALE covers all areas of the provision of infrastructural systems to support research and development of language processing software. In order to demonstrate the theory developed in relation to SALE, we present the design, implementation and evaluation of GATE, a General Architecture for Text Engineering, which illustrates in practice many of the theoretical points made.

The thesis represents the first discussion of software infrastructure for language computation that covers a large portion of the field. GATE is one of the most widely used such systems in existence, and very few other systems of similar design are freely available or in common use outside of their originating institutions.

Acknowledgements

If I used this opportunity to acknowledge all the intellectual and cultural debts that are in one way or another connected with this work I would be in peril of bankruptcy, so, briefly and incompletely, thanks to:

- Yorick Wilks, my supervisor and oracle;
- my other local colleagues, Robert Gaizauskas, Kevin Humphreys, Wim Peters, Mark Stevenson, Valentin Tablan, and the members of the Sheffield NLP group;
- my distributed colleagues who downloaded GATE, sometimes succeeded in installing it, and often sent us their thoughts;
- Gillian Callaghan, Gill Wells and Chris Stoddart for preventing the fabric of space and time from collapsing around me;
- my examiners, Harold Somers and Mark Hepple, for their constructive comments;
- Cunninghams Simon, Chris, Kath, Jill and Sandy, who have the sense not to give a monkey's about computers;
- most of all, Kalina Bontcheva.

Yorick, Kalina and Sandy all read and improved on previous drafts.

The work was supported by the UK Engineering and Physical Sciences Research Council (grant numbers GR/K25267 and GR/M31699) and several smaller grants.

Brief Contents

Contents

List of Figures

Part I

Motivation and Context

Chapter 1

Introduction

This thesis explores the intersection of two fields:

- computer processing of human language, or *Language Engineering*;
- computer infrastructure for software development, or *Software Architecture*.

The thesis reports on the analysis, design, implementation and evaluation of a Software Architecture for Language Engineering (SALE).

The system we have built is intended to be a general, extensible support tool for researchers and developers building software that processes human language. There is an analogy with infrastructural tools available in older, more mature, fields. The professional mathematician can take advantage of systems such as MatLab, SPSS or Mathematica. These systems are general, flexible, non-prescriptive software environments in which well-known results can be reused and speculative experiments pursued without the need to implement low-level computational tasks. Similarly, software developers can use integrated programming environments such as IBM Visual Age or Borland JBuilder. These tools offer a high degree of automation for debugging and user interface design without significantly restricting the type of application that can be built. In contrast, research and development in processing human language is usually conducted with the help of non-general tools which are frequently constructed specifically for the task at hand, and which constrain the types of theories that can be investigated.

Our system is called GATE, a General Architecture for Text Engineering, which was designed and implemented by the author and others[1]. GATE is an instance of a Software Architecture for Language Engineering.

[1]My colleagues in the University of Sheffield NLP group discussed and contributed to the design of GATE on many occasions. In particular Robert Gaizauskas and Kevin Humphreys participated in the

The thesis is organised around a progression from the general to the specific, capturing structural abstractions along the way. These abstractions eventually serve as the bones of the design of GATE. Thus in part I we cover generalities about language processing and software architecture; in part II we look more specifically at the entities to be found in language processing software, and review previous work on the architectural issues that arise in the domain. Part III presents GATE; part IV evaluates the work; part V supplies a technical appendix, bibliography and so on.

The contribution of the work is twofold:

1. to define the boundaries of SALE as a field, to elicit a requirement set for such systems and to discover the criteria by which a successful infrastructure may be measured;
2. to detail the design and implementation of GATE, and show the extent to which it meets the requirements identified in 1.

Element 1 is novel because although others have discussed architecture in particular contexts, this is the first such discussion to range over a large proportion of the Language Engineering (LE) field. Element 2 is novel because no other systems with comparable design have been made freely available and supported within the LE community. (There are other infrastructural systems that share some of GATE's characteristics, but none that combine all its elements, and fewer still that have a user community outside of their originating institutions.) GATE illustrates in practice the theoretical points that we make about SALE.

The rest of this chapter previews the material in the rest of the thesis.

1.1 Part I: Motivation and Context

GATE is a tool for:

- expert designers and programmers producing applications software that processes human language;

design of version 1, and are responsible for the structure of important elements of the user interface, and the annotation comparison and markup tools. Kevin also coded parts of the version 1 interface, and several others made contributions to the interface implementation. Kalina Bontcheva contributed significantly to the design of version 2; Wim Peters contributed to the design of language resource support in version 2; the implementation of version 2 is currently shared between myself, Kalina Bontcheva, Valentin Tablan and Christian Ursu; Mark Stevenson worked on the review of GATE for sense tagging in chapter 8. Although a large part of the work is my own, I could not have completed it without the intellectual environment that supported me, and the practical stimulus of the Information Extraction research led by Robert Gaizauskas and Yorick Wilks.

- non-expert programmers writing experimental software for research purposes in the same area;
- language researchers performing experiments with software written by others;
- teachers of language processing science and technology;
- systems administrators supporting language researchers.

These various client groups are all involved, in one way or another, with LE and its related disciplines. Chapter 2 discusses the nature of this field, and the relation between LE, Natural Language Processing (NLP) and Computational Linguistics (CL). The definitions given are: CL is a part of the science of language that uses computers as investigative tools; NLP is part of the science of computation whose subject-matter is computer systems that process human language; LE is the discipline of engineering software systems that perform tasks involving processing human language. There is crossover and blurring of these definitions in practice, but they capture some important generalisations and differences.

Chapter 3 discusses Software Architecture, and other aspects of software engineering of relevance to LE. The chapter characterises software infrastructure in a general sense, and argues for the explicit design of software, discussing how this may be realistically approached in the context of LE. It goes on to present architecture as a route to promoting software reuse, factoring out repetition and reducing development overheads in a number of ways, and notes the close relationship between architectures and development environments.

1.2 Part II: Domain Analysis

The nature of LE and the potential benefits of Software Architecture motivate the production of an architecture for language processing. Part II analyses the domain in which such an architecture must operate. Chapter 4 taxonomises LE software along a number of dimensions. It should be noted here that non-trivial taxonomies (and ontologies) are invariably inexact and may be expressed in many different ways. The classifications made in part II (and summarised just below) are important for expository and systems analysis purposes, but should not be interpreted as categorical statements regarding the nature of the field.

We make distinctions between *applications*, *technologies*, *methods* and *components*. Applications are complete software systems that perform some intrinsically useful task, and may be sold as products. For example, a translator's workbench is an application[2]. Technologies are those areas of research that contain relatively well-defined bodies of theory, methods

[2]In fact automated translation is probably the original LE application [Hutchins & Somers 92].

and tasks, and whose results are often the subject of active transfer from research labs into applications development. For example, Machine Translation (MT) is a technology in our terms (and underlies applications such as translators' workbenches). Methods are the means by which researchers perform their experiments, and developers implement technologies and applications. Examples of methods include symbolic computation over rule sets and statistical modelling. Components are the elements of which LE software systems are composed. For example, parsers and lexicons are typical components of MT systems (and others).

LE is concerned with how best to build *applications* that involve NLP technology. Such applications include:

- automated translation systems, and translators' aids;
- search engines for text corpora or the Web;
- spelling, grammar and style checkers;
- natural language interfaces to databases;
- speech transcription programs[3];
- automated text-to-database systems.

Section 4.1 discusses LE applications. Simply put, a SALE should support all the activities involved in the development, deployment and maintenance of these applications. In particular, anything that gets implemented regularly in these systems, and anything done by the support tools that their developers use, is a candidate for desiderata for the architecture. To make these statements more concrete, the discussion in section 4.1 begins with an imaginary application that is representative of the field, called the *Information Harvester*. The Harvester consists of a Web crawler that extracts information from diverse sources in order to populate a database of elementary facts. In addition, it allows the presentation of these facts as simple summaries in multiple languages, which may be written or spoken. The Harvester seeks to process all the Web pages it can find that contain any human language; when it comes across audio or video documents it attempts to locate and transcribe any spoken sections, and then to extract information from the transcription. The Harvester can usefully stand as a prototypical LE system because it uses technologies and methods that cover a large subset of LE. A SALE that fully supported the Harvester application, therefore, would be of widespread utility as LE software infrastructure.[4]

[3]Throughout this thesis 'program' refers to the computing sense of the word, while 'programme' refers to the planning or scheduling sense.

[4]The Harvester example serves expository purposes only; GATE is evaluated in the context of a number of existing applications and prototypes in chapter 8.

The language processing parts of LE applications are based on *technologies* from various sub-fields of NLP, for example:

- Automatic Speech Recognition (ASR);
- Information Extraction (IE);
- Natural Language Generation (NLG);
- Machine Translation (MT).

Technologies compose: MT, for example, may use NLG (amongst other things).

Section 4.2 discusses LE technologies, with detailed presentations of a representative set: ASR, IE and NLG. These are the LE technologies that underlie the Harvester application, and are intended to be an indicative set sufficient to guide work on SALE (the aim here is not to be comprehensive, which would only be appropriate for a textbook presentation of the subject). ASR translates speech into text; IE translates text into database records; NLG translates database records or semantic structures into text. These technologies and the methods they employ have been used to motivate and constrain the development of GATE. Rule-based IE systems were the first to be developed within the system, and this led to a certain bias towards symbolic processing of text, and towards language analysis as opposed to generation or translation. In order to address these biases, more recent work has also used the examples of ASR and NLG.

LE technologies are developed using general *methods* such as:

1. statistical modelling and search;
2. symbolic computation over rule sets and other structures;
3. quantitative and qualitative evaluation;
4. finite state transduction.

Various parts of section 4.2 give examples of 1–3; appendix A describes GATE's implementation of 4.

Applications, technologies and methods are associated with typical sets of software *components*. A translation application, for example, will probably include a parser component, which uses grammar and lexicon components. Section 4.3 covers LE components, beginning with a discussion of the way in which data and processing are often somewhat separate areas of study in LE work. The section adopts the following terminology:

Language Resource (LR): refers to data-only components such as lexicons, corpora, thesauri and ontologies.

Processing Resource (PR): refers to components whose character is principally task-oriented or algorithmic, such as lemmatisers, generators, translators, parsers and speech recognisers.

Having looked in some depth at various areas of LE we are then in a position to describe how the software entities that typify the field break down, giving a taxonomy of LE software components. The section uses UML (Unified Modelling Language) class diagrams (see chapter 3) to illustrate how various sorts of Language Resources and Processing Resources relate to each other.

Chapter 5 reviews the work of others who have developed infrastructural systems for LE, covering a wide range of different systems and categories of systems. In order to better organise the discussion we begin by extrapolating a set of architectural issues comprising the union of those addressed by the various researchers cited. This has the advantage of being easier to transform into software design (in part III) and the disadvantage that multipurpose infrastructures appear in several places. The chapter then discusses infrastructures aimed at Processing Resources, including the issues of component integration and execution. We then turn to Language Resource support, including the issues of access and the representation of information about text and speech. Finally we deal with issues relating to LE methods, applications and development environments.

The material in part II serves several purposes in the rest of the thesis. It is used to direct a requirements analysis for SALE, which in turn informs the design of GATE, and is later used to review the success of the system. Finally, our plans for the future development are based on our evaluation of GATE relative to the requirement set and therefore relative to the domain analysis of part II.

1.3 Part III: GATE

Part III moves on to the design and implementation of GATE, a General Architecture for Text Engineering. Chapter 6 presents a requirements analysis, expressed as use cases for GATE, exemplified by the Harvester application and informed by the review of previous work in chapter 5. Use cases are a requirements capture and project planning tool in which the external properties (or functional requirements) of a system are codified in terms of typical interactions with its users. The chapter closes with a summary of the desiderata for GATE encapsulated in the use cases.

The design and implementation of GATE is dealt with in chapter 7. GATE version 1 comprises three principal elements: GDM, the GATE Document Manager; CREOLE, a Collection of REusable Objects for Language Engineering (a set of LE modules integrated with the system); and GGI, the GATE Graphical Interface, a development tool for LE R&D, providing integrated access to the services of the other components and adding visualisation and debugging tools. The original design principles behind the system may be summarised as: to avoid tying developers into any particular theory; to support modular construction of LE systems and make it easy to swap components in and out; to provide graphical tools for the visualisation of data and the execution of processing components; to manage storage of data and loading of processors into programs.

1.4 Part IV: Results and Prospects

Part IV analyses the results and presents future prospects. The success of the work, relative to the requirements of chapter 6 and other measures, is assessed in chapter 8. The measures used are:

- take-up: the ultimate test of the system is whether or not LE practitioners want to use it;
- code reuse of systems developed in GATE;
- a review of functionality relative to the requirements discussed in chapter 6.

We cover each measure in turn, followed by two case studies of the system in operation:

- a word sense tagging system;
- the LaSIE IE system.

Finally, we summarise the strengths and weaknesses identified in the rest of the chapter. Work in progress and future plans appear in chapter 9.

1.5 Part V: Addenda

Part V contains an appendix describing GATE's finite state transduction language, a bibliography, author index and list of acronyms.

As noted in chapter 3, software development generally works best as an iterative process. The analysis, design and implementation presented below is in fact the result of such a process, but for expository reasons the discussion in later chapters generally ignores the actual chronology of the system's development, and presents a coherent, serial whole.

The creation of software infrastructure must be undertaken in conjunction with the development of systems on which the infrastructure is based and which will in turn use its services. GATE was originally developed in the context of work on IE, but has subsequently grown to encompass a wider range of LE technology. GATE is a live project and the software is constantly evolving. Inevitably there is a time-lag between design, implementation, deployment and evaluation, and some elements of evaluation can only cover versions that have been deployed some time ago. Chapters 7 and 8, on GATE and its evaluation, concentrate therefore on GATE versions up to 1.5.2, the final release of version 1 that was made in 1999. Chapter 9 includes material on current work in progress leading up to version 2.

1.6 Summary

To conclude this chapter, a summary the structure of the thesis and the function of its various elements:

- in part I this introduction describes the nature and contribution of our work, chapter 2 describes at a high level the field of LE, and chapter 3 plays a similar role for Software Architecture;
- in part II chapter 4 looks at LE applications, technologies, methods and components, and provides structure for the discussion that is then reflected in chapter 5 in a review of previous work on SALE, which fits the various systems cited into a classification of the field;
- in part III chapter 6 further refines the structure derived in II, and presents a requirements analysis for SALE; chapter 7 describes our own implementation, the GATE system;
- in part IV chapter 8 measures the extent to which GATE has succeeded in meeting the requirements defined in chapter 6, and in becoming a useful tool for LE workers; finally chapter 9 discusses how we plan to address areas of weakness in GATE and build on the system's strengths in the future.

Chapter 2

Language Engineering

What is *Language Engineering* (LE)? Is it new? Is it any different from applied Computational Linguistics (CL) or Natural Language Processing (NLP)? Where did it come from and where is it going? This chapter situates LE in the context of its parent and sister disciplines, and its recent history. The aim of this analysis is to provide a basis for constructing and evaluating infrastructural systems, such as software architectures, frameworks and development environments, that facilitate LE research and development.

Attempting to define something as dynamic and multi-faceted as a research field concerned with human language is a difficult task, rather like trying to paint a portrait of a galloping racehorse. Given ten researchers there are likely to be ten definitions, all using similar terminology. The temptation in such circumstances is to invent a new vocabulary of terms in order to inject a new precision, but the risk is that the Gestalt will then be lost. At the opposite pole, a phenomenological account might avoid definition altogether and simply compile anecdotes. We will try to steer a course between these extremes, using existing terminology and letting a number of researchers speak for themselves in quotation, but still arriving at definitions that try to capture the essentials.

Section 2.1 presents definitions of CL, NLP, and LE. Section 2.2 looks at the history of the field, and at some of the factors influencing its genesis. Finally, section 2.3 examines six salient characteristics of LE as it stands at the beginning of the millennium.

2.1 Definitions

We begin with the general terms *language* and *engineering*, then move on to CL, NLP and LE.

The Collins English Dictionary defines language as "a system for the expression of thoughts, feelings, etc., by the use of spoken sounds or conventional symbols" [Makins 91]. Language is a communication mechanism whose medium is text or speech, and LE is concerned with computer processing of text and speech. We will define *engineering* in contrast to *science* (and, later, LE in contrast to CL and NLP).

Herb Simon contends that "... far from striving to separate science from engineering, we need not distinguish them at all. But if we insist upon a distinction, we can think of engineering as science for people who are impatient". This point is undermined a little by a criterion which immediately follows and by which the two fields may be separated: "While the scientist is interested specifically in creating new knowledge, the engineer is interested also in creating systems that achieve desired goals" [Simon 95]. We will preserve the distinction here and use a definition of science as "the systematic study of the nature and behaviour of the material and physical universe, based on observation, experiment, and measurement, and the formulation of laws to describe these facts in general terms" [Makins 91].

Engineering connotes "creating cost-effective solutions to practical problems by applying scientific knowledge" [Shaw & Garlan 96], or "applying scientific principles to the design, construction and maintenance of engines, cars, machines etc." [Makins 91], or a "rigorous set of development methods" [Brown 89]. The "... basic difference between science and engineering is that science is concerned with finding out how the world works, while engineering is concerned with using that knowledge to build artifacts in such a way that one can expect them to perform as required" (Tony Cowling, personal communication, 1996.)

Buried in the phrases "achieve desired goals" and "perform as required" is the implication that engineered systems must conform to externally-specified criteria of adequacy, for example criteria relating to the space/time profile of an executable processing a certain type of data. This in turn has implications for the engineering process: unless the external criteria are floating free in time, a system will not perform as required or achieve its goals if the process of its development is unbounded with respect to resource requirements. In other words, engineers have to operate within finite time and with finite personnel resources and equipment. As well as implying that its outputs perform predictably, engineering implies that the process of constructing those outputs is also predictable.

If the *act* of engineering is a particular kind of construction process, the *field* of engineering is both the body of scientific knowledge relevant to a particular engineering task and also what we may call craft, art, lore, or heuristics: a body of known practice offering solutions to problems. For example: in electrical engineering, books with titles like "The Art of Electronics" [Horowitz & Hill 89]; in software engineering, books like "Numerical Recipes in C – the Art of Scientific Computing" [Press *et al.* 95], or "Expert C Programming – Deep C Secrets" [van der Linden 94], or "Design Patterns – Elements of Reusable Object-

Oriented Software" [Gamma *et al.* 95]. This craft knowledge takes many forms, e.g. a description of which scientific results are relevant to particular problems; example solutions to problems; notes of common problems or pitfalls in particular implementation paradigms; anecdotal evidence suggesting that certain solutions may be better than others in some circumstances. For example: a book on database design may refer to relational algebra; a book on algorithms may provide a proven, tested method for sorting with a known complexity and space requirement; a book on expert programming may contain hints like 'versions of the `indent` code formatting program up to the mid-80's silently transform `x=!y` into `x!=y`'[1]; a book on design patterns might suggest that implementing a monitor pattern under a particular thread regime is inefficient.

To summarise, we define engineering as a construction process that is directed both by intended conformance of the resultant artifact to well-specified criteria of fitness and by constraints operating on the nature of the process itself; both the construction process itself and its outputs should be measurable and predictable; the activity is informed by relevant scientific knowledge and practical experience.

Note that as LE involves producing software, the branch of engineering that concerns us in this thesis is *software engineering* [Pressman 94]; chapter 3 examines current software engineering practice in relation to LE.

Turning to *Computational Linguistics*, we can arrive at a semantics compositionally by referring to our definitions of science and language: CL is that part of the science of human language that uses computers to aid observation of, or experiment with, language. If "Theoretical linguists... attempt to characterise the nature of... either a language or Language" or "a grammar or Grammar", then "...theoretical Computational Linguistics proper consists in attempting such a characterisation *computationally*" [Thompson 85]. In other words, CL concentrates "on studying natural languages, just as traditional Linguistics does, but using computers as a tool to model (and, sometimes, verify or falsify) fragments of linguistic theories deemed of particular interest" [Boguraev *et al.* 95].

Natural Language Processing is a term used in a variety of ways in different contexts. Much work that goes under the heading of NLP could well fit under our definition of CL, and some could also fit the definition of LE that follows. Here we will use a narrow definition that makes CL and NLP disjoint. Whereas CL is "a branch of linguistics in which computational techniques and concepts are applied to the elucidation of linguistic and phonetic problems" [Crystal 91], NLP is a branch of computer science that studies computer

> ...systems for processing natural languages. It includes the development of algo-

[1] Bizarre but true – see [van der Linden 94] p.11. The bug was apparently introduced because of a change from old B syntax where `a =+ 3;` meant 'add 3 to a' to the C form `a += 3;`.

> rithms for parsing, generation, and acquisition of linguistic knowledge; the investigation of the time and space complexity of such algorithms; the design of computationally useful formal languages (such as grammar and lexicon formalisms) for encoding linguistic knowledge; the investigation of appropriate software architectures for various NLP tasks; and consideration of the types of non-linguistic knowledge that impinge on NLP. It is a fairly abstract area of study and it is not one that makes particular commitments to the study of the human mind, nor indeed does it make particular commitments to producing useful artifacts. [Gazdar 96][2]

There are elements of both CL and NLP in Winograd's early description of the work as

> ... part of a newly developing paradigm for looking at human behavior, which has grown up from working with computers. ... Computers and computer languages give us a formal metaphor, within which we can model the processes and test the implications of our theories. [Winograd 72]

To summarise: CL is a part of the science of language that uses computers as investigative tools; NLP is part of the science of computation whose subject matter is computer systems that process human language[3]. These definitions are too simple to apply perfectly in practice, but they represent some of the core meanings of the terms, and will suffice for present purposes.

We have by implication conflated spoken language and textual language, but this hides the fact that there is a distinct field concerned with computer processing of spoken language which is

> ... known as **speech processing** or just **Speech**. Surprisingly, perhaps, Speech and NLP are really rather separate disciplines: until recently there has been little overlap in the personnel involved, the journals and conferences are largely disjoint, and the theoretical approaches and methods used have had little in common. Speech research studies the problems that are peculiar to processing the spoken form of language whereas NLP research, in principle at least, studies the problems that are common to the processing of both spoken and written forms of natural languages. [Gazdar 96]

[2]Though note that Gazdar was referring here to 'theory of linguistic computation', which he takes to be a component of NLP, not NLP per se.

[3]And if NLP is a part of AI, then AI is a branch of computer science whose subject-matter is programs that display capabilities similar to facets of human intelligence.

We will have more to say about speech below, but now we turn to the main subject of this chapter, Language Engineering.

From our perspective, Gazdar's definition of "Applied NLP" is close to that of LE, a subject which

> ... involves the construction of intelligent computational artifacts that process natural languages in ways that are useful to people other than computational linguists. The test of utility here is essentially that of the market. Examples include machine translation packages, programs that convert numerical data or sequences of error codes into coherent text or speech, systems that map text messages into symbolic or numeric data, and natural language interfaces to databases. [Gazdar 96]

LE is the application of NLP to the construction of computer systems that process language for some task usually other than modelling language itself, or "... the instrumental use of language processing, typically as part of a larger system with some practical goal, such as accessing a database" [Thompson 85][4]. Probably as good a short definition as any of the *products* of LE (and other software disciplines) is given by Jacobs: LE "systems are meant to be fast, effective and helpful" [Jacobs 92]. A longer definition of the field appears in the editorial of the first issue of the *Journal of Natural Language Engineering*, which also contrasts the practices of NLP and CL with those of LE[5]:

> The principal defining characteristic of NLE work is its objective: to engineer products which deal with natural language and which satisfy the constraints in which they have to operate. This definition may seem tautologous or a statement of the obvious to an engineer practising in another, well established area (e.g. mechanical or civil engineering), but is still a useful reminder to practitioners of software engineering, and it becomes near-revolutionary when applied to natural language processing. This is partly because of what, in our opinion, has been the ethos of most Computational Linguistics research. Such research has concentrated on studying natural languages, just as traditional Linguistics does, but using computers as a tool to model (and, sometimes, verify or falsify) fragments of linguistic theories deemed of particular interest. This is of course a perfectly respectable and useful scientific endeavour, but does not necessarily (or even often) lead to working systems for the general public. [Boguraev *et al.* 95]

They go on to define the "constraints to be satisfied", including the existence of a target

[4]Note that Thompson was referring to 'Applied CL' here.

[5]LE and Natural Language Engineering, or NLE, are synonymous.

user group with a defined need that can be solved by the LE system, favourable cost-benefit profile and defined criteria for testing and evaluation of the system.

The final component of our definition of LE is that it is an art, or a body of rules of thumb. This is little written of at present, but has been noted at various workshops. For example, at panel discussions quoted in [Mitkov 95, Mitkov 96]: Tsujii called LE heuristic and empirical (and not normally interested in language per se); Matsumoto said it is art, craft, and technique. Not strictly relevant, but included here for flavouring is Zaharin's comment that language engineers make things work without knowing why, whereas computational linguists know why their systems do not work. As with any engineering discipline, LE rests on an ever-expanding body of craft, lore, art, heuristics, and practice.

To summarise, our gloss on these various definitions is this: 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 a body of practice.

Having arrived at an answer to "what is LE?", we now turn to the question of whether it constitutes a new field or just a new label.

2.2 History

The development of applications software that makes use of NLP techniques is not a new phenomenon: Systran (the first major commercial MT system), for example, was programmed in the 1960s, and there has been a steady stream of technology transfer to products in the subsequent decades (from the use of formal language theory in compiler construction to morphology or dictionary lookup or spell checking in word processors – see [Church & Rau 95] for a review of commercial applications of NLP). In the sense of the production of language processing systems, LE-type activities have been around for several decades; but in the sense of an engineering discipline with an established body of practice LE is much more recent. Two points illustrate this distinction: a) engineering is not the same as production; b) dissemination of the experience, art and craft of the field by explicit publication is a relatively recent phenomenon.

In support of point a), the existence of products does not in itself imply that the systems were engineered in the sense defined above. Although the current organisation of production dictates that the ultimate test of the usefulness of artifacts is the market, just because a system is marketed does not mean that is has been successfully engineered – it may have

come in way over budget, only run on machines far bigger than those available to the target user base, or be much less accurate than originally planned, and the methods used in its production may have been completely ad hoc.

In support of point b), the idea that mere experience of the field deserved discussion, definition and record is a relatively new one. It was in the 1990s that LE work became more prevalent and better defined, and in the process acquired its own name, journal and expanding literature.

The origin of the term is traced by [Mitkov 95] to a panel proposed by Nagao at COLING-88; the phrase entered common currency in Europe in the early 1990s with the eponymous funding programme of the European Commission (EC) [Boguraev *et al.* 95, Mitkov 96]. Referring to this programme, the EC wrote in 1996 that "Language Engineering has successfully promoted a shift from long-term research activities to more immediately feasible and industrially relevant RTD[6] themes" by "supporting projects aiming at market opportunities in the short term and the medium term" [CEC 96]. An EC requirement for projects in this programme is that they must be industrially led. In the late 1990s the EC changed the title of the programme to *Human Language Technologies*, a term which we can regard as roughly synonymous with LE for present purposes.[7] Also in the late 1990s the term came into common usage, and one research centre even changed its name from the Department of Language and Linguistics to the Department of Language Engineering, reflecting the fact that "Much of the work in the department is concerned with the production of *systems* – programs that do something useful when confronted with language-oriented tasks" [Ramsay 00].

On the other side of the Atlantic in recent years [D]ARPA[8] has moved towards requiring research software to be built within efficient delivery vehicles that use a standardised API for data management and storage [Grishman 97, Cunningham *et al.* 96b]. Part of the idea behind this is to encourage a competitive market in modules by enabling plug-and-play swapping of components from different suppliers, again promoting engineering criteria in the language processing context.

In parallel with these changes the body of knowledge that a practising language engineer can draw on has also begun to expand. The Journal of Natural Language Engineering has already been mentioned; we can also note the Applied Natural Language Processing (ANLP) conference series [ACL 97], and although none currently exists it is becoming possible to imagine cookbooks for building certain types of LE systems. For example, Named Entity recognition[9] is now sufficiently well-understood for a set of recipes to be collected for systems

[6]RTD: Research and Technological Development.

[7]This was also the name of a similar programme in the US.

[8]The [Defense] Advanced Research Projects Agency (the 'D' seems to come and go every few years).

[9]Named Entity recognition (see section 4.2.2) was one of the tasks of the MUC-6 and MUC-7 Message Understanding Conferences, a DARPA-sponsored bi-annual Information Extraction competition series of

based on

1. Markov models [Bikel *et al.* 97, Miller *et al.* 98], or
2. Brill-style transducers [Vilain & Day 96, Day *et al.* 98], or
3. chart parsing [Gaizauskas *et al.* 95, Humphreys *et al.* 98].

In each case the relevant algorithms and data structures could be described, along with the expected performance levels and run-time resource requirements relative to certain types of text (based on current practice, probably only news texts), and the profile of the production process required (in 1, parameter selection, text markup and training; in 2, the same as 1 followed by manual tweaking; manual rule collection in 3).

To summarise, in important respects LE is a creation of the last decade, during which time the term and a body of literature describing practice, relevant science and experience have arrived.

There are a number of commonly-cited reasons for the establishment of LE including:

- the demand for applications in a world where electronic text has grown exponentially in volume and availability, and where electronic communications and mobility have increased the importance of multi-lingual communication;
- computer hardware advances which have increased processor speeds and memory capacity, while reducing prices, and therefore made the computationally expensive processes that have typified language processing work more viable;
- the increasing availability of large-scale, language-related, on-line resources, such as dictionaries, thesauri, and 'designer' corpora – corpora selected for representativeness and perhaps annotated with descriptive information [Wilks *et al.* 96];
- maturing NLP technology which is now able, for some tasks, to achieve high levels of accuracy repeatedly.

The next section looks at the current state of the field.

which the seventh, in 1998, was the last [Grishman & Sundheim 96, Hobbs 93, Cunningham 99b].

2.3 Characteristics

> *Rumour has it that the word "hermeneutics" is now regularly being heard in the corridors of Palo Alto research centers and that may be a sure sign of desperation among some of the more theoretically oriented.* [Wilks 96].

The 1990s saw a "wholesale invasion of the NLP community and culture by the more engineering-oriented approaches and behaviour of speech engineers" [Wilks 96], a change dramatic enough to constitute a "revolution that has taken place in natural language processing research over the last five years" [Gazdar 96]. As noted earlier, speech research has in the past been largely separate from NLP and CL, a fact reflected in the ethos and methods of the respective groups. Speech research has for a number of years focused on large-scale or 'real-world' tasks, i.e. tasks driven by the possibilities for marketable applications software based on automatic speech recognition and synthesis. In parallel with this focus has been an emphasis on quantitative evaluation as the arbiter of the worth of different approaches, and a willingness to consider any technique irrespective of psychological plausibility. (We should note here that perhaps the most significant reason for the growth of empiricism in speech research has been DARPA's insistence on quantitative evaluation in their speech and language programmes. As we shall see below, this has also had an effect on NLP and LE.)

Church and Mercer note a correlation between use of empirical methods and large-scale tasks, and knowledge-based methods and small-scale tasks [Church & Mercer 93], and the most salient characteristic of recent approaches to speech recognition has been their reliance on statistical modelling of language using hand-annotated corpora.

The growth of LE has seen similar characteristics emerge in textual language processing, and

> While practical utility is something different from the validity of a theory, the usefulness of statistical models of language tends to confirm that there is something right about the basic approach. [Manning & Schütze 99, p.4]

Other significant trends involve social factors such as the increasing pressure for research groups to collaborate, and the nature and composition of these research groups themselves. The rest of this section discusses these trends in turn.

2.3.1 Large-Scale Tasks

It has been a common observation that early AI approaches to language tended towards a 'toy problem syndrome': AI has often chosen to investigate artificial, small-scale applications of the technology under development. These 'toy' problems are intended to be representative of the work involved in building applications of the technology for end-user or 'real-world' tasks, but scaling up problem domains from the toy to the useful has often shown the technology developed for the toy to be unsuitable for the real job. This has led to "... a growing skepticism about the importance of small-scale, research systems and whether many of them are genuinely original, as opposed to being notational variants in a field not very aware of its own history" [Wilks 96], and to an increasing focus on larger-scale tasks.

For example, Wilks writes in [Cunningham *et al.* 95] that he began a

> ... Prolog grammar project in 1985 [Farwell & Wilks 89]: by 1987 it was perhaps the largest DCG (Definite Clause Grammar) grammar anywhere, designed to cover a linguistically well-motivated test set of sentences in English. Interpreted by a standard parser it was able to parse completely and uniquely virtually no sentence chosen randomly from a newspaper. We suspect most large grammars of that type and era did no better, though reports are seldom written making this point.
>
> The mystery for linguists is how that can be: the grammar appeared to inspection to be virtually complete – it *had* to cover English, if thirty years of linguistic intuition and methodology had any value. It is a measure of the total lack of evaluation of parsing projects up to that time that such conflicts of intuition and result were possible, a situation virtually unchanged since Kuno's large-scale Harvard parser of the 1960's [Kuno & Oettinger 62] whose similar failure to produce a single, preferred, spanning parse gave rise to the AI semantics and knowledge-based movement. The situation was effectively unchanged in 1985 but the response this time around has been quite different.

In the late 1980s and the 1990s,

> ... the combination of pressure from the U.S. Government funders... and the Zeitgeist itself have pushed NLP towards specific applications, systems evaluation and above all, large-scale language processing systems. ... The move to the large scale, to applications and to evaluation in NLP is worldwide at the moment. [Wilks 96]

This shift to applications has had several knock-on effects. As with speech research, it has led to a belief that what matters about systems is performance, not the interesting things about language or mind that might be illustrated. In many cases, the technologies now being developed are assistive, rather than fully automatic, aiming to enhance or supplement a human's expertise rather than attempting to mimic it [Gaizauskas *et al.* 96b, Kay 97a, Kay 97b]. It has also meant that measures of performance have become much more important – see the next section – and a renewal of interest in performance-based models of language, with a corresponding renewal and extension of statistical techniques – see section 2.3.3. Lastly, software skills themselves have become more important. As an engineering discipline whose product is software, successful LE is predicated on successful software engineering: "... whoever builds a large software system inevitably runs up against software engineering concerns. Failing to handle them well harms the overall effort accordingly. ... System development issues should be as important as the dictionary definitions" [Cowie & Lehnert 96].

2.3.2 Evaluation

It may seem obvious that language processing systems should be subject to empirical criteria of effectiveness. The big problem, of course, is determining precisely what the criteria of success should be. There is now a substantial recent literature on this question [Crouch *et al.* 95, Sparck-Jones 94, Sparck-Jones & Galliers 96, Gaizauskas 97, Gaizauskas *et al.* 98], and practical solutions to the evaluation problem have emerged in a number of areas.

Participants in the MUC (Message Understanding Conference, an Information Extraction competition) and TREC (Text REtrieval Conference, a competition for Information Retrieval, or "document detection") competitions [ARPA 96], for example, build systems to perform precisely-defined tasks on selected bodies of news articles. Human analysts are employed to produce correct answers for some set of previously unseen texts, and the systems produce machine output for those texts. The performance of the systems relative to human annotators is then measurable quantitatively. Quantitative evaluation metrics bring numerically well-defined concepts like precision and recall, long used to evaluate Information Retrieval systems, to other areas of LE.

2.3.3 Performance vs. Competence

Following Chomsky, a distinction can be made between human linguistic capabilities, or language *competence*, and the observable usage of language in everyday communication, or its *performance*. "A linguistic *competence* model aims to characterise the set of grammati-

cal sentences together with the possible analyses that can be assigned to these sentences; a linguistic *performance* model aims to describe the actual production and perception of natural language sentences in specific situations" [Bod 95]. It has been noted that competence models associate strongly with rationalism and that performance models associate strongly with empiricism [Church & Mercer 93]. The performance of language is our experience of language; our capacity for language is a more abstract object, the subtler aspects of which can be reasoned about at length without stumbling over inconvenient realities. With the shift to large-scale tasks and quantitative evaluation, language research since the early 1980s has seen increasing reliance on large text collections (corpora) intended as representative samples of some aspect of language performance. These corpora are often annotated with the intended output of some task (e.g. part-of-speech tagging, or Named Entity recognition and classification, or alignment of translations) and used to train statistical or connectionist models of the language input / task output relationship embodied in the data. Interestingly, this focus on corpora has more in common with past practice in lexicography than in linguistics, a fact which informed some of the upsurge in Machine Tractable Dictionary research in the 1980s [Wilks *et al.* 96].

The shiny new empiricism eventually triumphed over the evil old rationalism and everyone lived happily ever after. Well, no, not really. Although statistical methods climbed the accuracy curve rapidly to begin with, it became apparent that two factors undermined the simplistic separation of the new and the old sketched above. First, the addition of markup to corpora has often been informed by linguistic intuition, for example about the nature of syntax [Marcus *et al.* 93]. It is not clear, then, that the statistical model built from this data is in fact independent of a rationalist explanation of language competence. Secondly, purely statistical systems soon encountered problems dealing with non-local reference, and with the sparse phenomena that Zipf showed in the 1930s was an intrinsic property of language [Zipf 35]. Although initial results were promising, statistical MT, for example, soon appeared to be hitting a hard ceiling at fairly low performance levels [Wilks 94]. Responses to these limitations were also twofold. First, many systems now incorporate both statistical and symbolic linguistic knowledge; this trend led Gazdar to write of "paradigm merger" in [Gazdar 96]. Secondly, research into statistical NLP has become more interested in models "... that yield good results with relatively small samples" [Dunning 93, Karov & Edelman 96].

Hybrid systems are indeed increasingly common, but it may be too early to talk of consensus breaking out into full paradigm merger. Charniak wrote in 1993 that "... it is fair to say that few, if any, consider the traditional study of language from an artificial-intelligence point of view a "hot" area of research. A great deal of work is still done on specific NLP problems... but for me at least it is increasingly hard to believe that it will shed light on broader problems, since it has steadfastly refused to do so in the past" [Charniak 93]. There are also still plenty of purely statistical systems out there, and for some tasks their results are

comparable with the best symbolic or rule-based systems available, for example the Named Entity recogniser developed at BBN that uses Markov models [Bikel *et al.* 97].

In sum, NLP is still a science encompassing paradigm conflicts arising from substantially different views on many issues. This theoretical diversity (with a parallel diversity of technical approaches in LE) means that the field is dynamic and rapidly changing, and, given the distance between our current capabilities and those of a fully language-capable computer, this state of affairs can be expected to last well into this century. The field may not be hot, but it is simmering.

2.3.4 Data Reuse vs. Algorithmic Reuse

The idea of reusing previously developed components in new software systems has been part of the Holy Grail for software engineering for some time now [Yourdon 96], and has also become an issue for LE [Cunningham *et al.* 94] in the context of the large-scale systems discussed earlier. If we distinguish the data used by LE systems for training or as internal knowledge resources (e.g. lexicons) from the algorithms used in processing, we get two quite different pictures of the current level of reuse in LE.

In line with the trend to empiricism, the last ten years have seen substantial interest in large-scale processing of Machine-Readable Dictionaries, a number of which have become available to the research public, each stimulating a flurry of investigation into their possible uses. For example:

- the Longman Dictionary of Contemporary English (LDOCE) [Michiels *et al.* 80, Michiels & Noel 82, Walker & Amsler 86, Boguraev *et al.* 87, Boguraev & Briscoe 87, Wilks *et al.* 87, Guthrie *et al.* 90, Bruce & Guthrie 92];
- the Merriam-Webster New Pocket Dictionary [Amsler & White 79, Amsler 80, Amsler 81];
- Webster's Seventh New Collegiate Dictionary [Chodorow *et al.* 85, Markowitz *et al.* 86, Binot & Jensen 87];
- the Wordnet thesaurus [Miller (Ed.) 90, Resnik 93];
- the EDR dictionary [Yokoi 95, Miller *et al.* 95, Lenat *et al.* 95].

Similarly, there are many reports of work with large corpora of texts in general [Ejerhed & Dagan 96], and the Penn Tree Bank [Marcus *et al.* 93] in particular. For example:

- extracting grammars [Gaizauskas 95];
- building taggers and parsers and measuring their performance [Bod 96, Magerman 94, Magerman 95, Brill 92b, Brill 95, Miller *et al.* 98].

There remain significant barriers to reuse, to do with the format differences and distribution mechanisms (see section 5.3.1 and [Bird & Liberman 98, Bird & Liberman 99a, Peters *et al.* 98, Cunningham *et al.* 98a]), but in general the picture is a healthy one.

In contrast to this frequent reuse of data resources (commonly called 'Language Resources' – see section 5.3), algorithmic reuse remains low, one key reason being that the integration and reuse of different components can be a major task. Referring to the need for algorithmic reuse, the EAGLES tools group noted that:

> Unfortunately, the linguistic software that exists at present (see for example the lists at the Natural Language Software Registry) only begins to cover the growing needs. Industrial software is often expensive or unavailable, and usually hard to adapt or extend. On the other hand, the substantial body of natural language processing academic software is often experimental and hard to get, hard to install and under-documented, and sometimes unreliable. In both cases tools are typically embedded in large, non-adaptable systems which are fundamentally incompatible. Although efforts to develop standards for data representation are underway, little effort has been made to develop standards for linguistic software, and **software reusability** is virtually non-existent.
>
> As a result, there is a serious lack of generally usable tools to manipulate and analyse the text and speech corpora and collections that are now becoming widely available. Worse, there is enormous duplication of effort: it is not at all uncommon for researchers to develop tailor-made systems that replicate much of the functionality of other systems and in turn create programs that cannot be reused by others, and so on in an endless **software waste cycle**. The reusability of data is a much-discussed topic these days; similarly, software reusability is needed, to avoid the re-inventing of the wheel characteristic of much language-analytic research in the past three decades. [Veronis & Ide 96] (Emphasis in the original.)

On a smaller scale, it is not unusual for the life-cycle of doctoral research in AI/NLP to be similarly wasteful, with software created during the research becoming unused fairly soon after completion (particularly at the original site when the student has moved on).

An infrastructure which enabled relatively easy reuse of past work could significantly increase research productivity, and this was the rationale for the work reported in the rest of this thesis.

2.3.5 Collaborative Research

If we think of NLP's theoretical diversity as a centrifugal force pushing researchers outwards from consensus, the EC and DARPA funding programme requirements that sites must collaborate is a centripetal tendency pushing them back together again[10]. These requirements show no signs of changing: collaborative development of LE systems is here to stay for at least the immediate future.

NLP and LE research groups are typically small: between 5 and 35 seems to be common, usually covering a large range of different subfields. This means that these groups are unable to bear large overheads on their development methods. Additionally, most personnel are not practising software engineers, or at least have little practical experience of engineering large systems. Together, these social factors militate against the adoption of rigorous development methods in language processing labs. This is a significant problem, as Cowie and Lehnert point out (with reference to developing Information Extraction components):

> Automation and speed-up of these components are critical for rapidly porting systems to new domains. The speed of the system is another critical factor in iterating to achieve high levels of accuracy through a fast development cycle. ... The idea that good software development always requires good software engineering is worth repeating in the IE context, because NLP researchers do not always think of software engineering as a critical NLP research weapon. [Cowie & Lehnert 96]

More knowledge and use of software engineering methods would be valuable; the next chapter discusses how software engineering (and particularly software architecture) may be applied to LE. An infrastructure which facilitated the construction of large-scale LE systems while taking care of the engineering issues involved could increase the autonomy and productivity of small research groups, and this has been an aim of the work reported in this thesis.

[10] This distinction was drawn in [Gaizauskas *et al.* 96b].

Chapter 3

Software Architecture

This chapter characterises software infrastructure in a general sense (part II characterises software infrastructure in the specific context of LE). The latter part of the chapter (section 3.2) discusses software architectures, frameworks and development environments. To begin with, however, section 3.1 examines the subject of software engineering (SE) in small group working. This digression is motivated by the current lack of support for small group software development in SE methods, and by the idea that "the critical programming concerns of software engineering and artificial intelligence tend to coalesce as the systems under investigation become larger" (Perlis, in [Abelson *et al.* 85]).

3.1 Small Group Software Engineering

A point seldom made by books on SE methods (e.g. [Yourdon 89, Booch 94, Booch *et al.* 99]; an honourable exception is [Fowler & Scott 00]) is that a large and complex engineering apparatus is inappropriate outside medium-to-large development teams. As noted in the previous chapter, most LE software is currently developed in research laboratories and startup companies. In small-group working there is often very little division of labour (everybody designs, everybody codes), and there is certainly little space for the overheads associated with a 'high ceremony' process that demands that every small action or change be formally requested, approved and documented. "A major problem with the large-scale methodologies is that they don't scale down well" [Fayad *et al.* 00]. Many small-group developers have concluded that the overhead involved in using *any* SE method is simply too high. Where this leads to a rejection of explicitly designing software, this can be a mistake.

All software may be said to have a design, but if the code itself is its only manifestation, the design is *implicit*. If, on the other hand, there is some additional description of the system,

perhaps in English, or a diagramming language, or a mathematical notation, then the design is *explicit*. Implicit design can only hope to rival explicit design when the program is small enough to fit comfortably into one person's head, and when it will never need to be fixed or upgraded in any way after its initial development. Such programs exist, but they are relatively few: software almost always needs an explicit design. This means developers need some conventions about how to express that design (a modelling language) and about how to arrive at the design and transform it into code (a process specification or development guide). There are a number of candidates for each of these; sections 3.1.1 and 3.1.2 below discuss them in the context of supporting small-group, low-overhead development.

3.1.1 Modelling Languages

It is now widely accepted that Object Orientation (OO) imposes a beneficial set of structural conventions on software models. These conventions are rather close to those that are habitually deployed for classification and description in many contexts, and include encapsulation, inheritance and composition. We will assume that developers will model using objects; this section looks at candidate *modelling languages*. Let's say that I decide that the system I am about to build will be composed of two separate programs, one fielding incoming messages and the other presenting the current set of them to a user via a graphical interface. If I sit down and write a couple of sentences in English about how this will work, then I have made a model of the software using English as a modelling language. (If, on the other hand, I create two files with a `.c` extension and start both of them with `int main(int argc, char *argv[])`, I have proceeded straight to implementation without using any modelling language.)

Amongst a myriad options, we will consider four modelling languages here: mathematics; English, or whatever natural language is shared by the majority of the developers and users; the Unified Modelling Language (UML) [Fowler & Scott 97, Booch *et al.* 99]; Java stubs and the `javadoc` HTML generation utility [Venners 98]. In each case we will bear in mind that for a language to be appropriate to small-group development projects of the sort we are interested in, it must be reasonably simple to produce, and must either have a widely-understood semantics or be reasonably easy to learn. With reference to the previous paragraph, it should also support Object-Oriented modelling.

3.1.1.1 Mathematical Notations

Computers perform numerical operations, and much of the behaviour of programs can be modelled using mathematical languages of one sort or another (e.g. Z or VDM,

[Spivey 89, Spivey 92, Jones 90]). These languages have the advantage of a precise and verifiable semantics. However, if we define the role of a modelling language in the present context as enabling the *explicit representation* of a software design for purposes of communication amongst and between systems developers and systems users, mathematics appears unsuitable. To be explicit to developers a design must be *less* complex than the code itself; a modelling language that is as complex or more complex simply trades one inscrutable artifact (code) for another, to no one's benefit.

In other words, programs "can be assigned a semantics (another model, by the way), and if a program's function can be specified, say, in the predicate calculus, the proof methods of logic can be used to make an acceptable correctness argument. Unfortunately, as programs get large and complicated, as they almost always do, the adequacy, consistency, and correctness of the specifications themselves become open to doubt, so that complete formal arguments of correctness seldom accompany large programs." [Abelson *et al.* 85]

As further anecdotal evidence of this point, van der Linden recounts a newsgroup discussion on program proofs:

> Readers of the Usenet network's C language forum were somewhat surprised to see the following strident posting one summer day. The poster (name omitted to protect the guilty) demanded the universal adoption of formal program proofs, because "anything else is just engineering hacks." His argument included a 45-line proof of the correctness of a three-line C program. ... Alert readers were even more startled to see the same poster follow up a few minutes later... [with various corrections to the proof]... Not only did the proof have two errors, but the C program that he "verified" was not in fact correct to start with! ... Phew! Let's stick with the "engineering hacks" for now. [van der Linden 94]

Add to the problems of complexity the fact that these languages are almost certainly difficult to learn (and most developers do not already know them), and it seems that mathematical notations are not a suitable candidate modelling language for our purposes.

This is not at all to say that formal mathematical models have no place in the specification of any of the data structures and algorithms that play a part in software systems. It is merely to say that they are appropriate only in the cases where either the properties of the software entities being described are inherently mathematical (for example a library of maths functions), or the model is describing domain objects that are themselves usefully described mathematically (for example a regular language described by a finite state grammar). The distinction is a fine one. The key point is that on the one hand the everyday modelling of software does not benefit from mathematical expression, but on the other hand

the application domain or the solution chosen may in some circumstances benefit from such expression. The necessary measure is that of utility for the developers involved: if a mathematical characterisation of a problem helps the developer understand it, or demonstrates the applicability of certain algorithms, then it is useful; if the characterisation simply results in a restatement of the design in terms no easier to manipulate than code itself, it is not useful.

Similar points may be made in relation to formal models of the complexity of algorithms. The standard way to estimate the performance of a computer program is to decide whether its algorithms are of type P or NP (roughly, whether they will run in polynomial or non-deterministic polynomial or exponential time in relation to constant factors such as size of data). Unfortunately it is quite possible for theoretically faster algorithms to perform worse in practical applications than those that theory says should be slower. A "striking example of the potential gulf between theory and practice is given by the so-called linear programming problem", one method for the solution of which is the *simplex algorithm*, invented in 1947, and "known to be, in theory, an exponential time algorithm, but when used in practice (on problems involving hundreds or even thousands of variables) it works extremely well... Indeed, the indications are that it tends to run in linear time" [Devlin 98]. In the 1970s a new solution was found, the *ellipsoidal algorithm*, which was shown to run in "polynomial time. Unfortunately, though this meant that the method was theoretically better than the simplex method, when applied to real-world problems it did not perform anything like as well as the simplex algorithm." The story so far seems to indicate that the practising software developer should take algorithmic complexity with a large pinch of salt, but there's a twist: in 1984 another linear programming algorithm was invented by a theoretician seeking a polynomial time solution, and on this occasion the work bore practical fruit, with the new method outperforming the simplex algorithm on real data sets. In this case "some highly sophisticated abstract mathematics" led "to a concrete product of crucial importance in the 'real' world of business, commerce and defence." [Devlin 98]

These anecdotes suggest that the formal complexity of algorithms is far removed from the operational performance of algorithms. Little wonder, then that Knuth refers to a "crisis of confidence that has unfortunately been growing between theoretical computer scientists and the programmers-on-the-street who have real problems to solve." [Knuth 93] His contribution to counteracting this crisis was the publication and free distribution of *The Stanford Graphbase*, a set of implemented graph algorithms and example programs. In each case a number of problems are described first, followed by a program that uses algorithms appropriate to their solution. If you can match your problem to one in the book, you have a solution whose operational properties can be tested reasonably straightforwardly. In this way the work provides a bridge between theoretical computer science and engineering practice.

What should we conclude from this tale? Neither that formal characterisation is a necessary step in building software nor that it is irrelevant. For the mathematically inclined, a formal model may seem to excuse lack of engineering sophistication. For engineers, having a working system may seem to excuse lack of knowledge of related formal models. While it is true that formal models are not executable, and that therefore the engineer's excuse is a little more plausible, in the end neither excuse is valid. Formalists have nothing without running code; engineers need to be aware of formal results in order not to miss opportunities to increase their system's conformance to its operational requirements.

3.1.1.2 Natural Language (NL)

Natural languages like English all share the great advantage for modelling of being the most expressive communication method available, and the great disadvantage of being the most expressive communication method available, and therefore inherently ambiguous. Another advantage of NL is that it involves no overhead to learn (for adults). Other disadvantages are that it is often more verbose than other languages ('a picture is worth a thousand words'), and that it is difficult to impose structure or stick to conventions.

3.1.1.3 The Unified Modelling Language (UML)

What is revolutionary about the UML is not that there is anything particularly revolutionary or new in it, but that it is *unified* – there's only one of it. Just as developing software with a group of people who speak the same natural language will be easier than doing so through translators, when developers come to associate certain sorts of icons with the same sorts of concepts, software modelling should become easier.

The UML has a formal semantics that makes it possible to translate into code. Modelling tools such as Rational Rose or Together/J allow 'round-trip engineering' of model and code: change the UML and the tool automatically changes the corresponding code; change the code and the UML model automatically reflects the change. There are things that UML can express that a programming language cannot, and vice versa, but nevertheless this technique can reduce coding time and help ensure consistency between the UML model and the code that implements it.

The notation used in this thesis is summarised in figure 3.1, which describes some simple elements of UML class diagrams expressing relationships of inheritance ('is-a'), composition ('has-a') and other forms of association such as 'uses'. Note that in general in our diagrams 'has-a' or 'part of' is modelled as UML association (not UML composition or aggregation) – see [Fowler & Scott 97] p.80.

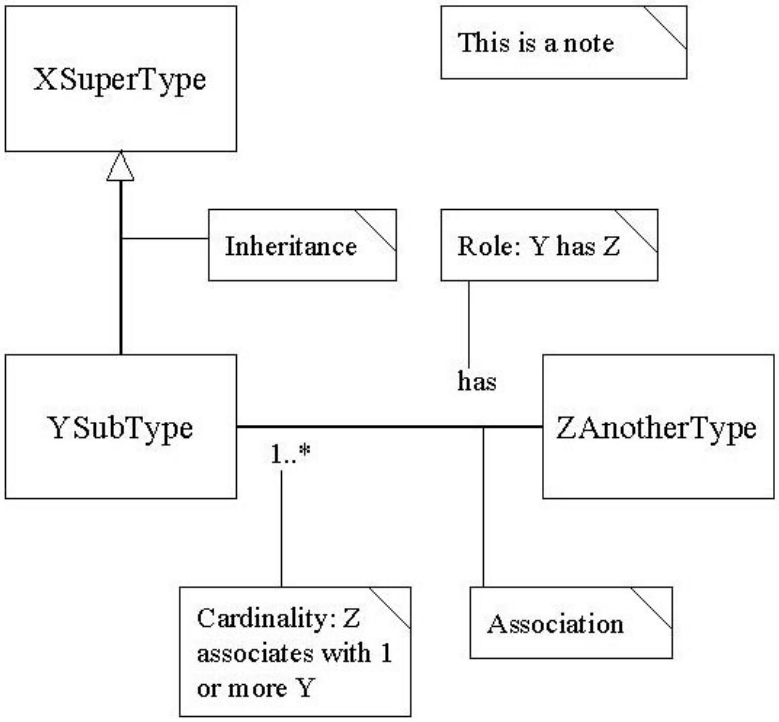

Figure 3.1: A fragment of UML: simple class diagrams

3.1.1.4 Java Stubs, `javadoc` and HTML

Different modelling techniques work better in some contexts than in others, and a modeller should always be sensitive to the preferences and aptitudes of the people who will use a model. There is little point expressing all sorts of complex information about a system in a class diagram whose intended use is for a system walkthrough with a user group who have only just figured out what an object is and what it is useful for. Similarly, there is little point giving a conceptual class diagram to a novice programmer and expecting them to come up with a good set of method signatures and instance variables. This latter example brings us to the last of the modelling techniques discussed in this chapter, namely the use of Java method stubs coupled with the `javadoc` HTML generation tool: classes can be written without method implementations but with comment documentation, and then run through the HTML generator to provide design documentation.

> A simple approach to design documentation is to express your design in Java code that has the structure and organization, but not the functionality, of your end product. You define and name classes and interfaces, and describe them in comments. You define the name and type of each field, and describe them in comments also. You define the name, return type, number of parameters and

> parameter types of each method, but you don't define the method body. Instead of a body you write a comment that describes what the body of the method, once implemented, will do. The comment can be pseudo-code, regular text description or both. If you take this approach, your design will remain relatively easy to change throughout the design process. [Venners 98]

How can *code* claim to be a modelling tool? The answer is that we use the term to denote any technique that helps us abstract from the final running program, and by this definition Venners' technique qualifies. The HTML documentation that is generated from the class or interface declarations is at a higher level of abstraction than runnable code, and certainly helps us discuss a design without having the final system in front of us. It is also an excellent way to provide a detailed specification to a programmer who is not experienced enough or closely involved with the project enough to fill in the gaps in a higher-level specification themselves. The technique also fits nicely with the common idiom of coding type hierarchies as interfaces with parallel class implementations, which makes later use of classes as supertypes much easier (if we want to subclass more than one type in Java we can only do so if all but one are interfaces, not classes, due to the lack of multiple implementation inheritance).

Other advantages are that the process of documenting the Application Programmers' Interface (API) is begun right from the start and is part of the same process as commenting the code, and when using a modelling tool that permits reverse engineering of model from code the stub code can be used to generate UML diagrams.

There are disadvantages. It is of necessity a technique that is close to the code, and so not suitable for modelling from a conceptual perspective. It is reliant on Java.

In conclusion, the technique may be a useful supplement to other modelling forms when used at the design specification end of the modelling continuum.

3.1.1.5 Patterns

This discussion of modelling ends by looking briefly at *patterns*, which may be considered an adjunct to modelling languages.

As noted above, one benefit of explicit modelling is an enhanced ability to communicate about software artifacts. This ability has been used recently to produce *patterns* that describe common analysis and design fragments in order for them to be reused [Fowler 97, Gamma *et al.* 95]. The same benefits apply as to reuse of code: less new work needs to be done, and the reused object is likely to be better understood, better documented

and better tested due to its use in numerous systems. "A *pattern* is a common solution to a common problem in a given context." [Booch *et al.* 99] Pattern work is still in its infancy. Precisely how to express patterns and how to index them so that users can get access to relevant patterns in an easy fashion are still research issues, but a modelling language and explicit design are probably prerequisites for benefiting from the technique.

The development of GATE currently uses a mixture of UML, `javadoc` and natural language descriptions – see chapters 7 and 9.

3.1.2 Development Processes

Software development presents special problems for small teams (such as academic research groups, specialised sections of corporate labs, or SME[1] development teams). Additional problems are present in collaborative multi-site development, as is often the case in LE. The rest of this section describes a simplified adaptation of the Objectory process [Jacobson *et al.* 99], tailored to small group development (following [Fowler & Scott 97]). Key features of the process are:

- Object-Oriented modelling (discussed above);
- iterative development;
- design driven by use cases.

The modelling language used is UML. The process is 'low ceremony': specification artifacts such as use case descriptions or class diagrams are constructed only where they are useful for developers and technical managers.

Use cases are snapshots of the external view of a software system. "In essence, a **use case** is a typical interaction between a user and a computer system" [Fowler & Scott 97, Fowler & Scott 00], but beyond this statement there is no fixed definition of what they should look like ([Cockburn 97] cites experience of 18 variations). They are partly defined by their purpose, which is to identify what a computer system should do for its users. Fowler identifies these properties:

- "A use case captures some user-visible function.
- A use case may be small or large.
- A use case achieves a discrete goal for the user."

[1] Small-to-Medium-sized Enterprise.

For example: *a developer will be able to export annotations on texts as an XML document.* Use cases describe interactions with users (who may in our case be software developers, researchers and others using a graphical interface, or other software components). They are used to break the desired functionality of the system into manageable chunks, which are then used as the unit of design and implementation planning.

The process uses **iterative development** and follows four stages: inception, elaboration, construction and transition. Iteration is built in because models and the requirements they reflect inevitably change during the process:

> Every computer program is a model, hatched in the mind, of a real or mental process. These processes, arising from human experience and thought, are huge in number, intricate in detail, and at any time only partially understood. They are modelled to our permanent satisfaction rarely by our computer programs. Thus even though our programs are carefully handcrafted discrete collections of symbols, mosaics of interlocking functions, they continually evolve: we change them as our perception of the model deepens, enlarges, generalises until the model ultimately attains a metastable place within still another model with which we struggle.
> Perlis, in [Abelson *et al.* 85].

Project **inception** produces the initial requirements statement and plan. It may be that the production of the project proposal itself and of technical details during contract negotiation constitutes the inception phase.

The **elaboration** phase occupies the first 10-20% of the project and deals with initial systems analysis, risk analysis and plan maintenance. In projects where the major software components are based on combination and extension of existing systems, elaboration first involves the analysis of these systems, yielding a domain model and use case set. To minimise risks this phase includes the selection of development technology (decisions such as 'Oracle for the database, with JDBC connection to clients') and skeleton prototypes used to predict and combat technology problems (e.g. by testing throughput of a client/database link under expected load conditions). Finally, the use case list is used to extend and add detail to the work plan, feeding into the next phase.

The **construction** phase occupies the remainder of the project. (The transition phase is interleaved with the construction phase.) A key feature of this phase is iteration: all the well-known activities of software development (design; code; test) are repeated in a number of cycles. Iterative development means lower risk because problems can be identified and attacked earlier, and planning revised on the basis of actual results as the project progresses.

It also means that the project can have several delivery points, instead of a single monolithic delivery late in the work programme. Construction iterations address a set of use cases and begin with domain and design modelling for these cases. (Note that in the small group development paradigm it is unwise to try to produce complete models; typical models are a page or two of text, a class diagram or two and possibly some interaction diagrams.) Coding is tightly coupled with design, using tools that allow production of code from UML models and automatic revision of models relative to code. Test development is a required part of coding; unit test suites are developed along with the main code, and are preserved throughout the life of the software. Integration with other completed components comes next, along with documentation. At the end of a construction iteration plan maintenance identifies schedule slippage and sets up targets for subsequent iterations.

The **transition** phase incorporates beta testing and system tuning. Transitioning occurs at each release.

As an aside, there is a parallel between the structure of this thesis and the iterative process just described. We begin at a fairly abstract level and iteratively become more specific, moving from general context, through domain analysis and into requirements analysis. The models at later stages of this process become part of the design of GATE; the final stage is their reflection in the code of the system itself.

3.1.3 Code Reuse

The use of code in more than one program, or *code reuse*, is a highly desirable property of a software engineering process. If a requirement can be expressed in a form abstract from a particular program, and if several programs are to be built to do similar work, then the implementation of that requirement may be reused. The obvious advantage is that a given problem is only solved once. Additionally, the more a piece of code is used, the better tested and debugged it will be. Patterns, architecture and frameworks can all encourage reuse, and reuse is one of the main pay-offs involved in the use of software infrastructure – see next section. Explicit modelling can facilitate reuse by making the link between code and design clearer.

There are significant barriers to successful code reuse, principally: culture; quality; documentation. Some software teams rely on lines of code produced as a metric of programmer productivity, thus encouraging code bloat and discouraging reuse. If library implementations are not completely reliable, programmers will be reluctant to use them. If code is not sufficiently well documented to make it easy to find and use in new circumstances, this will also inhibit reuse. On a related point, Abelson *et al.* argue that “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." [Abelson *et al.* 85]

3.2 Software Infrastructure and Software Architecture

As mentioned in chapter 1, the aim of this work is to construct software infrastructure for language engineering: software that is intended to apply to whole families of problems within this field, and to be like a craftsman's toolbox in service of construction and experimentation. This section describes what 'software infrastructure' means. We look at three types of infrastructural systems:

- frameworks;
- architectures;
- development environments.

These are examined in turn below; in each case we give definitions, rationale and examples.

All these types of infrastructure share the characteristic that *it is unreasonable to expect their successful development in isolation from the building of systems that use their services* [Booch 94, IBM 99a, IBM 99b]. Infrastructure must embody the shared features of a number of systems; it is very unlikely that this will be possible without direct feedback from construction of systems using the architecture. Frameworks, architectures and development environments all need to be developed in conjunction with systems on which they are based and which will in turn use their services. In our case, GATE has been developed in close coordination with the development of a large range of LE research and applications development projects – see chapter 8.

3.2.1 Frameworks and Components

3.2.1.1 Definitions

A **framework** "is a reusable design for all or part of a software system" [Johnson 97], made up of "a set of prefabricated software building blocks that programmers can use, extend, or customise for specific computing solutions." [IBM 99a]. In other words, a "*framework* is a collection of classes that provide a set of services for a particular domain; a framework

thus exports a number of individual classes and mechanisms which clients can use or adapt. [They]... represent reuse in the large." [Booch 94]

The idea rests heavily on previous practice, being somewhat equivalent to a well-documented procedure library that includes precise details of the API exported along with examples of how to use it in particular problem contexts. Similar facilities may also be referred to as platforms [Knuth 93].

The framework concept lies behind component-based development, such as that available in the Active X or Java Beans architectures. Here basic interfaces are specified that must be implemented by classes wishing to participate in the framework. These classes can then be manipulated in predictable ways by development environments and applications. These frameworks have a particularly visible extensibility mechanism; such mechanisms are a common feature of frameworks, and constitute one of their benefits (see next section).

In summary, a framework typically means an Object-Oriented class library that has been designed with a certain domain in mind, and which can be tailored and extended to solve problems in that domain. Frameworks may also be known as platforms, or component systems. We will see below that there is a crossover between frameworks and architectures.

3.2.1.2 Rationale

A principal benefit of frameworks is code reuse. "By providing an infrastructure, the framework dramatically decreases the amount of standard code that the developer has to program, test, and debug." [IBM 99b] This in turn can lower maintenance costs, as the framework part of systems is factored across the whole set of systems that use it. Frameworks also provide "infrastructure and architectural guidance", and "a mechanism for reliably extending functionality" [IBM 99b].

"Since large programs grow from small ones, it is crucial that we develop an arsenal of standard program structures of whose correctness we have become sure – we call them idioms – and learn to combine them into larger structures using organizational techniques of proven value." Perlis, in [Abelson *et al.* 85]. Frameworks are a way to express these types of idioms in a lasting form.

On a cautionary note: "The most profoundly elegant framework will never be reused, unless the cost of understanding it and then using its abstractions is lower than the programmer's perceived cost of writing them from scratch. The real payoff comes when these classes and mechanisms get reused over and over again, indicating that others are gaining leverage from the developer's hard work, allowing them to focus on the unique parts of their own particular problem." [Booch 94]

3.2.1.3 Examples

Examples of successful frameworks include:

- MacApp, a large toolkit available to developers on the Macintosh platform.
- A large number of UI libraries including OWL, Xvt.
- Active X control libraries available to e.g. Visual Basic programmers.
- The Java APIs, from collections to 3D imaging.

3.2.2 Architectures

3.2.2.1 Definitions

There is no single accepted definition of **Software Architecture**. At times the term seems to be used synonymously with Object-Oriented *framework* [Hendry & Harper 96]; at others it refers to patterns of modularity, control and data flow at the level of overall system structure [Pressman 94, Shaw 93]; at still others it is a vague reference to some aspect of system organisation (e.g. 'the client-server architecture', 'a Web-centric architecture', 'an Object-Oriented architecture').

Among those whose work is specific to architecture, there is more in common:

- "The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships between them." [Bass *et al.* 97]
- "The *architecture of a software system* defines that system in terms of computational components and interactions among those components. Components are such things as clients and servers, databases, filters, and layers in a hierarchical system. Interactions among components at this level of design can be simple and familiar, such as procedure call and shared variable access. But they can also be complex and semantically rich, such as client-server protocols, database-accessing protocols, asynchronous event multicast, and piped streams." [Shaw & Garlan 96]
- "The structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time." [Garlan & Perry 95]

These views all emphasise the high-level structure of systems, their components and interactions, but this view is not a consensus even within the software engineering community, as this quote from a 1999 book on UML indicates: "Software architecture is not only concerned with structure and behaviour, but also with usage, functionality, performance, resilience, reuse, comprehensibility, economic and technology constraints and trade-offs, and aesthetic concerns." [Booch *et al.* 99] However, for the purposes of discussing infrastructural software, we will adopt a definition that seems to be at the core of the consensus view of software architecture researchers:

> Software architecture is concerned with system structure – organization of the software, assignment of responsibilities to components, and assurance that the components' interactions satisfy the system requirements. [Shaw & Clements 97]

As with design, all software systems have an architecture. Sometimes the architecture is explicit, perhaps conforming to certain standards or patterns, sometimes it is implicit. Where an architecture is explicit and targeted on more than one system, it is known as a reference architecture, or a domain-specific architecture. The former is "a software architecture for a family of application systems." [Tracz 95] The term 'Domain-Specific Software Architecture' (DSSA – subject of an eponymous ARPA research programme [Hayes-Roth 94, Tracz 95]) "applies to architectures designed to address the known architectural abstractions specific to given problem domains." [Clements & Northrop 96]

> Recently there has been considerable interest in developing "reference" architectures for specific domains... These architectures provide an organizational structure tailored to a family of applications, such as avionics, command and control, or vehicle-management systems. By specializing the architecture to the domain, it is possible to increase the descriptive power of structures. Indeed, in many cases the architecture is sufficiently constrained that an executable system can be generated automatically or semi-automatically from the architectural description itself. [Shaw & Garlan 96]

It is this kind of architecture that can be thought of as infrastructural, and we will see in chapter 7 that the GATE system can be considered as a DSSA, and which can be used to automate the creation of executables based on combinations of user-supplied components with facilities of the architecture itself.

3.2.2.2 Rationale

As with frameworks, architectures are aimed at reuse and thence at software productivity. "Many would argue that future breakthroughs in software productivity will depend on our ability to combine existing pieces of software to produce new applications." [Garlan *et al.* 95] If a usable implementation of an architecture exists, or if one can be produced at acceptable cost, overheads for subsequent work are lowered: the architecture takes care of common tasks, provides a way to structure documentation and training, and enables the implementation of development tools.

Architectures may also facilitate interoperability [Wilson 99] by constraining external boundaries.

Architectures can encourage Object-Orientation by being implemented in an OO fashion. They can encourage modelling by being documented with a model. They can be based on patterns. They can reduce maintenance costs by making it easier to understand existing systems ("Much of the time spent on maintenance goes on understanding the existing code" [Shaw & Garlan 96]).

An architecture can help developers at all stages:

Requirements: knowing the parameters within which your system might work, and therefore what requirements might reasonably be put on it.

Design: suggesting patterns that may serve certain requirements.

Implementation: by association with framework library code to do the obvious and repetitive tasks in the domain.

Evaluation: suggesting and automating means of testing and scoring system performance.

In order to help software developers, an architecture intended for reuse should cover things that are common to many systems in the target domain, and that can be specified without tying the developer to a particular technique.

Reiter emphasises architecture as a descriptive aid: "Integration is not the only possible goal of a reference architecture. There are many other goals as well, which are less ambitious but may be easier to achieve. In particular, a reference architecture may help people describe systems, by giving them a reference point to base their descriptions on." [Reiter 99] This reference point can help build a shared terminology and promote understanding of the similarities and differences between systems.

Wilson notes that architectures are a means of communicating the requirements and capabilities of a technology to external users and potential users [Wilson 99].

As will be seen below, architectures can be built on to provide development environments that exploit their properties and ease their deployment in applications.

3.2.2.3 Examples

An architecture for systems for processing natural language, for example, might organise components as a collection of annotators of documents, and documents as 'fat pipes' implemented as SGML streams [McKelvie *et al.* 98]. An architecture for Graphical User Interface (GUI) programming might make a distinction between the data to display, or model, the various possible screen views of the data, and the methods that supervise the updating and manipulation of the display, or controller (this is a common architecture in use in a number of GUI toolkits called Model/View/Controller).

An architecture for component integration might specify certain API conventions that all components in a system must follow (as does the GATE component model – see chapter 7). For example, *Java Beans* is an architecture or framework for component integration which is used in development environments to help developers create applications from existing building blocks. The Beans architecture is a set of interface conventions from Sun Microsystems. Various development environments (e.g. Borland JBuilder or IBM VisualAge) help you create classes that conform to and use these conventions. Many beans are available, and when correctly written they plug easily into the development environments. One disadvantage of OO systems that Java Beans addresses[2] is "...that in order for one object to interact with another (via procedure call) it must know the identity of that other object" [Shaw & Garlan 96]. A Java Bean can, in contrast, be loaded by an appropriately aware tool or system without prior knowledge: the API details necessary to communicate with the Bean can be discovered at runtime.

Chapter 5 reviews a number of Language Engineering architectures.

[2] As does CORBA [The Object Management Group 92], the Common Object Request Broker Architecture.

3.2.3 Development Environments

3.2.3.1 Definitions

An implementation of an architecture that includes some graphical tools for building and testing systems is a **development environment**.

Architectures are often associated with integrated development environments (of which the most common are probably Integrated Programming Support Environments, or IPSEs). An architecture is not a development environment, and there is no necessary connection between the two. However, one of the benefits of an explicit and repeatable architecture is that a symbiotic relationship with a dedicated development environment can arise. In this relationship the development environment can help designers conform to architectural principles and visualise the effect of various design choices and can provide code libraries tailored to the architecture.

3.2.3.2 Rationale

Development environments can encourage adherence to an architecture, and can automate common tasks.

3.2.3.3 Examples

Many IPSEs, such as Borland JBuilder, provide functions such as rebuild (including calculation of source interdependencies), debug, etc. Conformance to architectures such as Java Beans is facilitated by 'wizards' that can be used to create new software entities that implement the correct conventions.

Environments also exist to support implementation of mathematical and statistical systems, e.g. Mathematica, SPSS.

3.2.4 Terminology

Finally, a note on terminology. As we have seen, architectures and frameworks are close relatives. "In fact, you can think of a framework as a kind of micro-architecture that encompasses a set of mechanisms that work together to solve a common problem for a common domain." [Booch *et al.* 99] We have also shown how architectures and frameworks can promote the capabilities of development environments. For these reasons, and because of the historical

genesis of the work, the system discussed in this thesis is generally referred to as an 'architecture', when, if we were a little more pedantic, it would be called a framework, architecture and development environment. The rest of this thesis will examine Software Architecture for Language Engineering (SALE), and our implementation, a General Architecture for Text Engineering (GATE).

Part II

Domain Analysis

Chapter 4

Entities and Activities

This part analyses the LE domain in some detail. As discussed in chapter 1, we distinguish *applications*, *technologies*, *methods* and *components*. Applications are complete software systems, often products. Technologies are those areas of research that are mature enough to be the subject of transfer from laboratories to software houses. Methods are the means by which researchers perform their experiments, and developers implement technologies and applications. Components are the elements which make up LE systems.

Section 4.1 covers applications; section 4.2 covers technologies and methods; section 4.3 covers components. Note that in each case our aim is to cover a representative sample of the types of work that a SALE might wish to support, not to be comprehensive in our coverage of the LE field. This sample will also serve during requirements analysis for GATE (see chapter 7). The models of component relationships given in section 4.3 are UML object models – see chapter 3.

Chapter 5 analyses previous work on SALE.

4.1 Applications

As discussed in chapter 2, LE is concerned with how best to build applications software or parts thereof that process human language. To begin our analysis of the domain that SALE addresses, this section presents examples of such applications.

[Church & Rau 95] gives a review of commercial applications of NLP. Examples include:

- speech transcription systems, e.g. Dragon Dictate;
- automated translation systems, e.g. Systran;

- spelling, grammar and style checkers, such as those found in word processors;
- summarisation;
- localisation and internationalisation;
- text generation;
- natural language interfaces to databases, and question-answering systems;
- speech synthesis;
- accessibility enhancement tools, e.g. [Newell *et al.* 98];
- named entity recognisers, e.g. Identifinder;
- second language learning;
- text comparison, versioning and authoring tools;
- text filtering and routeing, e.g. SRA's Assentor email screening system;
- authorship determination.

The rest of this section describes an imaginary application that is representative of many elements of the field. This application is called *Information Harvester*, and it consists of a Web crawler that extracts information from diverse sources in order to populate a database of elementary facts. In addition, it allows the presentation of these facts as simple summaries in multiple languages, which may be written or spoken. The Harvester seeks to process all the pages it can find that contain any human language; when it comes across audio or video documents it attempts to locate and transcribe any spoken sections, and then to extract information from the transcription.

The Harvester can usefully stand as a prototypical LE system because it uses techniques that cover a large subset of LE. A SALE that fully supported the Harvester application, therefore, would be of widespread utility as LE software infrastructure. We will use this application as a running example in chapter 6 to show how SALE usage scenarios relate to LE applications. The next section gives a high-level description of the example.

4.1.1 An Information Harvester

The Harvester operates in a manner similar to Web search engines like Infoseek[1] or AltaVista[2], using a crawler program to collect large numbers of pages which are then analysed.

[1] `http://www.infoseek.com/`
[2] `http://www.altavista.com/`

Web crawlers seek out pages by following hyperlinks, and are an interesting design challenge due to huge data volumes, server restrictions on mechanised access, the rapid change of many sources such as news, and the increase in dynamically generated pages for purposes such as Internet shopping. We will ignore these issues here, as they fall outside the LE domain.

When the crawler has downloaded new pages from the Web, the Harvester seeks to process any human language that they contain. In many cases this means processing text contained in HTML pages, but it may also mean dealing with spoken language present in audio or video documents. When it comes across such documents it attempts to locate and transcribe any spoken sections, using Automatic Speech Recognition (ASR – see below). The end result in either case is text, which is the input to the Information Extraction (IE – see below) stage.

IE deploys language analysis techniques to identify facts present in the text so as to be able to store those facts in a constrained format such as a database table. For example, the Harvester IE components might be set up to track product prices and price changes. A page derived from a newspaper article describing the falling price of gold bullion, for example, might lead to the creation of a database record with fields for the commodity name (gold), the direction of change (downwards), the price (n USD), and the amount of the change (3%).

In addition, the Harvester allows the presentation of these facts as simple summaries in ordinary language, which may be written or spoken, e.g. "Gold down by 3% from USD n to m". Results presented in this way may be made available in multiple languages, using Natural Language Generation (NLG – see below) techniques.

4.2 Technologies and Methods

This section discusses LE technologies and methods, focusing in particular on those that underlie the Information Harvester example given in section 4.1. Section 4.2.1 discusses ASR; section 4.2.2 discusses IE; section 4.2.3 discusses NLG. These three technologies respectively represent input, analysis and output stages for LE applications. Their characteristics serve as input to the GATE design and evaluation processes, and are presented in some detail for that reason. The section does not attempt a comprehensive review of LE technologies; wider coverage may be found in textbooks such as [Allen 95, Hutchins & Somers 92, Reiter & Dale 00, O'Shaugnessy 87, Gazdar & Mellish 89]. Note also that because technologies intersect, certain subjects appear incidentally in the following that could occupy sections in their own right (so, for example, grammar is discussed in the section on NLG; this is not meant to suggest that NLG is the sole or main user of grammars in LE).

Note that composition of technologies is common; for example, translation technology uses

analysis and generation technology.

LE technologies are developed using methods such as:

- statistical modelling and search – see section 4.2.1;
- symbolic computation over rule sets and other structures – see sections 4.2.2 and 4.2.3;
- finite state transduction – see appendix A;
- quantitative and qualitative evaluation – see section 4.2.2.

4.2.1 Automatic Speech Recognition

This section looks at ASR R&D. We begin with a look at human speech production and recognition, then at digital signal processing methods, and at various approaches to ASR. Finally we present the most common current ASR method, Hidden Markov Models, and note the role of training data in ASR systems development.

The relevance of this material extends beyond the problem of speech transcription. The methods employed in ASR systems are now common in textual LE, where statistical modelling became widely used in the previous decade (see chapter 2). One statistical method, Hidden Markov Models, is discussed in some detail in this section. The cross-over between spoken and typeset language is also of increasing importance due to the convergence of different transmission media around the Internet, which puts additional strain on indexing and search mechanisms. Text searching technology is an essential part of the Web due to the huge volumes available; in the coming world of video-on-demand, users will be faced with similar quantities of audio-visual material. This may be expected to increase the demand for R&D on video indexing, which in turn may lead to increased interest in combinations such as ASR-to-text-to-IE. Such work is already under evaluation in DARPA's Broadcast News programme [Renals *et al.* 99, Gotoh & Renals 00] and elsewhere.

4.2.1.1 Speech Production

Speech is the transfer of information via acoustic signals. People encode information in speech sounds when talking, and decode those sounds when listening, in order to recognise the words spoken to them. The mechanisms employed by people in the recognition process are of obvious interest for ASR research. As far as the exact cognitive and neurological processes involved are concerned, present knowledge is incomplete. It is clear, however, that a variety of information components are used in the decoding phase of speech communication. These

components range from the basic acoustic signal produced when talking to the semantic and pragmatic knowledge that underpins dialogue.

The set of speech sounds produced when talking are differentiable by their acoustic characteristics. It is this property that is exploited by the most direct (and currently most common) form of ASR, that which matches acoustic signals against stored representations of speech units, 'recognising' the best match.

The problem with this approach is the tremendous amount of variation imposed on the base acoustic building blocks of speech by physical factors and by conscious alterations used to enrich the information content of utterances.

Speech production is physically dependent upon the size, shape and even health of each particular speaker. Although different parts of speech share similar productive mechanisms and broad acoustic characteristics, the actual physical pressure wave will vary widely.

Even greater variations are purposefully employed by human speakers to enhance power of expression. Prosody, the patterning of stress, rhythm and intonation, is an integral component of spoken language. Prosody is deployed for numerous reasons. The relative social status of speaker and addressee indicates a particular prosodic delivery; emotional state, intended humour or gravity another. Regional accents also entail prosodic differences. More fundamentally, prosody

> ...serves several syntactic purposes: (a) segmenting long utterances into smaller phrasal or clausal units, (b) noting the relationship between such phrasal units, and (c) indicating whether an utterance is a question... [O'Shaugnessy 87]

A large amount of the expressivity afforded by speech communication is due to prosody [Waibel 88], contributing to huge variations in the form of acoustic signal used to represent each particular unit of speech.

A further complication is noise – few conversations take place between two people in a soundproofed anechoic chamber, and even those that do will likely be punctuated by coughs, murmurs of agreement and interruptions.

4.2.1.2 Speech Recognition

Given all these diversifying influences on the acoustic signals used in speech, how is it that human speech decoding is so accurate? People routinely follow conversations in a variety of dialects carried out in noise-filled environments. The answer is that we deploy linguistic

and world knowledge to supplement the raw acoustic information provided by our sense of hearing.

A language may be specified by a lexicon and a grammar. The lexicon is a dictionary of the words in the language; a grammar describes the language syntax, the ways in which words may be combined into sentences. Lexical units may be further decomposed into morphemes, indivisible grammatical or semantic units that may stand alone as words or be grouped to form other words, e.g. print-s, print-ed, print-ing, re-print [Gazdar & Mellish 89]. Similarly, the unit of word construction in speech is the phoneme, "defined such that if one phoneme is substituted for another in a word, the meaning of the word may be changed" [Ainsworth 88]. Only certain phoneme sequences are permissible in any given language. (It should be noted here that coarticulation, the overlapping of vocal tract articulator movements between one phoneme and the next, leads to each phoneme being represented in physical production terms by a number of phones, or sound units, known as the allophones of that phoneme.)

Grammatical information, although not explicitly present in speech, is also applied in the understanding process. There is no rigid mapping between formal sentential grammar (which tends to be thought of as textual) and the structures of everyday utterances, but grammatical utterances tend to be recognised more easily by listeners, and completely ungrammatical language is not ordinarily spoken. (We might say 'him and me' with dubious grammar, but not 'him me and' with no grammar.) Also present is semantic information – the *meaning* of speech – contained in the conjunction of words and in contextual references. The semantics of utterances is a key element of human speech usage – in the case of ambiguous word signals from other information layers, semantic knowledge usually provides a resolution (e.g. 'toucan' and 'two can' may be undifferentiable at the acoustic or lexical levels, but the meaning of a sentence or utterance framing them will almost certainly constrain one form or the other). A related area is pragmatics, which referentialises speech to its usage in an active dialogical form [Waterworth 87].

A human speech recognition process can use knowledge of all these areas. A string of phones are heard, and allocated to their respective phonemes; words are constructed according to morphological and lexical rules, and their parent sentences deduced from grammatical, semantic and pragmatic knowledge. This description suggests a hierarchical use of linguistic knowledge, but it is likely, given the neurological organisation of the brain and the speed of information processing attainable by people using component processors much slower than a microprocessor, that all these choices are performed in parallel, each constraining the others until an interpretation that maximally satisfies the various levels of 'rules' is synthesised.

The rest of this section details the approaches and techniques of ASR with computers.

4.2.1.3 ASR Approaches

ASR tasks are divisible into three classes. Human-like ASR is termed continuous speech recognition (CSR). Two simpler types exist, connected word recognition (CWR) where input speech is a string of words separated by small pauses, and isolated word recognition (IWR). CWR may be considered as a special case of IWR. Each class of ASR is further divisible into speaker-dependent and speaker-independent sub-classes, speaker-dependent systems being significantly easier to implement but of more limited utility than speaker-independent systems. The Information Harvester (see section 4.1.1) requires speaker-independent CSR, because in surfing the Web and seeking to transcribe speech it will encounter many different speakers who will be using normal (connected) speech.

Approaches to ASR have broadly followed two paths: the knowledge-based, or linguistic approach, that seeks to utilise morphological, syntactic and semantic information about language to develop speech understanding systems as opposed to simple recognisers; the pattern-based approach that relies on the acoustic information in speech, matching input utterances against stored templates.

The classic early example of the linguistic approach was funded by the [D]ARPA in the 1970s [Barr & Feigenbaum 81, Ainsworth 88, Cox 90]. Despite the obvious potential advantage for recognition of using all levels of information present in speech communication, the knowledge-based approach has had limited success, and recent ASR systems have relied heavily on pattern matching. Paradoxically, the approach that uses only acoustic information, ignoring other sources, is currently more successful than the intuitively better knowledge-based approach. Because of this the knowledge- and pattern-based techniques have been summarised as 'doing the right thing wrong or the wrong thing right'. Despite the prevalence of the pattern approach ('which is used by all commercial ASR devices' [Cox 90]) there is a general agreement that future work, particularly in CSR development, will need to incorporate higher levels of information [Ainsworth 88, O'Shaugnessy 87, Waterworth 87]. The level of current pattern-based technology is such that accurate speaker-independent IWR is possible for vocabularies typically less than 100 words, rising to a few hundreds in speaker-dependent mode.

4.2.1.4 Hidden Markov Models

Hidden Markov Models (HMMs) provide a means to represent probabilistically a statistical analysis of training data. HMMs are currently the most widespread method employed in ASR systems.

HMMs are derived from Markov chains (or processes). A Markov chain comprises a sequence

of states in which the probability of a state occurring depends upon the preceding state. This dependency restriction distinguishes Markov chains from the more general category of stochastic process. Models of Markov chains can be used to describe utterances such as words. In hidden Markov models states are multi-valued, with a probability specified for each value. These models are 'hidden' because a particular value does not imply a particular state – the state sequence in a chain derived from the model is masked by the value sequence. HMMs, used as templates representing vocabulary items, have been found to be of great utility in ASR. Good recognition accuracy relative to other methods can be achieved; templates are small compared to the set of training samples they are built from, yet they effectively capture data from multiple samples; temporal information is present and usable. HMMs are trained by iterative adjustment relative to samples of the speech units they are to represent. For each vocabulary item a set of training samples is collected; in general, the greater the cardinality of the set the higher the quality of the resultant model. (The SPHINX system [Lee 89], for example, used phone segments from 2240 sample sentences to train 45 phoneme models.) The training stage represents a statistical analysis of the sample data.

At recognition time each HMM template is compared with the input speech unit (e.g. a word). For each model the maximum probability of it generating the comparison unit is computed, and the model with the highest probability selected (subject to a rejection threshold used to exclude weakly recognised items).

The speech unit characterising an HMM system is that which a single model represents. Typically the unit is a word (SPHINX uses phonemes). At both training and recognition stages, before any processing using HMMs, sample utterances are segmented to give a number of sequential elements. An appropriate technique here is to derive a set of codewords via vector quantisation (VQ). Each segment is then a VQ codeword. The set of codewords in the VQ codebook then represents the possible values associated with each HMM state.

The following sections give the mathematical notation and formulæ (derived largely from [Huang *et al.* 90, Jelinek 97]) necessary for the initialisation, evaluation, decoding and estimation (training) of HMMs for ASR.

Modelling Notation

An HMM may be represented by

$$\lambda \; = \; \{A, B, \pi\}$$

A is an $N \times N$ matrix where N is the number of states in the model. It specifies transition probabilities between states:

$$A = \{a_{ij} \;\; | \; a_{ij} = Pr(s_{t+1} = j \;\; | \; s_t = i)\}$$

The probability function given to define a_{ij} here means the probability of moving to state j at time $t+1$ from state i at time t. (Note that as some transition must always occur, $\sum_{j=1}^{N} a_{ij} = 1$.)

B is an $N \times L$ matrix where L is $\mid v = \{v_1, v_2 \ldots v_L\} \mid$. v is the set of possible observation symbols; i.e. for any segmented utterance $O = \{O_1, O_2 \ldots O_T\}$, all segments $O_t \in v$. B specifies the probability of observing O_t at state j. (Continuous probability density functions are sometimes used; here we consider the more common case of discrete probability distributions.)

$$B = \{b_j(O_t) \mid b_j(O_t) = Pr(O_t \mid s_t = j)\}$$

π is a vector of length N giving a probability distribution for states starting a particular state sequence. N_I defines the number of valid initial states (giving S_I, the set of those states); π_i will be zero for $i > N_I$. N_F (with set S_F) is the number of permissible final states (valid sequence terminals).

$$\pi = \{\pi_i \mid \pi_i = Pr(s_1 = i)\}$$

Utterances (e.g. the word 'hello') are called observations and consist of symbol sequences of length T (T being proportional to the time frame of the utterance). Symbols index whatever units are produced by the signal processing front end of the ASR system – e.g. VQ codewords. Observations are denoted:

$$O = \{O_t \mid O_t \in v\},\ t = 1, 2 \ldots T$$

The set of observations produced by the same utterance (e.g. all samples derived from saying 'hello') is denoted

$$O^M = \{O^1, O^2 \ldots O^m\}$$

A final element of the notation concerns the topology of the model. HMMs for ASR must be feed-forward (i.e. a transition from s_i to s_j where $i > j$ is invalid, or has zero probability) in order to capture effectively the temporal ordering of speech sounds [Cox 90]. The most common topology [Huang *et al.* 90], and the one used here, also specifies all possible forward transitions up to a maximum 'jump' of states to pass over in a transition (e.g. given a maximum jump of 2, $j - i \leq 2$ is valid, but not $j - i > 2$). This type of model specifies a linear sequence of states, and is known as the left-to-right model. The max jump will be denoted here as J. Figure 4.1 shows a left-to-right HMM with $J = 2,\ N = 5$.

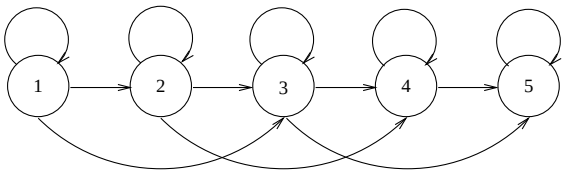

Figure 4.1: Left-to-right HMM

Notation Summary

$\lambda = \{A, B, \pi\}$ = an HMM.
$A = \{a_{ij} \mid a_{ij} = Pr(s_{t+1} = j \mid s_t = i)\},\ i = 1, 2 \ldots N$ = state transition probabilities.
$B = \{b_j(O_t) \mid b_j(O_t) = Pr(O_t \mid s_t = j)\},\ j = 1, 2, \ldots N$ = symbol probabilities.
$\pi = \{\pi_i \mid \pi_i = Pr(s_1 = i)\},\ i = 1, 2 \ldots N$ = initial state distribution.
N = number of states in λ.
$N_I = \mid S_I \mid$ = number of initial states.
$N_F = \mid S_F \mid$ = number of final states.
$O = \{O_t \mid O_t \in v\},\ t = 1, 2 \ldots T$ = an observation.
$T = \mid O \mid$ = length of the observation.
$v = \{v_1, v_2 \ldots v_L\}$ = set of possible observation symbols.
$L = \mid v \mid$ = cardinality of symbol set.
$O^M = \{O^1, O^2 \ldots O^m\}$ = set of observations relating to the same utterance.
$O^n = \{O^n_1, O^n_2 \ldots O^n_{T_n}\}$ = an observation $\in O^M$.
$T_n = \mid O^n \mid$.
$S = \{s\}$ = a set of states.
$(s_t = i)$ = state i at time t.
J = max forward transition.

Initialisation

Building a model for a vocabulary item begins with the initialisation of a 'naïve' HMM which does not relate to speech data but is formed so as to allow training using algorithms given below.

When using continuous probability densities, the initialisation of a model should be based on analysis of the training data. For discrete probability distributions, however, uniform initial estimates are valid [Huang *et al.* 90].

For A, an element is then zero for invalid transitions (a_{ij} where $j < i$ or $j - i > J$). For valid transitions (a_{ij} where $i \leq j \leq i + J$) the value is $\frac{1}{n}$ where n is the number of valid

transitions for a particular value of i. For example, in a model with $N = 3$,

$$A = \begin{bmatrix} \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \\ 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix}$$

Formally, initialisation of A is given by:

$$\begin{aligned} a_{ij} &= 0,\ j < i \text{ or } j - i > J \\ a_{ij} &= \frac{1}{n},\ i \leq j \leq (i + J),\ n = \mid \{j \mid i \leq j \leq (i + J)\} \mid \end{aligned} \tag{4.1}$$

Initialising B is simpler, each element being set to $\frac{1}{L}$. This means that $\sum_{k=1}^{L} b_{jk} = 1$, which is true because a state j must always emit some symbol k.

$$b_{jk} = \frac{1}{L} \tag{4.2}$$

With $N = 3$ and $L = 5$, B becomes:

$$B = \begin{bmatrix} \frac{1}{5} & \frac{1}{5} & \frac{1}{5} & \frac{1}{5} & \frac{1}{5} \\ \frac{1}{5} & \frac{1}{5} & \frac{1}{5} & \frac{1}{5} & \frac{1}{5} \\ \frac{1}{5} & \frac{1}{5} & \frac{1}{5} & \frac{1}{5} & \frac{1}{5} \end{bmatrix}$$

The initial distribution depends upon N_I: the first N_I members of π are set to $\frac{1}{N_I}$; others are zero. (The most common case sets N_I and N_F to 1, so $\pi_1 = 1, \pi_i = 0, i \neq 1$.)

$$\begin{aligned} \pi_i &= \frac{1}{N_I},\ i = 1, 2 \ldots N_I \\ \pi_i &= 0,\ i = N_I + 1, N_I + 2 \ldots N \end{aligned} \tag{4.3}$$

For $N = 3$, $N_I = 1$:

$$\pi = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}$$

The initialisation formulæ have the following input/output: input: N, J, N_I, L; output: λ.

Evaluation

Evaluation means testing an HMM against a speech sample and determining how accurately the HMM models the sample. This is the basis of HMM recognition – each model

is evaluated relative to speech inputs and the best scoring model selected. The vocabulary item represented by this best model is 'recognised' by the system (subject to a lowest permissible score threshold used to exclude non-vocabulary items). Although HMMs must be trained before any meaningful recognition can occur, the training algorithm uses data from the Baum-Welch evaluation algorithm given here, so this section precedes that on training. Evaluation involves calculating $Pr(O \mid \lambda)$. There are three algorithms for calculating $Pr(O \mid \lambda)$, derived from the work of L.E.Baum, R.Welch and A.J.Viterbi. The Baum-Welch algorithm gives two equivalent means of evaluating λ relative to O, the forward and backward probability calculations. The Viterbi algorithm is an alternative, computationally simpler method, which also provides a basis for 'unhiding' or decoding a state sequence (see section on decoding below). Note that the two Baum-Welch calculations yield the same result, so only one is needed for evaluation. Both are necessary for solving the estimation problem.

Baum-Welch Evaluation

The forward algorithm generates the probabilities for each state at each time t given a partial observation sequence of length t. This gives α, a $T \times N$ matrix where $\alpha_t(i) = Pr(O_1, O_2 \ldots O_t,\ s_t = i \mid \lambda)$.

$$\begin{aligned} \textit{Step } 1: \quad \alpha_1(i) &= \pi_i b_i(O_1), \quad i = 1, 2 \ldots N && (4.4) \\ \textit{Step } 2: \quad \alpha_t(j) &= \left[\sum_{i=1}^{N} \alpha_{t-1}(i) a_{ij}\right] b_j(O_t), \quad t = 2, 3 \ldots T && (4.5) \\ \textit{Step } 3: \quad Pr(O \mid \lambda) &= \sum_{i \in S_F} \alpha_T(i) && (4.6) \end{aligned}$$

Steps 1 and 2 generate α; step 3 calculates the probability (denoted P^{BW} below) that the model λ generated the observation O, the Baum-Welch solution to the evaluation problem.

Input: λ, O, T, N, N_I, S_F, S_I. Output: $\alpha[T][N]$, $P^{BW} = Pr(O \mid \lambda)$.

The backward algorithm does the same in reverse, as its name suggests. Here we generate β, specifying probabilities for each state happening at time T, given the partial sequence from t to T: $\beta_t(i) = Pr(O_{t+1}, O_{t+2} \ldots O_T \mid s_t = i,\ \lambda)$.

$$\begin{aligned} \textit{Step } 1: \quad \beta_T(i) &= \frac{1}{N_F},\ i \in S_F \\ \beta_T(i) &= 0,\ i \notin S_F && (4.7) \\ \textit{Step } 2: \quad \beta_t(j) &= \sum_{i=1}^{N} a_{ji} b_i(O_{t+1}) \beta_{t+1}(i), \quad t = T-1, \ldots 1 && (4.8) \\ \textit{Step } 3: \quad Pr(O \mid \lambda) &= \sum_{i \in S_I} \pi_i b_i(O_1) \beta_1(i) && (4.9) \end{aligned}$$

Note that step 3 generates the same value P^{BW} as the forward algorithm.

Input: λ, O, T, N_F, S_I, S_F, N; output: $\beta[T][N]$, P^{BW}.

Viterbi Evaluation

Where the Baum-Welch algorithms consider the likelihood of a model generating an observation summed over all state sequences, the Viterbi algorithm considers only the most likely sequence. An $N \times T$ matrix ψ is produced, which can be used to determine this best sequence (see section on decoding below). For evaluation, $P^V = Pr(O \mid \lambda)$ is derived. (Calculation of ψ is not necessary for evaluation only.).

$$\begin{aligned}
Step\ 1: \quad \delta_1(i) &= \pi_i b_i(O_1) \\
\psi_1(i) &= 0 \qquad (4.10) \\
Step\ 2: \quad \delta_t(j) &= \max_{i=1,2\ldots N} \left[\delta_{t-1}(i)a_{ij}\right] b_j(O_t), \quad t = 2,3\ldots T \\
\psi_t(j) &= \operatorname*{argmax}_{i=1,2\ldots N} \left[\delta_{t-1}(i)a_{ij}\right], \quad t = 2,3\ldots T \qquad (4.11) \\
Step\ 3: \quad P^{V^*} &= \max_{s \in S_F} \left[\delta_T(s)\right] \\
s_T^* &= \operatorname*{argmax}_{s \in S_F} \left[\delta_T(s)\right] \qquad (4.12)
\end{aligned}$$

where '*' indicates optimal path value (so s_T^* is the optimal final state).

Input: λ, O, T, N, S_F; output: $\psi[N][T]$ and s_T^* for decoding, P^{V^*} the evaluation figure.

Decoding Algorithm

It has been noted that Markov models with multiple symbols present at each state are 'hidden': the optimal state sequence of a model is not explicit. Using the Viterbi algorithm to derive ψ and s_T^*, the optimal state sequence of a model relative to an observation can be calculated:

$$\begin{aligned}
s_t^* &= s_T^*, \quad t = T \\
s_t^* &= \psi_{t+1}(s_{t+1}^*), \quad t = T-1, T-2\ldots 1 \qquad (4.13)
\end{aligned}$$

This generates in reverse order the optimal state sequence S^* that results in P^V.

Input: s_T^*, ψ, T; output: S^*.

Estimation

Once a model has been initialised as shown above it must be 'trained' – re-estimated relative

to training data to model accurately the vocabulary item it represents. The training problem involves the recalculation of a model λ as $\overline{\lambda}$ such that $Pr(O \mid \overline{\lambda}) > Pr(O \mid \lambda)$. Baum-Welch re-estimation using single observations for each utterance uses the α and β probabilities from the forward and backward algorithms. An iterative process adjusts λ incrementally – at each pass α, β, and P^{BW} are calculated and the model re-estimated as shown below. The difference between the new P^{BW} score and the previous one must be above a predefined threshold (set according to empirical data regarding the amount of training sufficient to build successful models) to justify another training pass. There are two essential components of the re-estimation formulæ. First, transition probabilities *for time* t (A is time non-specific):

$$\begin{aligned} \gamma_t(i,j) &= Pr(s_t = i,\ s_{t+1} = j \mid O,\ \lambda) \\ &= \frac{\alpha_t(i) a_{ij} b_j(O_{t+1}) \beta_{t+1}(j)}{\sum_{k \in S_F} \alpha_T(k)} \end{aligned} \tag{4.14}$$

The second component gives state probabilities for time t:

$$\begin{aligned} \gamma_t(i) &= Pr(s_t = i \mid O,\ \lambda) \\ &= \frac{\alpha_t(i)\beta_t(i)}{\sum_{k \in S_F} \alpha_T(k)} \end{aligned} \tag{4.15}$$

Note that the denominator of these equations is P^{BW} from the forward algorithm.

Given the values of γ for an existing model, re-estimation of λ as $\overline{\lambda}$ is given by:

$$\overline{a}_{ij} = \frac{\sum_{t=1}^{T-1} \gamma_t(i,j)}{\sum_{t=1}^{T-1} \gamma_t(i)} \tag{4.16}$$

$$\overline{b}_j(k) = \frac{\sum_{t \in O_t = v_k} \gamma_t(j)}{\sum_{t=1}^{T} \gamma_t(j)} \tag{4.17}$$

$$\overline{\pi}_i = \gamma_1(i) \tag{4.18}$$

The summation of the numerator of the $\overline{b}$ equation is read as: sum over all t where $O_t = v_k$.

Input: λ, N, O, S_F, N_F, T, α, β; output: $\overline{\lambda}$ where $Pr(O \mid \overline{\lambda}) \geq Pr(O \mid \lambda)$.

To extend Baum-Welch re-estimation to multiple observations (essential for successful training) the numerators and denominators of the above equations are summed over the observation set (O^M).

$$\overline{a}_{ij} = \frac{\sum_{n=1}^{M}\sum_{t=1}^{T_n-1}\gamma_t^n(i,j)}{\sum_{n=1}^{M}\sum_{t=1}^{T_n-1}\sum_{j=1}^{N}\gamma_t^n(i,j)} \tag{4.19}$$

$$\overline{b}_j(k) = \frac{\sum_{n=1}^{M}\sum_{t\in O_t^n=v_k}\gamma_t^n(j)}{\sum_{n=1}^{M}\sum_{t=1}^{T_n}\gamma_t^n(j)} \tag{4.20}$$

$$\overline{\pi}_i = \frac{\sum_{n=1}^{M}\alpha_t^n(i)\beta_t^n(i)}{\sum_{n=1}^{M}\sum_{k\in S_F}\alpha_T^n(k)}, \quad t=1 \tag{4.21}$$

The re-estimation of π is unnecessary given $N_I = 1$ as element one will be unity and all the others zero – these values will never change in any part of the model.

Input: λ, N, O^M, S_F, N_F, T^M, α^M, β^M; output: $\overline{\lambda}$ where $Pr(O^M \mid \overline{\lambda}) \geq Pr(O^M \mid \lambda)$.

Summary

HMMs allow the application of statistical models of large quantities of data to the ASR problem, and are currently very popular in ASR systems. HMM recognisers employ parameterisation/segmentation techniques to derive observation sequences from speech units (e.g. words). The probabilities at a point in a Markov chain of an element of the set of possible observation symbols occurring and the probabilities of transition onwards in the chain comprise the elements of states in an HMM. HMMs can be described by matrices A and B and vector π. Algorithms exist for the training and evaluation of HMMs.

4.2.1.5 Training Data

As with certain types of IE (see next section), ASR systems typically rely on supervised machine learning. Large quantities of recorded speech must be hand-annotated with phoneme and word boundaries to act as the input to HMM training algorithms.

4.2.2 Information Extraction

IE [Cowie & Lehnert 96, Appelt 99] is a process which takes unseen texts as input and produces fixed-format, unambiguous data as output. This data may be used directly for display to users, or may be stored in a database or spreadsheet for later analysis, or may be used for indexing purposes in Information Retrieval (IR) applications such as Internet search engines.

It is instructive to compare IE and IR: whereas IR finds relevant texts and presents them to the user, the typical IE application analyses texts and presents only the specific information from them that the user is interested in. For example, a user of an IR system wanting information on the share price movements of companies with holdings in Bolivian raw materials would typically type in a list of relevant words and receive in return a set of documents (e.g. newspaper articles) which contain likely matches. The user would then read the documents and extract the requisite information themselves. They might then enter the information in a spreadsheet and produce a chart for a report or presentation. In contrast, an IE system user could, with a properly configured application, automatically populate their spreadsheet directly with the names of companies and the price movements.

There are advantages and disadvantages with IE in comparison to IR. IE systems are more difficult and knowledge-intensive to build, and are to varying degrees tied to particular domains and scenarios (see below). They are also (for most tasks) less accurate than human readers. IE is more computationally intensive than IR. However, in applications where there are large text volumes IE is potentially much more efficient than IR because of the possibility of reducing the amount of time people spend reading texts. Also, where results need to be presented in several languages, the fixed-format, unambiguous nature of IE results makes this relatively straightforward in comparison with providing full translation facilities.

The rest of this section looks at types of IE, performance levels and domain dependence, and details the various subtasks of modern IE systems. The section finishes up with an extended example, and a discussion of multilingual results presentation.

4.2.2.1 Types of IE

There are five types of IE (or IE *tasks*) currently under R&D (as defined by the leading forum for this research, the Message Understanding Conferences [Grishman & Sundheim 96, SAIC 98]).

Named Entity recognition (NE)

Finds and classifies names, places, etc.

Coreference resolution (CO)
Identifies identity relations between entities in texts.

Template Element construction (TE)
Adds descriptive information to NE results (using CO).

Template Relation construction (TR)
Finds relations between TE entities.

Scenario Template production (ST)
Fits TE and TR results into specified event scenarios.

In simpler terms: NE is about finding entities; CO about which entities and references (such as pronouns) refer to the same thing; TE about what attributes entities have; TR about what relationships between entities there are; ST about events that the entities participate in. Consider these sentences:

> The shiny red rocket was fired on Tuesday. It is the brainchild of Dr. Big Head. Dr. Head is a staff scientist at We Build Rockets Inc.

NE discovers that the entities present are the *rocket*, *Tuesday*, *Dr. Head* and *We Build Rockets Inc.* CO discovers that *it* refers to the rocket. TE discovers that the rocket is *shiny red* and that it is Head's *brainchild*. TR discovers that *Dr. Head* works for *We Build Rockets Inc.* ST discovers that there was a rocket launching event in which the various entities were involved.

The various types of IE provide progressively higher-level information about texts. They are described in more detail below.

4.2.2.2 Performance Levels

Each of the five types of IE has been the subject of rigorous performance evaluation in MUC-7 (1998) [SAIC 98] and other MUCs, so it is possible to say quite precisely how well the current level of technology performs. Below we will quote percentage figures quantifying performance levels, which should be interpreted as a combined measure of precision and recall (or 'f-measure' [3]; see the section on evaluation in [SAIC 98]). Several caveats should be noted: most of the evaluation has been on English (with some Japanese, Chinese and

[3]The F-measure combines Precision and Recall into a uniformly weighted harmonic mean, in order to achieve an optimum balance between the two. Precision: $P = \frac{N_{correct}}{N_{correct}+N_{incorrect}}$. Recall: $R = \frac{N_{correct}}{N_{possible}}$. F-measure: $F = \frac{(\beta^2+1)PR}{(\beta^2 P)+R}$ [van Rijsbergen 79]. If recall and precision are of equal weight, $\beta = 1$.

Spanish); some applications of the technology may be easier or more difficult in different languages.

The performance of each IE task, and the ease with which it may be developed, is to varying degrees dependent on:

Text type: the kinds of texts we are working with, for example: Wall Street Journal articles, or email messages, or HTML documents from the Web, or the output of a speech recogniser.

Domain: the broad subject-matter of those texts, for example: financial news, or requests for technical support, or tourist information, and the style in which they are written, e.g. informal, formal.

Scenario: the particular event types that the IE user is interested in, for example: rocket launches, or mergers between companies, or problems experienced with a particular software package, or descriptions of how to locate parts of a city.

A particular IE application might be configured to process financial news articles from a particular news provider (written in the house style) and find information about mergers between companies and various other scenarios. The performance of the application would be predictable for only this conjunction of factors. If it was later required to extract facts from the love letters of Napoleon Bonaparte as published on wall posters in the 1871 Paris Commune, performance levels would no longer be predictable. Tailoring an IE system to new requirements is a task that varies in scale dependent on the degree of variation in the three factors listed above.

4.2.2.3 Training Data

Various approaches to automating the tailoring of IE systems are being researched, see e.g. [Cardie 97]. Many of these approaches involve supervised learning, where the results produced by humans for a particular task are collected in large quantities and used as inputs to machine learning algorithms [Mitchell 97]. The collection of large quantities of training data is a major factor in the success of this type of work (this also holds for the ASR work described in section 4.2.1).

4.2.2.4 Named Entity Recognition

The simplest and most reliable IE technology is *Named Entity recognition* (NE). NE systems identify all the names of people, places, organisations, dates, amounts of money, etc. So, for

example, if we run the Wall Street Journal text in figure 4.2 through an NE recogniser, the result is as in figure 4.3. (The viewers shown here and below are part of the GATE system – see chapter 7.) NE recognition can be performed at up to around 95% accuracy. Given that

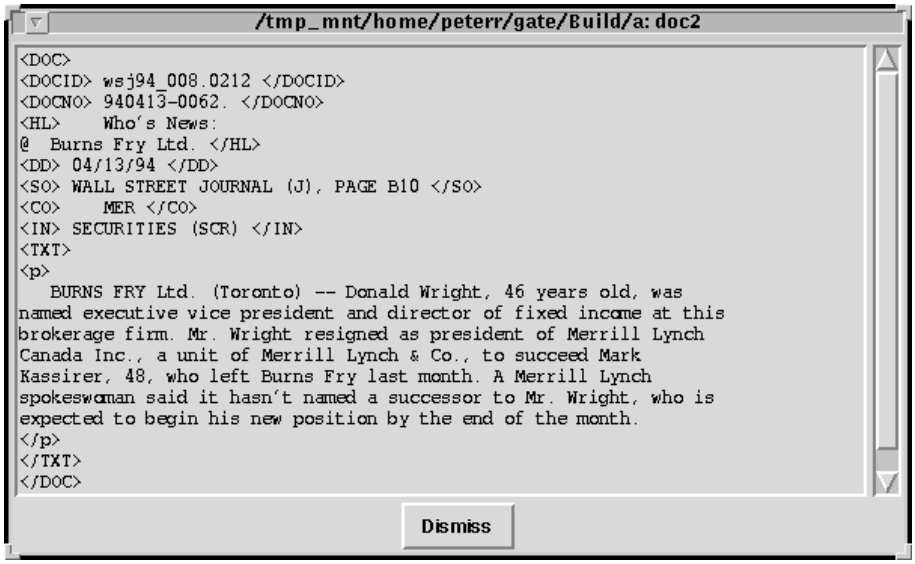

Figure 4.2: An example text

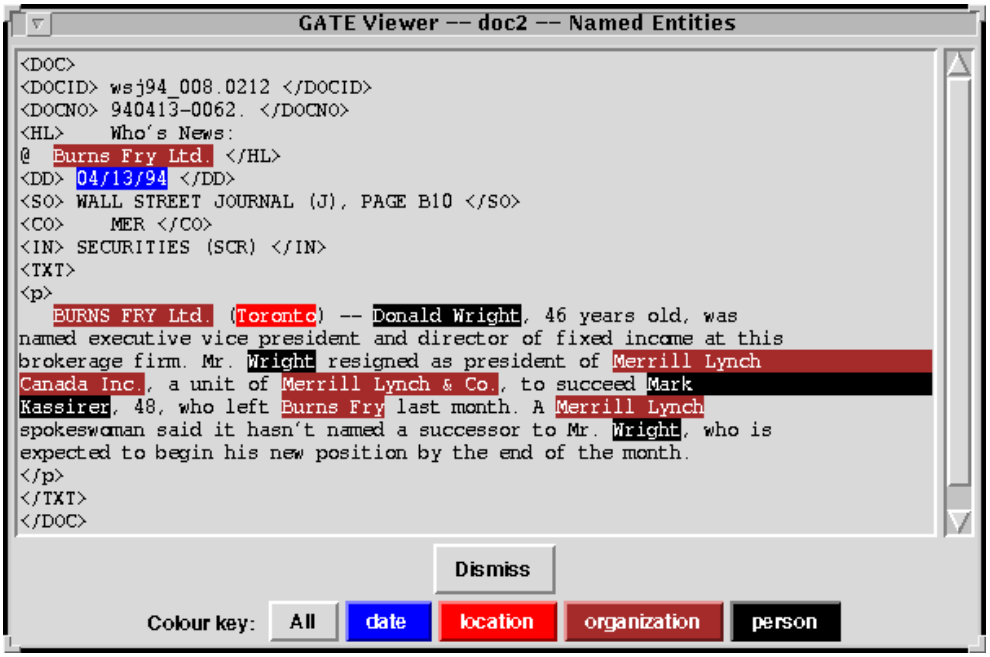

Figure 4.3: Named Entity recognition

human annotators do not perform to the 100% level (measured in MUC by inter-annotator comparisons), NE recognition can now be said to function at human performance levels, and applications of the technology are increasing rapidly as a result.

A recent evaluation of NE for Spanish, Japanese and Chinese ([Merchant *et al.* 96]) produced the following scores:

language	**best system**
Spanish	93.04 %
Japanese	92.12 %
Chinese	84.51 %

The process is weakly domain dependent, i.e. changing the subject-matter of the texts being processed from financial news to other types of news would involve some changes to the system, and changing from news to scientific papers would involve quite large changes.

[Bikel *et al.* 97, Bikel *et al.* 99] present an NE system based on HMMs (see section 4.2.1). [Vilain & Day 96, Day *et al.* 98] present a system based on finite state transduction (see appendix A). [Gaizauskas *et al.* 95, Humphreys *et al.* 98] present a system based on unification-based parsing.

4.2.2.5 Coreference Resolution

Coreference resolution (CO) involves identifying identity relations between entities in texts. These entities are both those identified by NE recognition and anaphoric references to those entities. For example, in

> Alas, poor Yorick, I knew him Horatio.

coreference resolution would tie 'Yorick' with 'him' (and 'I' with Hamlet, if that information was present in the surrounding text).

This process is less relevant to users than other IE tasks (i.e. whereas the other tasks produce output that is of obvious utility for the application user, this task is more relevant to the needs of the application developer). For text browsing purposes we might use CO to highlight all occurrences of the same object or provide hypertext links between them. CO technology might also be used to make links between documents, though this is not currently part of the MUC programme. The main significance of this task, however, is as a building block for TE and ST (see below). CO enables the association of descriptive information scattered across texts with the entities to which it refers. To continue the hackneyed Shakespeare example, coreference resolution might allow us to situate Yorick in Denmark. Figure 4.4 shows results for the example text from figures 4.2 and 4.3.

CO resolution is an imprecise process when applied to the solution of anaphoric reference. CO results vary widely; depending on domain perhaps only 50-60% may be relied upon. These scores are low (although problems with completing the task definition on schedule in MUC-6 complicated matters, and led to human scores of only around 80%), but note that this

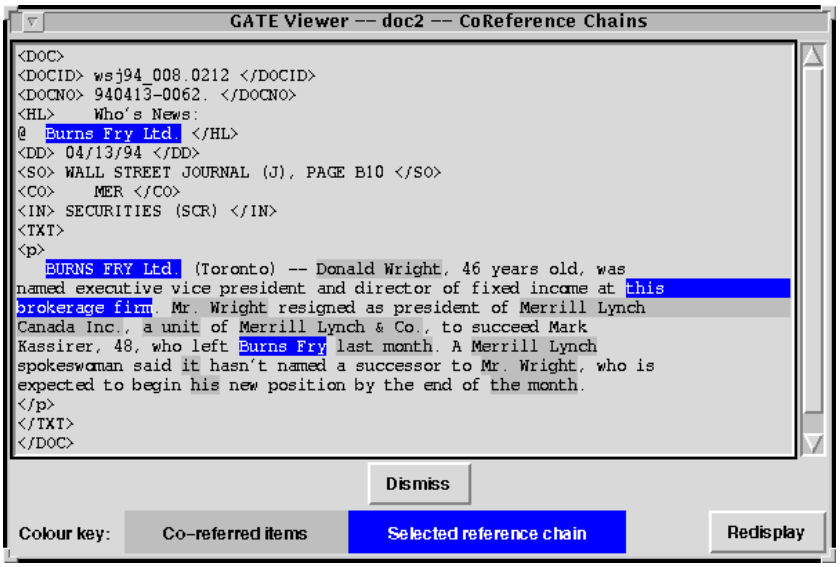

Figure 4.4: Coreference resolution

hides the difference between proper-noun coreference identification (same object, different spelling or compounding, e.g. 'IBM', 'IBM Europe', 'International Business Machines Ltd.', ...) and anaphora resolution, the former being a significantly easier problem.

CO systems are domain dependent.

4.2.2.6 Template Element Production

The TE task builds on NE recognition and coreference resolution. In addition to locating and typing (i.e. classifying, or assigning to a type – personal name, date, etc.) entities in documents, TE associates descriptive information with the entities. For example, from the figure 4.2 text the system finds out that Burns Fry Ltd. is located in Toronto, and it adds the information that this is in Canada.

Template elements for the figure 4.2 text are given in figure 4.5. The format is a somewhat arbitrary one developed at the behest of the American intelligence community (the original target user group of the MUC competitions). It is difficult to read; the main point to note is that it is essentially a database record, and could just as well be formatted for SQL store operations, or reading into a spreadsheet, or (with some extra processing) for multilingual presentation. See below for an extended example in a simplified format.

The best MUC-7 system scored around 80% for TE. Humans achieved 93%. MUC-6 was the first MUC to evaluate TE and ST tasks separately – TE scores may improve in future as developers gain more experience with the task.

As in NE recognition, the production of TEs is weakly domain dependent, i.e. changing

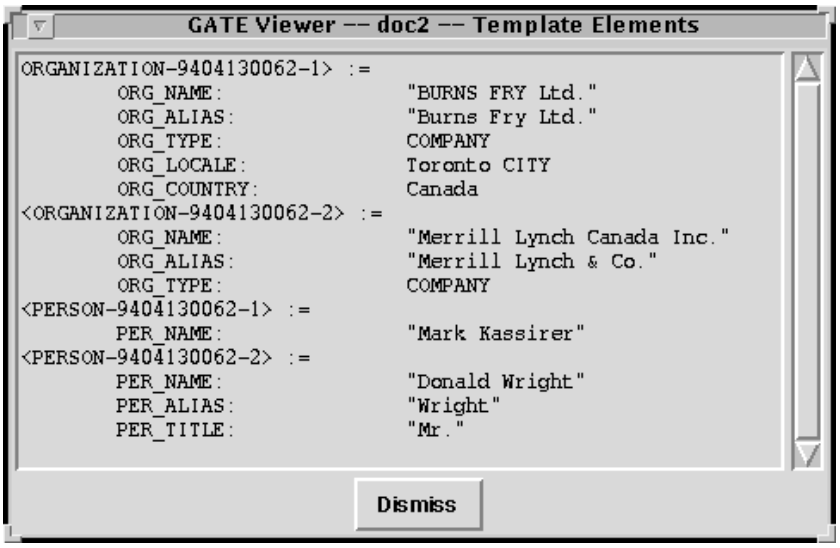

Figure 4.5: Template Elements

the subject-matter of the texts being processed from financial news to other types of news would involve some changes to the system, and changing from news to scientific papers would involve quite large changes.

4.2.2.7 Template Relation Production

Before MUC-7 relations between entities were part of the scenario-specific template outputs of IE evaluations. In order to capture more widely useful relations MUC-7 introduced the TR task.

> The template relation task requires the identification of a small number of possible relations between the template elements identified in the template element task. This might be, for example, an employee relationship between a person and a company, a family relationship between two persons, or a subsidiary relationship between two companies. Extraction of relations among entities is a central feature of almost any information extraction task, although the possibilities in real-world extraction tasks are endless. [Appelt 99]

In MUC-7 the best TR scores were around 75%. TR is a weakly domain dependent task.

4.2.2.8 Scenario Template Extraction

Scenario templates (STs) are the prototypical outputs of IE systems, being the original task for which the term was coined. They tie together TE entities and TR relations into event

descriptions. For example, TE may have identified Isabelle, Dominique and Françoise as people entities present in the Robert edition of Napoleon's love letters. ST might then identify facts such as that Isabelle moved to Paris in August 1802 from Lyon to be nearer to the little chap, that Dominique then burnt down Isabelle's apartment block and that Françoise ran off with one of Gerard Depardieu's ancestors. A slightly more pertinent example is given in figure 4.6. The same comments regarding format apply as for the TE task.

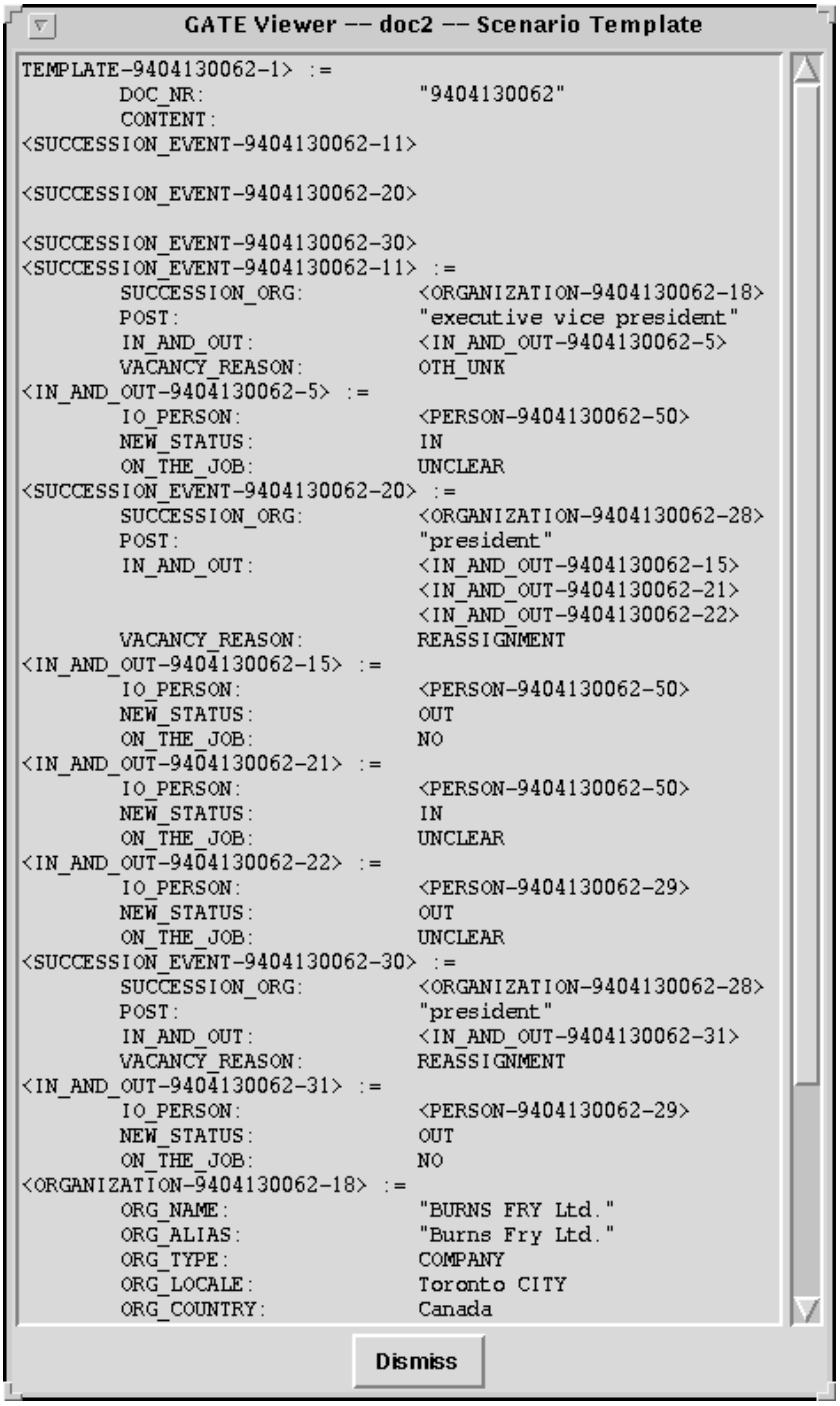

Figure 4.6: Scenario Template

ST is a difficult IE task; the best MUC systems score around 60%. The human score can be as low as around 80+%, which illustrates the complexity involved. These figures should be taken into account when considering appropriate applications of ST technology. Note, however, that it is possible to increase precision at the expense of recall: we can develop ST systems that do not make many mistakes, but that miss quite a lot of occurrences of relevant scenarios. Alternatively we can push up recall and miss less, but at the expense of making more mistakes.

The ST task is both domain dependent and, by definition, tied to the scenarios of interest to the users. Note however that the results of NE, TR and TE feed into ST. Note also that in MUC-6 and MUC-7 the developers were given the specifications for the ST task only 1 month before the systems were scored. This was because it was noted that an IE system that required very lengthy revision to cope with new scenarios was of less worth than one that could meet new specifications relatively rapidly. As a result of this, the scores for ST in MUC-6/7 were probably slightly lower than they might have been with a longer development period. Experience from previous MUCs suggests, however, that current technology has difficulty attaining scores much above 60% accuracy for this task.

4.2.2.9 An Extended Example

So far we have discussed IE from a general perspective. In this section we look at the capabilities that might be delivered as part of an application designed to support analysts tracking international drug dealing (loosely based on the AVENTINUS system [Thurmair 97, Kokkinakis 98]).

When the system is specified, our imaginary analysts state that "the operational domains that user interests are centred around are drug enforcement, money laundering, organised crime, terrorism, legislation". The entities of interest within these domains are cited as "person, company, bank, financial entity, transportation means, locality, place, organisation, time, telephone, narcotics, legislation, activity". A number of relations (or 'links') are also specified, for example between people, between people and companies, etc. These relations are not typed, i.e. the kind of relation involved is not specified. Some relations take the form of properties of entities – e.g. the location of a company – whilst others denote events – e.g. a person visiting a ship.

Working from this starting point an IE system is designed that:

1. is tailored to texts dealing with drug enforcement, money laundering, organised crime, terrorism, and legislation;

2. recognises entities in those texts and assigns them to one of a number of categories drawn from the set of entities of interest (person, company, ...);

3. associates certain types of descriptive information with these entities, e.g. the location of companies;

4. identifies a set (relatively small to begin with) of events of interest by tying entities and relations together into event descriptions.

For example, consider the following text:

> Reuters – New York, Wednesday 12 July 1996.
>
> New York police announced today the arrest of Frederick J. Thompson, head of Jay Street Imports Inc., on charges of drug smuggling. Thompson was taken from his Manhattan apartment in the early hours yesterday. His attorney, Robert Giuliani, issued a statement denying any involvement with narcotics on the part of his client. "No way did Fred ever have dealings with dope", Guliani[4] said.
>
> A Jay Street spokesperson said the company had ceased trading as of today. The company, a medium-sized import-export concern established in 1989, had been the main contractor in several collaborative transport ventures involving Latin-American produce. Several associates of the firm moved yesterday to distance themselves from the scandal, including the mid-western transportation company Downing-Jones.
>
> Thompson is understood to be accused of importing heroin into the United States.

From this IE might produce information such as the following (in some format to be determined according to user requirements, e.g. SQL statements addressing some database schema).

[4]The variation in spelling is intentional and illustrates that IE systems have to cope with mis-spellings.

First, a list of entities and associated descriptive information. Relations of property type are made explicit. Each entity has an id, e.g. `ENTITY-2`, which can be used for cross-referencing between entities and for describing events and relations involving entities. Each also has a type, or category, e.g. `company`, `person`. Additionally various type-specific information is available. For example, for dates, a `normalisation` slot gives the date in standard format.

```
Reuters
    id:             ENTITY-1
    type:           company
    business:       news

New York
    id:             ENTITY-2
    type:           location
    subtype:        city
    is_in:          US

Wednesday 12 July 1996
    id:             ENTITY-3
    type:           date
    normalisation:  12/07/1996

New York police
    id:             ENTITY-4
    type:           organisation
    location:       ENTITY-2

Frederick J. Thompson
    id:             ENTITY-5
    type:           person
    aliases:        Thompson; Fred
    domicile:       ENTITY-7
    profession:     managing director
    employer:       ENTITY-6

Jay Street Imports Inc.
    id:             ENTITY-6
    type:           organisation
    aliases:        Jay Street
    business:       import-export

Manhattan
    id:             ENTITY-7
    type:           location
    subtype:        city
    is_in:          ENTITY-2

Robert Giuliani
    id:             ENTITY-8
    type:           person
    aliases:        Guliani

1989
    id:             ENTITY-9
    type:           date
    normalisation:  ?/?/1989

Latin-America
    id:             ENTITY-10
    type:           location
    subtype:        country

Downing-Jones
    id:             ENTITY-11
    type:           organisation
    business:       transportation

heroin
    id:             ENTITY-12
    type:           drug
    class:          A

United States
    id:             ENTITY-13
    type:           location
    subtype:        country
```

(These results correspond to the combination of NE, TE and TR tasks; if we removed all but the type slots we would be left with the NE data.)

Second, relations of event type, or scenarios:

```
narcotics-smuggling
    id:             EVENT-1
    destination:    ENTITY-13
    source:         unknown
    perpetrators:   ENTITY-5, ENTITY-6
    status:         on-trial

joint-venture
    id:             EVENT-2
    type:           transport
    companies:      ENTITY-6, ENTITY-11
    status:         past
```

(These results correspond to the ST task.)

4.2.2.10 Multilingual IE

The results described above may be translated for presentation to the user or for storage in existing databases. In general this task is much easier than translation of ordinary text, and is akin to *software localisation*, the process of making a program's messages and labels on menus and buttons multilingual. Localisation involves storing lists of direct translations for known items. In our case these lists would store translations for words such as 'entity', 'location', 'date', 'heroin'. We also need ways to display dates and numbers in local formats, but code libraries are available for this type of problem.

Problems can arise where arbitrary pieces of text are used in the entity description structures, for example the descriptor slot in MUC-6 TE objects. Here a noun phrase from the text is extracted, with whatever qualifiers, relative clauses, etc. happen to be there, so the language is completely unrestricted and would need a full translation mechanism.

The process of constructing natural-sounding text from IE outputs (and other data sources) is known as Natural Language Generation – see section 4.2.3.

4.2.2.11 Summary

Unlike the more algorithmic approach taken to reviewing ASR, this discussion of IE has been structured around the tasks involved (NE, CO etc.). The examples given illustrate the types of result data structure that a SALE needs to represent and display in order to support this type of LE technology. The components that are typically used in IE systems are described in section 4.3.

4.2.3 Natural Language Generation

Natural Language Generation (NLG) "is concerned with the construction of computer systems that can produce understandable texts in English or other human languages from some underlying non-linguistic representation of information" [Reiter & Dale 99]. NLG is in increasing demand for applications such as:

- intelligent help and tutoring systems which are sensitive to user knowledge;
- production of technical documents in multiple languages from a single source;

- hypertext which adapts according to user goals, interests and prior knowledge, and to the presentation context.

NLG systems produce language output (ranging from a single sentence to an entire document) from data usually encoded in a knowledge base (KB) or a database. Often the input to a generator is a high-level communicative goal to be achieved by the system (which acts as a speaker or writer). Throughout the generation process, this high-level goal is refined into more concrete goals which eventually give rise to the generated utterance. Consequently, language generation can be regarded as a goal-driven process which aims at adequate communication with the reader/hearer, rather than as a process aimed entirely at the production of linguistically well-formed output.

The central problem of NLG for [McDonald 83] is "how specific utterances arise from specific communication goals in a specific discourse context". [Appelt 86] notes that "Human language behaviour is part of a coherent plan of action directed toward satisfying a speaker's goals".

The potential benefits of NLG technology include:

- Sensitivity to the current user and context: variable *information content* may be presented differently for different users.
- Low document maintenance costs: if changes are to be made, then they are performed once in the KB and are automatically propagated by the generator.
- Multilingual output.
- Guaranteed conformance to documentation standards.

However, these benefits come at the cost of building and maintaining various knowledge resources, particularly the declarative knowledge base, although some recent applications (e.g. software documentation from formal specifications) derive some or all of this knowledge from existing resources in a (semi-)automatic manner. In addition, recent advances in the field of IE [Pazienza 97] might allow the automatic construction of knowledge bases from text corpora.

The system construction costs may be further decreased with the development of re-usable components like the Penman surface generator [Mann 83] which has been integrated successfully in other NLG systems (e.g. [Hovy 91, Paris 91, Milosavljevic *et al.* 96]). Another

source of re-usability is grammar reversibility, i.e. grammars that are suitable both for understanding and generating natural language [Wilks 92].

NLG has been applied to various tasks including:

- automatic production of (multilingual) documents from a common knowledge source;
- flexible tutoring and advisory systems;
- intelligent user-sensitive explanations;
- generation of documents which automatically conform to certain style and language standards.

We can distinguish two types of NLG systems:

Static: systems that generate static texts (e.g. technical documentation [Paris & Linden 96]).

Interactive (dynamic): systems that generate explanations or dialogues in an interactive environment.

In the latter case the user may be able to formulate follow-up questions (e.g. [Moore 95]) to request additional information or clarification of a given explanation.

4.2.3.1 Generation Subtasks

Most existing systems divide the generation task into the following stages [Reiter 94], which are most often organised in a pipeline architecture:

Content determination; text planning: This stage involves decisions regarding the information which should be conveyed to the user (content determination) and the way this information should be rhetorically structured (text planning). Many systems perform these tasks simultaneously because rhetorical goals often influence what is relevant. Most text planners have hierarchically-organised plans and apply decomposition in a top-down fashion following AI planning techniques.

Sentence planning: Even though text planning specifies the text down to clausal units, it does not always address the problem of organising these clauses into sentences. In

general there are many possible ways in which the knowledge items could be combined into sentences ranging from one sentence per clause to a single complex sentence. The basic goal is to map conceptual structures onto linguistic ones, e.g. selection of a proper syntactic structure, referring expressions, and content words (lexical choice).

Surface realisation: Generate the individual sentences in a grammatically correct manner, e.g. agreement, morphology.

It should be noted that there is no agreement in the NLG community on the exact problems addressed in each one of these steps; they vary between different approaches and systems. A finer-grained breakdown is given in [Reiter & Dale 99] (content determination; discourse planning; sentence aggregation; lexicalisation; referring expression generation; linguistic realisation), though they also note that "From a pragmatic perspective the most common architecture in present-day applied NLG systems... use a three-stage pipeline" similar to the one outlined above.

4.2.3.2 Knowledge Sources

Language generators need various knowledge sources and language resources:

- **Domain knowledge:** an ontology or world model, usually divided into upper and domain models, where the former is domain-independent (e.g. the Penman Upper Model [Bateman 90]); factual knowledge from which the content of the generated utterance is selected.
- **Discourse history:** information about what has been presented so far. For instance, if a system maintains a list of previous explanations, then it can use this information to avoid repetitions, refer to already presented facts, or draw parallels.
- **User model:** a specification of the user's domain knowledge, plans, goals, beliefs, and interests.
- **Grammar:** a grammar of the target language which is used to generate linguistically correct utterances. Some of the grammars which have been used successfully in various NLG systems are:

  1. unification grammars, e.g.: Functional Unification Grammar [McKeown 85], Functional Unification Formalism [McKeown *et al.* 90];

2. phrase structure grammars, e.g.: Referent Grammar (GPSG with built-in referents) [Sigurd 91], Augmented Phrase Structure Grammar [Sowa 84];
3. Tree-Adjoining Grammar [Joshi 87, Nikolov *et al.* 95];
4. Systemic Grammar [Mann & Matthiessen 83];
5. Generalised Augmented Transition Network Grammar [Shapiro 82].

- **Lexicon:** a lexicon entry for each word containing typical information like part of speech, inflection class, etc.

The formalism used to represent the input semantics also affects the generator's algorithms and its output. For example, some surface realisation components expect a hierarchically structured input, while others use non-hierarchical representations. Examples of different input formalisms are:

- hierarchy of logical forms [McKeown *et al.* 90];
- SB-ONE (similar to KL-ONE) [Reithinger 91];
- functional representation [Sigurd 91];
- predicate calculus [McDonald 83];
- conceptual graphs [Nikolov *et al.* 95].

4.2.3.3 Summary

NLG transforms data into language; a SALE should be able to support this type of operation and able to store and display the types of intermediate data structure that are typical of the steps in the generation process (see also chapter 5 section 5.3.4 and the discussion of the RAGS project [Cahill *et al.* 99b]). Data sources such as knowledge bases, grammars and lexicons are also needed. [Reiter & Dale 99] also argue that corpora have a role in generation system design, to represent the types of target texts envisaged.

4.3 Components

The preceding sections have looked in some depth at various LE applications and technologies, and touched on the methods which their development entails. We turn now to focus on the elements that make up LE software. As a preliminary we discuss a common distinction between data and processing in LE. The function of this material is to inform the requirements analysis that appears in chapter 6, which will in turn inform the design and evaluation of GATE. In line with the development process discussed in chapter 3, we develop UML models of the domain objects under analysis here.

4.3.1 Language Resources and Processing Resources

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*. (Older development methods like Jackson Structured Design [Jackson 75] or Structured Analysis [Yourdon 89] kept them largely separate.) 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 choice is not quite so clear-cut. 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[5] 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. Improving the availability of LRs, particularly for non-indigenous and minority languages, is increasingly seen as an important task for the LE community [Somers 97, McEnery 97].

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

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

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' (see also chapter 2), 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.[6]

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 this thesis.

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 below and elsewhere in this thesis are themselves Object-Oriented.

4.3.2 A Taxonomy of LE Software Components

LE components may be classified according to a number of different dimensions:

- Are they primarily data or primarily algorithmic (LR or PR)?

[6] This point is due to Wim Peters.

- What methods do they employ? For example, are they based on neural networks or grammar rules?
- What tasks do they address? Do these tasks constitute applications or are they internal functions that are presumed useful to applications? For example, spell-checking vs. syntax analysis.
- Do they involve analysis of language, or generation, or a mixture of both? For example, morphological analysis vs. text planning.
- Do they involve spoken or written or typed (or even gestural) language?

These dimensions imply relations between entities. Some may involve part-of relations (a tokeniser is part of a translator), some may involve is-a relations (a parser is a language analyser). In addition a taxonomy must choose what level of granularity to adopt. Should we look at programs or internal program functions? Should we consider how to find a word in a dictionary, or how to identify each field of each entry? The choice of dimensions and of granularity will vary according to the purpose of the taxonomy under construction. In this case we are trying to identify common software components, our goal being to inform the construction of a SALE requirements model that is understandable, uncontroversial and useful (i.e. general and non-prescriptive). LE is a research field; creating a fully authoritative and conclusive taxonomy is impossible. Luckily our aim here is to describe the common and the typical, and to add to the foundations of our requirements model. Our choice of components will be influenced first-and-foremost by our own experiences of LE software, and that of the work reported in chapter 5. Since we are building an infrastructure which aims to be non-restrictive, the model must be amenable to extension and elaboration by client systems.

UML class diagrams are used below to express relationships of inheritance ('is-a'), composition ('has-a') and other forms of association such as 'uses'. (See figure 3.1 in chapter 3.) Note that in general in our diagrams 'has-a' or 'part of' is modelled as UML association (not UML composition or aggregation) – see [Fowler & Scott 97] p.80.

The component taxonomy itself appears in figures 4.7 – 4.9 which cover LRs, PRs

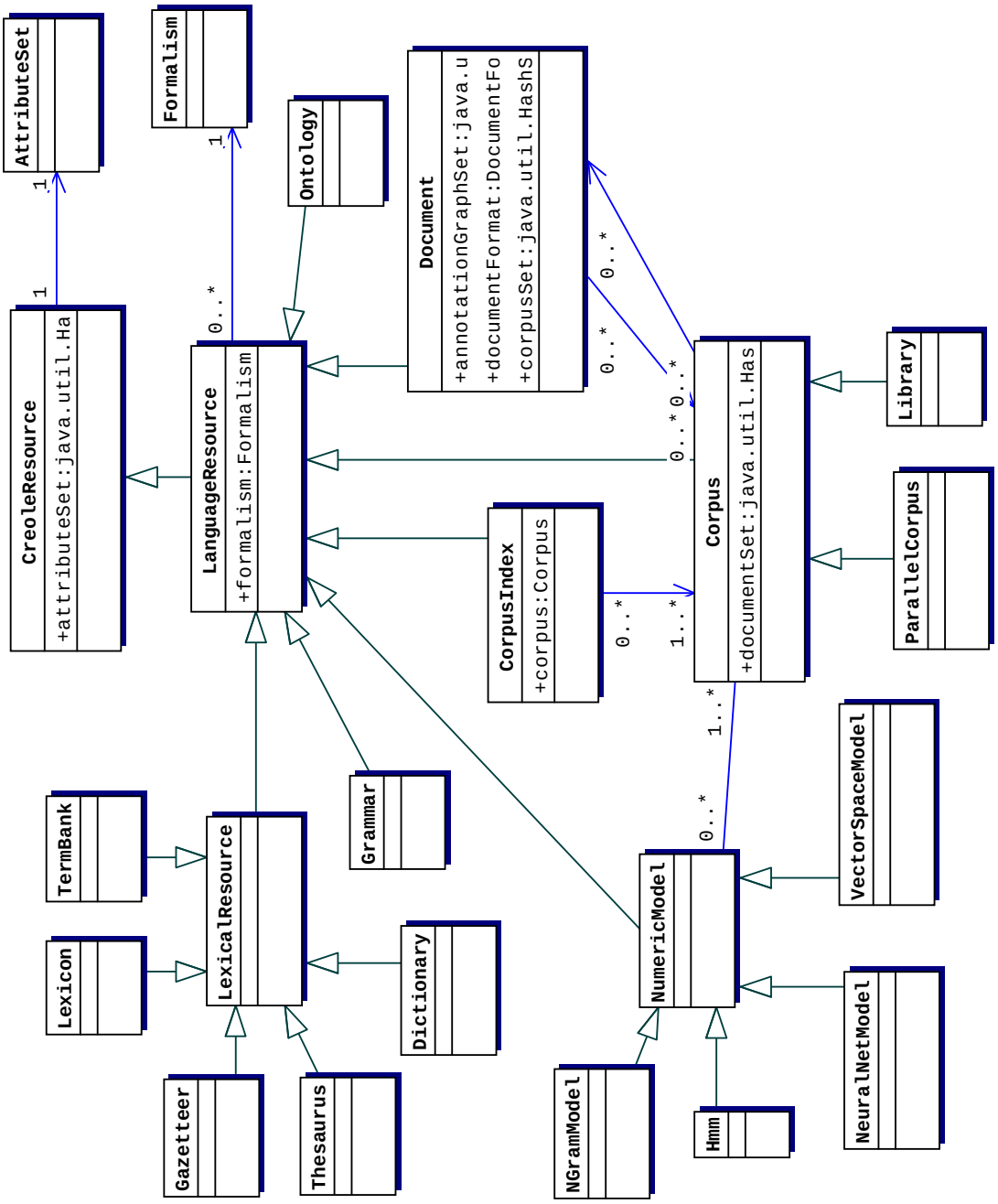

Figure 4.7: Taxonomy of LE software components: Language Resources

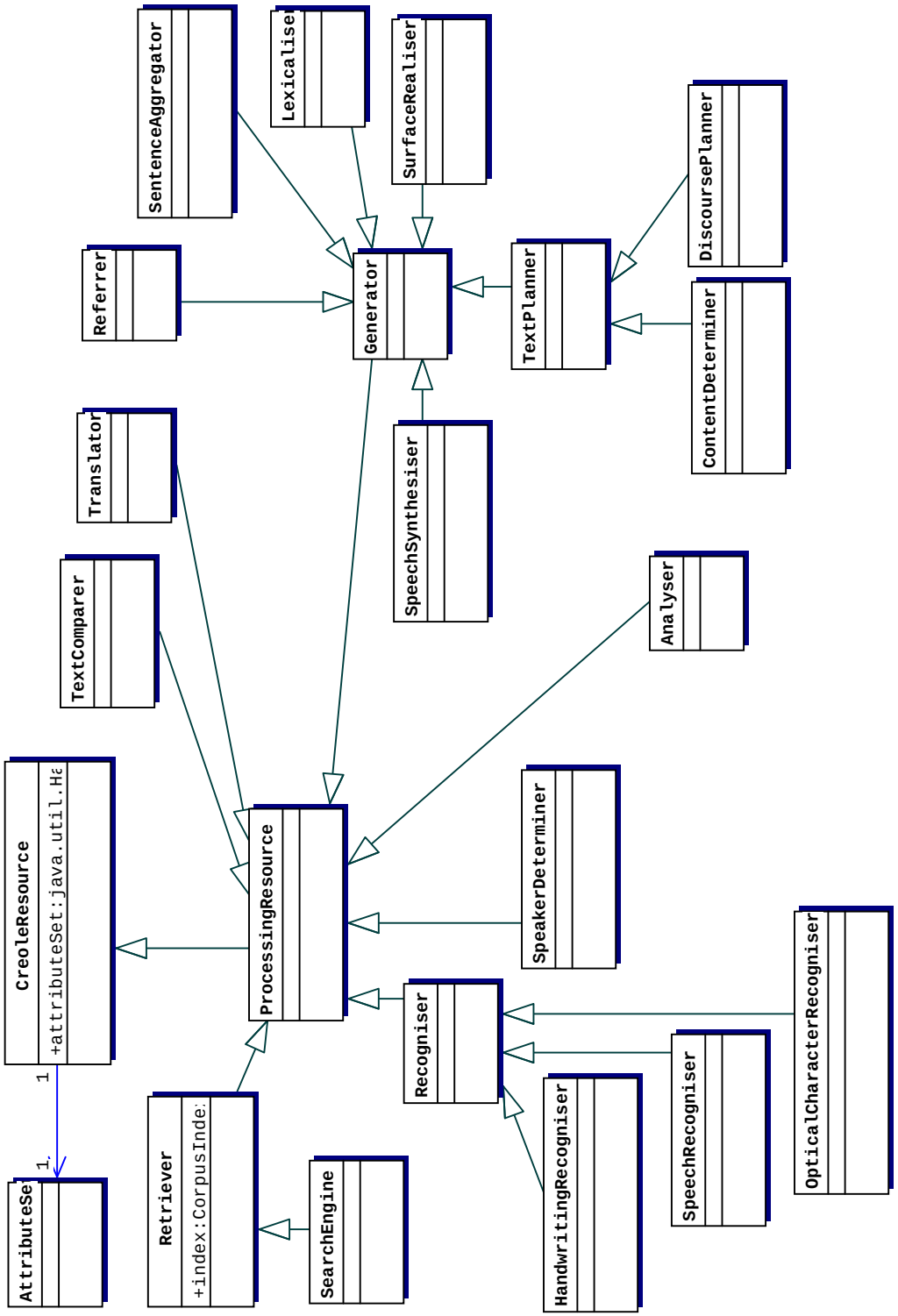

Figure 4.8: Taxonomy of LE software components: Processing Resources

(excluding analysers) and PR analysers respectively. (There are, of course, connections between the two.) The taxonomy diagrams are conceptual, concerned with types and their relationships (not classes and their implementations) – see [Fowler & Scott 97] p. 55.

The model is tied together at a single root, the `CreoleResource`[7] type, which encapsulates common attributes of LRs and PRs such as their origins, availability, etc. There are also many links between the LR and PR sides, but only a few examples are shown (e.g. a `LexicalAnalyser` PR uses a `LexicalResource` LR) partly in order to keep the diagrams comprehensible and partly because such links vary greatly.

In figure 4.7, various types of LR inherit from the `LanguageResource` class, including those representing corpora, documents, lexical resources (which in turn can be classified into lexicons, thesauri, etc.), and numerical models (which in turn has subtypes Markov model, ngram model, etc.).

In figure 4.8, various types of PR inherit from the `ProcessingResource` class, including language analysers, generators, retrievers (such as search engines) and recognisers (such as speech transcribers or handwriting recognisers). Language analysers come in many types, and are shown in figure 4.9.

Again it is worth stressing that there is nothing definitive about this model, and it is not comprehensive. The point is simply that there exist at least these types of components in LE software, and that they share certain properties (in ways that are shown here by inheritance and association). This knowledge will influence the use case analysis in chapter 6. We will show how the model is being used in the most recent version of GATE in chapter 9.

[7] CREOLE stands for Collection of REusable Objects for LE – see chapter 7.

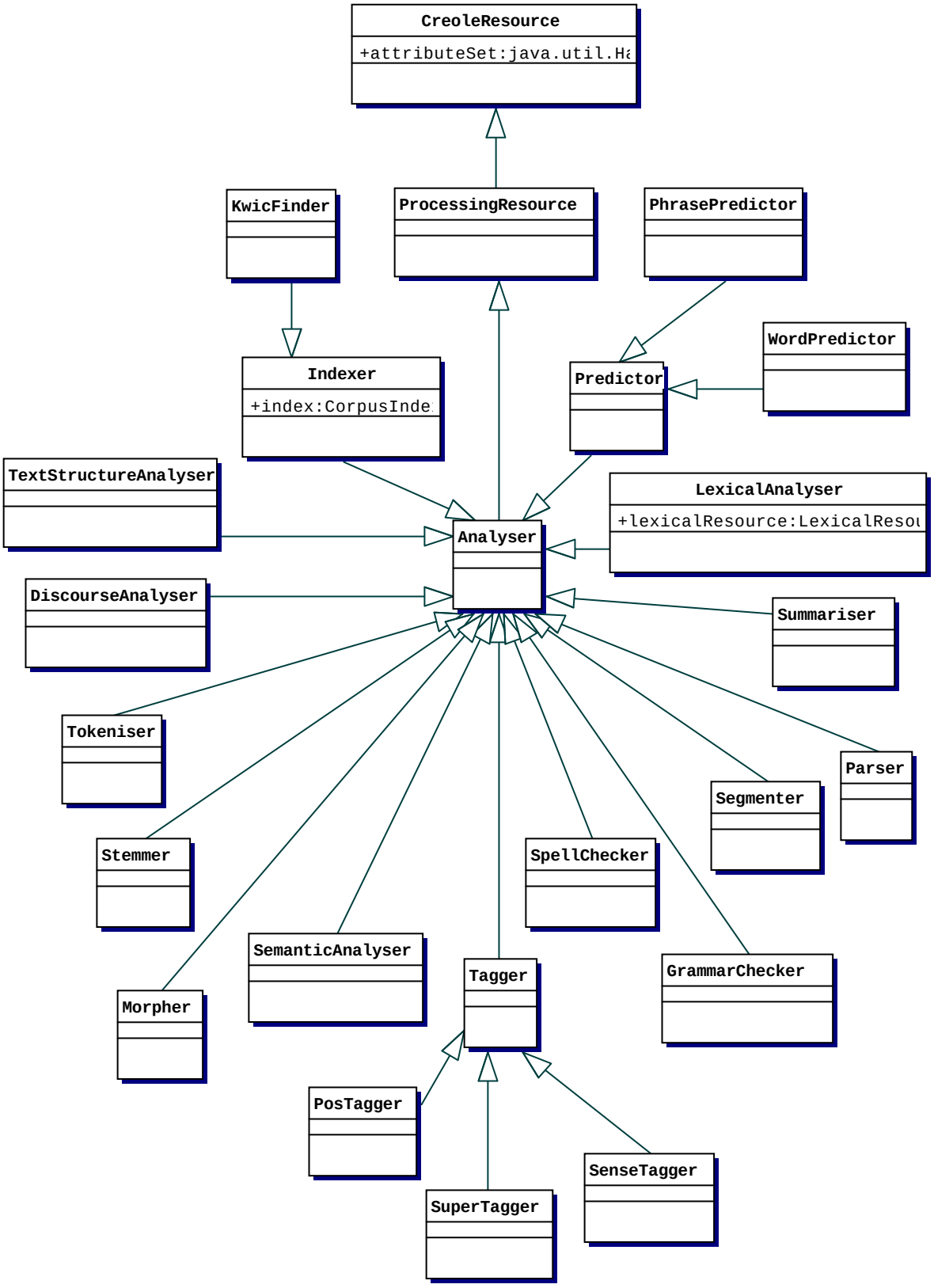

Figure 4.9: Taxonomy of LE software components: Processing Resources - Analysers

Chapter 5

Previous Work

This chapter gives a critical review of the various approaches that have been taken to SALE. The prime criterion for consideration in this part is being *infrastructural*, i.e. we consider work that is intended to support LE R&D in some way that extends beyond the boundaries of a single time-limited project.

We cover some sixteen categories of work, which ranges over a wide area. In order to provide an organising principle for the discussion we begin in section 5.1 by extrapolating a set of architectural issues that represents the union of those addressed by the various researchers cited. This has the advantage of being easier to transform into software design (see part III) and the disadvantage that multipurpose infrastructures appear in several categories.

Section 5.2 discusses infrastructures aimed at Processing Resources, including the issues of component integration and execution. Section 5.3 turns to Language Resource infrastructure, including the issues of access and the representation of information about text and speech. Section 5.4 deals with issues relating to LE methods, applications and development environments. Note that in some cases we refer to GATE itself, in order to place it in the context of other work in this field.

5.1 Categories of Work on SALE

This section categorises the work that is reviewed in following sections. The structure given here serves to organise the later sections and is also reflected in the requirements analysis

for GATE presented in the next chapter. It is informed by the previous chapter, and covers a similar set of LE entities and areas. Each category has been the subject of work on SALE, and each is explained in more detail below.

The breakdown of categories is as follows:

1. **Processing Resources**
   (a) Locating, loading and initialising components from local and non-local machines.
   (b) Executing processing components, serially or in parallel.
   (c) Representing information about components.
   (d) Factoring out commonalities amongst components.
2. **Language Resources, corpora and annotation**
   (a) Accessing data components.
   (b) Managing collections of documents (including recordings) and their formats.
   (c) Representing information about text and speech.
   (d) Representing information about language.
   (e) Indexing and retrieving information.
3. **Methods and applications**
   (a) **Method support**
      i. Finite state transduction, unification and statistics over information.
      ii. Comparing different versions of information.
   (b) **Application issues**
      i. Storing information.
      ii. Deployment and embedding.
   (c) **Development issues**
      i. Interoperation with other infrastructures.
      ii. Viewing and editing data components.
      iii. UI access to architectural facilities (development environments).

5.2 Processing Resources

It is commonly the case that a language processing system is composed of a number of discrete steps. For example, a translation application must first analyse the source text in order to arrive at some representation of meaning before it can begin deciding upon target language structures that parallel that meaning. A typical language analysis process will implement stages such as text structure analysis, tokenisation, morphological analysis, syntactic parsing,

and semantic analysis[1]. (The IE system whose results were used as examples in section 4.2.2 uses just such a pipeline [Gaizauskas *et al.* 95, Humphreys *et al.* 98].) Each of these stages is represented by components that perform processes on text (and use components containing data about language such as lexicons and grammars). In other words, the analysis steps are realised as a set of Processing Resources (PRs – see section 4.3). Several architectural questions arise in this context:

1. Is the execution of the PRs best done serially or in parallel?
2. How should PRs be represented such that their discovery on a network and loading into an executive process is transparent to the developer of their linguistic functions?
3. How should distribution across different machines be handled?
4. What information should be stored about components and how should this be represented?
5. How can commonalities between component sets be exploited?
6. How should the components communicate information between each other? (This question can also be stated as: how should information about text and speech be represented?)
7. How may the processing be embedded in client applications (e.g. a word processor or Web browser)?

This section reviews work that addresses questions 1 – 5. (The issue of representing information about language is addressed in section 5.3; those of embedding and deployment in section 5.4.) Section 5.2.1 looks at locating and loading components; section 5.2.2 looks at parallel and distributed execution; section 5.2.3 looks at associating metadata with components; section 5.2.4 looks at commonalities between components.

[1]The exact breakdown varies widely and is to some extend dependent on method; some statistical work early in the second wave of the application of these types of method completely ignored the conventional language analysis steps in favour of a technique based on a memory of parallel texts [Brown *et al.* 90]. Later work has tended to accept the advantages of some of these stages, however, though they may be moved into an off-line corpus annotation process such as [Marcus *et al.* 93].

5.2.1 Locating and Loading

There are several reasons why PR components should be separate from the controlling application that executes them:

- There will often be a many-to-one relation between applications and PRs. Any application using language analysis technology needs a tokeniser component, for example.
- A PR may have been developed for one computing platform, e.g. UNIX, but the application wishing to use it may operate on another, e.g. Windows.
- The processing regime of the application may require linear or asynchronous execution; this choice should be isolated from the component structures as far as possible in order to promote generality and encourage reuse.
- PR developers should not be forced to deal with application-level software engineering issues such as how to manage installation, distribution over networks, exception handling and so on.
- Explicit modelling of components allows exploitation of modern component infrastructures such as Java Beans or Active X.

Accordingly, many papers on infrastructural software for LE separate components from the control executive[2] [Boitet & Seligman 94, Edmondson & Iles 94a, Edmondson & Iles 94b, Koning *et al.* 95, Stefanini & Deamzeau 95, Görz *et al.* 96, Wolinski *et al.* 98, Poirier 99, Zajac 98b, Cahoon & McKinley 96]. The question then is how components become known to control processes or applications, and how they are loaded and initialised. (A related question is what data should be stored with components to facilitate their use by an executive – see section 5.2.3 below.) Much work ignores this issue; the rest of this section covers those SALE systems for which the data is available.

The TIPSTER architecture [Grishman 97] (see section 5.3.3.2) recognised the existence of the locating and loading problems, but didn't provide a full solution to the problem. The architecture document includes a place-holder for such a solution (in the form of a 'register annotator' API[3] call, which an implementation could use to provide component loading), but the semantics of the call were never specified.

[2]The term 'executive' is used here in the sense of a software entity which executes, or runs, other entities.
[3]Application Programmers' Interface.

The TalLab architecture "is embedded in the operating system" [Wolinski *et al.* 98] which allows them to "reuse directly a huge, efficient and reliable amount of code". The precise practicalities of this choice are unclear, but it seems that components are stored in particular types of directory structure, which are presumably known to the application at startup time.

ICE, the Intarc Communication Environment [Amtrup 95], is an 'environment for the development of distributed AI systems', and part of the Verbmobil real-time speech-to-speech translation project [Kay *et al.* 94]. ICE provides distribution based around PVM (Parallel Virtual Machine) and a communication layer based on channels. ICE is not specific to LE because the communication channels do not use data structures specific to NLP needs, and because document handling issues are left to the individual modules. ICE's answer to the locating and loading problem is the Intarc License Server, which is a kind of naming service, or registry, that stores addressing information for components. Components must themselves register with the server by making an API call (`Ice_Attach`). The components must therefore a) link to the ICE libraries and b) know the location of the license server (as must applications using ICE services).

Corelli [Zajac *et al.* 97, Zajac 97] and its successor Calypso [Zajac 98b] are also distributed systems that cater for asynchronous execution. The initial Corelli system implemented much of the CORBA standard [The Object Management Group 92], and component discovery used a naming and directory service. All communication and distribution was mediated by an Object Request Broker (ORB). Components ran as servers and implemented a small API to allow their use by an executive or application process. In the later Calyspo incarnation, CORBA was replaced with simpler mechanisms due to efficiency problems (for a usage example see [Amtrup 99]). In Calypso components are stored in a centralised repository, sidestepping the discovery problem. Loading is catered for by requiring components to implement a common interface.

GATE version 1 is a single-process, serial execution system. Components must reside in the same file system as the executive; location is performed by searching a path stored in an environment variable (see section 7.2.2 in chapter 7). Loading is performed in three ways, depending on the type of component and which of the GATE APIs it uses.

5.2.2 Execution

It seems unlikely that people process language by means of a set of linear steps involving morphology, syntax and so on. More probably we deploy our cognitive faculties in a parallel fashion (hence, for example, the term 'parallel distributed processing' in neural modelling work [McClelland & Rumelhart 86]). These kinds of ideas have motivated work on non-linear component execution in NLP; [von Hahn 94] gives an overview of a number of approaches; a significant early contribution was the Hearsay speech understanding system [Erman & Lesser 75, Erman *et al.* 80].

Examples of asynchronous infrastructural systems include Kasuga [Boitet & Seligman 94], Pantome [Edmondson & Iles 94a, Edmondson & Iles 94b], Talisman [Koning *et al.* 95, Stefanini & Deamzeau 95], Verbmobil [Görz *et al.* 96], TalLab [Wolinski *et al.* 98], Xelda [Poirier 99], Corelli [Zajac *et al.* 97, Zajac 97], Calyspo [Zajac 98b] and Distributed Inquery [Cahoon & McKinley 96]. Motivations include that mentioned in the previous paragraph, and also the desire for feedback loops in ambiguity resolution, for example:

> NLP raises the problem of ambiguities and therefore the multiple solutions derived. Architectures based on sequential levels, in which each module corresponds to a linguistic level (preprocessing, morphology, syntax, pragmatics, semantics) have shown their limits. A sequential architecture does not provide an adequate framework for exchanging necessary information between different modules in order to reduce ambiguities. [Koning *et al.* 95]

In the Inquery and Verbmobil cases an additional motivation is efficiency. ICE, the Verbmobil infrastructure, addressed two problems: distributed processing and incremental interpretation. Distribution is intended to contribute to processing speed in what is a very compute-intensive application area (speech-to-speech translation). Incremental interpretation is both for speed and to facilitate feedback of results from downstream modules to upstream ones (e.g. to inform the selection of word interpretations from phone lattices using part-of-speech information). ICE's PVM-based architecture provides for distributed asynchronous execution.

Most of the systems cited above are concerned with an experimental language processing system that involves non-linear execution. ICE, Xelda and Calypso go further and develop infrastructural software to enable the building of non-linear component-based systems.

5.2.3 Metadata

A distinction may be made between the data that language processing components use (or Language Resources) and data that is associated with components for descriptive and other reasons. The latter is sometimes referred to as *metadata* to differentiate it from the former. In a similar fashion, the content of the Web is largely expressed in HTML; data that *describes* Web resources, e.g. 'this HTML page is a library catalogue', is also called metadata. W3C, the Web consortium, has produced a standard for Web metadata called RDF, the Resource Description Framework [Lassila & Swick 99, Berners-Lee *et al.* 99].

There are several reasons why metadata should be part of a component infrastructure, including:

- to facilitate the interfacing and configuration of components;
- to encode version, author and availability data;
- to encode purpose data and allow browsing of large component sets.

When components are reused across more than one application or research project it will often be the case that their input/output (I/O) characteristics have not been designed alongside the other components forming the language processing capability of the application. For example, one part-of-speech tagger may require tokens as input in a one-per-line encoding. Another may require SGML[4] input. In order to reuse the tagger with a tokeniser that produces some different flavour of output, that output must be transformed to suit the tagger's expectations. In cases where there is an isomorphism between the available output and the required input a straightforward syntactic mapping of representations will be possible. In cases where there is a semantic mismatch, additional processing will be necessary.

[Busemann 99] addresses component interfacing and describes a method for using feature structure matrices to encode structural transformations on component I/O data structures. These transformations are essentially re-ordering of the data structures around pre-existing unit boundaries; therefore the technique assumes isomorphism between the representations concerned. The technique also allows for type checking of the output data during restructuring.

[4]The Standard Genralised Markup Language [Goldfarb 90].

TIPSTER [Grishman 97], GATE and Calypso [Zajac 98b] deal with interfacing in two ways. First, component interfaces share a common data structure (corpora of annotated documents), thus ensuring that the syntactic properties of the interface are compatible. Component wrappers are used to interface to other representations as necessary (so, for example, the Brill tagger [Brill 92a] wrapper writes out token annotations in the required one-per-line format, then reads in the tags and writes them back to the document as annotations). Second, where there is semantic incompatibility between the output of one component and the input of another a dedicated transduction component can be written to act as an intermediary between the two.

In Verbmobil a component interface language is used [Bos *et al.* 98], which constrains the I/O profiles of the various modules. This language is a Prolog term which encodes logical semantic information in a flat list structure. The principle is similar to that used in TIPSTER-based systems, but the applicability is somewhat restricted by the specific nature of the data structure.

Provision of descriptive metadata has been addressed by the Natural Language Software Registry (NLSR) [DFKI 99] and by the EUDICO distributed corpora project [Brugman *et al.* 98a, Brugman *et al.* 98b]. In each case Web-compatible data (HTML and XML respectively) are associated with components. The NLSR is purely a browsable description; the EUDICO work links the metadata with the resources themselves, allowing the launching of appropriate tools to examine them with[5].

In addition to the issue of I/O transformation, in certain cases it may be desirable to be able to identify automatically which components are plug-compatible with which other ones, in order to identify possible execution paths through the component set.

GATE 1 addresses automatic identification of execution paths by associating a configuration file with each processing component that details the input (pre-conditions) and output (post-conditions) in terms of TIPSTER annotation and attribute types (see section 5.3.3.2 below). This information is then used to auto-generate an execution graph for the component set (see section 7.2.3 in chapter 7).

[5]Note that EUDICO has only dealt with Language Resource components at the time of writing.

5.2.4 Commonalities

To conclude our survey of infrastructural work related to processing, this section looks at the exploitation of commonalities between components. As we saw in section 4.3, LE components can be modelled as a graph of entities that inherit characteristics from each other. So, for example, both parsers and taggers have the characteristics of language analysers. One of the key motivating factors for SALE is to break the 'software waste cycle' [Veronis & Ide 96] and promote reuse of components. Various researchers have approached this issue by identifying typical component sets for particular tasks [Hobbs 93, TIPSTER 95, Reiter 94, Reiter 99, Reiter & Dale 99]. Some work goes on to provide implementations of common components [Cheong *et al.* 94, Ibrahim & Cummins 89]. The rest of this section describes these approaches.

Reiter and Dale have reviewed and categorised Natural Language Generation (NLG) components and systems in some detail. [Reiter 94] argues that a consensus component breakdown has emerged in NLG (and that there is some psychological plausibility for this architecture); the classification is extended in [Reiter & Dale 99, Reiter & Dale 00]. (The discussion in section 4.2.3 is based on this classification.) They also discuss common data structures in NLG (as does the RAGS project – see section 5.3.4 below), and appropriate methods for the design and development of NLG systems. [Reiter 99] argues that the usefulness of this kind of architectural description is to 'make it easier to describe functionalities and data structures' and thus facilitate research by creating a common vocabulary amongst researchers. He states that this is a more limited but more realistic goal than supporting integration of diverse NLG components in an actual software system. The term he uses for this kind of descriptive work is a 'reference architecture', which is also the subject of the workshop at which the paper was presented [Mellish & Scott 99].

The TIPSTER research programme developed descriptive or reference architectures for Information Extraction and for Information Retrieval. [Hobbs 93] describes a typical module set for an IE system. The architecture comprises [Gaizauskas & Wilks 98]:

1. Text Zoner. Divides the input text into a set of segments.

2. Preprocessor. Converts a text segment into a sequence of sentences, where each sentence is a sequence of lexical items, with associated lexical attributes (e.g. part-of-speech).

3. Filter. Eliminates some of the sentences from the previous stage by filtering out irrelevant ones.

4. Preparser. Detects reliable small-scale structures in sequences of lexical items (e.g. noun groups, verb groups, appositions).

5. Parser. Analyses a sequence of lexical items and small-scale structures and attempts to produce a set of parse tree fragments, possibly complete, which describes the structure of the sentence.

6. Fragment Combiner. Turns a set of parse tree or logical form fragments into a parse tree or logical form for the whole sentence.

7. Semantic Interpreter. Generates a semantic structure or meaning representation or logical form from a parse tree or parse tree fragments.

8. Lexical Disambiguation. Disambiguates any ambiguous predicates in the logical form.

9. Coreference resolution or discourse processing. Builds a connected representation of the text by linking different descriptions of the same entity in different parts of the text.

10. Template generator. Generates final templates from the semantic representation of the text.

For IR [TIPSTER 95] describes two functions, search and routeing, each with a typical component set (some of which are PRs and some LRs.)

An architecture for spoken dialogue systems is presented in [LuperFoy *et al.* 98], which divides the task into dialogue management, context tracking and pragmatic adaptation. This in turn leads to an architecture in which various components (realised as agents) collaborate in the dialogue:

1. Speech recognition: from waveform to word string.

2. Natural language interpretation: from word string to semantic representation of utterances.

3. Context tracking on input, which tracks discourse entities of input utterances to resolve dependent references.

4. Pragmatic adaptation on input, which converts logical form to internal command forms.
5. Back-end communication with the underlying application.
6. Pragmatic adaptation on output, which converts applications responses to semantic representation of communicative acts.
7. Context tracking on output.
8. Natural language generation.
9. Speech synthesis.

In addition a dialogue manager component provides high-level control and routeing of information between components.

The preceding discussion illustrates that there is considerable overlap between component sets developed for various purposes. A SALE that facilitated multipurpose components would cut down on the waste involved in continual reimplementation of similar components in different contexts.

In several cases, work on identifying component commonalities has led to the development of toolkits that aim to implement common tasks in a reusable manner. For example: TARO [Ibrahim & Cummins 89] is an OO syntactic analyser toolkit based on a specification language. A toolkit for building IE systems and exemplified in the MFE IE system is presented in [Cheong *et al.* 94].

5.3 Language Resources

Recall from section 4.3 that Language Resources are data components such as lexicons, corpora and language models. They are the raw materials of LE. This section covers five issues relating to infrastructure for LRs:

1. Computational access (local and non-local).
2. Managing document formats and document collections (corpora).
3. Representing information about corpora (language data, or performance modelling).

4. Representing information about language (data about language, or competence modelling).

5. Indexing and retrieval of language-related information.

Note also that the advantages of a component-based model presented (in relation to PRs) in section 5.2.1 also apply to LRs.

5.3.1 Programmatic Access

LRs are of worth only inasmuch as they contribute to the development and operation of PRs and the language processing research prototypes, experiments and applications that are built from them. A key issue in the use of LRs for language processing purposes is that of computational access. Suppose that a developer is writing a program to generate descriptions of museum catalogue items; they may have a requirement for synonyms, for example, in order to lessen repetition. A number of sources for synonyms are available, e.g. WordNet [Miller (Ed.) 90], or Roget's *Thesaurus*. In order to reuse these sources the developer needs to access the data in these LRs from their program.

Although the reuse of LRs has exceeded that of PRs [Cunningham *et al.* 94] in general, there are still two barriers to LR access and hence LR reuse:

I. each resource has its own representation syntax and corresponding programmatic access mode (e.g. SQL for Celex, C or Prolog for WordNet);

II. resources must generally be installed locally to be usable, and how this is done depends on what operating systems are available, what support software is required, etc., which varies from site to site.

A consequence of (I) is that although resources of the same type usually have some structure in common (for example, at one of the most general levels of description, lexicons are organised around words), this commonality cannot be exploited when it comes to using a new resource. In each case the user has to adapt to a new data structure; this adaptation is a significant overhead. Work which seeks to investigate or exploit commonalities between resources has first to build a layer of access routines on top of each resource. So, for example, if we wished to do task-based evaluation of lexicons, by measuring the relative performance

of an IE system with different instantiations of lexical resource, we would typically have to write code to translate several different resources into SQL or some other common format. Similarly, work such as [Jing & McKeown 98] on merging large scale lexical resources (including WordNet and Comlex) for NLG has to deal with this problem.

A consequence of (II) is that users may have to adjust their compute environments to suit resources tailored to particular platforms. Also, there is no way to 'try before you buy': no way to examine an LR for its suitability for one's needs before licensing it *in toto*. Correspondingly there is no way for a resource provider to give limited access to their products for advertising purposes, or gain revenue through piecemeal supply of sections of a resource.

There have been two principal responses to problem (I): standardisation and abstraction.

The standardisation solution seeks to impose uniformity by specifying formats and structures for LRs. So, for example, the EAGLES working groups have defined a number of standards for lexicons, corpora and so on [EAGLES 99].

While standardisation would undoubtedly solve the representation problem, there remains the question of existing LRs (and of competing standards). [Peters *et al.* 98, Cunningham *et al.* 98a] describe experiments with an abstraction approach based on a common Object-Oriented model for LRs that encapsulates the union of the linguistic information contained in a range of resources, and encompasses as many object hierarchies as there are resources. At the top of the resource hierarchies are very general abstractions; at the leaves are data items specific to individual resources. Programmatic access is available at all levels, allowing the developer to select an appropriate level of commonality for each application. Generalisations are made over different object types in the resources, and the object hierarchies are linked at whatever levels of description are appropriate. No single view of the data is imposed on the user, who may choose to stay with the 'original' representation of a particular resource, or to access a model of the commonalities between several resources, or a combination of both.

Problem (II), non-local access, has also attracted two types of response, which can be broadly categorised as: Web browsing; distributed databases.

A number of sites now provide querying facilities from HTML pages, including:

- the Linguistic Data Consortium at `http://www.ldc.upenn.edu/`;

- the British National Corpus server at `http://info.ox.ac.uk/bnc/`;
- the W3 Corpora site at `http://clwww.essex.ac.uk/w3c/`[6].

So, for example, all occurrences of a particular word in a particular corpus may be found via a Web browser. This is a convenient way to access LRs for manual investigative purposes, but is not suited to (or intended for) use by programs for their access purposes.

Moving beyond browsing, several papers report work on programmatic access using distributed databases. [Fikes & Farquhar 99] show how ontologies may be distributed; [Brugman *et al.* 98a, Brugman *et al.* 98b] describe the EUDICO distributed corpus access system; [Peters *et al.* 98, Cunningham *et al.* 98a] propose a system similar to EUDICO, generalised to other types of LR.

Other issues in the area of access to LRs include that of efficient indexing and search of corpora (see section 5.3.5), and that of annotation of corpora (see section 5.3.3). The issue of how to access SGML documents in an efficient manner is discussed in [Olson & Lee 97], who investigated the use of Object-Oriented databases for storing and retrieving SGML documents. Their conclusions were essentially negative due to the slowness of the databases used. [Hendler & Stoffel 99] discuss how ontologies may be stored and processed efficiently using relational databases, and here the results are more positive.

5.3.2 Documents, Formats and Corpora

Documents play a central role in LE. They are the subject of analysis for technologies such as IE; they are both analysed and generated in technologies such as MT. In addition a large amount of work uses annotated documents as training data for machine learning of numerical models. Previous work on LE infrastructure has developed models for documents and corpora, provided abstraction layers for document formats, and investigated efficient storage of documents in particular formats.

Documents may contain text, audio, video or a mixture of these (the latter are referred to as multimedia documents). The underlying data will frequently be accompanied by formating information (delineating titles, paragraphs, areas of bold text, etc.) and, in the LE context, annotation (storing linguistic data such as gesture tags, part-of-speech tags or syntax

[6]Not to be confused with W3C, the World-Wide Web consortium at `http://www.w3c.org`.

trees). Both formatting and annotation come in a wide variety of flavours. Formats include proprietary binary data such as MS Word's `.doc` or Excel's `.xls`, semi-open semi-readable formats such as Rich Text Format (Word's exchange format), and non-proprietary standardised formats such as HTML, GIF (Graphics Interchange Format) or WAV (Windows Audio format).

The Text Encoding Initiative (TEI) [Sperberg-McQueen & Burnard 94] and the Corpus Encoding Standard (CES) [Ide 98a, Ide 98b, Ide & Priest-Dorman 99] are models of documents and corpora that aim to standardise the representation of structural and linguistic data for textual documents. The general approach is to represent *all* information about document structure, formatting and linguistic annotation using SGML, a markup (or annotation) language which is covered in section 5.3.3.

The issue of document formats has been addressed by several TIPSTER-based systems, including GATE and Calypso, and by the HTK speech recognition toolkit [Young *et al.* 99]. In the latter case the approach is to provide API calls that deal with documents in various known formats (e.g. WAV, MPEG) independent of those formats. So, for example, a speech recogniser can access the raw audio from these documents without knowing anything about the representation format.

The TIPSTER systems deal with formats by means of input filters that contain knowledge about the format encoding and use that knowledge to unpack format information into annotations. TIPSTER also supplies a model of corpora and data associated with both corpus and documents [Grishman 97]. Unlike the TEI/CES approach, document format is not constrained because of the use of reference annotations (see below). Note that the two approaches are not mutually exclusive: [Ogden 99] has defined a mapping between TEI/CES and TIPSTER annotations.

5.3.3 Annotation

One of the key issues for much work in this area is how to represent information about text and speech. (This kind of information is sometimes called *language data*, distinguishing it from *data about language* in the form of lexicons, grammars, etc.)

Two broad approaches to annotation have been taken: to embed markup; to use references

or pointers to the original[7].

5.3.3.1 Embedded Markup

Language data can be represented by embedding annotation in the document itself (at least in the case of text documents; users of embedding typically transcribe speech documents before markup, or use 'stand-off markup' – see below). The principal examples of embedded markup for language data use the Standard Generalized Markup Language (SGML [Goldfarb 90]). SGML is a *meta-language*, a language used to create other languages. The syntax of SGML is therefore abstract, with each document filling in this syntax to obtain a concrete syntax and a particular markup language for that document. In practice certain conventions are so widespread as to be *de facto* characteristics of SGML itself. For example, annotation is generally delimited by `<TAG>` and `</TAG>` pairs, often with some attributes associated such as `<TAG ATTRIBUTE=value>`. The legitimate tags (or *elements*) and their attributes and values must be defined for each class of document. An SGML document must have the following components:

- an SGML declaration referring to its *document type definition* (DTD);
- the DTD, which defines the elements and attributes which are legal in the document;
- an SGML *document instance*, the document text itself, marked up with appropriate tags.

The DTD is required because SGML specifies three things at an abstract level:

- what markup is allowed, and where;
- what markup is required;
- how markup is to be distinguished from other text (here the `<TAG>` convention is almost always adopted).

It does *not* specify what the markup means; the DTD is the grammar which defines how the elements may be legally combined and in what order in a particular class of text: 'the

[7]We will see, however, that even those working in the embedded markup field are increasingly using reference, i.e. there's a degree of convergence.

type definition and the marked up document together... constitute the rigorously defined document that machine processing requires' [Goldfarb 90].

The British National Corpus

The British National Corpus (BNC) [Burnard 95, Burnage & Dunlop 92] is a large collection of documents marked up in SGML, and is widely used in UK LE. This section gives some detail on the BNC as a good example of SGML in practical use for annotation. The documents are divided into spoken and written text and each type has specific markup. Each one is further divided into two parts, the first being a header which contains details of who created the text, how large it is and so on. The remainder of the document is delimited by tags `<stext>` and `</stext>` for transcripts of spoken texts, `<text>` and `</text>` for written texts. The whole document is enclosed between `<bncDoc id=x>` and `</bncDoc>`. The text is tagged by sentence, e.g. `<s n=001>`, with the opening tag for the next sentence (or the end of the document) automatically closing the previous sentence (this is referred to as mimimised markup). The same automatic closure applies to the `<w XXX>` and `<c YYY>` tags (for words and punctuation, respectively). Each word has part of speech as an attribute. Other tags include, for example, in the spoken texts, `<pause>` and `<unclear>`, and the speaker is also indicated, e.g. `<u who=DCJPS000>`.

The Web, HTML and XML

The HyperText Markup Language (HTML) is an application of SGML, and is specified by its own DTD. A difference with ordinary SGML is that the DTD is often cached with software such as Web browsers, rather than being a separate file associated with the documents that instantiate it. In practice Web browsers have been lenient in enforcing conformance to the HTML DTD, and this has led to diversity amongst Web pages, meaning that HTML DTDs now represent an idealised specification of the language which often differs from its usage in reality.

Partly in response to this problem the eXtensible Markup Language, XML [Goldfarb & Prescod 98] has been developed. SGML is a complex language: DTDs are difficult to write, and full SGML is difficult to parse. XML made the DTD optional, and disallowed certain features of SGML such as markup minimisation.

LT NSL and LT XML

One of the problems in the SGML/XML world is that of computational access to and manipulation of markup information. Addressing this problem, the Language Technology group at the University of Edinburgh developed an architecture and framework based on SGML called the LT Normalised SGML Library (LT NSL) [Thompson & McKelvie 96, McKelvie *et al.* 97, McKelvie *et al.* 98]. This in turn led to the development of LT XML [Brew *et al.* 99] tracking the introduction of the XML standard.

Tools in an LT NSL system communicate via interfaces specified as SGML DTDs (essentially tag set descriptions), using character streams on pipes: a pipe-and-filter arrangement modelled after UNIX-style shell programming. To avoid the need to deal with certain difficult types of SGML (e.g. minimised markup) texts are converted to a normal form before processing. A tool selects what information it requires from an input SGML stream and adds information as new SGML markup. LT XML is an extension of LT NSL to XML; in this case the normalisation step is unnecessary.

Advantages and Disadvantages of XML Annotation

The embedding approach to storing information about language cannot be applied directly to binary formats such as audio or video streams; when applied to speech these systems first make a transcription and then add markup to that. Standoff markup, where embedding is abandoned in favour of pointers (also known as hyperlinks in this context) that are stored in separate SGML files, may also be used to refer back to time-sync points in the audio, and this type of arrangement is becoming increasingly popular [Isard *et al.* 98]. This is a convergence point with reference annotation (see below), where pointers are also used.

A disadvantage is that although graph-structured data may be expressed in SGML, doing so is complex: "When SGML fanciers say 'structure,' they mean structure where everything is contained and sequential, with no overlap, no sharing of material in two places, no relations uncontained" [Nelson 97]. Graph-structured information might be present in the output of a parser, for example, representing competing analyses of areas of text.

An advantage here is a degree of data-structure independence: so long as the necessary information is present in its input, a tool can ignore other markup that inhabits the same stream; unknown markup is simply passed through unchanged (so, for example, a semantic interpretation module might examine phrase structure markup, but ignore the POS tags).

LT NSL and LT XML use a pipe/filter architecture, and as such: "LT NSL... is a good system for sequential corpus processing where there is locality of reference... LT NSL is not so good for: Applications... which need to access markup at random from a text..." [McKelvie *et al.* 97]. Applications that need to access markup at random include any application that has a GUI component where the user can navigate around a text at will. This is a typical disadvantage of pipe/filter architectures, which "often lead to a batch organization of processing... [and] are typically not good at handling interactive applications" [Shaw & Garlan 96].

Readability is an oft-cited advantage of SGML, but "Although SGML is human readable, in practice once the amount of markup is of the same order as the textual content, reading SGML becomes difficult." [McKelvie *et al.* 97]

All the markup passing through an LT XML processing pipeline potentially has to be parsed at each step and then regenerated (again a recognised problem with pipe/filter arrangements, which "may force a lowest common denominator on data transmission, resulting in added work for each filter to parse and unparse its data" [Shaw & Garlan 96]). Later versions of the architecture have moved to a partly compiled representation of the streams to reduce this problem. This also means, however, that the advantages that accrue to SGML (such as readability or standardisation) may no longer apply to the stream format.

As noted above, binary formats such as audio or MS Word necessitate a pointing or reference approach to annotation. Although [McKelvie *et al.* 97] state that the "only storage overhead of LT NSL is the generation of normalised SGML" there would also seem to be an overhead associated with the practice of using hyperlinking to split an SGML document into smaller and more manageable chunks or to apply markup to a binary document. In the example given in [McKelvie *et al.* 97], where paragraph and token markup is stored in one file and phrasal markup in another, 53 out of 99 non-whitespace characters in the phrasal markup deal with linking. This means that over 50% of this example is a redundant storage overhead.

Markup and Translation

An interesting problem in this area is that of preserving annotation and/or formatting data through translation. [Thurmair 96] describes a Text Handling Interface (based on SGML) and also reviews numerous MT and other representation formats.

5.3.3.2 Reference Annotation (I): TIPSTER

The ARPA-sponsored TIPSTER programme in the US, which finished in 1998, produced a data-driven architecture for NLP systems [Grishman 97]. A number of sites implemented the architecture; the initial prototype was written by Ted Dunning at the Computing Research Lab of New Mexico State University. In contrast to the embedding approach, in TIPSTER the text remains unchanged while information about it is stored in a separate database (DB). The database refers back to the text by means of offsets. The data is stored *by reference*.

Information is stored in the database in the form of *annotations*, which associate arbitrary information (*attributes*), with portions of documents (identified by sets of start/end character offsets or *spans*). Attributes will often be the result of linguistic analysis, e.g. POS tags. In this way information about texts is kept separate from the texts themselves. In place of an SGML DTD, an *annotation type declaration* defines the information present in annotation sets (though note that few implementations instantiated this part of the architecture). Figure 5.1 gives an example which "shows a single sentence and the result of three annotation procedures: tokenisation with part-of-speech assignment, name recognition, and sentence boundary recognition. Each token has a single attribute, its part of speech (pos), ...; each name also has a single attribute, indicating the type of name: person, company, etc." [Grishman 97].

Text

```
Kevin admired his bike.
0...|5...|10..|15..|20
```

Annotations

Id	Type	Span		Attributes
		Start	End	
1	token	0	5	pos=NP
2	token	6	13	pos=VBD
3	token	14	17	pos=PP
4	token	18	22	pos=NN
5	token	22	23	
6	name	0	5	name_type=person
7	sentence	0	23	

Figure 5.1: TIPSTER annotations example

Documents are grouped into *collections* (or corpora), each with an associated database storing annotations and document attributes such as identifiers, headlines, etc. The definition of

documents and annotations in TIPSTER forms part of an Object-Oriented model that can deal with inter- as well as intra-textual information by means of reference objects that can point at annotations, documents and collections. The model also describes elements of IE and IR systems relating to their use, providing classes representing queries and information needs.

Advantages and Disadvantages of TIPSTER Annotations

Texts may appear to be one-dimensional, consisting of a sequence of characters, but this view is incompatible with structures like tables, which are inherently two-dimensional. Their representation and manipulation is much easier in a referential model like TIPSTER than in an embedding one like SGML where markup is stored in a one-dimensional text string. In TIPSTER, a column of a table can be represented as a single object with multiple references to parts of the text (an *annotation* with multiple *spans*, or a document attribute with multiple references to annotations). Marking columns in SGML requires a tag for each row of the column, and manipulation of the structure as a whole necessitates traversal of all the tags and construction of some other, non-SGML data structure.

Distributed control has a relatively straightforward implementation path in a database-centred system like TIPSTER: the DB can act as a blackboard, and implementations can take advantage of well-understood access control technology.

On the issue of storage overhead and contrasting with the hyperlinking used in LT XML, we see that:

1. There is no need to break up a document into smaller chunks as the DBMS[8] in the document manager can deal efficiently with large data sets and visualisation tools can give intelligible views into this data.

2. To cross-refer between annotations is a matter of citing ID numbers (which are themselves indexes into DB records and can be used for efficient data access). It is also possible to have implicit links: simple API calls will find all the token annotations subsumed by a sentence annotation, for example, via their respective byte ranges without any need for additional cross-referencing information.

Another disadvantage of embedded markup compared with TIPSTER is that an SGML structure like `<w id=p4.w1>` has to be parsed in order to extract the fact that we have a

[8]DataBase Management System.

"w" tag whose "id" attribute is "p4.w1". A TIPSTER annotation is effectively a database record with separate fields for type (e.g. "w"), ID and other attributes, all of which may be indexed and none of which ever requires parsing.

There are three principal disadvantages of the TIPSTER approach.

1. Editing of texts requires offset recalculation.
2. TIPSTER specifies no interchange format, and TIPSTER data is weakly typed (there is no effective DTD mechanism; though this is also to an extent an advantage, as a complex typing scheme can inhibit unskilled users).
3. The reference classes can introduce brittleness in face of changing data: unless an application chases all references and updates them in face of changes in the objects they point to, the data can become inconsistent.

(The latter problem also applies to hyperlinking in embedded markup.)

5.3.3.3 Reference Annotation (II): LDC

The Linguistic Data Consortium (LDC) have proposed the use of Directed Acyclic Graphs (DAGs), or just Annotation Graphs (AGs), as a unified data structure for text and speech annotation [Bird & Liberman 98, Bird & Liberman 99a, Bird *et al.* 00]. [Bird & Liberman 99b] provides an example of using these graphs to to mark up discourse-level objects. This section compares the structure of TIPSTER annotations with the graph format.

TIPSTER annotations as discussed above are associated with documents and have:

- a type, which is a string;
- an ID, which is a string unique amongst annotations on the document;
- a set of spans that point into the text of the document;
- a set of attributes.

TIPSTER attributes, which are associated with annotations and with documents and collections of documents have:

- a name, which is a string;
- a value, which may be one of several data types including a string, a reference to annotation, document or collection, or a set of strings or references.

Some implementors of the architecture, including GATE and Corelli, have relaxed the type requirements on attribute values, allowing any object as a value. This has the advantage of flexibility and the disadvantage that viewing, editing and storage of annotations is made more complex.

TIPSTER explicitly models references between annotations with special reference classes. These rely on annotations, documents and collections of documents having unique identifiers.

LDC annotations are arcs in a graph whose nodes are time points (or, by extension, character offsets in a text). Each annotation has a type and a value, which are both atomic. A document may have a number of different graphs, and graphs can be associated with more than one document; this is not specified in the model.

There are no explicit references. References are handled implicitly by equivalence classes: if two annotations share the same type and value, they are considered coreferential. In order to refer to particular documents or other objects, an application or annotator must choose some convention for representing those references as strings, and use those as annotation values. This seems problematic: an annotation of type CoreferenceChain and value Chain23 should be equivalent to another of the same type and value, but this is not true for an annotation of type PartOfSpeech and value Noun.

LDC annotation values are atomic. Any representation of complex data structures must define its own reference structure to point into some other representation system.

TIPSTER is a richer formalism, both in terms of the complexity of the annotation/attribute part of the model and also because documents and collections of documents are an explicit part of the model, as are references between all these objects.

The LDC formalism is simple, mathematically specified and likely to be easy to manipulate by finite state algorithms, e.g. for purposes of search. There is a simple file encoding available, and work has been done on the possible shape of annotation indices.

TIPSTER is a software model of annotation. It provides an architecture that systems like GATE can easily implement and that then forms a backbone which is easy to program to and

easy to understand for system users. The LDC formalism is a mathematical specification of how annotations may be represented as DAGs. Because the properties of DAGs are well understood, deriving a software model of the formalism should be feasible. The problem would seem to be that there is still be a rather large hole left for the application developer. What should we do about documents, references, or representation of complex attribute values? From this perspective the LDC format would benefit from the inclusion of:

- Documents and collections of documents in the model, and the association of information with them in similar fashion to that on annotations.
- Annotation identifiers, which would allow for referencing without the equivalence class notion.
- A referencing convention within and between annotations, their graphs and the documents that contain them.
- An extensibility model that allows other classes of object to be added to the model and play a part in annotation as references values (for example lexicons and their entries).

The inherent problems with developing a model of a task to be solved in software in isolation from the development of instances of that software are evident in [Cassidy & Bird 00], where the properties of the LDC AG model when stored and indexed in a relational database are discussed. At this point the authors added ID fields to annotations, as suggested above.

The TIPSTER world might learn from LDC:

- To make annotation sets more explicitly graph-based. Implementations have been moving this way for some time, due to increased efficiency of traversal and simpler editing when offsets are moved from annotations themselves into a separate node object.
- To have multiple annotation sets on single documents. Consider the situation when two people are adding annotation to the same document, and later wish to compare and merge their results. TIPSTER would handle this by having an "annotator" attribute on all the annotations. It would be simpler to have disjoint sets.

In addition, both LDC and TIPSTER need an annotation meta-language, to describe for purposes of validation or configuration of viewing and editing tools the structure and permissible value set of annotations.

We return to these points in chapter 9.

5.3.4 Data About Language

Preceding sections described language data – information related directly to examples of human performance of language. This section considers work on data about language, or description of human language competence. Much work in this area has concentrated on formalisms for the representation of the data, and has advocated declarative, constraint-based representations (using feature-structure matrices manipulated under unification) as an appropriate vehicle with which "many technical problems in language description and computer manipulation of language can be solved" [Shieber 92]. One example of an infrastructure project based on Attribute-Value Matrices (AVMs) is ALEP – the Advanced Language Engineering Platform. ALEP aims to provide "the NLP research and engineering community in Europe with an open, versatile, and general-purpose development environment" [Simkins 92, Simkins 94, Eriksson 96]. ALEP, while in principle open, is primarily an advanced system for developing and manipulating feature structure knowledge-bases under unification. Also provided are several parsing algorithms, algorithms for transfer, synthesis and generation [Schütz 94]. As such, it is a system for developing particular types of LR (e.g. grammars, lexicons) and for *doing* a particular set of tasks in LE in a particular way.

The system, despite claiming to use a theory-neutral formalism (in fact an HPSG-like formalism), is still committed to a particular approach to linguistic analysis and representation. It is clearly of utility to those in the LE community using that class of theories and to whom those formalisms are relevant; but it excludes, or at least does not actively support, those who are not, including an increasing number of researchers committed to statistical and corpus-based approaches.

Other systems that use AVMs include: [Zajac 92], a framework for defining NLP systems based on AVMs; the Eurotra architecture [Schütz *et al.* 91]: an 'open and modular' architecture for MT promoting resource reuse; the DATR morphological lexicon formalism [Evans & Gazdar 96]; the Shiraz MT Architecture [Amtrup 99], a chart and unification-based architecture for MT; [Zajac 98a], a unified FST/AVM formalism for morphological lexicons; the RAGS architecture (see below).

A related issue is that of grammar development in an LE context – see [Netter & Pianesi 97, Estival *et al.* 97]. [Fischer *et al.* 96] presents an abstract model of Thesauri and terminology

maintenance in an OO framework. ARIES [Goni *et al.* 97] is a formalism and development tool for Spanish morphological lexicons.

The RAGS (Reference Architecture for Generation Systems) project [Cahill *et al.* 99b, Paiva 98, Cahill & Reape 98, Cahill *et al.* 99a] has concentrated on describing structures that may be shared amongst NLG component interfaces. This choice is motivated by the fact that the input to a generator is not a document but a meaning representation. RAGS describes component I/O using a nested feature matrix representation, but doesn't describe the types of LR that an NLG system may use, or the way in which components may be represented, loaded, etc.

5.3.5 Indexing and Retrieval

Modern corpora, and annotations upon them, frequently run to many millions of tokens. To enable efficient access to this data the tokens and annotation structures must be indexed. In the case of raw corpora, this problem equates to Information Retrieval (IR; aka document detection), a field with a relatively well-understood set of techniques based on treating documents as bags of stemmed words and retrieving based on relative frequency of these terms in documents and corpora – see for example [van Rijsbergen 79]. Although these processes are well understood and relatively static, nevertheless IR is an active research field, partly because existing methods are imperfect and partly because that imperfection becomes more and more troubling in face of the explosion of the Web. There have been a number of attempts to provide SALE systems in this context.

As noted above, the TIPSTER programme developed a reference model of typical IR component sets [TIPSTER 95]. More concretely, this programme also developed a communication protocol based on Z39.50 for detection interactions between querying application and search engine [Buckley 98]. The annotation and attribute data structures described in section 5.3.3.2 were also applied for IR purposes, though the practical applications of the architecture were found in general to be too slow for the large data sets involved.

[Cutting *et al.* 91] developed an OO architecture for IR systems that was used by at least one site other than the originator; however this site reported[9] that the developers had lost interest in the idea after the initial version. The InfoGrid system [Rao *et al.* 92] developed a UI and interaction model framework for IR systems, but again take-up was limited. More

[9] At the final TIPSTER meeting, Baltimore, September 1998.

recently FireWorks [Hendry & Harper 96] also developed a UI framework for building IR systems; P-OQL [Henrich 96] describes IR extensions for the PCTE repository interface standard.

Whereas the problem of indexing and retrieving documents is well understood, the problem of indexing complex structures in annotations is more of an open question. The Corpus Query System [Christ 94, Christ 95] is the most cited source in this area, providing indexing and search of corpora and later of WordNet. Similar ideas have been implemented in CUE [Mason 98] for indexing and search of annotated corpora, and at the W3-Corpora site [University of Essex 99] for searchable on-line annotated corpora. Some work on indexing in the LT XML system is reported in [McKelvie & Mikheev 98].

5.4 Methods and Applications

5.4.1 Method Support

A number of infrastructural toolkits exist that are explicitly targeted on particular LE methods. The CMU-Cambridge Statistical Modelling toolkit [Clarkson & Rosenfeld 97] is a set of command-line tools for building ngram language models.

HTK [Young *et al.* 99] is a large and comprehensive system for building speech recognisers, and gives extensive support to Hidden Markov Modelling.

The TIPSTER programme defined the Common Pattern Specification Language, which provides finite state transduction over TIPSTER annotations based on regular expressions. The original implementation is Doug Appelt's TextPro system: `http://www.ai.sri.com/~appelt/TextPro`. A version of CPSL has been adopted in GATE version 2, called JAPE [Cunningham 99c], a Java Annotation Patterns Engine. JAPE is described in detail in appendix A.

5.4.2 Application Issues

Application level issues are those that occur when LE technology and LE components are combined to form practical end-user systems, or are used as part of such systems. How

to deploy applications, and how to embed components, has been the focus of considerable effort in the commercial software world in recent years. Deployment raises questions of code portability; interpreted and bytecode languages such as Perl and Java are intended to be mobile across diverse computing platforms, but problems remain relating to installation of interpreters and virtual machines, their various versions and so on. Commercial systems such as InstallShield support dynamic configuration of applications during deployment. In the free software world GNU `autoconf` plays the same role.

The issue of embedding is more complex. Significant approaches at present include: COM/OLE; CORBA; Java Beans. COM, a Common Object Model, and OLE, an Object Linking and Embedding standard, are proprietary Microsoft mechanisms that allow applications to share document components. For example, a spreadsheet might embed a portion of a word processing document. Also available are means to customise menu and command functionality in Office applications. CORBA, the Common Object Request Broker Architecture [The Object Management Group 92], is a heavyweight approach in which all inter-application communication takes place via a mediator (ORB) which contains methods for data conversion between languages. So, for example, a C++ application could send an object to a Java application without either application knowing what order the bits of an integer are stored in by the collaborating application. Java Beans is a lightweight component integration architecture that has conventions for Java object signatures which aid application builders and development environments wiring objects together.

A meta-level problem in producing infrastructure in a research context is that few publications regard application issues as relevant to their mission. Accordingly there is little published work in this area; we can cite experience with GATE version 1 and with LT XML, however, both of which are C-based systems that use the GNU configuration utilities for deployment. For embedding both systems make available their object code as libraries that may be linked with client code from other applications. The Corelli architecture [Zajac *et al.* 97, Zajac 97] used CORBA, but found it difficult to implement fast enough procedures due to the extreme decoupling of components mandated by the use of ORBs. Chapter 9 gives more details of the approach adopted in GATE version 2.

Part III

A General Architecture for Text Engineering

Chapter 6

Requirements Analysis

Parts I and II have described Software Architecture, Language Engineering and the software that typifies LE work, and reviewed infrastructures that support various aspects of the development of such software. This part presents a Software Architecture for LE called GATE, a General Architecture for Text Engineering. This chapter is a requirements analysis; chapter 7 describes the implementation. Part IV evaluates GATE and looks to its future.

As its name suggests, GATE is intended to be general, to cater for a wide range of LE activities and to encompass a large number of the useful infrastructural activities that have been identified by other work in this area. Given that language processing is still very much a research field, this task is potentially open-ended, and the question arises of how to restrict the endeavour to manageable proportions. The approach that we have taken is to make a distinction between software that aims to solve some research goal and software that provides an implementation of tasks that are common to a number of research goals (and that are not themselves active subjects of research). The latter are candidates for inclusion in the architecture; the former are not. In other words, GATE provides infrastructure tools and not research results: anything that is an open research topic is not properly part of the architecture. *How* to implement certain common algorithms or data structures may be part of the system; *what* algorithms and data structures to choose for a particular application or research project is not.

Exceptions to the 'no research' rule are made for two reasons. First, where a task is a valid research subject, but nonetheless so common amongst LE systems that a SALE will benefit greatly from the inclusion of a default implementation. Tokenisation of text is such a

subject: whilst not a solved problem (particularly for languages such as Japanese or Chinese), still most researchers would prefer to take a simple tokenisation scheme as read. Secondly, SALEs must be developed *in context*, along with the processing systems that will use them. In our case the primary context has been Information Extraction research, and we have actively distributed IE components along with versions of GATE in order to promote both the architecture and work on IE itself. (Note that these components are actually separate from GATE itself [Cunningham *et al.* 96a]. See chapter 8 for more details of IE systems developed within GATE.)

A SALE should support all the activities involved in the development, deployment and maintenance of LE software. In particular, anything that represents common ground amongst LE applications (i.e. gets implemented regularly) and anything done by support tools that help LE workers is a candidate for desiderata for the architecture. Following part II we may identify roles for SALE in the development of LE applications, technologies, methods and components. Examples of these roles:

For applications, allow easy embedding in mainstream software architectures (e.g. by exploiting component-based development, Java, the Internet).

For technologies, provide measurement apparatus (e.g. precision and recall of IE outputs relative to manual annotation).

For methods, implement common algorithms (e.g. Baum-Welch for HMMs, FST[1] over annotation) and support common data structures (e.g. annotation).

For components, provide abstractions that model their commonalities.

Following chapter 5 we can classify SALE roles according to the issues that have been addressed by previous work on LE infrastructure using the categories from section 5.1. This has the advantage of being a reflection of infrastructure requirements that have been developed in close association with the client group, and it is this classification that we will use in this chapter, which presents a set of use cases (see chapter 3) encapsulating the requirement set for GATE.

The clients of a SALE are the *actors* in these use cases; they may be human (software developers, researchers, teachers or students) or software (the programs written and used by the human clients). The client set includes:

[1] Finite State Transduction.

- expert programmers producing applications software that processes human language;
- non-expert programmers writing experimental software for research purposes;
- systems administrators supporting language researchers;
- non-programming language researchers performing experiments with software written by others;
- teachers of language technologies.

Section 6.1 gives general use cases; section 6.2 PR and LR use cases; section 6.3 use cases relating to methods; section 6.4 those specific to applications; section 6.5 those for development environments.

In order to give a reference point to the use cases below, each includes an example drawn from the *Information Harvester* application of section 4.1.

6.1 General Desiderata

Use Case 1: LE research and development

Goal: To support LE R&D workers producing software and performing experiments.

Description: During design, developers use the architectural component of SALE for guidance on the overall shape of the system. During development they use the framework for implementations of the architecture, and of commonly occurring tasks. The development environment is used for convenient ways of exploiting the framework and of performing common tasks. For deployment the framework is available independently of the development environment and can be embedded in other applications.

Example: Harvester developers centre their data management strategy on a model of documents and their annotations, implementation of which is provided by the framework. Individual components process the documents in different ways (ASR and NLG create documents; IE analyses them and adds extraction data to their annotations); in each case the development environment provides ways to run modules, visualise and evaluate their results.

Use Case 2: Documentation, maintenance, and support

Goal: To document, maintain and support the architecture.

Description: Without adequate documentation of its facilities an architecture is next to useless. Without bug fixes and addition of new features to meet changing requirements it will not evolve and will fall into disuse. Without occasional help from experts, users will learn more slowly than they could.

Example: Harvester developers need to find quickly what is available in the system that could help them. They need to be able to request bug fixes and explanations from the infrastructure developer, and get access to previous discussion *re.* bugs, features and so on.

Use Case 3: Localisation and internationalisation

Goal: To allow the use of the architecture in and for different languages.

Description: Users of the architecture need to be able to have menus and at least some of the documentation in a familiar language, and they need to be able to build applications which process and display virtually any human language.

Example: The Harvester is developed at multiple sites by people who have a wide variety of native languages. The software itself must process many languages.

Use Case 4: Software development good practice

Goal: To promote good software engineering in LE development.

Description: With reference to chapter 3, we can derive a number of general desiderata for SALEs on the basis that they are used for software development. In common with other software developers, SALE users need extensibility; interoperability; openness; explicit design documentation in appropriate modelling languages; graphical development environments; usage examples, or patterns.

Example: Applications software of the size and complexity of the Harvester must be engineered to high standards in order to succeed.

Use Case 5: Framework requirements

Goal: To exploit the benefits of the framework.

Description: Some general requirements for frameworks:
Orthogonality of elements: a user should not have to learn everything in order to use one thing.
Availability of abstractions at different levels of complexity: a user should be able to do something basic in a simple fashion, but also be able to fiddle under the hood if necessary.

Example: Some Harvester developers may only be involved in hand-annotating training data for machine learning algorithms. These developers do not want to know how the finite state transducers treat the data they create. They may be satisfied with the built-in data editing facilities to begin with, but want to add their own variations later on, without rewriting the whole system.

6.2 Components, PRs and LRs

Use Case 6: Locate and load components

Goal: To discover components at runtime, load and initialise them.

Description: R&D workers create LR and PR components and reuse those created by others. Experimenters, students and teachers use components provided for them. Systems administrators install components. Applications developers embed components in their systems. The set of components required in different cases is dynamic and loading should be dynamic as a consequence. The SALE should find all available components given minimal clues (perhaps a list of URLs), load them and initialise them ready for use.

Example: Harvester developers need not load all the components of the final system in order to work on their own – they may use annotations created manually to mimic the output of upstream[2] components, for example. When reusing they may try tagger A today and tagger B tomorrow. They may want to go back to the version they wrote last week and check how that worked.

[2]The terms 'upstream' and 'downstream' are used in this context to refer to points in an execution pipeline; upstream components execute early in the pipeline, downstream components execute later.

Use Case 7: PR and LR management

Goal: To allow the building of systems from sets of components.

Description: Developers need to be able to choose a subset of the available components and wire them together to form systems. These configurations should be shareable with other developers.

Example: If the Harvester is configured as a pipeline of components, developers construct the pipeline by selecting from a set of processing components and describing their order to the development environment. The environment then allows them to execute the processes, singly or as a batch, and to manipulate the results.

Use Case 8: Distributed processing

Goal: To allow the construction of systems based on components residing on different host computers.

Description: Components developed on one computer platform are seldom easy to move to other platforms. In order to reuse a diverse set of such components they must be made available over the Net for distributed processing. Networks are often slow, however, so there must also be the capability to do all processing on one machine if the component set allows.

Example: Different groups collaborating on the application make their most recent stable components available on their own site for remote running, so that each group does not need to install everyone else's software locally.

Use Case 9: Parallel processing

Goal: To allow asynchronous execution of processing components.

Description: Certain tasks can be carried out in parallel in some language processing systems. This implies that the execution of PRs should be multithreaded and means made available for parallel execution.

Example: While ASR analyses the next piece of audio input IE should be able to process the text output from a previous section without waiting.

Use Case 10: Component metadata

Goal: To allow the association of structured data with LR and PR components.

Description: Components are wired together with executive and task-specific code to form experimental systems or applications. Component metadata helps automate the wiring process, e.g. by describing the I/O constraints of the component. To use components they have to be found: metadata can be used to allow categorisation and description for browsing component sets.

Example: The Harvester uses ASR, which is increasingly a commodity product and might be expected to be present in various pre-configured forms in a well-found language processing lab. Given the appropriate metadata the Harvester programmers could browse the local component installations to find and exploit the available recognisers.

Use Case 11: Component commonalities

Goal: To factor out commonalities between related components.

Description: Where there are families of components that share certain characteristics those commonalities should be modelled in the architecture. For example, language analyser PRs characteristically take a document as input and add certain annotations to the document. Developers of analysers should be able to extend a part of the model which captures this and other characteristics.

Example: The Harvester uses lexical LRs in a number of contexts. In each case the means of looking up a word is identical, and the signature of the methods used is specified by the framework.

Use Case 12: LR access

Goal: To provide uniform, simple methods for accessing data components.

Description: Just as the execution of PRs should be normalised by a SALE, so access to data components should be done in a uniform and efficient manner.

Example: The Harvester uses Wordnet, EuroWordnet and the British National Corpus for various purposes. A SALE supporting the developers should abstract from the three underlying methods of opening these LRs for reading, getting to the data they contain and writing back changes where required.

Use Case 13: Corpora

Goal: To manage (possibly very large) corpora of documents in an efficient manner.

Description: Documents (texts and audiovisual materials) are grouped into collections which may have data associated with them. Operations which relate to documents should be generalisable to collections of documents.

Example: Complex LE application development is normally corpus-driven.

Use Case 14: Format-independent document processing

Goal: To allow SALE clients to use documents of various formats without knowledge of those formats.

Description: Documents can be processed independent of their formats. For example, an IE system can get to the text in an RTF[3] document or an HTML document without worrying about the structure of these formats. The structure is available for access where needed.

Example: The Harvester component language analysis technologies (IE, ASR) have to be isolated from the large number of formats available on Web pages (HTML, PostScript, PDF, Word, RTF, WAV, MPEG, MP3, etc.) in order to make them tractable.

Use Case 15: Annotations on documents

Goal: To support theory-neutral format-independent annotation of documents.

Description: Many of the data structures produced and consumed by PR components are associated with text and speech. Even NLG components can be viewed as producing data structures that relate to nascent texts that become progressively better specified, culminating in surface strings of words. See also the interoperation use case (annotation import/export to/from SGML/XML).

Example: Harvester developers both create manual annotation for training materials, and use annotation on documents as a method for sharing data between components.

[3]Rich Text Format, Microsoft's Word document interchange format.

Use Case 16: Data about language

Goal: To support creation and maintenance of LRs that describe language.

Description: Lexicons, grammars, ontologies, etc., all require support tools for their development, for example for consistency checking, browsing and so on. (Note that this use case is potentially very large.) In addition, developers of these types of resource use tools such as concordancers (e.g. KWIC) which should be provided by the development environment.

Example: If the Harvester had a parser based on HPSG, the ALEP grammar development environment might be used to construct linguistic resources for the parser.

Use Case 17: Indices

Goal: To cater for indexing and retrieval of diverse data structures.

Description: The architecture includes data structures for annotating documents and for associating metadata with components. These data structures need efficient indices to make computation and search over large data sets tractable.

Example: The Harvester language analysis modules may well create millions of annotations on large documents, and then require random access to them. (This use case is not subsumed by the annotation case because it would be advantageous to generalise its application across other data structures and across text itself in some cases.)

6.3 Method Support

Use Case 18: Common algorithms

Goal: To provide a library of well-known algorithms over native data structures.

Description: Although infrastructure should not in general stray into open research fields, where a particular algorithm is well-known it would be advantageous to provide a baseline implementation. For example, finite state transduction over annotation data structures, perhaps unification, ngram models and so on.

Example: The Harvester parser may use regular expression-based pattern/action rules

that recognise and produce annotations.

Use Case 19: Data comparison

Goal: To provide simple methods for comparing data structures.

Description: Machine learning methods, evaluation methods and introspective methods all need ways of comparing desired results on a particular language processing task with the results that a set of components has produced. In some cases this is a complex task (e.g. the comparison of MUC templates was found in some circumstances to be NP complete), but in many cases a simple comparison measure based on identity is useful for a first-cut approximation of success. This measure can be expressed as precision/recall where appropriate. (This use case is not subsumed by the annotation case because it would be advantageous to generalise its application across other data structures.)

Example: The Harvester's FST parser uses Brill's error-driven transformation-based learning algorithm, which depends on an evaluation measure.

6.4 Application Issues

Use Case 20: Persistence

Goal: All data structures native to the architecture should be persistent[4].

Description: The storage of data created automatically by components or manually by editing should be managed by the framework. This management should be transparent to a large degree, but must also be efficient and therefore should be amenable to tinkering where necessary. Access control may also be provided here.

Example: Let us imagine that the developers of the Harvester's NLG component need a data structure to encode rhetorical structure of utterances. If they define the data structure to extend or inherit from one of the architecture's own classes they should then be able to save that data via the framework and development environment without any further work. If they find that this default persistence is too slow or uses too much space, they should be able to write some more code to optimise the process, but still stay within the architecture's

[4]That is, their lifecycle should span more than one invocation of the programs that create and use them. In general 'persistent' implies 'stored on disk' in some form or other.

persistence framework.

Use Case 21: Deployment

Goal: To allow the use of the framework in diverse contexts.

Description: The framework must be available in many contexts in order to allow the transfer of experimental and prototype systems from the development environment to external applications and parts of applications. Users must be able to use framework classes as a library, including classes of their own that are derived from the framework classes. They should also be able to build programs based on the framework by supplying their own executive code, and be able to access data resources from other contexts using standard database protocols.

Example: Groups of Harvester developers create components running inside the SALE development environment. They then deliver these to form part of the final application as libraries that can be called externally as required.

6.5 Development Issues

Use Case 22: Interoperation and embedding

Goal: To enable data import from and export to other infrastructures and embedding of components in other environments.

Description: Formats and formalisms for the expression of LRs come in many shapes and sizes. Some of these are dealt with by wrapping those formats in code that talks the language of the SALE framework. Other, widespread formats should be made more generally accessible via import/export filters. The prime case here is SGML/XML.

Certain common execution environments should be catered for, such as MS Office (or COM, Common Object Model, applications) and Netscape Communicator.

Example: The Harvester has an off-line part and an on-line part. The former crunches Web pages and builds a database of facts; the latter presents those facts to users. Users access Web browsers and are presented with various ways to access the facts database. They may also submit new text for crunching.

The developers of the Harvester use the SALE's embedding and interoperation facilities to implement front-ends that are available in Netscape and MS Word. For example, a new menu is added to Word on PCs where the Harvester is installed that allows them to search for facts from the database, and to submit the current document for processing.

Use Case 23: Viewing and editing

Goal: To manipulate LE data structures.

Description: SALEs are used to view and edit the data structures that LE systems process. This applies to both LRs and PRs.

Example: For the Harvester IE parser a developer must code algorithms that do things like manipulate a chart of syntactic constituents, applying rules to create new constituents. A linguist must code grammar rules and lexicon entries. All these data structures are typically large and complex, and require visualisation tools. Editing is needed to develop test data or training data.

Use Case 24: Development UI

Goal: To give access to all the framework and architectural services and support development of LE experiments and applications.

Description: A large part of the SALE story is components, which can be viewed, edited, stored, accessed from the framework API and so on. The final element is a UI for developers that wires all these together and gives top-level access to storage and component management, and execution of PRs.

Example: A developer working on a noun phrase parser for the Harvester using an FST pattern language will repeatedly write regular expression pattern/action rules, compile the rules into a transducer, then run the transducer over certain texts and check the results, perhaps comparing them with some hand-annotated results for the same texts.

6.6 Summary

We have identified 24 use cases. They serve as input to the design and project planning processes, and, in chapter 8, a means to measure the extent to which GATE has addressed

the range of concerns which exist in LE and in previous work on SALE. The use cases are summarised here to provide a succinct point of reference for the set:

1. **LE research and development** To support LE R&D workers producing software and performing experiments.
2. **Documentation, maintenance, and support** To document, maintain and support the architecture.
3. **Localisation and internationalisation** To allow the use of the architecture in and for different languages.
4. **Software development good practice** To promote good software engineering in LE development.
5. **Framework requirements** To exploit the benefits of the framework.
6. **Locate and load components** To discover components at runtime, load and initialise them.
7. **PR and LR management** To allow the building of systems from sets of components.
8. **Distributed processing** To allow the construction of systems based on components residing on different host computers.
9. **Parallel processing** To allow asynchronous execution of processing components.
10. **Component metadata** To allow the association of structured data with LR and PR components.
11. **Component commonalities** To factor out commonalities between related components.
12. **LR access** To provide uniform, simple methods for accessing data components.
13. **Corpora** To manage (possibly very large) collections of documents in an efficient manner.
14. **Format-independent document processing** To allow SALE users to use documents of various formats without knowledge of those formats.
15. **Annotations on documents** To support theory-neutral format-independent annotation of documents.
16. **Data about language** To support creation and maintenance of LRs that describe language.
17. **Indices** To cater for indexing and retrieval of diverse data structures.
18. **Common algorithms** To provide a library of well-known algorithms over native data structures.

19. **Data comparison** To provide simple methods for comparing data structures.
20. **Persistence** All data structures native to the architecture should be persistent.
21. **Deployment** To allow the use of the framework in diverse contexts.
22. **Interoperation and embedding** To enable data import from and export to other infrastructures and embedding of components in other environments.
23. **Viewing and editing** To manipulate LE data structures.
24. **Development UI** To give access to all the framework and architectural services and support development of LE experiments and applications.

Ideally, a SALE should support all of these usage scenarios.

Chapter 7

Design and Implementation

This chapter concentrates on GATE version 1, which was initially released in 1996 and saw its final release in 1999. Version 2, currently under development, is refered to in passing here; a fuller description may be found in chapter 9. Version 1 comprises three principal elements (see figure 7.1):

- GDM, the GATE Document Manager, based on the TIPSTER document manager described in chapter 5;
- CREOLE, a Collection of REusable Objects for Language Engineering: a set of LE components integrated with the system;
- GGI, the GATE Graphical Interface, a development tool for LE R&D, providing integrated access to the services of the other components and adding visualisation and debugging tools.

The rest of this chapter describes these three elements in the context of the usecases that they refer to. Section 7.1 describes GDM; 7.2 describes CREOLE; 7.3 describes GGI.

GATE has been developed over a period of several years, and has been partly reported in [Cunningham *et al.* 95, Cunningham *et al.* 96d, Cunningham *et al.* 96c, Cunningham *et al.* 96e, Cunningham *et al.* 96a, Cunningham *et al.* 96b, Cunningham *et al.* 97b, Gaizauskas *et al.* 96b, Cunningham *et al.* 97a, Cunningham *et al.* 98b, Cunningham *et al.* 98a, Peters *et al.* 98, Cunningham *et al.* 99, Cunningham 99a, Cunningham *et al.* 00, Stevenson *et al.* 98]. The original design principles upon which the system was based may be summarised as:

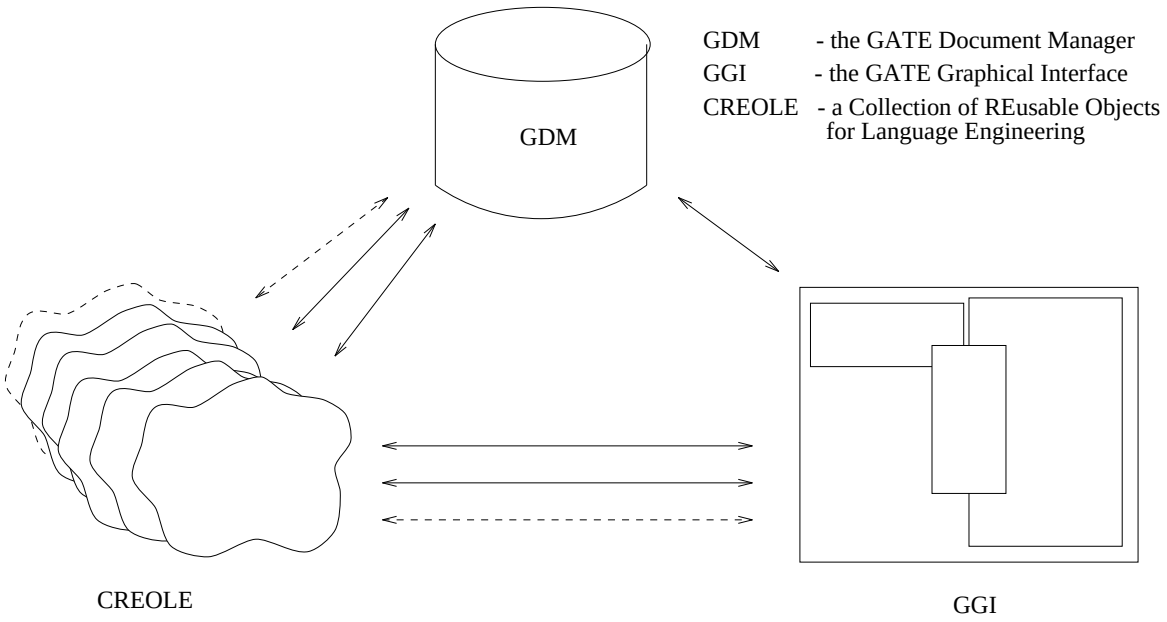

Figure 7.1: The three elements of GATE

- data structures for language information should not tie the user into a particular theory – see section 7.1;
- LE systems should adopt a component-based model where different modules may be exchanged – see section 7.2;
- visualisation of data and execution of components should be provided by a development environment – see section 7.3;
- facilities for managing storage of data and loading of components into programs should be transparent to the user – see sections 7.1 and 7.2.

As noted elsewhere, GATE version 1 was developed in the context of Information Extraction and other language analysis projects, and so the data structures and modules supported are concerned with documents, corpora and their analysis by components which add annotation to them. There is no dedicated support for Language Resources; all the modules discussed are in fact Processing Resources according to the terminology developed in section 4.3. Where developers using GATE wanted to model an LR separately from the PRs that used it, a component was created that annotated a document with the results of looking up the words in the document in the LR. For example, the Longmans Dictionary of Contemporary English (LDOCE) was represented in GATE as a PR which took all the token annotations in a document, looked them up in the dictionary, and added attributes to the tokens containing the dictionary definitions. This arrangement was adequate for dictionary access, but falls

short of supporting diverse LRs in their own right (see chapter 8), and version 2 of the system has adopted a model in which LRs are more privileged (see chapter 9).

7.1 Corpora, Documents and Annotations

Both the additive and referential approaches to representing information about text (discussed in section 5.3.3 of chapter 5) were serious candidates for GATE. Both were available as standards (SGML; TIPSTER architecture) when the project began, and both are theory-neutral (i.e. many different views of the appropriate data structures to describe human language can be accommodated). TIPSTER has a better time/space profile, can take advantage of database technology straightforwardly, is better for random access to data (SGML being inherently sequential [McKelvie *et al.* 97]), and can represent graph-structured and non-contiguous text structures easily. For these reasons the TIPSTER model was chosen for the core of the GATE Document Manager, with support for the import and export of SGML using the LT-NSL framework.

The GDM provides a central repository that stores all the information an LE system generates about the texts it processes. All communication between the components of an LE system goes through GDM, which insulates these components from direct contact with each other and provides them with a uniform API for manipulating the data they produce and consume. As pointed out in [Zajac 97], this reduces the number of interfaces between components in a serial execution system comprising n components from $n-1$ to 1. This is shown in figure 7.2: A, B and C are the components; the interfaces are shown in red. Whereas in the upper half of the diagram two interfaces (i1 and i2) are needed to connect the components, in the lower half only one (i3) is required.

The basic concepts of the data model underlying the GDM have been explained in the discussion of the TIPSTER model in section 5.3.3 above. The core of the model is the *collection* (equivalent to a corpus), whose members are *documents* containing texts and *annotations* upon them (see also figure 9.9 in chapter 9). The GDM is fully conformant with the document management subset of the specification [Grishman 97], and isolates the user from data storage issues by hiding behind the scenes the database layer that does the actual storage of annotations and other structures.

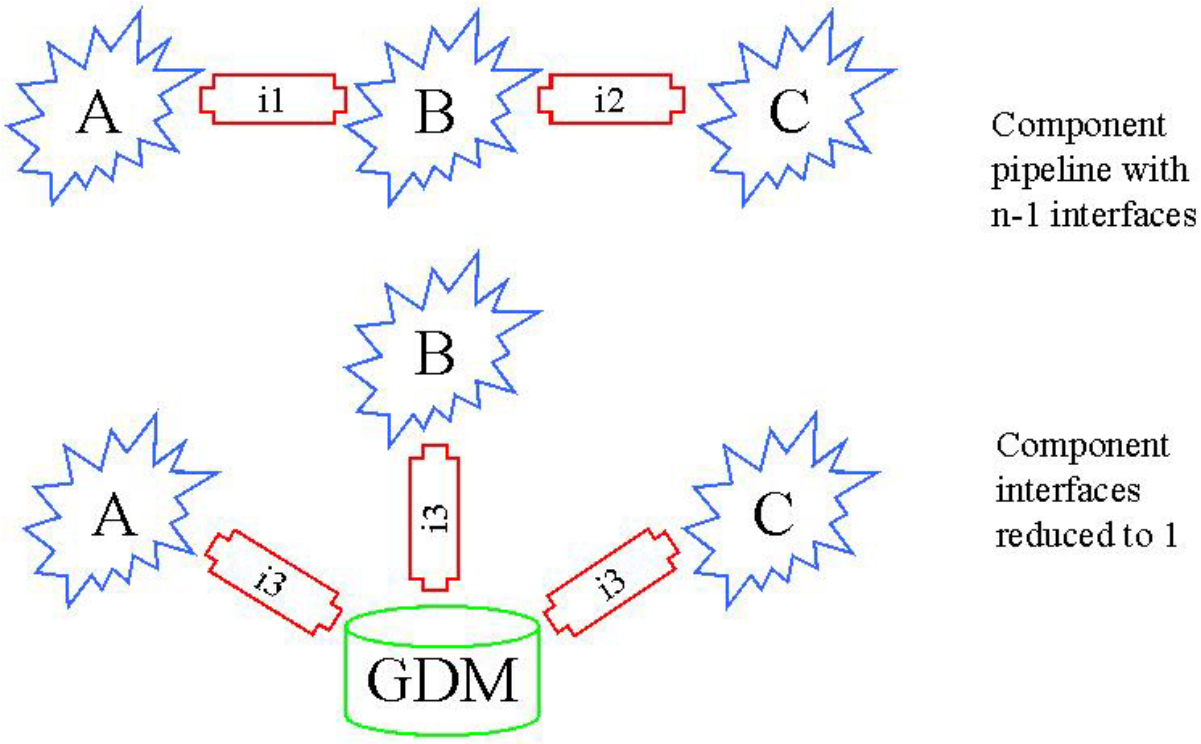

Figure 7.2: Interfaces between n components

7.2 A Component-Based Framework

All the real work of analysing texts in a GATE-based LE system is done by CREOLE modules or objects (we use the terms *module* and *object* rather loosely to mean PRs). Often, a CREOLE object will be a wrapper around a pre-existing LE module or database: a tagger, parser or dictionary, for example. Alternatively, objects may be developed from scratch for the architecture; in either case the object represents a standardised API to the underlying resources which allows access via the GGI and I/O via the GDM. The CREOLE APIs may also be used for programming new objects.

When the user initiates a particular CREOLE object via the GGI (or when a programmer does the same via the GATE API when building an LE application) the object is run, obtaining the information it needs (the document source and the annotations created by other PRs) via calls to the GDM API. Its results are then stored in the GDM database and become available for examination via the GGI or as the input to other CREOLE objects.

The GDM imposes constraints on the I/O format of CREOLE objects, namely that all information must be associated with byte offsets and conform to the annotation model of the TIPSTER architecture. The principal overhead in integrating a PR into GATE is making

the component use byte offsets, if it does not already do so.

7.2.1 Persistence

All the data created by the CREOLE PRs integrated within GATE is part of the TIPSTER model. In this model any particular object may be reached from the collection (or corpus) of which it is part: a collection has documents which have annotations which have attributes. Persistence facilities are associated with collections, which have methods for creating, opening, closing, and deleting on-disk representations of the data.

7.2.2 Locating and Loading Components

This section describes the process of integrating existing modules into GATE ('CREOLEising'), and how GATE finds and loads components. The developer is required to produce some C++, Tcl or Java code that uses the GDM API to get information from the database and write back results. The underlying module can be in C/C++, Java or Tcl, or be an external executable written in any language (the current set includes Prolog, Lisp and Perl programs, for example). When the module pre-dates integration, this code is called a *wrapper* as it encapsulates the module in the 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.

There are three ways to provide the CREOLE wrapper functions. Packages written in C, or in languages which obey C linkage conventions, can be compiled into GATE directly as a Tcl package [Ousterhout 94]. This is *tight coupling* and is maximally efficient but necessitates recompilation of GATE when modules change. On platforms which support shared libraries, C-based wrappers can be loaded at run-time, which is *dynamic coupling*. This is also efficient (with a small penalty at load time) and allows developers to change CREOLE objects and run them within GATE without recompiling the GATE system. Wrappers written in Tcl can also be loaded at run-time using a built-in Tcl interpreter – *loose coupling*. There is a performance penalty in comparison with using the C APIs, but for simple cases this is the easiest integration route. In each case the implementation of CREOLE services is completely transparent to GATE.

When a module has been adapted to use GATE for its input/output, and the configuration

information supplied, its location is specified by adding its parent directory to an environment variable (`GATE_CREOLE_PATH`). When the system starts up it searches all of these directories and automatically loads all the PRs that it finds.

7.2.3 Component Metadata

CREOLE wrappers encapsulate information about the pre-conditions 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 the 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.

An example configuration file appears in figure 7.3. This data structure (which is a Tcl array

```
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}
    }
}
```

Figure 7.3: A PR configuration file

[Ousterhout 94]) 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 a developer has to produce in order to integrate a module

with GATE. See [Cunningham *et al.* 96a] for more details.

Given an integrated module, all other interface functions are available 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's 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.

7.3 The Developer User Interface

One of the key benefits of adopting an explicit architecture for LE data management is that it becomes straightforward to add graphical interface access to architectural services and data visualisation tools. GGI, the GATE graphical interface, is a graphical tool that encapsulates the GDM and CREOLE resources in a fashion suitable for interactive building and testing of LE components and systems. The GGI has functions for creating, viewing and editing collections of documents that are managed by the GDM and that form the corpora which LE systems in GATE use as input data. The GGI also has facilities to display the results of component 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 manner, 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 normally 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 as for Named Entity markup. Thus CREOLE developers reuse the GATE data visualisation code with negligible overhead.

A central function of the GGI is to provide a graphical launchpad for the various LE subsystems available in GATE. Each subsystem is a grouping of PRs which have been selected from the pool of CREOLE objects that were found and loaded at startup time. Figure 7.4 shows an example; a user chooses a particular system by clicking on one of the jigsaw-like icons. The user having chosen a system, a window appears displaying a connected graph of the PRs that need to be run to achieve the task. This graph is called a *task graph*[1].

[1]Kevin Humphreys originated the task graph mechanism

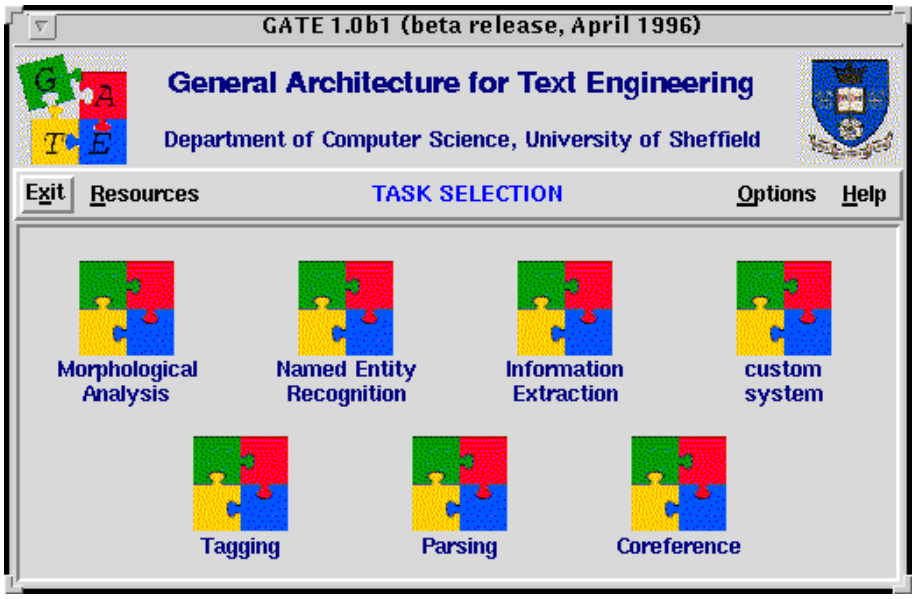

Figure 7.4: The GATE main window.

7.3.1 Executing Processing Components

An example of a task graph is shown in figure 7.5. In this graph, the boxes denoting PRs are active buttons: clicking on them will, if appropriate conditions have been met, cause the module to be executed. Paths through the graph indicate the dependencies amongst

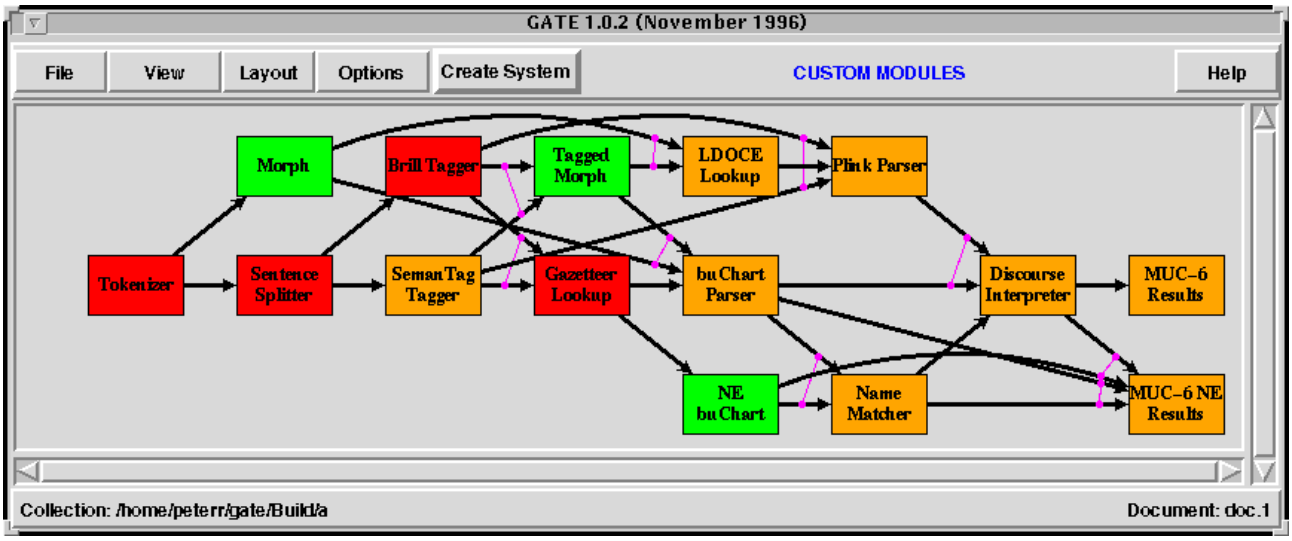

Figure 7.5: A GATE task graph

the various modules making up this subsystem. At any point, the state of execution of the system (or, more accurately, the availability of data from various PRs that have been executed) is depicted via colour-coding of the module boxes. After execution, the results of completed modules are available for viewing by clicking again on the module box, and are

displayed using an appropriate annotation viewer as described above. In addition, PRs can be 'reset', i.e. their results removed from the GDM, so as to allow the user to pick another path through the graph, or re-execute the PRs having altered some tailorable data-resource (such as a grammar or lexicon) interpreted by the module at run-time. (PRs running as external executables might also be recompiled between runs.)

A system graph is an executable graph, and may be thought of as a data flow program. Note that the execution of PRs in a graph is serial: GATE version 1 does not support parallel or distributed processing.

The arcs in the graph represent dependencies between components; each arc that is incident on a component indicates that the component uses one or more annotation or attribute that is created by the component from which the arc originated. Where a dependency may be satisfied by more than one source (i.e. the data item concerned may be created by more than one component), the arcs that represent the dependency are joined by a purple line and are termed 'disjunctive arcs' (or 'or arcs'). For example, in figure 7.5, the Gazetteer Lookup component may be run if either the SemanTag or Brill tagger has run; the Discourse Interpreter may run if either the Plink or buChart parser has run and the Name Matcher component has run.

Green modules are ready for execution (all their input conditions are currently satisfied by the state of the annotation/attribute data). Modules that cannot be executed are amber (some or all of their input conditions are unsatisfied). Modules that have already been executed are red, and the results that they produced may be accessed by right-clicking on them.

To make it easier to run a number of modules in one go without waiting for each to finish before selecting the next, a single click on a green module adds it to the current 'execution pending' path: the module turns white, but is not actually run, and other modules which rely on the results of the white module turn green (i.e. the graph behaves as if the white module has actually run). A path all the way through the graph can be selected in this way; clicking again on a white module will run all the modules in the graph up to that point.

7.3.2 PR Management

After a PR has been loaded by GATE, it may be selected for inclusion in a set of modules making up a language processing system in the tasks graphs described in section 7.3.1. These graphs are automatically generated, based on the pre- and post-conditions of the module's metadata. A component that has a particular annotation or attribute type in its list of post-conditions (as specified in the configuration file – see 7.2.3) will become the source of an arc incident on a component which shares that type in its list of pre-conditions. In figure 7.5 the Gazetteer component has the annotation type `lookup` in its post-conditions, and buChart has the same type in its pre-conditions, so an arc connects the two components. As noted earlier, where more than one component can produce output of the same type, the arcs indicent on components requiring input of that type are disjunctive and are joined by a line.

The task graphs are displayed with a graph drawing package[2] (also used in tree-based visualisation tools available for display of parse trees) which uses a layout algorithm similar to that used by the daVinci tool [Fröhlich & Werner 95].

7.3.3 Data Visualisation

When a module has been run, or when data has been loaded from a previously processed document, clicking on the module pops up a menu of the results viewers available. These viewers display all the annotations of a particular type that are currently present on a document. A default viewer displays annotations in a tabular form; this viewer can display all annotations, regardless of their structure, albeit in a form which is not particularly easy to read. To make viewing of annotation data easier, special viewers exist for a number of special cases.

- Single span annotations that associate a value with a range of text are displayed as in figure 4.3 (in chapter 4) or figure 7.11 below.
- Multiple span annotations that describe discontiguous structures such as coreference chains are displayed as in figure 7.6.

[2]Pete Rodgers implemented the graph drawing package.

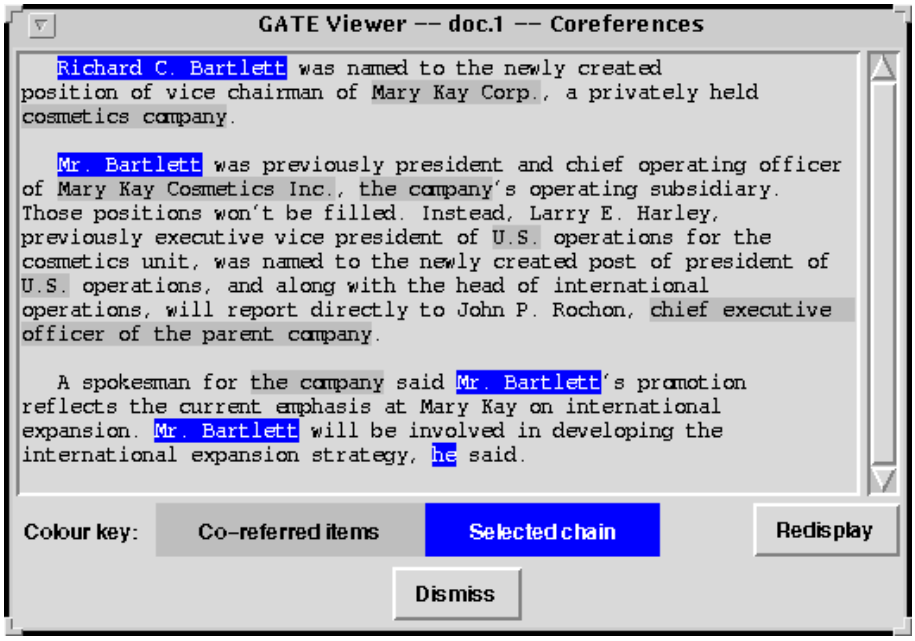

Figure 7.6: A multiple-span annotation viewer

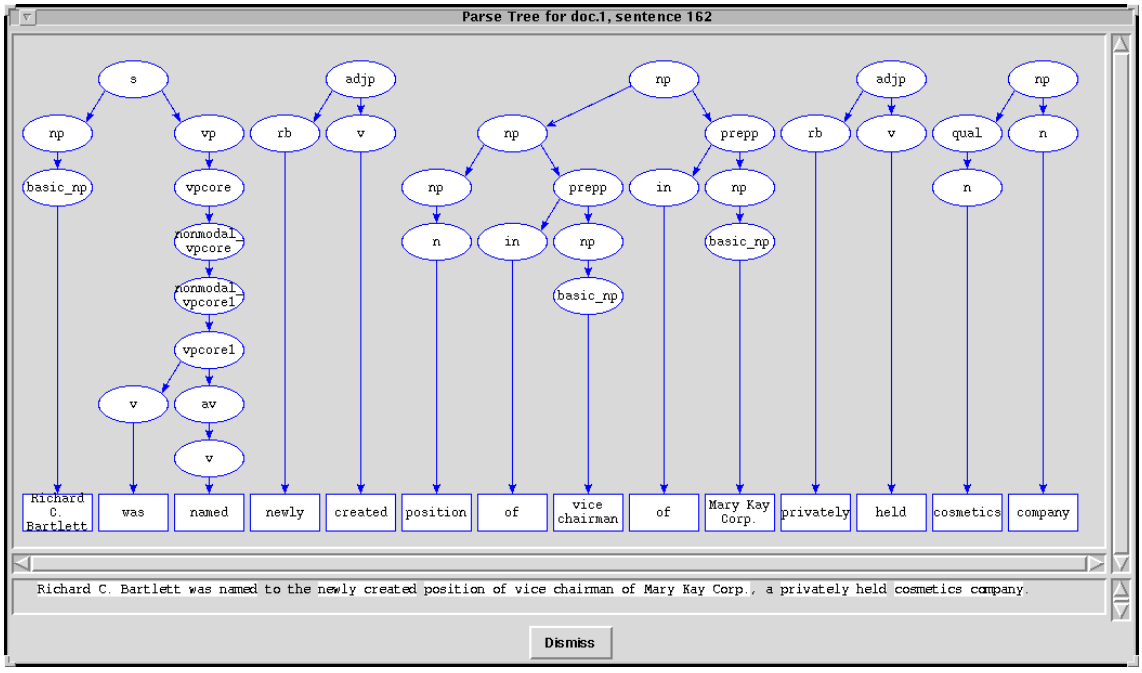

Figure 7.7: A tree viewer

- Interrelated annotations that describe tree structures such as syntax trees are displayed as in figure 7.7.

A developer can exploit these viewers by selecting them in a module's configuration file – see section 7.2.3. The viewers are connected to an extent: it is possible to click on an annotation in the tabular (default) viewer, or a node in a parse tree display, and see the area of text that the annotation refers to highlighted in the display of the text itself. See figure 7.8.

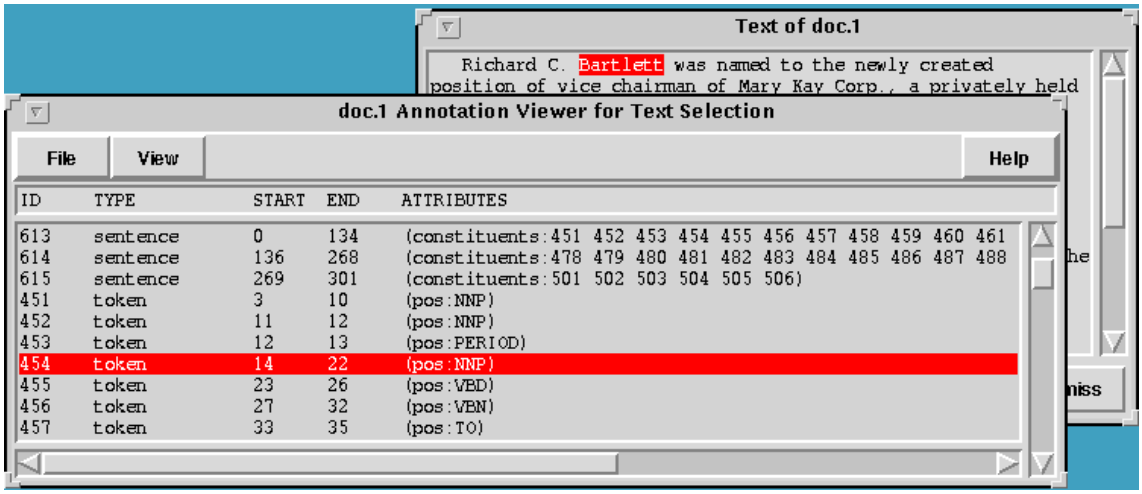

Figure 7.8: Integration between viewers

7.3.4 Data Comparison

GGI includes a facility for simple comparison of different sets of annotations, called the Compare Annotations Tool (CAT[3]), which operates on single-span annotations.

> An example CAT main window is shown in figure 7.9... The two sets of compared annotations are shown in the the CAT main window. The annotations are spaced so that annotations that cover the same area text appear at the same point in the window. The scroll bar in the middle of the window allows both sets of annotations to be scrolled up and down together. The metrics *Precision*, *Recall* and *F-Measure* for the two sets are displayed at the bottom of the window. Annotations with a grey background are the same in both annotation sets. Those

[3]Both CAT and MAT (see below) were designed by Robert Gaizauskas and implemented by Pete Rodgers.

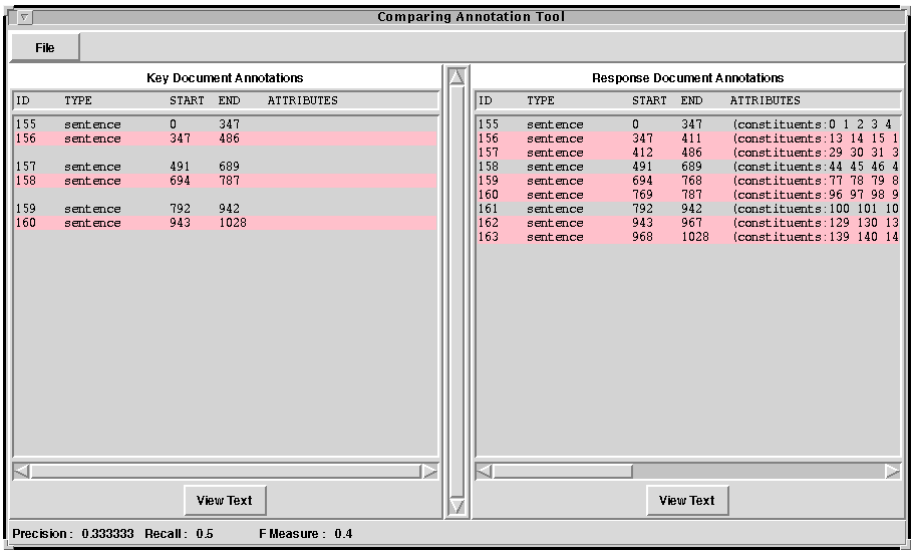

Figure 7.9: The CAT main window

> with a pink background are different... Clicking on an annotation will produce a window containing the document text with the chosen annotation highlighted...
>
> [Gaizauskas *et al.* 96a]

This tool provides a simple way to identify the differences between two runs of a module, or between manually-annotated data and data produced by the system under development. In more complex cases, a dedicated scoring system may be used (for example, the MUC scoring tools were integrated with GATE during development of IE systems for that competition).

7.3.5 Data Editing

The Manual Annotation Tool (MAT) allows the manual creation of annotations on documents, and also the editing of existing annotations. An example is shown in figure 7.10. The user selects types of annotations to be shown; clicking in the text will show the annotation at that position and allow editing. If a text range is selected, new annotations can be created on that range. For more details see [Gaizauskas *et al.* 96a].

As an alternative to a single general-purpose manual annotation tool, the visualisation components in GATE version 2 allow editing of the annotation data they display, thus providing a suite of special-purpose manual annotation facilities.

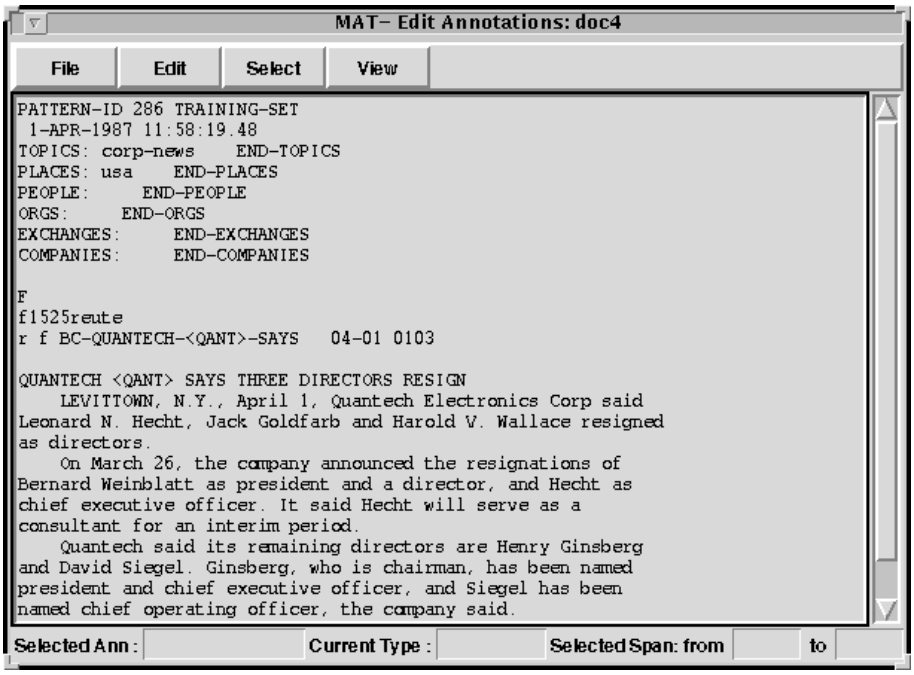

Figure 7.10: The MAT main window

7.4 Internationalisation and Deployment

7.4.1 Localisation and Internationalisation

GATE version 1 is restricted to 8-bit characters, due to limitations in the programming languages that it was written in. This makes it impossible to display languages with very large character sets, such as Japanese or Chinese. With a little work, however, the system can be persuaded to display non-ASCII character sets such as Greek – see figure 7.11.

7.4.2 Deployment, Interoperation and Embedding

GATE's implementation technology dictates its deployment, interoperation and embedding characteristics. Both old and new versions are intended to be useable in several contexts; this flexibility is achieved by the provision of multiple 'entry points':

- a development environment with its own graphical interface;
- a command-line batch processing system;
- a library which can be linked into other applications (that may or may not use elements of the graphical interface as required).

Figure 7.11: Results from a Greek gazetteer.

The deployment characteristics are quite different, however, because of the different implementation technologies used:

GATE Version 1 GDM is coded in C++, and GGI is in Tcl/Tk, as are the CREOLE integration facilities. Makefiles are provided for building the system on various platforms, and GNU auto configuration is used to tailor these Makefiles and the C code dependent on the peculiarities of the available system software and 'standard' C library. The core of GATE is built as a set of object code libraries, which may be linked to other applications. The Tcl `load` call is used for dynamic loading of CREOLE modules. The system runs on flavours of UNIX including SunOS, Solaris, Linux, AIX and HP-UX, and on Windows NT (using a port of the GNU tools).

GATE Version 2 GATE version 2 is in Java, and benefits from the cross-platform portability of that language, and from the built-in facilities for multi-threading, distributed processing and database access. It is tested on Solaris, Windows NT and Linux.

GATE version 1 has proved quite difficult to install, although it has been used successfully on diverse platforms; this was one of reasons for the choice of Java for version 2.

Interoperation with systems based on SGML is provided by input/output filters. For input from SGML documents, the LT NSL [McKelvie *et al.* 98] library was used in version 1 to transform SGML annotations into TIPSTER annotations. For output to SGML, an algorithm to transform TIPSTER annotations into SGML annotations was developed. For example, figure 7.12 shows the result of dumping name annotations from GATE into SGML

```
<ENAMEX TYPE="LOCATION">Alabama</ENAMEX> troopers find
<NUMEX TYPE="MONEY">$23 million</NUMEX> in cocaine.
<ENAMEX TYPE="LOCATION">WASHINGTON</ENAMEX>,
<TIMEX TYPE="DATE">April 4</TIMEX> (<NAME>Reuter</NAME>)

A cocaine haul valued at
<NUMEX TYPE="MONEY">$23 million</NUMEX> was
found in a load of bananas at a routine truck inspection
near <ENAMEX TYPE="LOCATION">Evergreen</ENAMEX>,
<ENAMEX TYPE="LOCATION">Ala.</ENAMEX>,
<ENAMEX TYPE="ORGANIZATION">NBC</ENAMEX>
television reported on <TIMEX TYPE="DATE">Friday</TIMEX>.

An estimated 1,100 pounds (1.1 short tons) of cocaine,
<ENAMEX TYPE="LOCATION">Alabama</ENAMEX>'s
largest drug seizure ever, was discovered
after traffic control officers were
alerted by a drug-sniffing dog,
<ENAMEX TYPE="ORGANIZATION">NBC</ENAMEX> said.

<NAME>Narcotics</NAME> officers were then called in.
The driver of the vehicle, a semi-trailer which was
carrying about 30,000 pounds (15 short tons) of bananas,
had been detained but had not been charged early on
<TIMEX TYPE="DATE">Friday</TIMEX>,
<ENAMEX TYPE="ORGANIZATION">NBC</ENAMEX> said.
```

Figure 7.12: An example SGML dump

(using the format specified in MUC: ENAMEX is an entity name expression; TIMEX a time expression; NUMEX a number expression; NAME signifies an entity which could not be categorised).

Part IV

Results and Prospects

Chapter 8

Evaluation

Part IV gives an evaluation of the work and a roadmap of the future of GATE. This chapter assesses the success of GATE; chapter 9 looks ahead.

A definitive set of quantitative metrics for the evaluation of this type of system does not exist. The measures used here are:

- take-up: the ultimate test of the system is whether or not LE practitioners want to use it;
- code reuse of systems developed in GATE;
- a review of functionality relative to the desiderata discussed in chapter 6.

Sections 8.1, 8.2 and 8.3 evaluate these in turn; the rest of the chapter presents two case studies of the system in operation:

- a word sense tagging system [Cunningham *et al.* 98b] (section 8.4);
- the LaSIE Large-Scale Information Extraction system [Gaizauskas *et al.* 95] (section 8.5).

Section 8.6 summarises the strengths and weaknesses identified in the rest of the chapter.

8.1 Usage of GATE

GATE is used as a research tool, for applications development and for teaching. Below we note examples of each, in reverse order.

Our colleagues in the Universities of Edinburgh, UMIST in Manchester, and Sussex (amongst others) have reported using the system for teaching, and the University of Stuttgart produced a tutorial in German for the same purposes (see `http://www.dcs.shef.ac.uk/gate/contrib/michael.dorna`).

Numerous postgraduates in locations as diverse as Israel, Copenhagen and Surrey are using the system in order to avoid having to write simple things like sentence splitters from scratch, and to enable visualisation and management of data.

Turning to applications development, ESTEAM Inc., of Gothenburg and Athens are using the system for adding name recognition to their MT systems (for 26 language pairs) to improve performance on unknown words.

Syntalex Ltd., of London, are developing a product that automatically applies amendments to legal documents within the GATE framework.

Both British Gas Research and Master Foods NV (owner of the Mars confectionery brand) used the LaSIE system for competitor intelligence systems. LaSIE was configured as an embedded library using GATE's deployment facilities.

[Gotoh *et al.* 98] report experiments using Named Entity tagged language models for large vocabulary connected speech recognition. The modelling data was created by running LaSIE as a batch process using GATE's command line interface.

The Swedish Institute of Computer Science have used GATE for integrating a national collection of NLP tools (for Swedish) – see [Eriksson 97, Eriksson & Gambäck 97, Olsson *et al.* 98], and `http://www.sics.se/humle/projects/svensk/`.

Competitor intelligence researchers in Finland are using GATE in the BRIEFS project [Keijola 99]:

> We at the Department of Industrial Engineering and Management of Helsinki University of Technology have been users of Gate since June of this year. The Computer Linguistics Department of Helsinki University is also involved in our

> project which is called BRIEFS (for Brief driven Information retrieval and Extraction for Strategy). The project is about business intelligence for companies. Our industrial partners include Nokia Corporation.
>
> After some initial pain in getting the system to run on Red Hat 6.0 we have been quite satisfied with Gate and have been able to get some results quickly which indeed was our main criteria for choosing Gate.
> [Matti Keijola, personal communication, October 1999.]

The Polytechnic of Catalunya in Barcelona

> ... have used GATE as a frame for integrating our tools for Spanish/Catalan with other tools for Basque as part of the ITEM project... we are now working in Multilingual Information Extraction and Information Retrieval for Basque/Spanish/Catalan... GATE helped us very much as a way to integrate our tools and to visualise the results and have a friendly environment which can be used for people who are not familiar with the tools input/output formats.
> [Jordi Batalla, personal communication, March 1999.]

The main problem experienced by this group was insufficient speed of processing large corpora.

GATE has been used for Information Extraction in English, German, French, Spanish, Greek and Swedish [Kokkinakis 98, Kokkinakis & Johansson-Kokkinakis 99]. Figures 8.1 – 8.2

Figure 8.1: Results from Greek tagger.

illustrate the output of the GIE (Greek IE) project which constructed a Greek Named Entity system based on LaSIE and GATE [Spyropoulos 99]. The GIE project identified these advantages of GATE:

- "Quick and easy CREOLE module development entirely in the Tcl language.
- Easy embedment of external 'ready to use' programs, through Tcl or C++ wrappers.
- A set of powerful and easy to use viewers.
- Easy integration of modules into complete systems with the additional capability of batch processing.
- A small set of powerful tools that allow the end user to intervene in the output of any module (MAT Annotator), compare two documents and calculate various measures such as Recall or Precision (CAT Comparison) and export SGML." [Spyropoulos 99]

The problems experienced using GATE for GIE were:

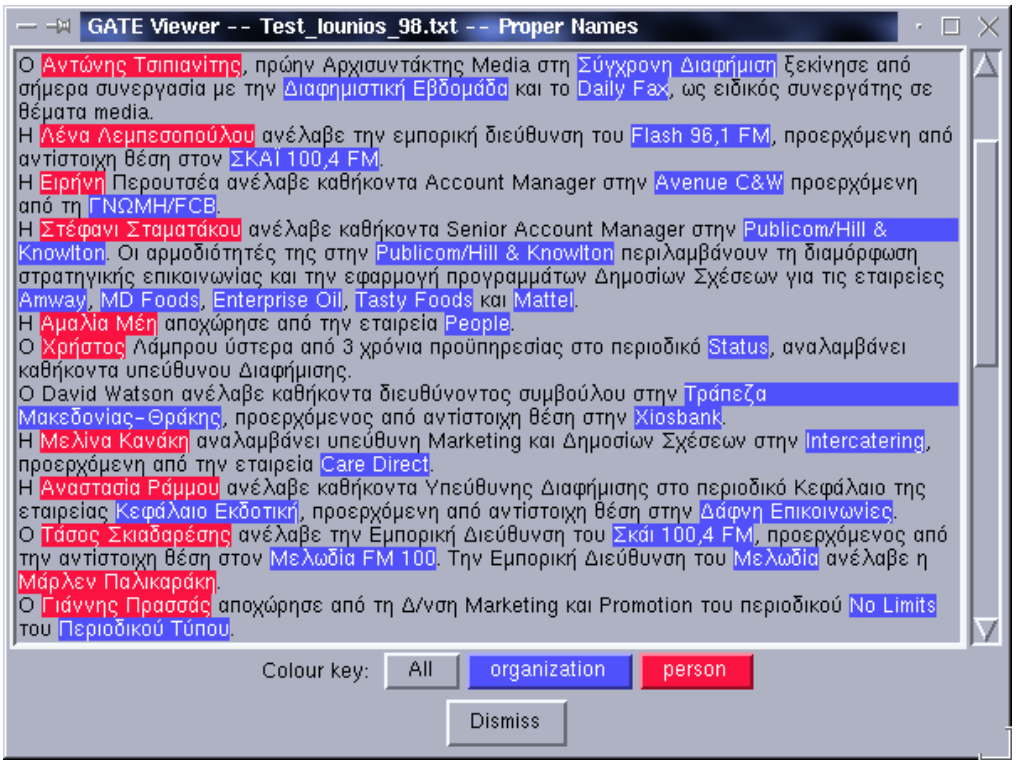

Figure 8.2: Results from Greek NE.

- Time/space profile of the executable.
- Hard-coding of non-internationalised font specifiers into the GGI code, creating difficulties with non-Latin character sets.
- Difficulties with the task graph generation process when there are complex sets of modules, for example differentiated by language.
- Various minor complexities and non-fluencies in the GUI.
- Non-extensibility of the tool model (extensibility is only available for PRs, not for viewers, or other tools).
- Complexity of compilation and installation.

In the projects cited above and others, researchers have contributed a diverse and growing set of components to CREOLE. Members of CREOLE include:

- the VIE[1] English IE components (tokeniser and text structure analysers; sentence splitter; several POS taggers; morphological analyser; chart parser; name matcher; discourse interpreter);

[1] A 'Vanilla IE' system related to LaSIE – see below.

- the Alvey Natural Language Tools morphological analyser and parser;
- the Plink parser;
- the Parseval tree comparison software;
- the MUC scoring tools;
- French parsing and morphological analysis tools from Fribourg and INRIA; Italian corpus analysis tools from Rome Tor Vergata;
- a wide range of Swedish language processing tools.

The main deficiency in this set is a bias towards language analysis, and towards Processing Resources above Language Resources.

A partial list of GATE licensees (of which there were over 300 at the end of 1999) is available at `http://www.dcs.shef.ac.uk/nlp/gate/users.html`. Along with the experiences cited above, this list indicates that take-up of the system is healthy and that LE R&D workers have found the system useful in many contexts.

8.2 Code Reuse

GATE encourages code reuse in two ways. First, systems developed from CREOLE components result in a modular design where inter-component boundaries are well-specified, and installation of components is a matter of telling GATE where to find them when it starts up. These components are then easy to transport into other contexts for other purposes. To use a language analogy, GATE takes care of all the syntactic issues to do with data representation, configuration, component interfacing and so on. The developer is then free to concentrate on the substantive, semantic issues. Such component reuse has proved common in projects with which we have experience; examples appear in the rest of this section and in the case studies which follow.

The second way in which GATE promotes code reuse is by providing GUI and data management code that are general across diverse contexts and can therefore be reused in different projects. For example, figure 8.3 shows three lines from the configuration file for a parser integrated into GATE. Each line lists an annotation/attribute type, names a viewer appro-

```
{name type} single_span {Proper Names}
{syntax category} parse {Syntax}
{semantics qlf} raw {Semantics}
```

Figure 8.3: Viewer configuration example

priate to that type and gives a title for the window that will be displayed. E.g. line two is for annotations of type 'syntax', with attribute 'category', and will use GATE's 'parse' viewer under the title 'Syntax'. When this module is run from the GATE development environment, one of the results available will be the syntax annotations, and they will be displayed in succinct graphical form without the developer writing any GUI code at all. Similarly, the data management substrate in the TIPSTER document manager is reused whenever modules communicate or their results are stored. In this way the developer is achieving a high level of code reuse by using the GATE core facilities during development and deployment.

Returning to the issue of code reuse via component reuse, the experience of seven projects at Sheffield suggests that a high degree of code reuse is possible using GATE, with the following projects all using modules from LaSIE, along with newly developed modules: AVENTINUS; ECRAN; PASTA; EMPATHIE; TRESTLE; ELSE; STOBBS (see `http://www.dcs.shef.ac.uk/nlp/research.html` for details).

Elsewhere, a Swedish IE system also benefited from code reuse, as reported in [Kokkinakis 98, Kokkinakis & Johansson-Kokkinakis 99]:

> Here are some figures on the number of lines I changed in some of the 'VIE' components:
>
> **Tokeniser:** New: 0, Reuse: all. Minor modifications on some SGML tokens and names; both in the lex files and the config files.
>
> **Sentence splitter:** On the Perl script I use the same functions as VIE but with different conditions: New: 250, Reuse: 100. Minor modifications on names in the wrapper and the config file: New: 0, Reuse: all
>
> **Semantag/Brill taggers:** New: 0, Reuse: all. Minor modifications on names in the Tcl and config. file. The Brill resources (lexicons etc.) are of course different.
>
> **Morphological analyser:** In the wrapper: New: 20, Reuse: 20, on the cases

where you "strncmp" the tags. In the morph.ll I used the same functions: Reuse: 70, New: 1000. Minor modifications on names in the config file.

Gazetteer lists: Only the content of the lists is changed of course since there are few language specific names: Reuse: All, New: 200. The programs, except on some minor modifications on names in the config and the lookup files: New: 0, Reused:All

buchart: Reused: all of the non-rule files, with some changes on file names etc. NE grammar: New: 20 Reused: all. Minor changes on designators and translation of some key words to Swedish. Grammar and vp-rules: Reused: 800, New: 400

(Dimitri Kokkinakis, personal communication, 1998.)

Similarly, GIE achieved a high level of reuse in translating the English LaSIE NE system to Greek texts [Spyropoulos 99].

We can conclude from these and other experiences that GATE lives up to the promise of SALE with respect to code reuse.

8.3 Requirements Review

This section discusses GATE functionality relative to the desiderata and the use cases given in chapter 6, which are summarised below (excluding some of the general cases). GATE 1 catered for 10/19 of these cases, either fully or partially. The current design of GATE 2 is intended to satisfy all or nearly all of these use cases. Chapter 9 describes how GATE 2 addresses these requirements; this evaluation is based on GATE 1.

1. **Localisation and internationalisation** To allow the use of the architecture in and for different languages.
   GATE 1 didn't really meet this goal, as the range of languages that could be displayed is restricted, and the systems menus, documentation, etc. was only available in English.

2. **Locate and load components** To discover components at runtime, load and initialise them.
   $\surd$ GATE 1 met this goal.

3. **PR and LR management** To allow the building of systems from sets of components.
   $\surd$ GATE 1 met this goal for PR components.

4. **Distributed processing** To allow the construction of systems based on components residing on different host computers.

5. **Parallel processing** To allow asynchronous execution of processing components.

6. **Component metadata** To allow the association of structured data with LR and PR components.
   $\surd$ GATE 1 partially met this goal for PR components.

7. **Component commonalities** To factor out commonalities between related components.

8. **LR access** To provide uniform, simple methods for accessing data components.

9. **Corpora** To manage (possibly very large) collections of documents in an efficient manner.

10. **Format-independent document processing** To allow SALE users to use documents of various formats without knowledge of those formats.

11. **Annotations on documents** To support theory-neutral format-independent annotation of documents.
    $\surd$ GATE 1 met this goal.

12. **Data about language** To support creation and maintenance of LRs that describe language.

13. **Indices** To cater for indexing and retrieval of diverse data structures.

14. **Common algorithms** To provide a library of well-known algorithms over native data structures.

15. **Data comparison** To provide simple methods for comparing data structures.
    $\surd$ GATE 1 met this goal.

16. **Persistence** All data structures native to the architecture should be persistent.
    $\surd$ GATE 1 met this goal.

17. **Deployment** To allow the use of the framework in diverse contexts.
    $\surd$ GATE 1 met this goal, with the caveat that cross-platform installation was difficult.

18. **Interoperation and embedding** To enable data import from and export to other infrastructures and embedding of components in other environments.
    $\surd$ GATE 1 partially met this goal for SGML.

19. **Viewing and editing** To manipulate LE data structures.
    $\surd$ GATE 1 met this goal.

20. **Development UI** To give access to all the framework and architectural services and support development of LE experiments and applications.
    $\surd$ GATE 1 met this goal.

8.4 Case Study 1: Sense Tagging

Note: excepting this paragraph, this section is adapted from [Cunningham *et al.* 98b], reporting work done by Yorick Wilks and Mark Stevenson[2] – see also [Wilks & Stevenson 97a, Wilks & Stevenson 97b, Wilks & Stevenson 98, Stevenson *et al.* 98, Wilks 98, Wilks & Stevenson 99]. The section describes the experience of building a sense tagging system within GATE, and the lessons learned for the GATE project. We give only a cursory description of the sense tagging problem in itself, concentrating on the implications for evaluating and developing GATE.

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.

The sense tagger was implemented as a set of 11 CREOLE modules, 6 of which had been implemented as part of VIE (see below) and the remaining 5 were developed specifically for the sense tagger. 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 section 7.2.2). 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

[2]Thanks are due to Mark for helping to produce this case study.
[3]This is often loosened to each *content* word.

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 8.4.

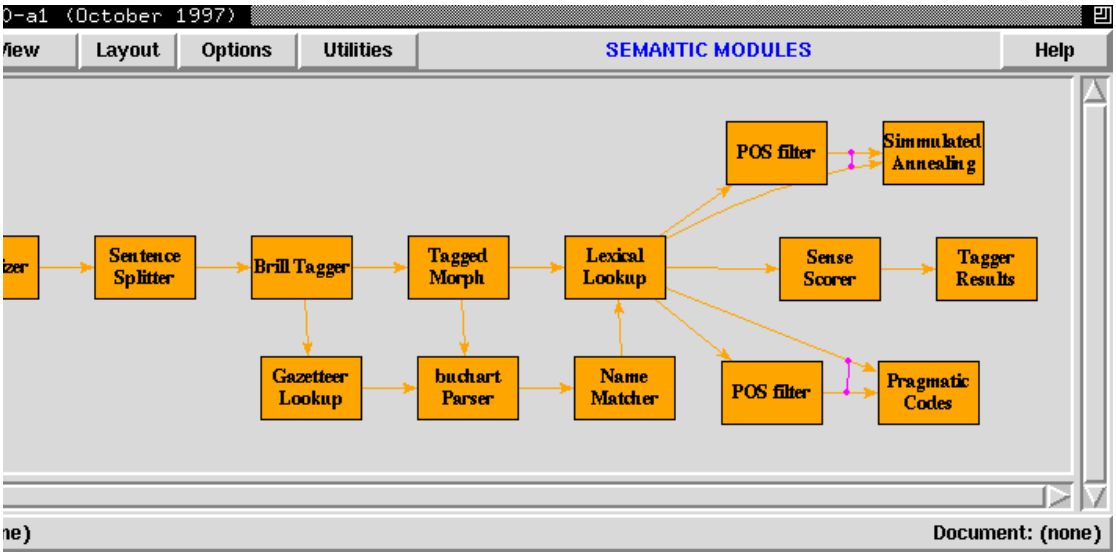

Figure 8.4: The sense tagger in GATE

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 with the ambiguous words highlighted (see figure 8.5). 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 8.6). 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.

8.4.1 Lessons Learned for GATE

The experience of implementing a novel system within GATE gave insights into the usefulness of the architecture which could not have been gained otherwise, and the result was encouraging. With the TIPSTER annotation scheme, GATE provides a well-defined standard for data-transfer between components. This approach allows for the rapid re-use of existing components and removed the need to specify the interfaces between components. Almost the entire preprocessing of the text was carried out using components which had al-

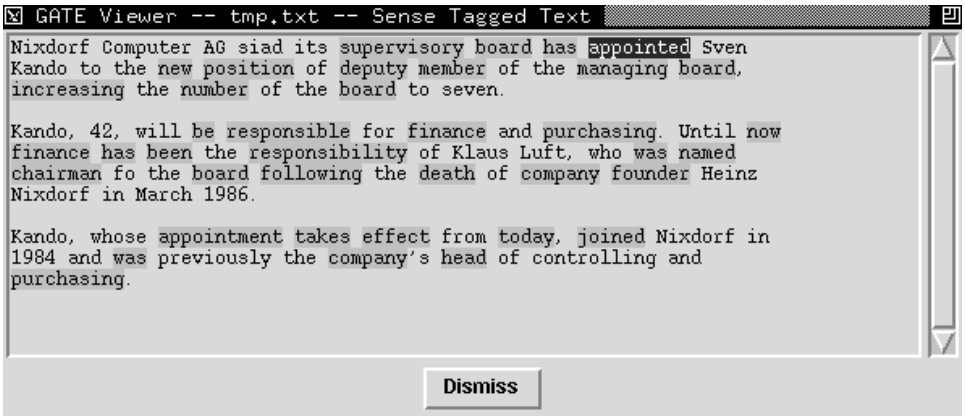

Figure 8.5: Words disambiguated by the tagger

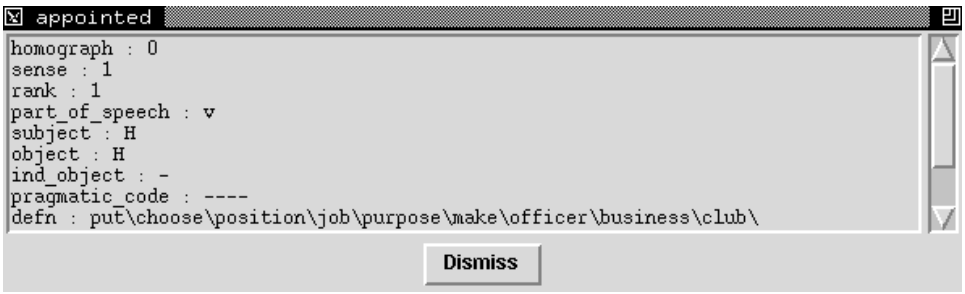

Figure 8.6: View of senses assigned to *"appointment"*

ready been implemented within GATE: the tokeniser, sentence splitter, Brill part-of-speech tagger and the components which made up the Named Entity identifier. This meant that the developers could quickly implement the components which carried out the disambiguation, and those were the components in which they were most interested. The implementation was further speeded up by the use of results viewers which allowed the examination of the annotations in the GATE document manager after a component had been run, allowing discovery of bugs more quickly. One aspect of sense tagging in which the project staff were interested is the effect of including and excluding different components, 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 component reset in the presence of non-monotonic database updates. For example, the POS filter components remove some of the sense definitions associated with words by the lexical preprocessing stages. When resetting these components 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 components alter previously existing annotations by examination of the pre-/post-conditions of the component supplied by the developer as configuration information prior to loading). This leads to redundant processing, and in the case of slow components (like the LDOCE lookup component) this can be an appreciable brake on the development cycle. (The planned solution is to change the implementation of the reset function. Currently this simply deletes the database objects created by a component. 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 difficulty of managing these graphs as the number of components 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 components; 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, as it is not obvious how the linear model currently embodied in the graphs could be extended to support non-linear control structures.

8.5 Case Study 2: LaSIE

To illustrate the process of converting pre-existing LE systems into GATE-compatible CREOLE sets we use as an example the creation of VIE (Vanilla Information Extraction system) from LaSIE (Large-Scale Information Extraction system) [Gaizauskas *et al.* 95], Sheffield's entry in the MUC-6 system evaluations[4]. LaSIE module interfaces were not standardised when originally produced and its CREOLEisation gives a good indication of the ease of integrating other LE tools into GATE. The resulting system, VIE, is freely distributed with GATE.

In various incarnations the system was entered in the MUC-6/7 evaluations [Gaizauskas *et al.* 95, Humphreys *et al.* 98] and in the MET Japanese entity recogni-

[4]Confusingly, LaSIE is both the name of our pre-GATE MUC-6 IE system, *and* our post-VIE internal development IE system. Unless implied otherwise, in this section 'LaSIE' refers to the MUC-6 system, which was integrated into GATE to form VIE. We later developed LaSIE separately from VIE, but within GATE. Most other usages of the term in this thesis refer to the later system.

tion task [Takemoto *et al.* 96, Wakao *et al.* 96]. Work on high-performance coreference resolution algorithms [Gaizauskas & Humphreys 96a, Gaizauskas & Humphreys 96b, Gaizauskas & Humphreys 97b, Gaizauskas & Humphreys 97a, Azzam *et al.* 98b] attained the highest score in MUC-7, and the system has been used for detailed evaluation of the use of semantic networks for IE [Gaizauskas & Humphreys 97c]. The system now processes multiple languages using an interlingual domain model to minimise cross-language porting costs by maximising the extent to which system resources are shareable across languages [Azzam *et al.* 97a, Azzam *et al.* 97b, Azzam *et al.* 98a, Azzam *et al.* 99, Kameyama 97].

LaSIE was designed as a research system for investigating approaches to Information Extraction and for entering in the MUC-6 conference [ARPA 95]. As such it was a standalone system that was aimed at specific tasks and, while based on a modular design, none of its modules were specifically designed with reuse in mind, nor was there any attempt to standardise data formats passed between modules. Modules were written in a variety of programming languages, including C, C++, Flex, Perl and Prolog. In this regard LaSIE was probably typical of existing LE systems and modules.

The high-level tasks LaSIE performed include the four MUC-6 tasks (carried out on *Wall Street Journal* articles): Named Entity recognition, coreference resolution and two template filling tasks.

The system had a pipelined architecture which processed a text one sentence at a time and consisted of three principal processing stages: lexical preprocessing, parsing plus semantic interpretation (creating a quasi-logical form [Alshawi 92] with Davidsonian semantics [Davidson 67]), and discourse interpretation.

As described in chapter 7, section 7.2.2, CREOLEisation of existing LE modules involves providing them with a wrapper so that the modules communicate via the GDM, by accessing TIPSTER document annotations and updating them with new information.

The major work in converting LaSIE to VIE involved defining useful module boundaries, unpicking the connections between them, and then writing wrappers to convert annotations relating to text spans into the module's native input format and to convert module output back into annotations relating to text spans.

The complete VIE system comprises ten modules, each of which is a CREOLE object integrated into GATE. The resulting system has all the functionality of the original LaSIE

system, but the interface makes it much easier to use. And, of course, it is now possible to swap in modules, such as a different parser, with significantly less effort than would have been the case before.

Of course, GATE does not 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, based on the work of the TIPSTER architecture group. Because GATE places no constraints on the linguistic formalisms or information content used by CREOLE modules, the latter problem must be solved by dedicated translation functions – e.g. tagset-to-tagset mapping – and, in some cases, by extra processing – e.g. adding a semantic processor to complement a bracketing parser.

8.6 Strengths and Weaknesses

The strengths of the final version 1 release of GATE are that it:

- facilitates reuse of NLP components by reducing the overheads of integration, documentation and data visualisation;
- facilitates multi-site collaboration on IE research by providing a modular base-line system (VIE) with which others can experiment;
- facilitates comparative evaluation of different methods by making it easy to interchange components;
- facilitates task-based evaluation, both of 'internal' components such as taggers and parsers, and of whole systems, e.g. by using materials from the ARPA MUC programme [Grishman & Sundheim 96] (whose scoring software is available in GATE, as is the Parseval tree scoring tool [Harrison 91], and a generic annotation scoring tool [Rodgers *et al.* 97]);

- provides a reasonably easy to use, rich graphical interface;
- contributes to increased software-level robustness, quality and efficiency in NLP systems, and to their cross-platform portability (all UNIX systems, Windows NT and Windows 95; native support for Java, C++ and Tcl);
- contributes to the portability of NLP systems across problem domains by providing a markup tool for generating training data for example-based learning (it can also take input from the Alembic tool [Day *et al.* 97] for this purpose);
- unifies the two best approaches to managing information about text by combining a TIPSTER-style database with SGML input/output filters (developed using tools from Edinburgh's Language Technology Group [McKelvie *et al.* 97]).

The principal problems with version 1 are that:

- It is biased towards *algorithmic* components for language processing, and neglects *data resource* components (PRs vs. LRs).
- It is biased towards text *analysis* components, and neglects text *generation* components.
- The database implementation is space and time inefficient.
- The visual interface is complex and somewhat non-standard.
- The task graph generation and management process does not scale beyond small component sets: "GGI suffers from the scaling problem [Burnett *et al.* 87], as the size of the custom graph quickly becomes unmanageable" [Rodgers *et al.* 97].
- Only the annotator component model is extensible; adding new viewers or tools is not possible.
- Installing and supporting the system is a skilled job, and it runs better on some platforms than on others (UNIX vs. Windows).
- Sharing of components depends on sharing of annotation definitions (but isomorphic transformations are relatively easy to implement).
- It only caters for textual documents, not for multi-media documents.
- It only supports 8-bit character sets.

- Module reset cascades through all previously run PRs that made non-monotonic database updates.

The next chapter looks at how we plan to combat these problems.

Chapter 9

Future Work

Looking back on one of the two project proposals that resulted in the original approval of funding for GATE in 1994[1] it seems that the original meaning of the acronym was 'General Architecture for Text *Extraction*'; in other words the original brief for the architecture was to support Information Extraction. As we saw in chapter 8, IE has been one of the major areas where the system has succeeded (partly because of the success of the LaSIE-based tools which have been distributed with it in various forms), but there are also a range of other R&D projects that have been pursued using GATE. This chapter describes the ways in which we are attempting to extend these successes to a greater proportion of the language processing field. The project is now known as GATE2[2].

Chapter 8 also indicated that there are significant areas of SALE that are not covered by GATE version 1; in most cases version 2 aims to close these gaps, in ways that are described below.

Parts of GATE2 exist in prototype form at the time of writing. Other parts are based on GATE version 1; still others are yet to be implemented. Snapshots of the development process are available at `http://gate.ac.uk/demos/`. In keeping with the idea that infrastructures should be built in close coordination with systems that use them (see chapter 3), GATE2

[1] Written by Yorick Wilks and Robert Gaizauskas. Funding from the EPSRC on grant GR/K25267.

[2] Funding from the EPSRC on grant GR/M31699, from July 1999 to June 2002; proposal written by Hamish Cunningham and Yorick Wilks; project managed by Hamish Cunningham. Some work on support for non-indigenous minority language writing systems within GATE is also funded by the EPSRC on grant GR/N19106 (EMILLE), written by Tony McEnery at Lancaster University and Robert Gaizauskas. Several smaller grants supported prototyping work in 1998 and 1999, including two from the US TIPSTER programme and one from the Max Planck Institute in Nijmegen, Holland (proposals written by Hamish Cunningham and Yorick Wilks).

prototypes have been used in several projects including:

- integration with the EUDICO multimedia architecture for language technology [Brugman *et al.* 99] (which gave us the opportunity to experiment with supporting the annotation of speech and video data);
- the LOTTIE (Low-Overhead Triage from Text with IE) demonstrator system (see section 9.2);
- the AVENTINUS Named Entity recognition system [Thurmair 97, Kokkinakis 98]).

A first public release of GATE2 is scheduled for 2000.

The design is structured around the requirements analysis of chapter 6, which in turn was influenced by the taxonomy of previous work on SALE in section 5.1. In this way we hope to keep the design close to the requirements that have been exposed by the R&D community which it is intended to serve.

Reflecting the terminology developed in chapter 3, this chapter begins with a discussion of the GATE2 architecture and framework (section 9.1), and then looks at the GATE2 development environment (section 9.2). Finally, the way in which each use case is reflected by design goals is presented (section 9.3).

Section 9.4 concludes the thesis.

9.1 The GATE2 Architecture and Framework

There are large differences between the currently evolving version of GATE and the original system presented in chapter 7, but the underlying principles remain the same:

- theory-neutral data structures for language data;
- a component-based model of language processing modules;
- a development environment for visualisation of data and execution of processing;
- facilities for managing storage of data and loading of processors into programs;
- support for LE applications, technologies, methods and components.

Applications developers can embed a set of components and the GATE framework in their systems to provide LE functionality in a conveniently packaged form that places few constraints on the type of software being constructed. Developers cherry-pick facilities as required, while simply ignoring architectural elements of which they have no need.

Many of the modules integrated with GATE version 1 are related to IE. To expand the range of technologies supported, our plan is to extend the set of CREOLE objects to cover as many core areas of LE R&D as possible. We hope to integrate modules for MT, Information Retrieval, ASR, and dialogue processing.

Another new feature is the inclusion of well-known algorithms (or methods) operating over some types of LR data structure. For example, finite state transduction over annotations is provided (see appendix A), and various options for reusing statistical modelling toolkits are under investigation.

GATE2 adds a number of new data structures to support a wider range of LE subfields, but in each case aims to provide an unconstraining model. In addition to the corpora and documents of the original system, GATE2 will support LRs such as lexicons, ontologies and thesauri. All LRs are modelled as data source objects (in general these are implemented using relational databases) coupled with a Java abstraction layer that encapsulate their structure in an OO form. Taking advantage of the Java DataBase Connectivity (JDBC) standard, the architecture allows the set of LRs used by an application to be distributed across the Net.

Processing components in GATE2 may be applied to any LR, and may also create LRs. In this way the model loses the previous bias towards language analysis, and makes support for generation and translation possible. Processing components may also be run on remote machines (using the Java Remote Method Invocation – RMI – protocol), and a multi-threaded execution model makes possible asynchronous execution control algorithms.

As before, the implementation of the system is structured as a framework library which can be embedded in applications in a flexible manner. One such application is the development environment supplied with the system, which is designed to support as many of the day-to-day language-related tasks of LE workers as possible. Using the Java Beans component integration standard, the framework allows modular access to graphical objects, so that the code that makes up the data visualisation and editing facilities of the development environment can now be reused in other applications, and their set may also be extended by users.

9.2 The GATE2 Development Environment

Figure 9.1 shows a GATE2 corpus viewing component[3]. Figure 9.2 shows a GATE2 annotation viewing component that displays trees, in this case syntax trees. This component also provides editing, and figure 9.3 shows the same tree with some additions.

Figure 9.4[4] shows a snapshot of the demonstrator for the GATE/EUDICO project [Brugman *et al.* 99] which used GATE components to display annotations on speech data. The annotations were created and the audio managed by the EUDICO framework [Brugman *et al.* 98a, Brugman *et al.* 98b]; the editing and display of the annotation (in this case part-of-speech) is done using GATE2 components. The top window is the audio player, which also scrolls the transcription. The lower two windows are annotation tiers for the two speakers involved in the dialogue. The red and grey highlighting in the lower windows indicates the annotation that is current with the audio as it plays.

The LOTTIE (Low-Overhead Triage from Text with IE) demonstrator application referred to above is also an example of how applications can embed GATE2 components. To quote the (somewhat optimistic) publicity materials:

> LOTTIE is a demonstrator project for the GATE language processing infrastructure. Parts of it are real, based on a project carried out in a different domain but with similar goals. Parts of it are faked, and are included as a test case for GATE development and as an illustration of where language processing technology may be applied in the future.
>
> Imagine this scenario:
>
> A high-speed passenger train collides with a truck on the Swiss border near Basel. The truck is carrying hazardous chemicals, which are blown by the wind towards a major population centre. The emergency services of three countries are mobilised (Germany, France and Switzerland share a border near Basel), and ambulances, paramedics, fire fighters and army personnel converge on the crash scene. These services are tasked with: rescuing wounded people from the train and transferring them to hospitals; dousing the chemical spillage with neutralising agents; evacuating those downwind of the chemical cloud; rescuing those already

[3]Kalina Bontcheva did the user interface programming on this and other GATE2 viewers.
[4]Principal programmers Kalina Bontcheva and Hennie Brugman.

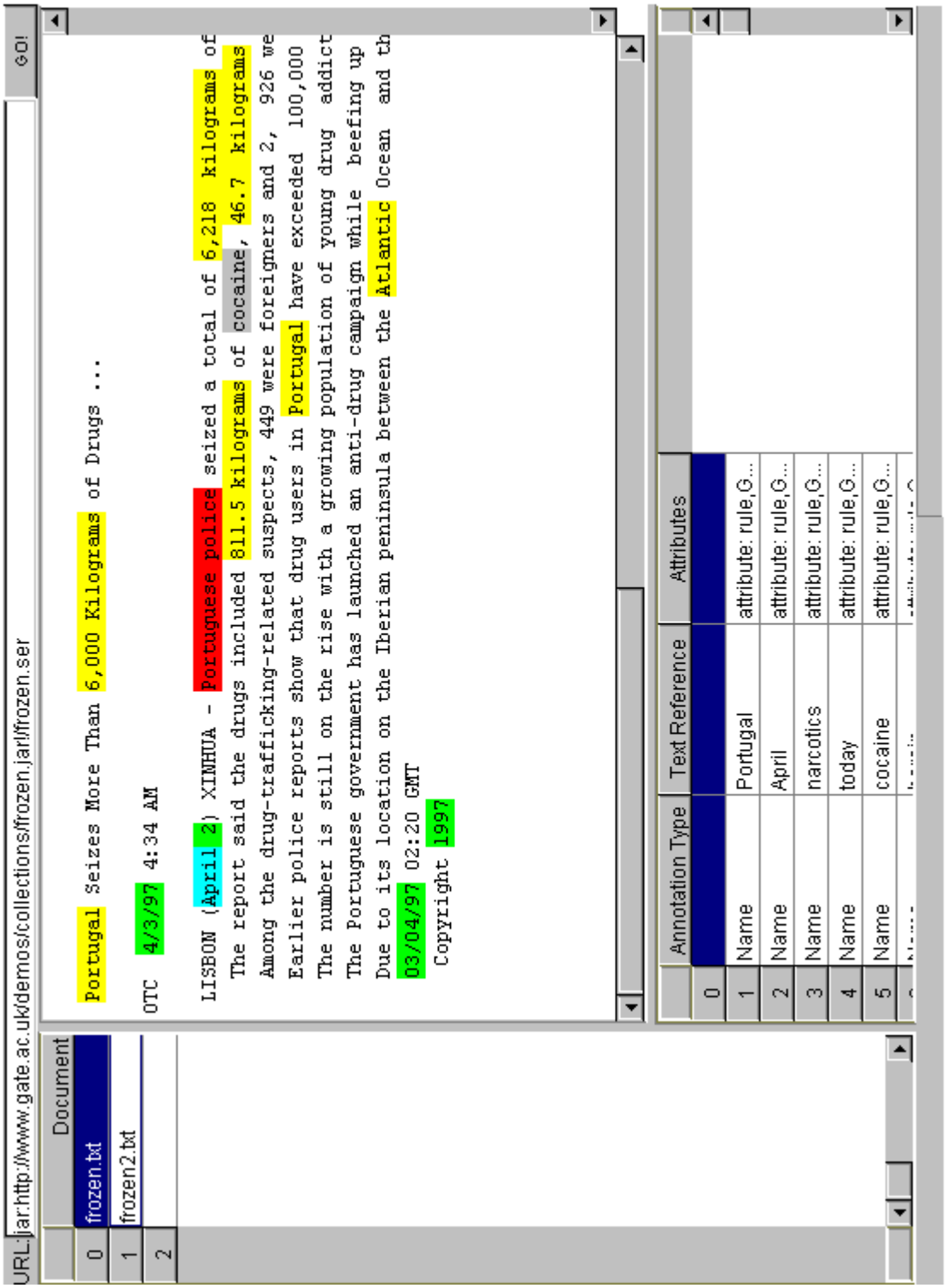

Figure 9.1: A corpus viewer

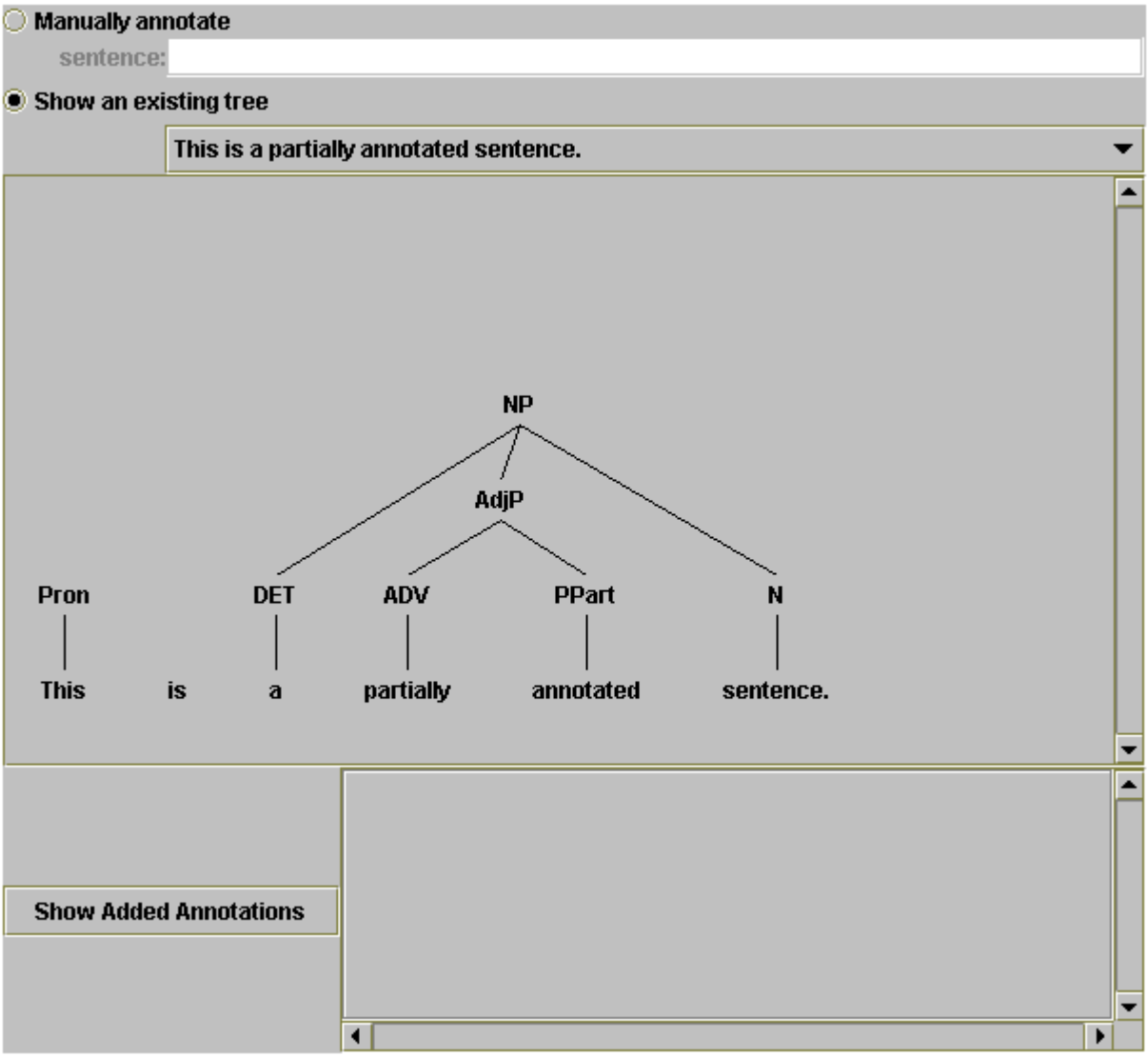

Figure 9.2: A syntax tree viewer

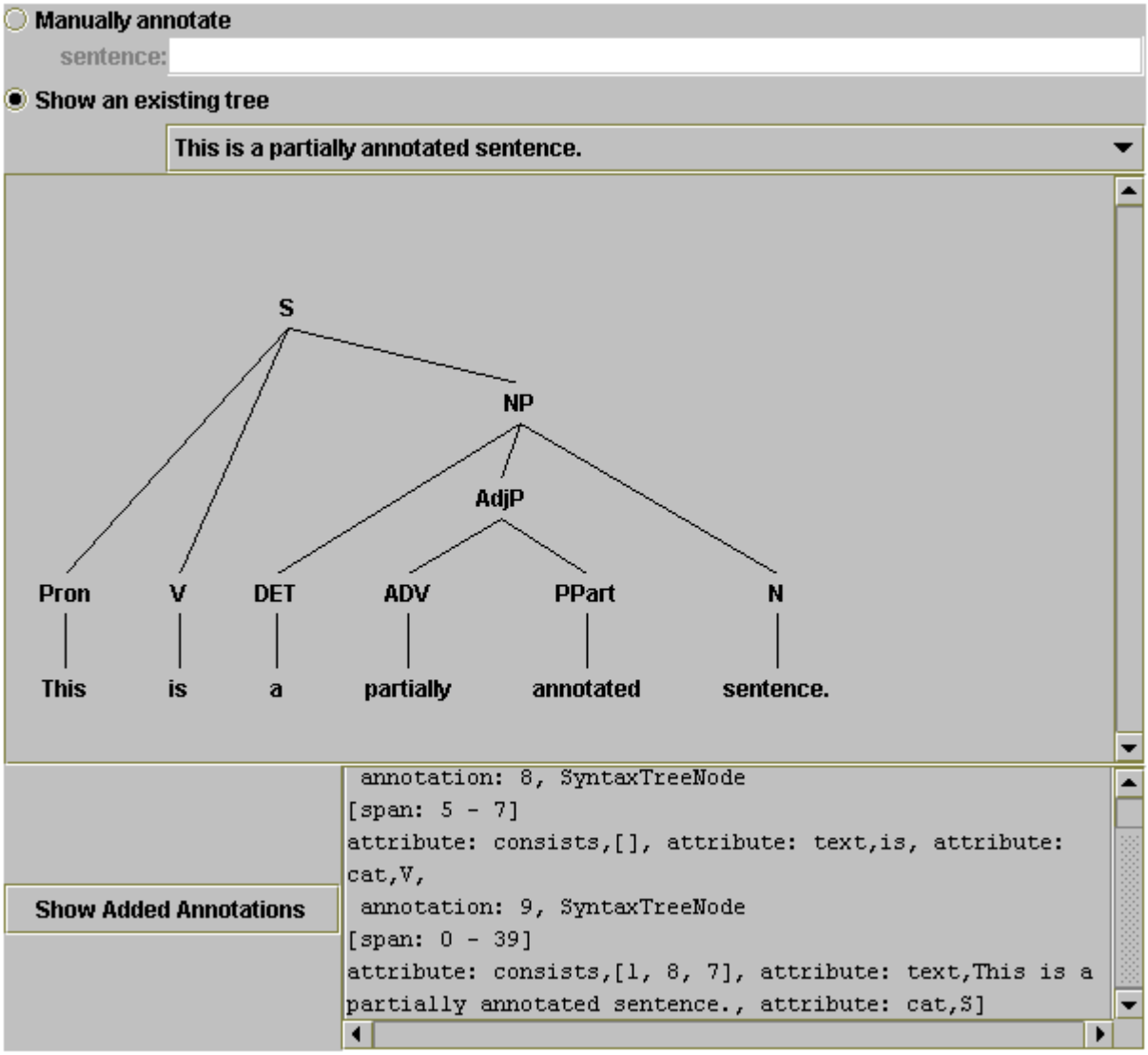

Figure 9.3: Syntax tree editing

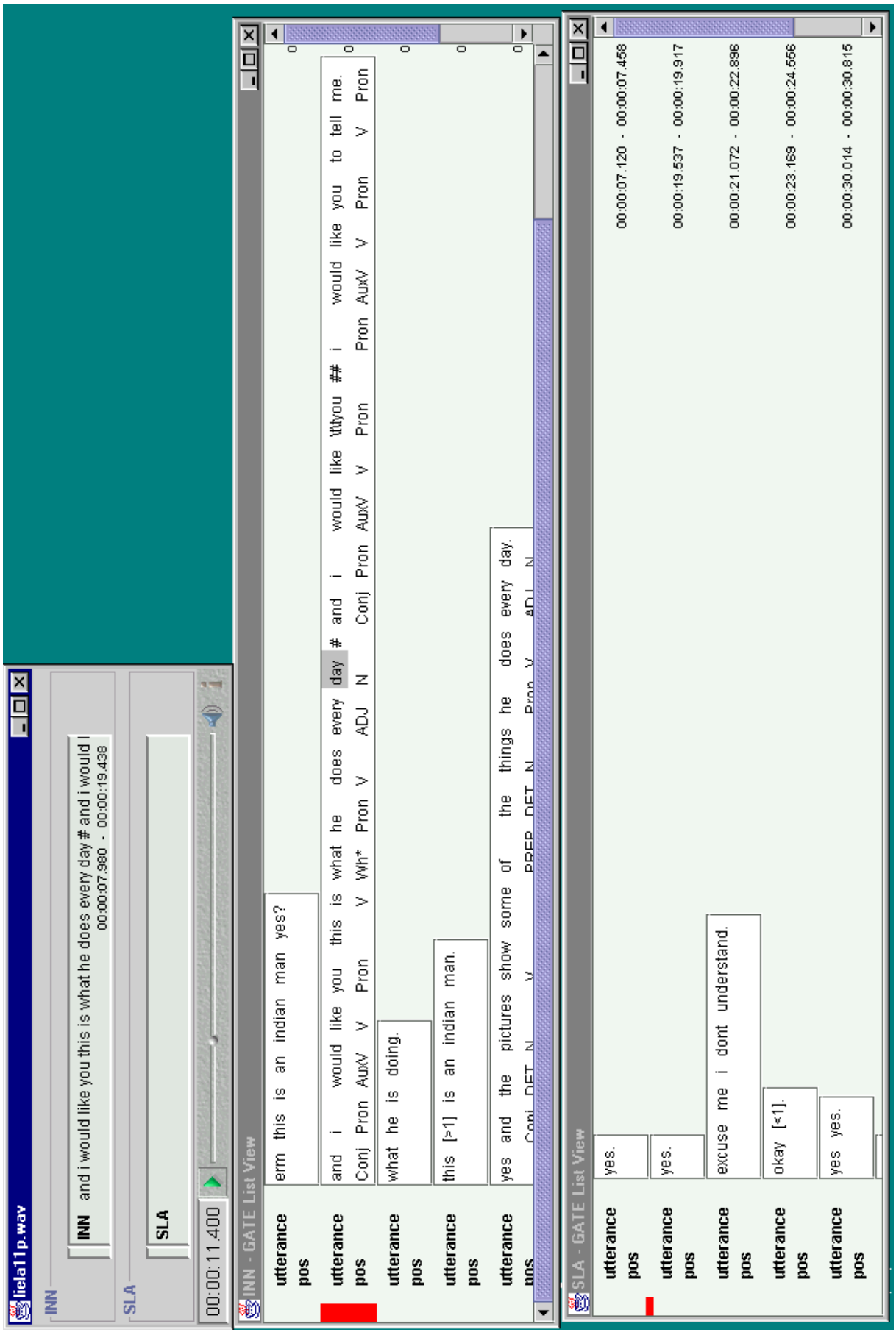

Figure 9.4: Displaying annotation in sync with speech

> suffering respiratory problems due to inhalation of chemicals; reporting the progress and dispersal of the cloud.
>
> How do these services communicate, plan and coordinate their activities?
>
> A combination of recent advances in language processing technology, coupled with more established geographical and spatial modelling tools, may go a long way to answering these questions in the near future. These advances have made the provision of computer-assisted triage possible. The aim of LOTTIE is to provide proof-of-concept by implementing demonstration software that deals with the major technological problems involved.
>
> Returning to the example scenario above, imagine further that the three countries concerned assign central controllers to coordinate emergency service operations. These controllers will rapidly receive hundreds of messages from staff on the ground, detailing, for example, their locations and the condition and number of patients, and requesting allocation to a suitable hospital.
>
> As part of the LOTTIE triage system, the emergency staff are equipped with speech recognition systems that allow them to send voice messages to the controllers' computer systems, where they are converted to text (possibly with human checking; this checking can be unskilled, however, and quicker than manual data entry).
>
> The controllers need to process the information in these messages as quickly as possible, and need to be able to search out subsets of recent messages according to criteria such as the location of the paramedics, or type of symptoms reported, etc. The system extracts the most salient information from them, allowing their display in an easy-to-assimilate form.
>
> For example, locations are converted into grid references and can be fed to map displays. Numbers of patients and injury severity are also recognised and can be displayed visually for rapid assimilation.

(From `http://gate.ac.uk/demos/lottie/lottie.html`.)

Figure 9.5[5] shows the LOTTIE main window, which is divided into four regions:

- the query field (top);

[5]LOTTIE was sigificantly enhanced and ported to the Java Web browser plugin by Valentin Tablan.

- the document list (left side, under the query field);
- the text panel (right side, under the query field);
- the document entities list (under the text panel).

The text panel is used to display the contents of the currently-selected document. The document list shows all the documents forming the opened collection. The list can also be used to switch between documents. The query field can be used to enter a word or phrase to be found within all the documents in the collection. All occurrences of the word are highlighted with the chosen colour (red by default). The document entities list shows all the text segments that are semantically marked-up in the current document. The set of entities changes dynamically during various operation over the text. For example, if the user enters a query, all the occurrences of the sought word are marked-up as entities of type 'query'.

Figure 9.6 shows how a map viewer appears when a user right clicks on a location entity (in this case 'Mulhouse').

Figure 9.7 shows a user making a query on the word 'air'. Queries can also be forwarded to Web search engines via a pop-up menu that is active over highlighted entities; figure 9.8 shows the results of querying Altavista on the word 'Bamlach'.

The components in LOTTIE are a mixture of GATE2 facilities (such as annotation viewers), external PRs (performing Named Entity recognition) and application-specific code (for showing maps, making queries to search engines, and so on). Note that the reuse of GATE viewers in application-specific user interfaces is a new feature with version 2.

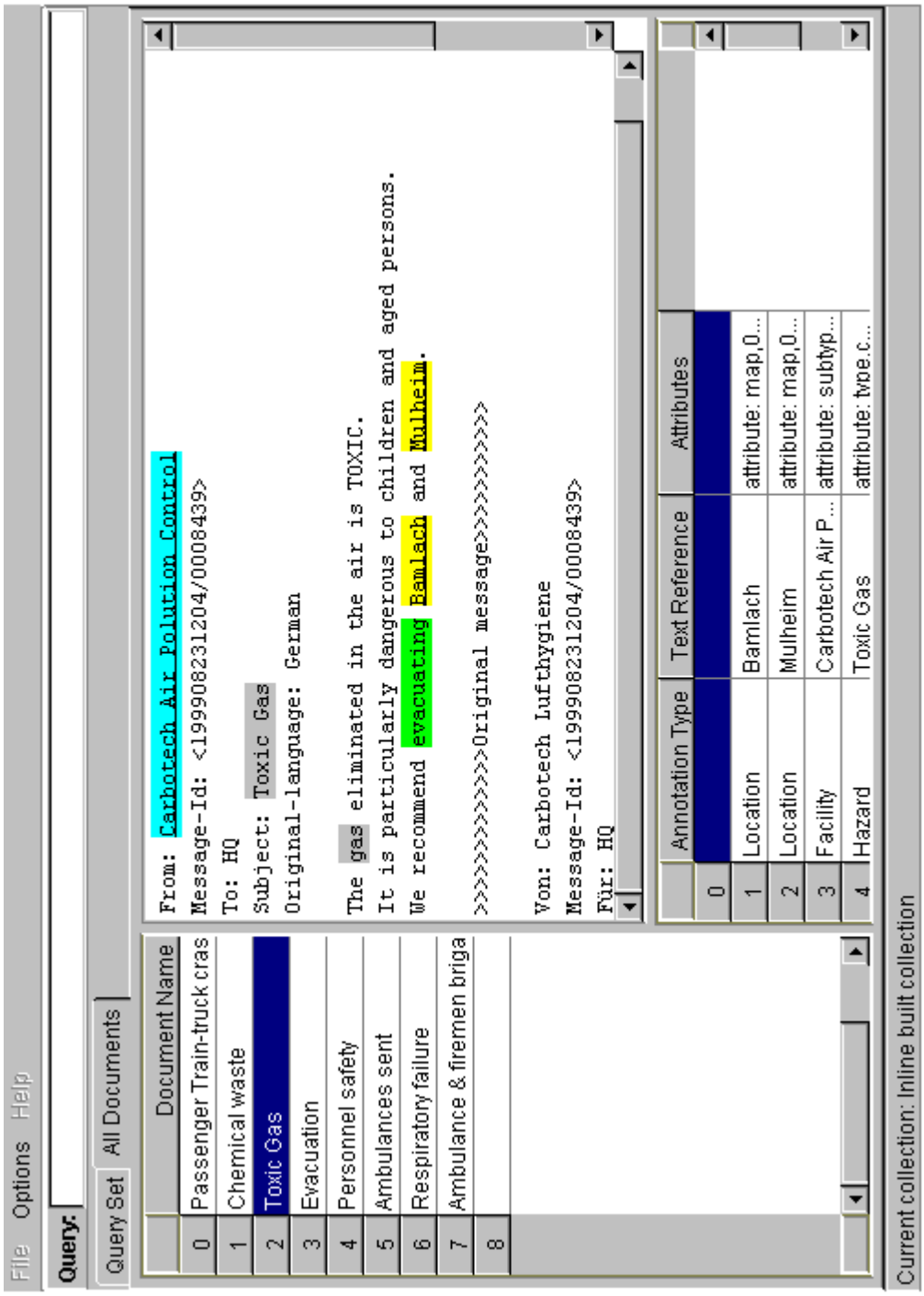

Figure 9.5: LOTTIE (1) – highlighted Named Entities

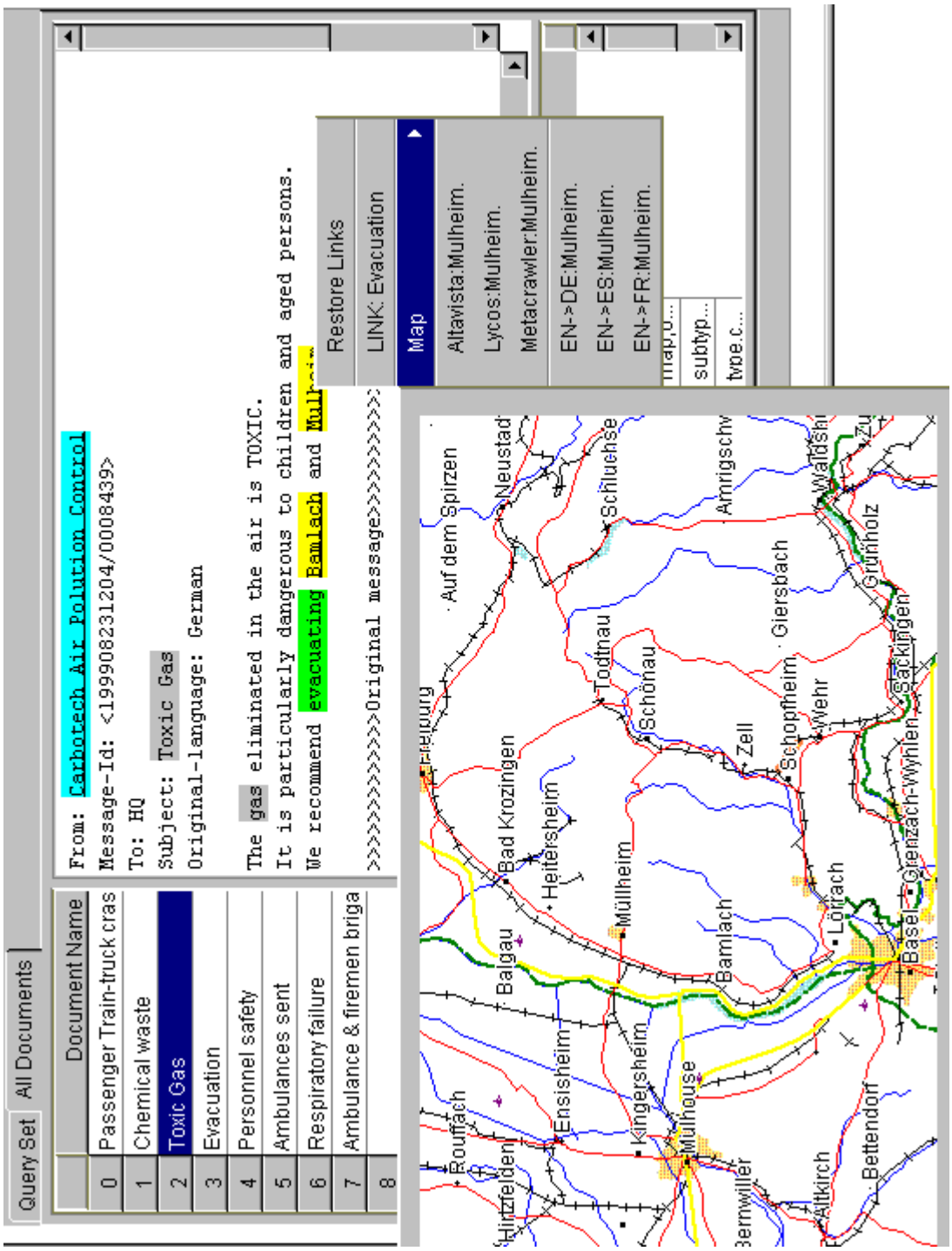

Figure 9.6: LOTTIE (2) – looking at a map of a location entity

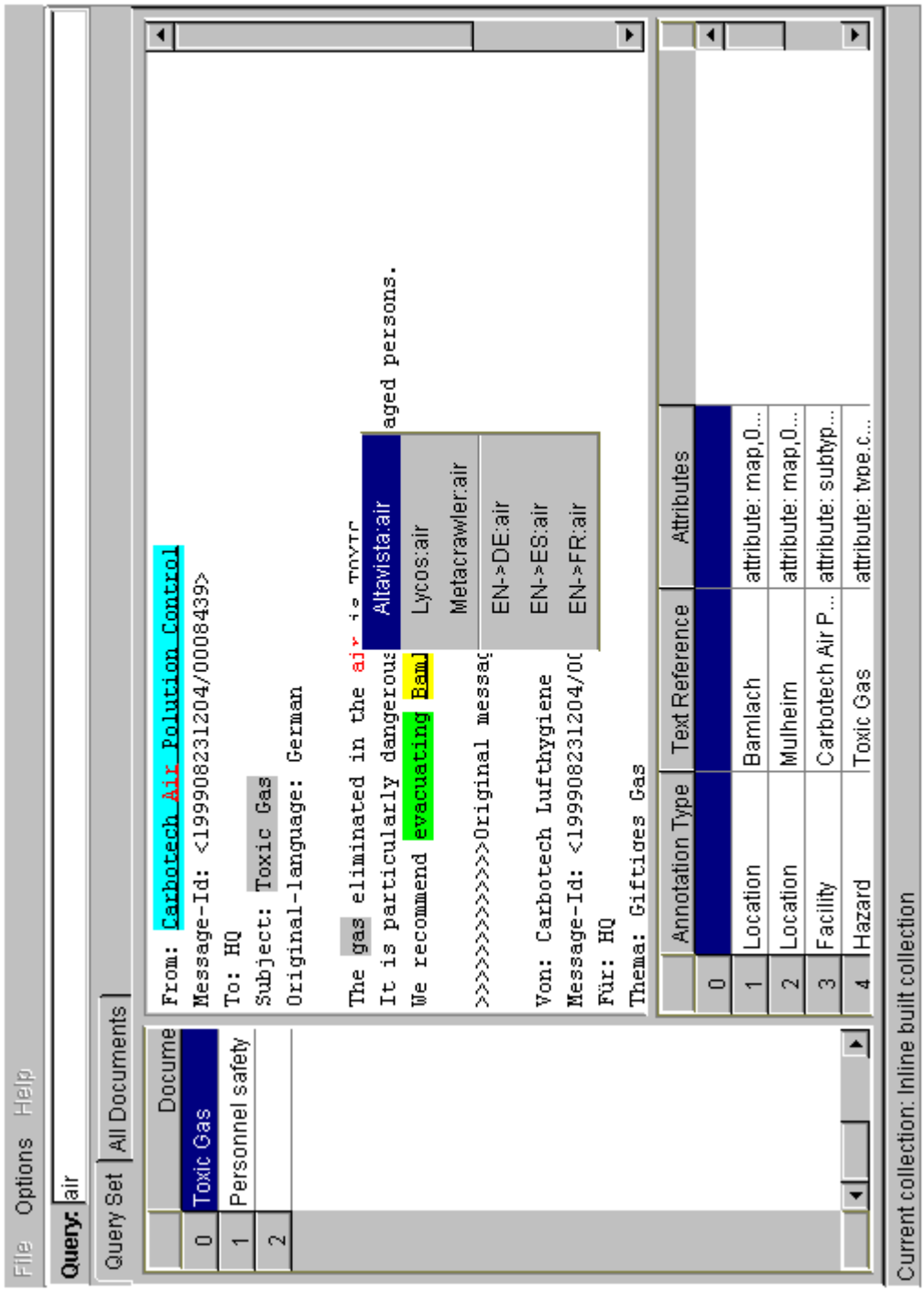

Figure 9.7: LOTTIE (3) – making a query

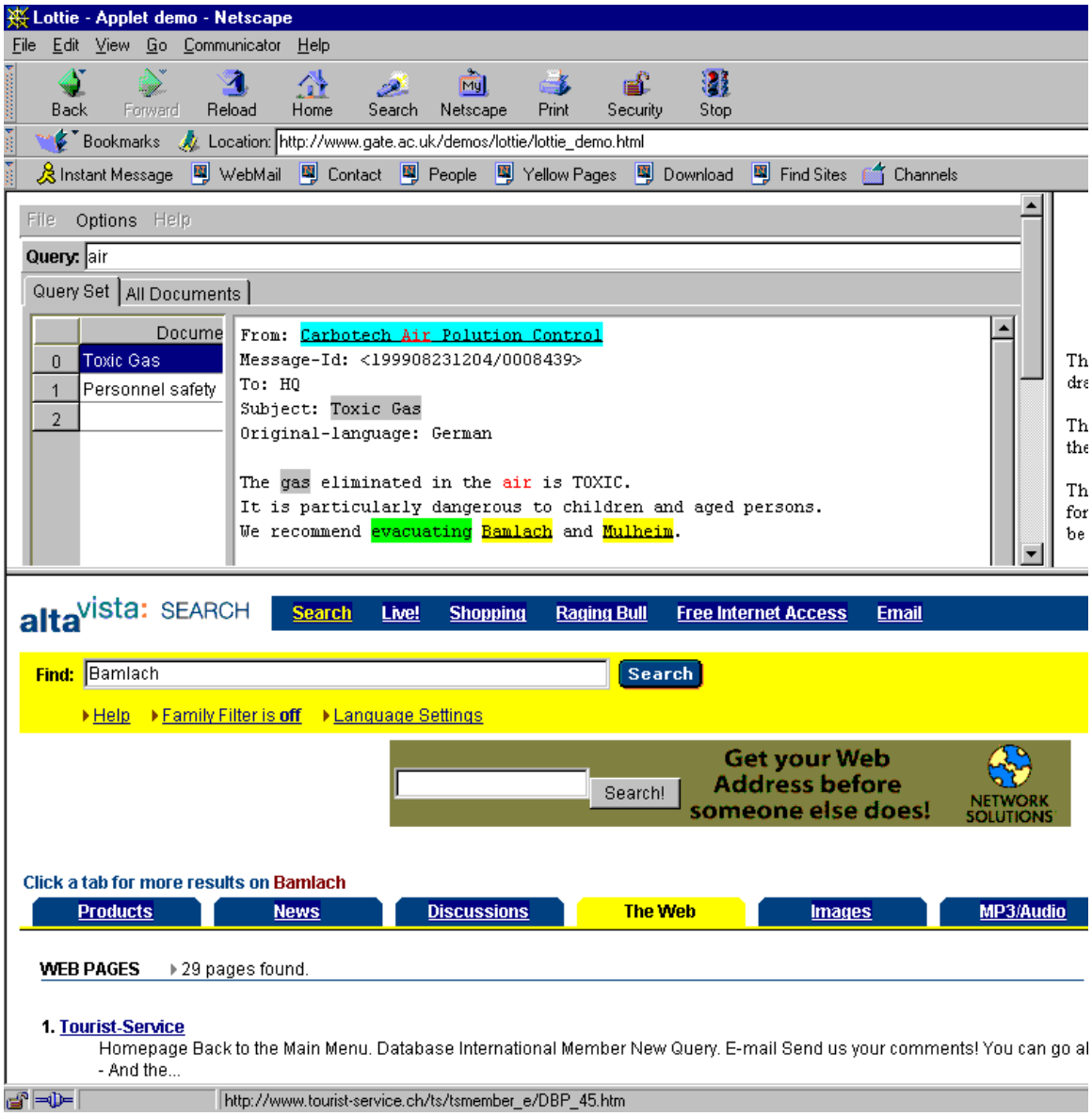

Figure 9.8: LOTTIE (4) – query results from Altavista

9.3 Design Goals

This section describes the design goals of GATE2, which parallel the use cases of chapter 6.

Goal 1: To support LE R&D workers producing software and performing experiments.

As noted in chapter 2, the practitioner group in LE is diverse, and, to achieve generality, a SALE must cater for a wide range of abilities and interests, including expert programmers who may extend the system core, and non-programmers who merely wish to examine data or run programs written by others. GATE addresses this diversity by providing multiple 'entry points' to the system. The development environment is intended to be usable by non-experts (GATE2 improves ease-of-use over the original version by adopting common user interface conventions such as left-click select, right-click action, and so on). The framework has a public interface which provides simple abstractions of LE entities and allows extensibility of the resource set with minimal overhead to the client programmer. For those who need to get under the hood (and who have the requisite skills), the full system source code and developer documentation is available.

To cater for different data storage needs several underlying relational databases are supported, including Oracle (which is expensive and difficult to administer, but fast) and a free database (which is slower).

Goal 2: To document, maintain and support the architecture.

GATE2 is documented at `http://gate.ac.uk/`, which intended to become a *Web community* [Greenspun 99]: an interactive site with facilities for discussion lists, visitor comments and user-tailored pages. We use both `javadoc` and UML to document the code and the design model. The system is 'open-source' and will be distributed under the GNU licence, and it is hoped that the user community will come to contribute to the maintenance, support and development of the system, as happens with other free software.

It is also hoped that facilities for data storage and processing will be made available on `gate.ac.uk` for those who want to take advantage of a centrally-managed database and server environment, for example other UK academic sites with heavy corpus-crunching requirements.

Goal 3: To allow the use of the architecture in and for different languages.

GATE2 is based on Java, which uses Unicode [The Unicode Consortium 96] to represent characters. Most languages can be represented in Unicode, but difficulties remain relating to the conversion of other existing character sets into the format, and in the omission of certain characters from the model [McEnery *et al.* 99]. The EMILLE project referred to above aims to address these problems in the GATE2 context.

Goal 4: To promote good software engineering in LE development.

In one sense GATE2 exists in order to make it *less* necessary for language processing workers to worry about software engineering by taking over responsibility for tasks like data storage. Nevertheless, the GATE2 development team uses and promotes practical small-group engineering methods, as discussed in chapter 3. We use explicit design, iterative development, version control and regression testing. Almost all the tools used are free (the exceptions being the UML modelling system and the Java development environment used; in the latter case free alternatives are available).

Goal 5: To exploit the benefits of frameworks.

The GATE2 framework has a number of features to make it easy to use in diverse client programs, including:

1. an interface-based design;
2. use of well-known Java patterns and idioms.

Point 1 refers to the *interface* facility of the Java language which allows the representation of types by a collection of public methods, not including constructors [Arnold & Gosling 98], hence isolating client code from the implementation of types. The public interface of the GATE2 framework often hides the construction of objects and obliges client code to use interface definitions. This means that the framework can supply different instantiations of resources depending on the underlying storage mechanism in operation, or the machine which a process is running on, or the control algorithm under which processes are to be executed.

Point 2 refers to the framework's use of the extensive Java standards and libraries that cater for tasks such as collection management, distributed processing and database access, to name but a few. Using these libraries means that programmers using the framework have less system-specific patterns to learn, and may already be familiar with the mechanisms deployed. For example, instead of the custom-designed arrangements for managing collections

of objects in the TIPSTER architecture, GATE2 classes inherit from the Java collections library (corpora are sets of documents, and behave just like other sets).

Goal 6: To discover components at runtime, load and initialise them.

GATE2 starts up with a URL list, and searches it for components, reading the metadata for each one and loading the component accordingly.

Goal 7: To allow the building of systems from sets of components.

The GATE2 framework includes abstractions for both Language Resources and Processing Resources, and also for the data visualisation and editing components associated with them. In each case the resources are expected to live outside of the GATE2 codebase and be loaded dynamically at run-time. The selection of a set of components for an application may then be made in a simple fashion, both when using the development environment and when programming with the framework.

Goal 8: To allow the construction of systems based on components residing on different host computers.

GATE2 defines a `ProcessingEngine` class, which is responsible for the construction of `ProcessingResource` objects. PRs are accessed by the clients of the framework via interfaces, and GATE2 will supply proxy objects that use the RMI protocol to execute on a remote server while still appearing identical to a local PR.

Goal 9: To allow asynchronous execution of processing components.

The model currently adopted is that each PR may run in its own thread if asynchronous processing is required (by default PRs will be executed serially in a single thread). The set of LRs being manipulated by a group of multi-threaded PRs must be synchronised (i.e. all their methods must have locks associated with whichever thread is calling them at a particular point). Synchronisation of LRs is performed in a manner similar to the Java collections framework. This arrangement allows the PRs to share data safely. Responsibility for the semantics of the interleaving of data access (who has to write what in what sequence in order for the system to succeed) is a matter for the client, however.

Goal 10: To allow the association of structured data with LR and PR components.

The language used for component metadata has not been selected; under evaluation are

RDF [Lassila & Swick 99, Berners-Lee *et al.* 99], plain XML as used in the EUDICO corpus browsing system [Brugman *et al.* 98a], and Java properties.

Goal 11: To factor out commonalities between related components.

The component model given in section 4.3 will be available in the GATE2 framework. This model is based on inheritance: a parser is a type of language analyser which is a type of processing resource. The client can choose, therefore, between implementing a more specific interface and adhering to the choices made by the GATE2 developers for that type, or implementing a more general interface and making their own choices about the specifics of their particular resource.

Goal 12: To provide uniform, simple methods for accessing data components.

The life-cycle of all types of LR within GATE2 is the same. For example, an instance of WordNet may be created by constructing a `DataStore` object from a reference to the URL of a database that stores the thesaurus data (the system communicates with the database using the JDBC protocol, which provides both distribution and vendor independence). The client then asks the `DataStore` for a WordNet object, which is returned as a `Thesaurus` interface. The client then queries this interface for the data it requires (it may also make SQL queries on the object if required). If the client has permission to update the WordNet data, they can call mutator methods on the interface, then ask the `DataStore` to make the changes permanent, or discard them as necessary.

Goal 13: To manage (possibly very large) collections of documents in an efficient manner.

To cater for large LR data sets GATE2 uses a relational database model with several optional underlying implementations. For optimal efficiency, sites with access to Oracle can store LRs there; for reduced cost, free alternatives are also provided.

Goal 14: To allow use of documents of various formats without knowledge of those formats.

GATE2 separates the treatment of documents from their formats. Any document may be annotated without concern for the format of its source, and multiple input formats are supported by various filter components. Figure 9.9 gives a UML diagram of the GATE2 document model.

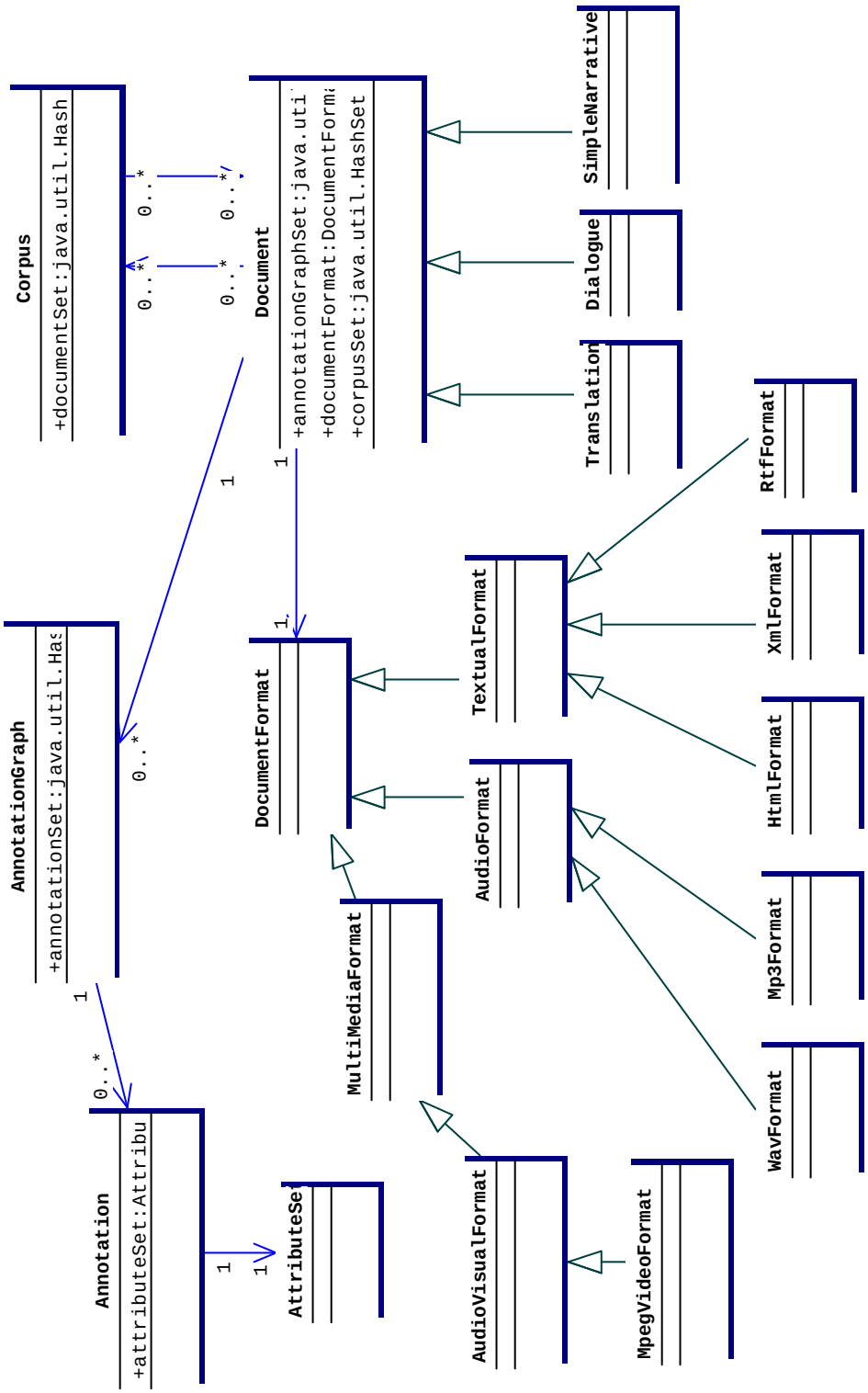

Figure 9.9: Documents etc.

Goal 15: To support theory-neutral format-independent annotation of documents.

GATE2 has a new model of annotations, intended to incorporate the best of both the TIPSTER world and the work done at the Linguistic Data Consortium (see section 5.3.3). The main changes from GATE 1 are:

- the introduction of a `Node` type, which is the bearer of the reference offsets into the underlying document content (text or audiovisual time-line); annotations now have nodes, not offsets (i.e. they now behave more like arcs in a graph);
- an annotation only has two nodes, which means that the multiple-span annotations of TIPSTER are no longer supported (the workaround is to store non-contiguous data structures as features of the document, and point from there to the multiple annotations that make up the structures);
- documents may now have multiple annotation sets.

Goal 16: To support creation and maintenance of LRs that describe language.

This design goal can probably only be partially met. The problem is that whereas language data LRs share much structure in common (they are made up of corpora, which are collections of annotated documents), LRs such as grammars and lexicons may be structurally complex and require special tools for their creation and maintenance. For example, an HPSG grammar of any size would be very difficult to write without a comprehensive unification-based environment such as ALEP [Simkins 94] or ALE [Carpenter & Penn 92]. In these cases, the LE developer is probably best served by tools specific to the individual LRs in question, such as grammar development environments [Netter & Pianesi 97]. In other cases there is less of a problem providing support within GATE. For example, the WordNet thesaurus is available in a simple relational format that can be stored in a database in a straightforward manner, and made available from the GATE2 framework relatively easily. In these cases GATE2 plans to provide integration routes for as many LRs as possible. Often there will be a licensing problem that will prevent us from distributing the data; instead we will distribute code to map the data into a relational form, whence it can be accessed using GATE mechanisms.

Goal 17: To cater for indexing and retrieval of diverse data structures.

The TIPSTER document management architecture uses a single attribute/value data repre-

sentation to associate information with corpora, documents and ranges of text. One of the benefits of this arrangement is that it is easier to provide a common indexing mechanism to support searching. GATE2 plans to extend this idea to other types of LRs.

Goal 18: To provide a library of well-known algorithms over native data structures.

Where LE methods are well-known, GATE2 aims to provide baseline implementations of the algorithms running over the GATE2 data structures. For example, the Markov chain algorithms described in section 4.2.1 are a candidate for inclusion (probably via the integration of a 3rd-party toolkit such as [Rosenfeld 95]). Unification over attribute-value matrices is another candidate; finite state transduction over annotations is already a facility – see appendix A.

Goal 19: To provide simple methods for comparing data structures.

The evaluation of LE systems frequently involves marking up LRs with the results which a human being would produce for a certain task and then measuring the difference between this and the results that a system produces on the same material. The measurement process can be complex but in other cases quite simple measures leading to figures such as precision and recall can suffice. GATE2 will provide comparison mechanisms for data such as annotations sufficient for these latter cases.

Goal 20: All data structures native to the architecture should be persistent.

The GATE2 persistence mechanism is designed around the following principles:

1. Multi-user access: Several users should be able to work simultaneously on the same LRs – documents, corpora, etc. Deadlocks and data integrity will be handled with the help of transactions. User authentication and accessibility of LRs should be dealt with.

2. API Simplicity: It needs to be kept as small and intuitive as possible. SQL statements, transactions, commits and rollbacks must be hidden from the API user.

3. Extensibility: The persistence architecture must offer third-party extensibility. In other words, programmers must be able to add their own LR types and have them saved via the same mechanism as that by which our LRs are saved. It is acceptable for them to have to explicitly use SQL, deal with commits and rollbacks, and define/alter tables.

4. Data exchange: LRs should be movable from one machine to another. Since documents

are identified by URLs (which might be machine-dependent), it should be possible for them to be internalised, i.e. their content to be copied and stored inside the DBMS.

5. Efficient data access: Storing and retrieving data has to be efficient (but this should not come at the expense of over-complicating the API or compromising other design goals).

6. Data rollback: Users should be able to rollback the objects to a previous, marked state. For instance, when three PRs have been run consecutively, the user must be able to either save the state, or rollback the data undoing the work of one, two, or all three modules. Similarly, no work should finally be saved in the database until a user explicitly asks for 'save'.

7. Remote Data Access: Client processes should be able to access and manipulate the persistent objects remotely. This also entails some means of locating or configuring remote resources on a per site/user basis.

Goal 21: To allow the use of the framework in diverse contexts.

Embedding the GATE2 framework in a Java application involves the placing of a single file (`gate.jar`) in a system library directory. Embedding in applications based on other languages is more complex, but is known to be relatively straightforward for at least C, C++, Tcl and Perl. It is our intention to provide hooks for calling the system from Microsoft Word macros, from Emacs LISP and as a Web browser plugin, development resources permitting.

Goal 22: To enable data import from and export to other infrastructures and embedding of components in other environments.

GATE2 provides input/output to/from XML as a means for exchanging data with other environments.

Goal 23: To provide graphical manipulation of LE data structures.

Each LR type is associated with one or more visualisation components. For example, annotations of various types are associated with components for displaying trees, single-range structures such as part-of-speech tags, and multi-range structures such as coreference chains. Where appropriate, the display components also allow editing (see for example the syntax tree viewer/editor at `http://gate.ac.uk/demos/`).

Goal 24: To give access to all the framework and architectural services and support development of LE experiments and applications.

GATE2 will duplicate most of the graphical facilities of GATE version 1, as described in chapter 7. The task graph interface, and the autogeneration of paths through PR sets, however, will not be implemented, because the resulting complexity and scalability problems outweigh their benefits.

9.4 Conclusion

In part I we discussed the nature of Language Engineering, and saw how Software Architecture has the potential to reduce R&D overheads in the field. In part II we described LE more concretely, from a number of different perspectives, and gave a classification and critical review of the various issues that previous work on SALE has covered. In part III we refined the structure derived in part II into a detailed requirements analysis for SALE, and described our own implementation, the GATE system. In part IV we measured the extent to which GATE has succeeded in the meeting its goals, and discussed the present and future of the system.

As noted in chapter 1, we have taken two novel steps: defining SALE as a field and mapping out the objectives of SALE systems, and presenting software that validates a significant number of these objectives.

Part V

Addenda

Appendix A

Java Annotation Patterns Engine Specification

This specification describes a version of CPSL, the Common Pattern Specification Language, which provides finite state transduction over TIPSTER annotations based on regular expressions. The best description of this language is in Doug Appelt's TextPro manual: `http://www.ai.sri.com/~appelt/TextPro`.

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.

Section A.1 gives a formal definition of the JAPE grammar, and some examples of its use. Section A.2 describes JAPE's relation to CPSL. Section A.3 discusses rule application strategies.

A.1 Grammar of JAPE

JAPE is similar to CPSL with a few exceptions. Figure A.1 gives a BNF[1] 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.

[1] Backus-Naur Format.

```
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 A.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'.

A.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 A.3.
5. Expressions relating to labels unbound on the LHS are not evaluated on the RHS. (In TextPro they evaluate to "false".)
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.

A.3 Rule Application Algorithms

TextPro implements the usual FSA lexical analysis rule application strategy, plus the CPSL priority mechanism. So it looks for the longest applicable rule; if there are several, it checks priorities; if there are still several it goes for the first in the input file. JAPE also implements this strategy, but it also has an alternative 'find all' strategy, where every rule is applied everywhere in the document (this is like Brill's algorithm, or the 'g' option in, for example, `sed`). (To trigger one strategy or another, use an 'Option: appelt' or 'brill" line in your grammar.)

The rule application algorithm that mimics Appelt's looks like this:

1. for each document position (starting at 0):
   (a) apply all rule LHSs and collect the set that succeed;
   (b) select one of these according to length/priority/file position;
   (c) apply RHS of selected rule, and advance position according to the spans of the annotations consumed by the LHS;
   (d) if no rules matched, advance position to next possible match point.

This implementation of Appelt's style is potentially expensive where rules share common prefixes, as all these will be evaluated individually.

The 'find all' variant:

1. for each rule:
   (a) position = 0
   (b) if LHS matches then apply the RHS and advance position accordingly
   (c) else advance position to next possible match point.

Bibliography

[Abelson *et al.* 85]

H. Abelson, G. Sussman, and J. Sussman. *The Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, Mass., 1985.

[ACL 97]

ACL. *Proceedings of the Fifth Conference on Applied Natural Language Processing (ANLP-97).* Association for Computational Linguistics, 1997.

[Ainsworth 88]

W. Ainsworth. *Speech Recognition by Machine.* Peter Peregrinus / IEE, London, 1988.

[Allen 95]

J. Allen. *Natural Language Understanding.* Benjamin/Cummings, Redwood City, CA, 2 edition, 1995.

[Alshawi 92]

H. Alshawi, editor. *The Core Language Engine.* MIT Press, Cambridge MA, 1992.

[Amsler & White 79]

R. A. Amsler and J. S. White. Development of a Computational Methodology for Deriving Natural Language Semantic Structures via Analysis of Machine-readable Dictionaries. Technical Report MCS77-01315, NSF, 1979.

[Amsler 80]

R. A. Amsler. The Structure of the Merriam-Webster Pocket Dictionary. Technical Report TR-164, University of Texas at Austin, 1980.

[Amsler 81]

R. A. Amsler. A Taxonomy of English Nouns and Verbs. In *Proceedings of the 19th*

Annual Meeting of the Association for Computational Linguistics, pages 133–138, Stanford, CA, 1981.

[Amtrup 95]

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

[Amtrup 99]

J. Amtrup. Architecture of the Shiraz Machine Translation System. `http://crl.nmsu.edu/shiraz/archi.html`, 1999.

[Appelt 86]

D. E. Appelt. Planning English referring expressions. In B. L. W. B. Grosz, K. S. Jones, editor, *Readings in Natural Language Processing*, pages 501 – 517. Morgan Kaufmann, California, 1986.

[Appelt 99]

D. Appelt. An Introduction to Information Extraction. *Artificial Intelligence Communications*, 12(3):161–172, 1999.

[Arnold & Gosling 98]

K. Arnold and J. Gosling. *The Java Programming Language, Second Edition.* Addison-Wesley, Reading, MA, 1998.

[ARPA 95]

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

[ARPA 96]

Advanced Research Projects Agency. *Proceedings of the TIPSTER Text Program (Phase II)*. Morgan Kaufmann, California, 1996.

[Azzam *et al.* 97a]

S. Azzam, K. Humphreys, R. Gaizauskas, H. Cunningham, and Y. Wilks. A Design for Multilingual Information Extraction (poster). In *IJCAI-97*, 1997.

[Azzam *et al.* 97b]

S. Azzam, K. Humphreys, R. Gaizauskas, H. Cunningham, and Y. Wilks. Using a Language Independent Domain Model for Multilingual Information Extraction. In

C. Spryrodopoulos, editor, *Proceedings of the IJCAI-97 Workshop on Multilinguality in the Software Industry: the AI Contribution (MULSAIC-97)*, 1997.

[Azzam *et al.* 98a]

S. Azzam, K. Humphreys, and R. Gaizauskas. Coreference Resolution in a Multilingual Information Extraction System. In *Linguistic Coreference Workshop, in First International Conference on Language Resources and Evaluation, Granada, Spain*, 1998.

[Azzam *et al.* 98b]

S. Azzam, K. Humphreys, and R. Gaizauskas. Evaluating a Focus-Based Approach to Anaphora Resolution. In *Proceedings of COLING-ACL'98*, pages 74–78, 1998.

[Azzam *et al.* 99]

S. Azzam, K. Humphreys, and R. Gaizauskas. Using a Language Independent Domain Model for Multilingual Information Extraction. *Journal of Applied Artificial Intelligence*, 1999.

[Barr & Feigenbaum 81]

A. Barr and E. Feigenbaum. *The Handbook of Artificial Intelligence Vol. I.* Pitman, London, 1981.

[Bass *et al.* 97]

L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice.* Addison-Wesley, Reading, MA, 1997.

[Bateman 90]

J. A. Bateman. Upper Modeling: organizing knowledge for natural language processing. In *5th. International Workshop on Natural Language Generation, 3-6 June 1990*, Pittsburgh, PA., 1990.

[Berners-Lee *et al.* 99]

T. Berners-Lee, D. Connolly, and R. Swick. Web Architecture: Describing and Exchanging Data. Technical report, W3C Consortium, `http://www.w3.org/-1999/04/WebData`, 1999.

[Bikel *et al.* 97]

D. Bikel, S. Miller, R. Schwartz, and R. Weischedel. Nymble: a High-Performance

Learning Name-finder. In *Proceedings of the Fifth conference on Applied Natural Language Processing*, 1997.

[Bikel *et al.* 99]

D. Bikel, R. Schwartz, and R. Weischedel. An Algorithm that Learns What's in a Name. *Machine Learning, Special Issue on Natural Language Learning*, 34(1-3), Feb. 1999.

[Binot & Jensen 87]

J. Binot and K. Jensen. A Semantic Expert Using an Online Standard Dictionary. In *Proceedings of the 10th International Joint Conference on Artificial Intelligence (IJCAI-87)*, pages 709–714, Milan, Italy, 1987.

[Bird & Liberman 98]

S. Bird and M. Liberman. Towards a Formal Framework for Linguistic Annotation. In *Proceedings of the ICLSP*, Sydney, December 1998.

[Bird & Liberman 99a]

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`.

[Bird & Liberman 99b]

S. Bird and M. Liberman. Annotation graphs as a framework for multidimensional linguistic data analysis. In *Towards Standards and Tools for Discourse Tagging, Proceedings of the Workshop. ACL-99*, pages 1–10, 1999.

[Bird *et al.* 00]

S. Bird, D. Day, J. Garofolo, J. Henderson, C. Laprun, and M. Liberman. ATLAS: A flexible and extensible architecture for linguistic annotation. In *Proceedings of the Second International Conference on Language Resources and Evaluation*, Athens, 2000.

[Bod 95]

R. Bod. *Enriching Linguistics with Statistics: Performance Models of Natural Language*. Institute for Logic, Language and Computation, University of Amsterdam, 1995.

[Bod 96]

R. Bod. Two Questions about Data-Oriented Parsing. In *Proceedings of the Fourth Workshop on Very Large Corpora*, Copenhagen, 1996.

[Boguraev & Briscoe 87]

B. Boguraev and T. Briscoe. Large Lexicons for Natural Language Processing: Exploring the Grammar Coding System of LDOCE. *Computational Linguistics*, 13, 1987.

[Boguraev *et al.* 87]

B. K. Boguraev, T. Briscoe, J. Carroll, D. Carter, and C. Grover. The Derivation of a Grammatically Indexed Lexicon from the Longman Dictionary of Contemporary English. In *Proceedings of the 25th Annual Meeting of the ACL*, pages 193–200, Stanford, CA, 1987.

[Boguraev *et al.* 95]

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

[Boitet & Seligman 94]

C. Boitet and M. Seligman. The "Whiteboard" Architecture: A Way to Integrate Heterogeneous Components of NLP Systems. In *Proceedings of COLING '94*, pages 426–430, Kyoto, Japan, 1994.

[Booch 94]

G. Booch. *Object-Oriented Analysis and Design 2nd Edn.* Benjamin/Cummings, 1994.

[Booch *et al.* 99]

G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modelling Language User Guide*. Addison-Wesley, Reading, MA, 1999.

[Bos *et al.* 98]

J. Bos, C. Rupp, B. Buschbeck-Wolf, and M. Dorna. Managing information at linguistic interfaces. In *Proceedings of the 36th ACL and the 17th COLING (ACL-COLING '98)*, pages 160–166, Montreal, 1998.

[Brew *et al.* 99]

C. Brew, D. McKelvie, R. Tobin, H. Thompson, and A. Mikheev. *The XML Library*

LT XML version 1.1 User documentation and reference guide. Language Technology Group, Edinburgh, 1999. `http://www.ltg.ed.ac.uk/`.

[Brill 92a]

E. Brill. A simple rule-based part-of-speech tagger. In *Proceedings of the Third Conference on Applied Natural Language Processing*, Trento, Italy, 1992.

[Brill 92b]

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

[Brill 95]

E. Brill. Transformation-Based Error-Driven Learning and Natural Language. *Computational Linguistics*, 21(4), December 1995.

[Brown 89]

A. Brown. *Database Support for Software Engineering*. Kogan Page, 1989.

[Brown *et al.* 90]

P. Brown, J. Cocke, S. D. Pietra, V. D. Pietra, F. Jelinek, J. Lafferty, R. Mercer, and P. Roossin. A Statistical Approach to Machine Translation. *Computational Linguistics*, 16(June):79–85, 1990.

[Bruce & Guthrie 92]

R. Bruce and L. Guthrie. An Automatically Generated Semantic Hierarchy. In *Proceedings of COLING '92*, Nantes, 1992.

[Brugman *et al.* 98a]

H. Brugman, A. Russel, P. Wittenburg, and R. Piepenbrock. Corpus-based Research using the Internet. In *Workshop on Distributing and Accessing Linguistic Resources*, pages 8–15, Granada, Spain, 1998. `http://www.dcs.shef.ac.uk/~hamish/dalr/`.

[Brugman *et al.* 98b]

H. Brugman, H. Russel, and P. Wittenburg. An infrastructure for collaboratively building and using multimedia corpora in the humaniora. In *Proceedings of the ED-MEDIA/ED-TELECOM Conference*, Freiburg, 1998.

[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.

[Buckley 98]

C. Buckley. TIPSTER Advanced Query (DN2). TIPSTER programme working paper, 1998.

[Burnage & Dunlop 92]

G. Burnage and D. Dunlop. Encoding the British National Corpus. In *Proceedings of the 13th International Conference on English Language Research on Computerised Corpora*, 1992.

[Burnard 95]

L. Burnard. Users Reference Guide for the British National Corpus. `http://info.ox.ac.uk/bnc/`, May 1995.

[Burnett *et al.* 87]

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

[Busemann 99]

S. Busemann. Constraint-Based Techniques for Interfacing Software Modules. 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.

[Cahill & Reape 98]

L. Cahill and M. Reape. Component Tasks in Applied NLG Systems. RAGS deliverable, `http://www.tri.bbrighton.ac.uk/projects/rags/`, 1998.

[Cahill *et al.* 99a]

L. Cahill, C. Doran, R. Evans, C. Mellish, D. Paiva, M. Reape, D. Scott, and N. Tipper. Towards a Reference Architecture for Natural Language Generation Systems. Technical Report ITRI-99-14; HCRC/TR-102, University of Edinburgh and Information Technology Research Institute, Edinburgh and Brighton, 1999.

[Cahill *et al.* 99b]

L. Cahill, C. Doran, R. Evans, D. Paiva, D. Scott, C. Mellish, and M. Reape. Achieving Theory-Neutrality in Reference Architectures for NLP: To What Extent

is it Possible/Desirable? 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.

[Cahoon & McKinley 96]

B. Cahoon and K. McKinley. Performance Evaluation of a Distributed Architecture for Information Retrieval. In *Proceedings of SIGIR '96*, pages 110–118, Zurich, 1996.

[Cardie 97]

C. Cardie. Empirical Methods in Information Extraction. *AI Magazine*, 18(4), 1997.

[Carpenter & Penn 92]

B. Carpenter and G. Penn. ALE 2.0 User's Guide. Technical report, Carnegie Mellon University Laboratory for Computational Linguistics, Pittsburgh, 1992.

[Cassidy & Bird 00]

S. Cassidy and S. Bird. Querying databases of annotated speech. In *Eleventh Australasian Database Conference*, Australian National University, Canberra, 2000.

[CEC 96]

CEC. *Telematics Applications Programme, Language Engineering Project Directory.* European Commission Directorate General XIII, Luxembourg, 1996.

[Charniak 93]

E. Charniak. *Statistical Language Learning.* MIT Press, Cambridge MA, 1993.

[Cheong *et al.* 94]

T. Cheong, A. Kwang, A. Gunawan, G. Loo, L. Qwun, and S. Leng. A Pragmatic Information Extraction Architecture for the Message Formatting Export (MFE) System. In *Proceedings of the 2nd Singapore Conference on Intelligent Systems (SPICIS '94)*, pages B371–B377, Singapore, 1994.

[Chodorow *et al.* 85]

M. S. Chodorow, R. J. Byrd, and G. E. Heidorn. Extracting Semantic Hierarchies from a Large On-Line Dictionary. In *Proceedings of the 23rd Annual Meeting of the ACL*, pages 299–304, Chicago, IL, 1985.

[Christ 94]

O. Christ. A Modular and Flexible Architecture for an Integrated Corpus Query

System. In *Proceedings of the 3rd Conference on Computational Lexicography and Text Research (COMPLEX '94)*, Budapest, 1994. `http://xxx.lanl.gov/-abs/cs.CL/9408005`.

[Christ 95]

O. Christ. Linking WordNet to a Corpus Query System. In *Proceedings of the Conference on Linguistic Databases*, Groningen, 1995.

[Church & Mercer 93]

K. Church and R. Mercer. Introduction. *Special issue on Computational Linguistics Using Large Corpora, Computational Linguistics*, 19(1), 1993.

[Church & Rau 95]

K. Church and L. Rau. Commercial Applications of Natural Language Processing. *Communications of the ACM*, 38(11), November 1995.

[Clarkson & Rosenfeld 97]

P. Clarkson and R. Rosenfeld. Statistical Language Modeling using the SMU-Cambridge Toolkit. In *Proceedings of ESCA Eurospeech*, Greece, 1997.

[Clements & Northrop 96]

P. Clements and L. Northrop. Software Architecture: An Executive Overview. Technical Report CMU/SEI-96-TR-003, Software Engineering Institute, Carnegie Mellon University, 1996.

[Cockburn 97]

A. Cockburn. Structuring Use Cases with Goals. *Journal of Object-Oriented Programming*, Sept-Oct and Nov-Dec, 1997.

[Cowie & Lehnert 96]

J. Cowie and W. Lehnert. Information Extraction. *Communications of the ACM*, 39(1):80–91, 1996.

[Cox 90]

S. Cox. Hidden Markov Models for Automatic Speech Recognition: Theory and Application. In *Speech and Language Processing*. Chapman and Hall, London, 1990.

[Crouch *et al.* 95]

R. Crouch, R. Gaizauskas, and K. Netter. Interim Report of the Study Group on As-

sessment and Evaluation. Technical report, EAGLES project, Language Engineering Programme, European Commission, 1995.

[Crystal 91]

D. Crystal. *A Dictionary of Linguistics and Phonetics. 3rd Edn.* Blackwell Publishers, Oxford, 1991.

[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 *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.

[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. `http://xxx.lanl.gov/abs/cs.CL/9607025`.

[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.

[Cunningham *et al.* 97b]

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

[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.

[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.

[Cunningham *et al.* 00]

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/`.

[Cutting *et al.* 91]

D. Cutting, J. Pedersen, and P.-K. Halvorsen. An Object-Oriented Architecture for Text Retrieval. In *Proceedings of RIAO '91*, pages 285–298, Barcelona, 1991.

[Davidson 67]

D. Davidson. The logical form of action sentences. In N. Rescher, editor, *The Logic of Decision and Action*. University of Pittsburgh Press, Pittsburgh, 1967.

[Day *et al.* 97]

D. Day, J. Aberdeen, L. Hirschman, R. Kozierok, P. Robinson, and M. Vilain. Mixed-Initiative Development of Language Processing Systems. In *Proceedings of the 5th Conference on Applied NLP Systems (ANLP-97)*, 1997.

[Day *et al.* 98]

D. Day, P. Robinson, M. Vilain, and A. Yeh. MITRE: Description of the *Alembic* System Used for MUC-7. In *Proceedings of the Seventh Message Understanding Conference (MUC-7)*. `http://www.itl.nist.gov/iaui/894.02/-related_projects/muc/index.html`, 1998.

[Devlin 98]

K. Devlin. *Mathematics: The New Golden Age, Second Edition*. Penguin Books, London, 1998.

[DFKI 99]

DFKI. The Natural Language Software Registry. `http://www.dfki.de/-lt/registry/`, 1999.

[Dunning 93]

T. Dunning. Accurate Methods for the Statistics of Surprise and Coincidence. *Computational Linguistics*, 19(1), March 1993.

[EAGLES 99]

EAGLES. EAGLES recommendations. `http://www.ilc.pi.cnr.it/-EAGLES96/browse.html`, 1999.

[Edmondson & Iles 94a]

W. Edmondson and J. Iles. A Non-linear Architecture for Speech and Natural Language Processing. In *Proceedings of International Conference on Spoken Language Processing (ICSLP '94)*, volume 1, pages 29–32, Yokohama, Japan, 1994.

[Edmondson & Iles 94b]

W. Edmondson and J. Iles. Pantome: An architecture for speech and natural language processing. Paper distributed at Dept. of CS Seminar, Sheffield University., 1994.

[Ejerhed & Dagan 96]

E. Ejerhed and I. Dagan, editors. *Proceedings of the Fourth Workshop on Very Large Corpora*, Copenhagen, 1996. Association for Computational Linguistics.

[Eriksson & Gambäck 97]

M. Eriksson and B. Gambäck. SVENSK: A Toolbox of Swedish Language Processing Resources. In *Proceedings of the 2nd Conference on Recent Advances in Natural Langauge Processing (RANLP-2)*, Tzigov Chark, Bulgaria, 1997.

[Eriksson 96]

M. Eriksson. ALEP. `http://www.sics.se/humle/projects/svensk/platforms.html`, 1996.

[Eriksson 97]

M. Eriksson. Final Report of Svensk. Technical report, SICS, `http://www.sics.se/-humle/projects/svensk/`, 1997.

[Erman & Lesser 75]

L. Erman and V. Lesser. A multi-level organisation for problem solving using many, diverse, cooperating sources of knowledge. In *Proceedings of IJCAI-75*, pages 483–490, 1975.

[Erman *et al.* 80]

L. Erman, F. Hayes-Roth, V. Lesser, and D. Reddy. The Hearsay II speech understanding system: integrating knowledge to resolve uncertainty. *Computing Surveys*, 12, 1980.

[Estival *et al.* 97]

D. Estival, A. Lavelli, K. Netter, and F. Pianesi, editors. *Computational Environments for Grammar Development and Linguistic Engineering*. Association for Computational Linguistics, July 1997. Madrid, ACL-EACL'97.

[Evans & Gazdar 96]

R. Evans and G. Gazdar. DATR: A Language for Lexical Knowledge Representation. *Computational Linguistics*, 22(1), 1996.

[Farwell & Wilks 89]

D. Farwell and Y. Wilks. ULTRA - a machine translation system. Memoranda in Computer and Cognitive Science NM1989, Computing Research Lab, New Mexico State University, 1989.

[Fayad *et al.* 00]

M. Fayad, M. Laitinen, and R. Ward. Software Engineering in the Small. *Communications of the ACM*, 43(3):115–118, 2000.

[Fikes & Farquhar 99]

R. Fikes and A. Farquhar. Distributed Repositories of Higly Expressive Reusable Ontologies. *IEEE Intelligent Systems*, 14(2):73–79, 1999.

[Fischer *et al.* 96]

D. Fischer, W. Mohr, and L. Rostek. A Modular, Object-Oriented and Generic Approach for Building Terminology Maintenance Systems. In *TKE '96: Terminology and Knowledge Engineering*, pages 245–258, Frankfurt, 1996.

[Fowler & Scott 97]

M. Fowler and K. Scott. *UML Distilled*. Addison-Welsey, Reading, MA, 1997.

[Fowler & Scott 00]

M. Fowler and K. Scott. *UML Distilled, Second Edition*. Addison-Welsey, Reading, MA, 2000.

[Fowler 97]

M. Fowler. *Analysis Patterns: Reusable Object Models*. Addison-Welsey, Reading, MA, 1997.

[Fröhlich & Werner 95]

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

[Gaizauskas & Humphreys 96a]

R. Gaizauskas and K. Humphreys. Quantitive Evaluation of Coreference Algorithms in an Information Extraction System. In *DAARC96 - Discourse Anaphora and Anaphor Resolution Colloquium*. Lancaster University, 1996.

[Gaizauskas & Humphreys 96b]

R. Gaizauskas and K. Humphreys. Using verb semantic role information to extend partial parses via a co-reference mechanism. In J. Carroll, editor, *Proceedings of the Workshop on Robust Parsing*, pages 103–113, Prague, Czech Republic, August 1996. European Summer School in Language, Logic and Information.

[Gaizauskas & Humphreys 97a]

R. Gaizauskas and K. Humphreys. Quantitative Evaluation of Coreference Algorithms in an Information Extraction System. Technical report CS–97–19, Department of Computer Science, University of Sheffield, 1997.

[Gaizauskas & Humphreys 97b]

R. Gaizauskas and K. Humphreys. Quantitive Evaluation of Coreference Algorithms in an Information Extraction System. In S. Botley and T. McEnery, editors, *Discourse Anaphora and Anaphor Resolution*. University College London Press, 1997.

[Gaizauskas & Humphreys 97c]

R. Gaizauskas and K. Humphreys. Using a semantic network for information extraction. Technical Report CS-97-03, Department of Computer Science, University of Sheffield, 1997.

[Gaizauskas & Wilks 98]

R. Gaizauskas and Y. Wilks. Information Extraction: Beyond Document Retrieval. *Journal of Documentation*, 54(1):70–105, 1998.

[Gaizauskas 95]

R. Gaizauskas. Investigations into the grammar underlying the Penn Treebank II. Research Memorandum CS-95-25, Department of Computer Science, University of Sheffield, 1995.

[Gaizauskas 97]

R. Gaizauskas. A Review of *Evaluating Natural Language Processing Systems*, Sparck-Jones and Galliers, 1996. *Journal of Natural Language Engineering*, 1997.

[Gaizauskas *et al.* 95]

R. Gaizauskas, T. Wakao, K. Humphreys, H. Cunningham, and Y. Wilks. Description of the LaSIE system as used for MUC-6. In *Proceedings of the Sixth Message Understanding Conference (MUC-6)*. Morgan Kaufmann, California, 1995.

[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.

[Gaizauskas *et al.* 98]

R. Gaizauskas, M. Hepple, and C. Huyck. A scheme for comparative evaluation of diverse parsing systems. In *Proceedings of the 1st International Conference on Language Resources and Evaluation (LREC'98)*, pages 143–149, Granada, Spain, 1998.

[Gamma *et al.* 95]

E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[Garlan & Perry 95]

D. Garlan and D. Perry. Introduction to the Special Issue on Software Architecture. *IEEE Transactions on Software Engineering*, 21(4), April 1995.

[Garlan *et al.* 95]

D. Garlan, R. Allen, and J. Ockerbloom. Architectural Mismatch or Why it's hard to build systems out of existing parts. In *Proceedings of the Seventeenth International Conference on Software Engineering*, Seattle WA, April 1995.

[Gazdar & Mellish 89]

G. Gazdar and C. Mellish. *Natural Language Processing in Prolog.* Addison-Wesley, Reading, MA, 1989.

[Gazdar 96]

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

[Goldfarb & Prescod 98]

C. Goldfarb and P. Prescod. *The XML Handbook.* Prentice Hall, New York, 1998.

[Goldfarb 90]

C. F. Goldfarb. *The SGML Handbook.* Oxford University Press, 1990.

[Goni *et al.* 97]

J. Goni, J. Gonzalez, and A. Moreno. ARIES: A lexical platform for engineering Spanish processing tools. *Journal of Natural Language Engineering*, 3(4):317–347, 1997.

[Görz *et al.* 96]

G. Görz, M. Kessler, J. Spilker, and H. Weber. Research on Architectures for Integrated Speech/Language Systems in Verbmobil. In *Proceedings of COLING-96*, Copenhagen, 1996.

[Gotoh & Renals 00]

Y. Gotoh and S. Renals. Information extraction from broadcast news. *Philosophical Transactions of the Royal Society of London, Series A*, 358(1769):1295–1309, 2000. http://www.dcs.shef.ac.uk/~sjr/pubs/2000/rs00-preprint.html.

[Gotoh *et al.* 98]

Y. Gotoh, S. Renals, R. Gaizauskas, G. Williams, and H. Cunningham. Named Entity Tagged Language Models for LVCSR. Technical Report CS-98-05, Department of Computer Science, University of Sheffield, 1998.

[Greenspun 99]

P. Greenspun. *Philip and Alex's Guide to Web Publishing*. Morgan Kaufman, California, 1999.

[Grishman & Sundheim 96]

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

[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/`.

[Guthrie *et al.* 90]

L. Guthrie, B. Slator, Y. Wilks, and R. Bruce. Is there content in empty heads? In *Proceedings of the 13th International Conference on Computational Linguistics (COLING-90)*, volume 3, pages 138–143, Helsinki, Finland, 1990.

[Harrison 91]

P. Harrison. Evaluating Syntax Performance of Parsers/Grammars of English. In *Proceedings of the Workshop on Evaluating Natural Language Processing Systems, ACL*, 1991.

[Hayes-Roth 94]

F. Hayes-Roth. Architecture-Based Acquisition and Development of Software: Guidelines and Recommendations from the ARPA Domain-Specific Software Architecture (DSSA) Program. Technical report, Techknowledge Federal Systems, 1994. `http://www.oswego.com/dssa/`, visited 29th March 1999.

[Hendler & Stoffel 99]

J. Hendler and K. Stoffel. Back-End Technology for High-Performance Knowledge Representation Systems. *IEEE Intelligent Systems*, 14(3):63–69, 1999.

[Hendry & Harper 96]

D. Hendry and D. Harper. An Architecture for Implementing Extensible Information-Seeking Environments. In *Proceedings of SIGIR-96.*, pages 94–100, 1996.

[Henrich 96]

A. Henrich. Document Retrieval Facilities for Repository-Based System Development Environments. In *Proceedings of SIGIR '96*, pages 101–109, Zurich, 1996.

[Hobbs 93]

J. Hobbs. The Generic Information Extraction System. In *Proceedings of the Fifth Message Understanding Conference (MUC-5)*. Morgan Kaufmann, California, 1993. `http://www.itl.nist.gov/div894/894.02/related_projects/tipster/gen_ie.htm`.

[Horowitz & Hill 89]

P. Horowitz and W. Hill. *The Art of Electronics*. Cambridge University Press, 1989.

[Hovy 91]

E. H. Hovy. Approaches to the planning of coherent text. In C. L. Paris, W. R. Swartout, and W. C. Mann, editors, *Natural language generation in artificial intelligence and computational linguistics*. Kluwer Academic Publishers, July 1991. Presented at the Fourth International Workshop on Natural Language Generation. Santa Catalina Island, CA, July, 1988.

[Huang *et al.* 90]

D. Huang, Y. Ariki, and M. Jack. *Hidden Markov Models for Speech Recognition*. Edinburgh University Press, 1990.

[Humphreys *et al.* 98]

K. Humphreys, R. Gaizauskas, S. Azzam, C. Huyck, B. Mitchell, H. Cunningham, and Y. Wilks. Description of the LaSIE system as used for MUC-7. In *Proceedings of the Seventh Message Understanding Conference (MUC-7)*. `http://www.itl.nist.gov/iaui/894.02/related_projects/muc/index.html`, 1998.

[Hutchins & Somers 92]

W. J. Hutchins and H. L. Somers. *An Introduction to Machine Translation*. Academic Press, London, 1992.

[IBM 99a]

IBM. Building Object-Oriented Frameworks. `http://www.ibm.com/-java/education/oobuilding/index.html`, March 1999.

[IBM 99b]

IBM. Leveraging Object-Oriented Frameworks. `http://www.ibm.com/-java/education/ooleveraging/index.html`, March 1999.

[Ibrahim & Cummins 89]

M. Ibrahim and F. Cummins. TARO: An Interactive, Object-Oriented Tool for Building Natural Language Systems. In *IEEE International Workshop on Tools for Artificial Intelligence*, pages 108–113, Los Angeles, 1989.

[Ide & Priest-Dorman 99]

N. Ide and G. Priest-Dorman. Corpus Encoding Standard. `http://www.cs.vassar.edu/CES/`, 1999.

[Ide 98a]

N. Ide. Corpus Encoding Standard: SGML Guidelines for Encoding Linguistic Corpora. In *Proceedings of the First International Language Resources and Evaluation Conference*, pages 463–470, Granada, Spain, 1998.

[Ide 98b]

N. Ide. Encoding Linguistic Corpora. In *Proceedings of the Sixth Workshop on Very Large Corpora*, pages 9–17, Montreal, 1998.

[Isard *et al.* 98]

A. Isard, D. McKelvie, and H. Thompson. Towards a Minimal Standard for Dialogue Transcripts: A New Sgml Architecture for the HCRC Map Task Corpus. In *Proceedings of the 5th International Conference on Spoken Language Processing (ICSLP '98)*, Sydney, 1998.

[Jackson 75]

M. Jackson. *Principles of Program Design.* Academic Press, London, 1975.

[Jacobs 92]

P. Jacobs, editor. *Text-Based Intelligent Systems: Current Research and Practice in Information Extraction and Retrieval.* Lawrence Erlbaum, Hillsdale, NJ, 1992.

[Jacobson *et al.* 99]

I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, Reading, MA, 1999.

[Jelinek 97]

F. Jelinek. *Statistical Methods for Speech Recognition*. MIT Press, Cambridge, MA, 1997.

[Jing & McKeown 98]

H. Jing and K. McKeown. Combining Multiple, Large-Scale Resources in a Reusable Lexicon for Natural Language Generation. In *Proceedings of the 36th ACL and the 17th COLING (ACL-COLING '98)*, pages 607–613, Montreal, 1998.

[Johnson 97]

R. Johnson. Frameworks Home Page. `http://st-www.cs.uiuc.edu/users/johnson/-frameworks.html`, 1997.

[Jones 90]

C. Jones. *Systematic Software development using VDM*. Prentice Hall, New York, 1990.

[Joshi 87]

A. K. Joshi. The relevance of tree adjoining grammar to generation. In G. Kempen, editor, *Natural Language Generation: Recent Advances in Artificial Intelligence, Psychology, and Linguistics*. Kluwer Academic Publishers, Boston/Dordrecht, 1987. Paper presented at the Third International Workshop on Natural Language Generation, August 1986, Nijmegen, The Netherlands.

[Kameyama 97]

M. Kameyama. Information Extraction across Linguistic Boundaries. In *AAAI Spring Symposium on Cross-Language Text and Speech Processing*, Stanford University, 1997.

[Karov & Edelman 96]

Y. Karov and S. Edelman. Learning similarity-based word sense disambiguation from sparse data. In E. Ejerhed and I. Dagan, editors, *Proceedings of the Fourth Workshop on Very Large Corpora*, Copenhagen, 1996.

[Kay 97a]

M. Kay. It's Still the Proper Place. *Machine Translation*, 12:3–23, 1997.

[Kay 97b]

M. Kay. The Proper Place of Men and Machines in Language Translation. *Machine Translation*, 12:3–23, 1997. Originally appeared as a Xerox PARC Working Paper in 1980.

[Kay *et al.* 94]

M. Kay, J. Gawron, and P. Norvig. *Verbmobil, A Translation System for Face-to-Face Dialog.* CSLI, Stanford, CA, 1994.

[Keijola 99]

M. Keijola. BRIEFS – Gaining Information of Value in Dynamical Business Environments. `http://www.tuta.hut.fi/briefs`, 1999.

[Knuth 93]

D. Knuth. *The Stanford GraphBase, A Platform for Combinatorial Computing.* ACM Press, New York, 1993.

[Kokkinakis & Johansson-Kokkinakis 99]

D. Kokkinakis and S. Johansson-Kokkinakis. A Cascaded Finite-State Parser for Syntactic Analysis of Swedish. Technical report, Department of Swedish, University of Göteborg, Göteborg, 1999.

[Kokkinakis 98]

D. Kokkinakis. AVENTINUS, GATE and Swedish Lingware. In *Proceedings of the 11th NODALIDA Conference*, pages 22–33, Copenhagen, 1998.

[Koning *et al.* 95]

J. Koning, M. Stefanini, and Y. Deamzeau. DAI Interaction Protocols as Control Strategies in a Natural Language Processing System. In *Proceedings of IEEE Conference on Systems, Man and Cybernetics*, 1995.

[Kuno & Oettinger 62]

S. Kuno and A. Oettinger. Multiple path syntactic analyser. In *Information Processing 1962, proceedings IFIP, Popplewell (ed.)*, Amsterdam, 1962. North-Holland.

[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.

[Lee 89]

K. Lee. *Automatic Speech Recognition, the Development of the SPHINX System.* Kluwer, Dordrecht, Netherlands, 1989.

[Lenat *et al.* 95]

D. Lenat, G. Miller, and T. Yokoi. CYC, WordNet, and EDR: Critiques and Responses. *Communications of the ACM*, 38(11), November 1995.

[LREC-1 98]

Conference on Language Resources Evaluation (LREC-1), Granada, Spain, 1998.

[LuperFoy *et al.* 98]

S. LuperFoy, D. Loehr, D. Duff, K. Miller, F. Reeder, and L. Harper. An Architecture for Dialogue Management, Context Tracking, and Pragmatic Adaptation in Spoken Dialogue Systems. In *Proceedings of the 36th ACL and the 17th COLING (ACL-COLING '98)*, pages 794–801, Montreal, 1998.

[Magerman 94]

D. Magerman. *Natural Language Parsing as Statistical Pattern Recognition.* Unpublished PhD thesis, Department of Computer Science, Stanford University, CA, 1994.

[Magerman 95]

D. Magerman. Statistical Decision Tree Models for Parsing. In *Proceedings of ACL*, 1995.

[Makins 91]

M. Makins, editor. *Collins English Dictionary, 3rd Edition.* Harper Collins, 1991.

[Mann & Matthiessen 83]

W. C. Mann and C. M. Matthiessen. Nigel: A systemic grammar for text generation. Technical Report ISI/RR-83-105, Information Sciences Institute, February 1983. 4676 Admiralty Way, Marina del Rey, CA.

[Mann 83]

W. C. Mann. An overview of the PENMAN text generation system. In *Proceedings*

of the National Conference on Artificial Intelligence, pages 261–265. AAAI, August 1983. Also appears as USC/Information Sciences Institute, RR-83-114.

[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/` .

[Marcus *et al.* 93]

M. Marcus, B. Santorini, and M. Marcinkiewicz. Building a Large Annotated Corpus of English: The Penn Treebank. *Computational Linguistics*, 19(2):313–330, 1993.

[Markowitz *et al.* 86]

J. Markowitz, T. Ahlswede, and M. Evens. Semantically significant patterns in dictionary definitions. In *Proceedings of the 24th Annual Meeting of the Association for Computational Linguistics*, pages 112–119, New York, 1986.

[Mason 98]

O. Mason. The CUE Corpus Access Tool. In *Workshop on Distributing and Accessing Linguistic Resources*, pages 20–27, Granada, Spain, 1998. `http://www.dcs.shef.ac.uk/~hamish/dalr/`.

[McClelland & Rumelhart 86]

J. McClelland and D. Rumelhart. *Parallel Distributed Processing*. MIT Press, Cambridge, MA, 1986.

[McDonald 83]

D. McDonald. Natural Language Generation as a Computational Problem: an Introduction. In M. Brady and R. Berwick, editors, *Computational Models of Discourse*, Cambridge, MA, 1983. MIT Press.

[McEnery 97]

A. McEnery. Multilingual Corpora – Current Practice and Future Trends. In *13th ASLIB Machine Translation Conference*, pages 75–86, London, 1997.

[McEnery *et al.* 99]

A. McEnery, S. Botley, and A. Wilson, editors. *Multilingual Corpora: Teaching and Research*. Rodopi, Amsterdam, 1999.

[McKelvie & Mikheev 98]

D. McKelvie and A. Mikheev. Indexing SGML files using LT NSL. LT Index documentation, from `http://www.ltg.ed.ac.uk/`, 1998.

[McKelvie *et al.* 97]

D. McKelvie, C. Brew, and H. Thompson. Using SGML as a Basis for Data-Intensive NLP. In *Proceedings of the fifth Conference on Applied Natural Language Processing (ANLP-97)*, Washington, DC, 1997.

[McKelvie *et al.* 98]

D. McKelvie, C. Brew, and H. Thompson. Using SGML as a Basis for Data-Intensive Natural Language Processing. *Computers and the Humanities*, 31(5):367–388, 1998.

[McKeown 85]

K. R. McKeown. *Text Generation: Using Discourse Strategies and Focus Constraints to Generate Natural Language Text*. Cambridge University Press, Cambridge, 1985.

[McKeown *et al.* 90]

K. McKeown, M. Elhadad, Y. Fukumuto, J. Lim, C. Lombardi, J. Robin, and F. Smadja. Natural Language Generation in COMET. In R. Dale, C. Mellish, and M. Zock, editors, *Current Research in Natural Language Generation*, London, 1990. Academic Press.

[Mellish & Scott 99]

C. Mellish and D. Scott. Workshop preface. In *Proceedings of the AISB'99 Workshop on Reference Architectures and Data Standards for NLP*. The Society for the Study of Artificial Intelligence and Simulation of Behaviour, April 1999.

[Merchant *et al.* 96]

R. Merchant, M. Okurowski, and N. Chinchor. The Multi Lingual Entity Task (MET) Overview. In *Advances in Text Processing – TIPSTER Programme Phase II*. DARPA, Morgan Kaufmann, California, 1996.

[Michiels & Noel 82]

A. Michiels and J. Noel. Approaches to Thesaurus Production. In *Proceedings of the 9th International Conference on Computational Linguistics (COLING-82)*, pages 227–232, Prague, Czechoslovakia, 1982.

[Michiels *et al.* 80]

A. Michiels, J. Mullenders, and J. Noel. Exploiting a Large Data Base by Longman. In *Proceedings of the 8th International Conference on Computational Linguistics (COLING-80)*, pages 374–382, Tokyo, Japan, 1980.

[Miller (Ed.) 90]

G. A. Miller (Ed.). WordNet: An on-line lexical database. *International Journal of Lexicography*, 3(4):235–312, 1990.

[Miller *et al.* 95]

E. Miller, M. Kado, M. Hirakwa, and T. Ichikawa. HI-VISUAL as a User-Customizable Visual Programming Environment. In *Proceedings VL'95 11th International IEEE Symposium on Visual Languages, Darmstadt.* IEEE Computer Society Press, 1995.

[Miller *et al.* 98]

S. Miller, M. Crystal, H. Fox, L. Ramshaw, R. Schwartz, R. Stone, and R. Weischedel. Description of the BBN System Used for MUC-7. In *Proceedings of the Seventh Message Understanding Conference (MUC-7).* `http://www.itl.nist.gov/iaui/894.02/related_projects/muc/index.html`, 1998.

[Milosavljevic *et al.* 96]

M. Milosavljevic, A. Tulloch, and R. Dale. Text Generation in a Dynamic Hypertext Environment. In *Proceedings of 19th Australian Computer Science Conference*, Melbourne, 1996.

[Mitchell 97]

T. Mitchell. *Machine Learning.* McGraw-Hill, New York, 1997.

[Mitkov 95]

R. Mitkov. Language Engineering on the Highway: New Perspectives for the Multilingual Society. In *Proceedings of NLPRS*, 1995.

[Mitkov 96]

R. Mitkov. Language Engineering: towards a clearer picture. In *Proceedings of the International Conference on Mathematical Linguistics (ICML '96)*, 1996.

[Moore 95]

J. D. Moore. *Participating in Explanatory Dialogues*. MIT Press, Cambridge, MA, 1995.

[Nelson 97]

T. Nelson. Embedded Markup Considered Harmful. In D. Connolly, editor, *XML: Principles, Tools and Techniques*, pages 129–134. O'Reilly, Cambridge, MA, 1997.

[Netter & Pianesi 97]

K. Netter and F. Pianesi. Preface. In *Proceedings of the Workshop on Computational Environments for Grammar Development and Linguistic Engineering*, pages iii–v, Madrid, 1997.

[Newell *et al.* 98]

A. Newell, S. Langer, and M. Hickey. The role of natural language processing in alternative and augmentative communication. *Journal of Natural Language Engineering*, 4(1):1–17, 1998.

[Nikolov *et al.* 95]

N. Nikolov, C. Mellish, and G. Ritchie. Sentence Generation from Conceptual Graphs. In *Proceedings of 3rd Int. Conf. on Conceptual Structures (ICCS'95)*, number 954 in LNAI, Santa Cruz, CA, 1995. Springer-Verlag.

[Ogden 99]

B. Ogden. TIPSTER annotation and the Corpus Encoding Standard. `http://crl.nmsu.edu/Research/Projects/tipster/annotation`, 1999.

[Olson & Lee 97]

M. Olson and B. Lee. Object Databases for SGML Document Management. In *IEEE International Conference on Systems Sciences*, 1997.

[Olsson *et al.* 98]

F. Olsson, B. Gambäck, and M. Eriksson. Reusing Swedish Language Processing Resources in SVENSK. In *Workshop on Minimising the Efforts for LR Acquisition*, Granada, Spain, 1998.

[O'Shaugnessy 87]

D. O'Shaugnessy. *Speech Communication, Human and Machine*. Addison-Wesley, Reading, MA, 1987.

[Ousterhout 94]

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

[Paiva 98]

D. Paiva. A Survey of Applied Natural Language Generation Systems. Technical Report ITRI-98-03, Information Technology Research Institute, Brighton, 1998.

[Paris & Linden 96]

C. Paris and K. V. Linden. DRAFTER: An Interactive Support Tool for Writing Multilingual Instructions. *IEEE Computer, Special Issue on Interactive NLP*, July 1996.

[Paris 91]

C. L. Paris. Generation and explanation: Building an explanation facility for the explainable expert systems framework. In C. L. Paris, W. R. Swartout, and W. C. Mann, editors, *Natural language generation in artificial intelligence and computational linguistics*. Kluwer Academic Publishers, July 1991. Presented at the Fourth International Workshop on Natural Language Generation. Santa Catalina Island, CA, July, 1988.

[Pazienza 97]

M. Pazienza, editor. *Information Extraction: A Multidisciplinary Approach to an Emerging Information Technology*. Springer-Verlag, Berlin Heidelberg, 1997.

[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.

[Poirier 99]

H. Poirier. The XeLDA Framework (presentation at Baslow workshop on Distributing and Accessing Linguistic Resources, Sheffield, 1999). `http://www.dcs.shef.ac.uk/~hamish/dalr/baslow/xelda.pdf`, 1999.

[Press *et al.* 95]

W. Press, S. Teukolsky, W. Vetterling, and B. Flannery. *Numerical Recipes in C – the Art of Scientific Computing*. Cambridge University Press, U.K., 1995.

[Pressman 94]

R. Pressman. *Software Engineering, a Practitioner's Approach (European Edition)*. McGraw Hill, New York, 1994.

[Ramsay 00]

A. Ramsay. Research in the Department of Language Engineering. `http://www.ccl.umist.ac.uk/research/booklet.html`, UMIST Department of Language Engineering, 2000.

[Rao *et al.* 92]

R. Rao, H. J. S.K. Card, J. Mackinlay, and G. Robertson. The Information Grid: a Framework for Information Retrieval and Retrieval-Centered Applications. In *Proceedings of the fifth annual ACM symposium on User interface software and technology (UIST '92)*, pages 23–32, Monterey, CA, 1992.

[Reiter & Dale 99]

E. Reiter and R. Dale. Building Natural Language Generation Systems. *Journal of Natural Language Engineering*, Vol. 3 Part 1, 1999.

[Reiter & Dale 00]

E. Reiter and R. Dale. *Building Natural Language Generation Systems*. Cambridge University Press, Cambridge, 2000.

[Reiter 94]

E. Reiter. Has a Consensus NL Generation Architecture Appeared, and is it Psycholinguistically Plausible? In *Proceedings of the Seventh International Workshop on Natural Language Generation (INLGW-1994)*, 1994. `http://xxx.lanl.gov/-abs/CS.cl/9411032`.

[Reiter 99]

E. Reiter. Are Reference Architectures Standardisation Tools or Descriptive Aids? 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.

[Reithinger 91]

N. Reithinger. POPEL—a Parallel and Incremental Natural Language Generation

System. In C. Paris, W. Swartout, and W. Mann, editors, *Natural Language Generation in Artificial Intelligence and Computational Linguistics*, Dordrecht, Netherlands, 1991. Kluwer.

[Renals *et al.* 99]

S. Renals, Y. Gotoh, R. Gaizauskas, and M. Stevenson. Baseline IE-NE Experiments Using The SPRACH/LASIE System. In *Proceedings of the DARPA Broadcast News Workshop*, Herndon, VA, 1999.

[Resnik 93]

P. Resnik. *Selection and Information: A Class-Based Approach to Lexical Relationships*. Unpublished PhD thesis, Institute for Research in Cognitive Science, University of Pennsylvania, 1993.

[Rodgers *et al.* 97]

P. Rodgers, R. Gaizauskas, K. Humphreys, and H. Cunningham. Visual Execution and Data Visualisation in Natural Language Processing. In *IEEE Visual Language*, Capri, Italy, 1997.

[Rosenfeld 95]

R. Rosenfeld. The CMU statistical language modelling toolkit and its use in the 1994 ARPA CSR evaluation. In *Proceedings of the ARPA Spoken Language Technology Workshop*, Austin TX, 1995.

[SAIC 98]

SAIC. Proceedings of the Seventh Message Understanding Conference (MUC-7). `http://www.itl.nist.gov/iaui/894.02/related_projects/muc/index.html`, 1998.

[Schütz 94]

J. Schütz. Developing Lingware in ALEP. *ALEP User Group News, CEC Luxemburg*, 1(1), October 1994.

[Schütz *et al.* 91]

J. Schütz, G. Thurmair, and R. Cencioni. An Architecture Sketch of Eurotra-II. In *MT Summit III*, pages 3–11, Washington D.C., 1991.

[Shapiro 82]

S. Shapiro. Generalized Augmented Transition Network Grammars for Generation

from Semantic Networks. *American Journal of Computational Linguistics*, 8(2):12 – 25, 1982.

[Shaw & Clements 97]

M. Shaw and P. Clements. A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems. In *Proceedings COMPSAC97, 21st Int'l Computer Software and Applications Conference*, pages 6–13, 1997.

[Shaw & Garlan 96]

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

[Shaw 93]

M. Shaw. Software Architecture: Beyond Objects. In *OOPSLA 93*, Washington, DC, 1993.

[Shieber 92]

S. Shieber. *Constraint-Based Grammar Formalisms*. MIT Press, Cambridge, MA, 1992.

[Sigurd 91]

B. Sigurd. Referent Grammar in Text Generation. In C. Paris, W. Swartout, and W. Mann, editors, *Natural Language Generation in Artificial Intelligence and Computational Linguistics*, Dordrecht, Netherlands, 1991. Kluwer.

[Simkins 92]

N. K. Simkins. ALEP User Guide. CEC Luxemburg, 1992.

[Simkins 94]

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

[Simon 95]

H. Simon. Artificial intelligence and empirical science. *Artificial Intelligence*, 77(1), August 1995.

[Somers 97]

H. L. Somers. Machine Translation and Minority Languages. In *Translating and the Computer 19: Papers from the Aslib conference*, London, 1997.

[Sowa 84]

J. Sowa. *Conceptual Structures: Information Processing in Mind and Machine.* Addison-Wesley, Reading, MA, 1984.

[Sparck-Jones & Galliers 96]

K. Sparck-Jones and J. Galliers. *Evaluating Natural Language Processing Systems.* Springer, Berlin, 1996.

[Sparck-Jones 94]

K. Sparck-Jones. Towards Better NLP System Evaluation. In *Proceedings of the Second ARPA Workshop on Human Language Technology.* Morgan Kaufmann, California, March 1994.

[Sperberg-McQueen & Burnard 94]

C. Sperberg-McQueen and L. Burnard. *Guidelines for Electronic Text Encoding and Interchange (TEI P3).* ACH, ACL, ALLC, 1994. `http://etext.virginia.edu/-TEI.html`.

[Spivey 89]

J. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall, New York, 1989.

[Spivey 92]

J. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall, New York, 2nd edition, 1992.

[Spyropoulos 99]

C. Spyropoulos. Final Report of the Greek Information Extraction (GIE) Project. Technical report, NKSR Demokritus, Athens, 1999.

[Stefanini & Deamzeau 95]

M. Stefanini and Y. Deamzeau. TALISMAN: a multi-agent system for Natural Language Processing. In *Proceedings of IEEE Conference on Advances in Artificial Intelligence, 12th Brazilian Symposium on AI*, 1995.

[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.

[Takemoto *et al.* 96]

Y. Takemoto, T. Wakao, H. Yamada, R. Gaizauskas, and Y. Wilks. Description of the NEC/Sheffield System Used for MET Japanese. In *Proceedings of theTIPSTER Phase II Workshop*, 1996.

[The Object Management Group 92]

The Object Management Group. *The Common Object Request Broker: Architecture and Specification.* John Wiley, New York, 1992.

[The Unicode Consortium 96]

The Unicode Consortium. *The Unicode Standard, Version 2.0.* Addison-Wesley, Reading, MA, 1996.

[Thompson & McKelvie 96]

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

[Thompson 85]

H. Thompson. 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, Chichester, 1985.

[Thurmair 96]

G. Thurmair. WP 4.1 Task S6: Text Handling, Detailed Functional Specifications. Technical Report WP41-S6-V1.1, Sail Labs GMBH., Munich, 1996. LE project LE1-2238 AVENTINUS internal report.

[Thurmair 97]

G. Thurmair. Information extraction for intelligence systems. In *Natural Language Processing: Extracting Information for Business Needs*, pages 135–149, London, March 1997. Unicom Seminars Ltd.

[TIPSTER 95]

TIPSTER. The Generic Document Detection System. `http://www.itl.nist.gov/-div894/894.02/related_projects/tipster/gen_ir.htm`, 1995.

[Tracz 95]

W. Tracz. Domain-Specific Software Architecture (DSSA) Frequently Asked Questions (FAQ). `http://www.oswego.com/dssa/faq/faq.html`, March 1995.

[University of Essex 99]

University of Essex. Description of the W3-Corpora web-site. `http://clwww.essex.ac.uk/w3c/`, 1999.

[van der Linden 94]

P. van der Linden. *Expert C Programming – Deep C Secrets*. Prentice Hall, New York, 1994.

[van Rijsbergen 79]

C. van Rijsbergen. *Information Retrieval*. Butterworths, London, 1979.

[Venners 98]

B. Venners. Introduction to Design Techniques. `http://www.javaworld.com/-jw-02-1998/jw-02-techniques.html`, February 1998.

[Veronis & Ide 96]

J. Veronis and N. Ide. Considerations for the Reusability of Linguistic Software. Technical report, EAGLES, April 1996. `http://w3.lpl.univ-aix.fr/-projects/multext/LSD/LSD1.html`.

[Vilain & Day 96]

M. Vilain and D. Day. Finite-state phrase parsing by rule sequences. In *Proceedings of COLING-96*, Copenhagen, 1996.

[von Hahn 94]

W. von Hahn. The Architecture Problem in Natural Language Processing. *Prague Bulletin of Mathematical Linguistics*, 61:48–69, 1994.

[Waibel 88]

A. Waibel. *Prosody and Speech Recognition*. Pitman, London, 1988.

[Wakao *et al.* 96]

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

[Walker & Amsler 86]

D. E. Walker and R. A. Amsler. The use of machine-readable dictionaries in sub-

language analysis. In R. Grishman and R. Kittredge, editors, *Analyzing Language in Restricted Domains*. Lawrence Erlbaum, Hillsdale, NJ, 1986.

[Waterworth 87]

T. Waterworth. *Speech and Language-Based Interaction With Machines*. Ellis Horwood, Chichester, 1987.

[Wilks & Stevenson 97a]

Y. Wilks and M. Stevenson. Combining Independent Knowledge Sources for Word Sense Disambiguation. In *Proceedings of the Conference 'Recent Advances in Natural Language Processing'*, pages 1–7, Tzigov Chark, Bulgaria, 1997.

[Wilks & Stevenson 97b]

Y. Wilks and M. Stevenson. Sense Tagging: Semantic Tagging with a Lexicon. In *Proceedings of the SIGLEX Workshop on Tagging Text with Lexical Semantics*, pages 74–78, Washington, DC, 1997.

[Wilks & Stevenson 98]

Y. Wilks and M. Stevenson. Word sense disambiguation using optimised combinations of knowledge sources. In *Proceedings of COLING-ACL98*, Montreal, 1998.

[Wilks & Stevenson 99]

Y. Wilks and M. Stevenson. The Grammar of Sense: Using part-of-speech tags as a first step in semantic disambiguation. *Journal of Natural Language Engineering 4(3)*, 4(3), 1999.

[Wilks 92]

Y. Wilks. Where am I coming from: The reversibility of analysis and generation in natural language processing. In M. Puetz, editor, *Thirty Years of Linguistic Evolution*. John Benjamins, 1992.

[Wilks 94]

Y. Wilks. Developments in MT Research in the US. *Proceedings of ASLIB*, 46(4), 1994.

[Wilks 96]

Y. Wilks. Natural Language Processing, guest editorial. *Communications of the ACM*, 39(1), January 1996.

[Wilks 98]

Y. Wilks. Is word-sense disambiguation just one more NLP task? In *Proceedings of the SENSEVAL Conference*, Herstmonceux, Sussex, 1998. Also appears as Technical Report CS-98-12, Department of Computer Science, University of Sheffield, 1998.

[Wilks *et al.* 87]

Y. Wilks, D. C. Fass, C. Guo, J. E. McDonald, T. Plate, and B. M. Slator. A tractable machine dictionary as a resource for computational semantics. In B. Boguraev and T. Briscoe, editors, *Computational Lexicography for Natural Language Processing*. Longman Group Limited, Harlow, Essex, 1987. Also MCCS-87-105, CRL/NMSU.

[Wilks *et al.* 96]

Y. Wilks, L. Guthrie, and B. Slator. *Electric Words*. MIT Press, Cambridge, MA, 1996.

[Wilson 99]

M. Wilson. Standards and Reference Architectures: Their Purpose and Establishment. 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.

[Winograd 72]

T. Winograd. *Understanding Natural Language*. Academic Press, New York, 1972.

[Wolinski *et al.* 98]

F. Wolinski, F. Vichot, and O. Gremont. Producing NLP-based On-line Contentware. In *Natural Language and Industrial Applications*, Moncton, Canada, 1998. `http://xxx.lanl.gov/abs/cs.CL/9809021`.

[Yokoi 95]

T. Yokoi. The EDR Electronic Dictionary. *Communications of the ACM*, 38(11), November 1995.

[Young *et al.* 99]

S. Young, D. Kershaw, J. Odell, D. Ollason, V. Valtchev, and P. Woodland. *The HTK Book (Version 2.2)*. Entropic Ltd., Cambridge, 1999. `ftp://ftp.entropic.com/pub/htk/`.

[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.

[Zajac 92]

R. Zajac. Towards Computer-Aided Linguistic Engineering. In *Proceedings of COLING '92*, pages 828–834, Nantes, France, 1992.

[Zajac 97]

R. Zajac. 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)*, 1997.

[Zajac 98a]

R. Zajac. Feature Structures, Unification and Finite-State Transducers. In *International Workshop on Finite State Methods in Natural Language Processing*, Ankara, Turkey, 1998.

[Zajac 98b]

R. Zajac. Reuse and Integration of NLP Components in the Calypso Architecture. In *Workshop on Distributing and Accessing Linguistic Resources*, pages 34–40, Granada, Spain, 1998. `http://www.dcs.shef.ac.uk/~hamish/dalr/`.

[Zajac *et al.* 97]

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

[Zipf 35]

G. K. Zipf. *The Psycho-Biology of Language*. Houghton Mifflin, Boston, 1935.

LOA (List of Acronyms)

AI	Artificial Intelligence
ALEP	the Advanced Language Engineering Platform
API	Applications Programmers' Interface
ASR	Automatic Speech Recognition
AVM	Attribute-Value Matrix
CL	Computational Linguistics
CO	Coreference
CREOLE	a Collection of REusable Objects for Language Engineering
[D]ARPA	the [Defence] Advanced Research Projects Agency
DSSA	Domain-Specific Software Architecture
EC, EU	the European Commission, the European Union
GATE	a General Architecture for Text Engineering
GDM	the GATE Document Manager
GGI	the GATE Graphical Interface
GUI	Graphical User Interface
HLT	Human Language Technologies
ICE	the Intarc Communication Environment
IE	Information Extraction
IO or I/O	Input/Output
IPSE	Integrated Programming Support Environment
IR	Information Retrieval
JAPE	Java Annotation Patterns Engine
JDBC	Java DataBase Connectivity
LaSIE	the Large-Scale Information Extraction system
LE	Language Engineering
LOTTIE	Low-Overhead Triage from Text with Information Extraction

LR	Language Resource
LT-NSL	the Language Technology group Normalised SGML Library
MT	Machine Translation
MUC	Message Understanding Conference
NE	Named Entity
NLE	Natural Language Engineering – synonymous with Language Engineering
NLG	Natural Language Generation
NLP	Natural Language Processing
NLU	Natural Language Understanding
PR	Processing Resource
R&D	Research and Development
RMI	Remote Method Invocation
SALE	Software Architecture for Language Engineering
SA	Software Architecture
SE	Software Engineering
SGML	the Standard Generalised Markup Language
SME	Small-to-Medium sized Enterprise
SQL	the Structured Query Language
ST	Scenario Template
TE	Template Element
TIPSTER	not an acronym; the name of a US IE/IR research programme
TLA	Three Letter Acronym
TR	Template Relation
TREC	Text REtrieval Conference
VIE	the Vanilla Information Extraction system

Author Index