Journal of Visual Languages and Computing (2001) **12,** 375–412
doi: 10.1006/jvlc.2000.0203, available online at http://www.idealibrary.com on IDEAL®

Visual Tools for Natural Language Processing

R. GAIZAUSKAS*, P. J. RODGERS† AND K. HUMPHREYS*

**Department of Computer Science, University of Sheffield, U.K. and † Computing Laboratory, University of Kent, U.K.*

Received 9 February 2000; revised 1 December 2000; accepted 22 December 2000

We describe GATE, the General Architecture for Text Engineering, an integrated visual development environment to support the visual assembly, execution and analysis of modular natural language processing systems. The visual model is an executable data flow program graph, automatically synthesised from data dependency declarations of language processing modules. The graph is then directly executable: modules are run interactively in the graph, and results are accessible via generic text visualisation tools linked to the modules. These tools lighten the 'cognitive load' of viewing and comparing module results by relating data produced by modules back to the underlying text, by reducing the amount of search in examining results, and by displaying results in context. Overall, the GATE integrated visual development environment leads to rapid understanding of system behaviour and hence to rapid system refinement, therefore demonstrating the utility of visual programming and visualisation techniques for the development of natural language processing systems.
© 2001 Academic Press

Keywords: visual programming systems, natural language processing, data flow program graphs.

1. Introduction

IN THIS PAPER, we describe a visual environment for natural language processing (NLP) called GATE—the General Architecture for Text Engineering—that is designed to support researchers and developers of large-scale NLP systems. By a visual natural language processing environment we mean an environment that supports the visual representation of NLP programs and the visual representation and manipulation of associated data. More specifically, GATE provides for the visual assembly of processing modules into larger systems, the visual execution of multi-module systems with large, real-world text corpora as data, and the visual display and analysis of the results of module execution. All of these activities are catered for within a single, integrated visual development environment.

That there is a need for such an environment we believe is indisputable. First, following wider trends in software engineering towards component-based development and within NLP research where recent notable successes have been achieved in developing specialised programs that solve a small piece of the language processing puzzle, modularisation and integration of software objects have become key issues for NLP development environments. The assembly of components into larger wholes, where there are complex dependencies between the components, is a process quite naturally carried out via manipulation of diagrammatic representations. Second, in modularised NLP systems a single complex datum, a document, is processed by a series of modules, each of which works on the underlying document as well as on additional information about the document supplied by its predecessor modules. To understand the complex forms of behaviour that result when developing such a system, rapid and simple access to the results of different modules in the system is essential, as is the ability to alter and rerun individual components while holding other components constant. This style of development is admirably supported by a visual execution model, e.g. in the form of an executable graph, that permits the developer to execute modules and access their results by operations on iconic representations of the modules, with data dependencies displayed as links. Finally, given the volume and complexity of the underlying textual data being processed, and the data structures built by modules about this textual data, visual tools to enable a user to 'see' significant results in a sea of text and to navigate amongst these results are crucial to support system development.

Previous work in visual programming systems (we use this term, as in [1], to include visual programming and visualisation) has to date found rather limited application in NLP. Visual versions of some text processing languages, such as *awk* and *grep* have been produced [2, 3]. Tools, that incorporate visualisation techniques, have been developed to help with some NLP tasks. These include:

- graphical tools such as Syntactica [4] and Semantica [5] designed for teaching syntax and semantics, and making extensive use of graphically presented trees;
- graphical browsers for complex (nested) feature structures, a formalism widely used in computational linguistics [6];
- tools for annotating text corpora with SGML tags, such as the Alembic Workbench for marking up text [7], or with associated tree structures [8, 9];
- tools for creating and exploring text corpora, such as XKWIC [10].

However, useful as these tools are individually, they do not form a general visual environment for building, executing and viewing the results of complex, multi-module NLP systems.

The underlying model we have adopted for a visual NLP system is that of data flow program graphs [11]. This model has been widely used for visual programming languages in a variety of application domains [12], but not, as far as we are aware, for natural language processing. In the data flow model, functions are typically represented as boxes and data flows are shown as lines between the boxes. We follow this tradition, the functions in our case corresponding to NLP modules that perform tasks ranging from tokenisation and morphological analysis of individual words, to parsing of sentences and discourse interpretation of whole texts. The user is supplied with facilities to assemble systems—data flow graphs—visually, from component modules, to execute systems visually by simply clicking on nodes in the graph, and to view results by selecting specialised viewing modules associated with nodes in the graph. The result is an integrated visual environment for building, running and analysing natural language systems, which may be viewed as high-level programs. See Figure 1 for an example of an executable data flow graph in GATE.

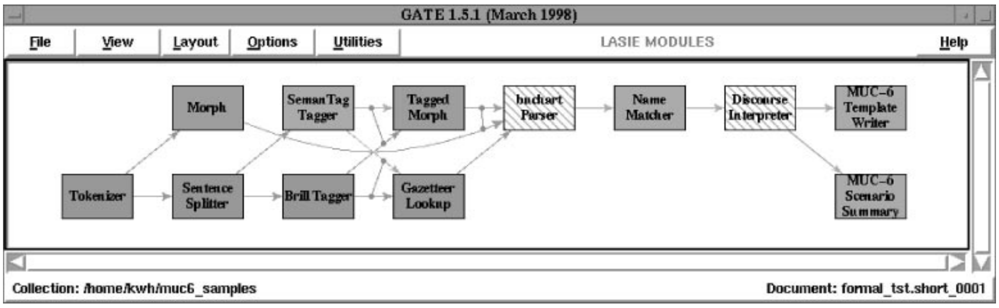


Figure 1. An executable graph for an information extraction system

We should note that GATE is not a general-purpose visual programming language, such as e.g. PROGRAPH [13]. It has been designed as a special-purpose environment for assembling relatively simple—though by no means trivial—programs that represent NLP systems and not for expressing arbitrary computations (the component modules in GATE, which represent the computational primitives of the visual data flow language, are themselves written in conventional text-based programming languages). Nevertheless, aside from being a tool of considerable utility for NLP researchers and developers, we believe that the data flow model we have designed and implemented has potential applicability beyond NLP. Any application which involves the multi-stage processing of a complex data source, where there is a need to assemble, execute and view the results of individual modules in isolation, as well as in conjunction with other components with which it forms a system, could make use of a similar model. Examples of such applications include speech and image processing.

In addition to adapting a visual data flow model for the construction, execution and access to results of modular NLP systems, GATE also contributes to the area of visual programming systems through the provision of generic visualisation tools which assist in the text-based programming of individual component modules. Most NLP modules are developed by iterative refinement of declarative rule bases or statistical models, which claim to model some aspect of language processing. These refinements rely on spotting and correctly interpreting anomalies in the processing of large amounts of real textual data. 'Seeing' the results of a module in a fashion that supports appropriate interpretation, i.e. that permits a language model to be improved, is a form of visual programming. GATE provides cognitively motivated tools for visualising certain classes of information about texts. These tools are generic in that they are reusable by any module that produces information about texts that falls into one of the classes for which the tools are designed.

The GATE system described in this paper is fully implemented and is freely available for research purposes. To date it has been distributed to hundreds of research organisations around the world and is being used in a number of national and international research projects (see http://www.dcs.shef.ac.uk/nlp/gate/users.html for details).

As an historical note, it is worth pointing out that GATE was not initially conceived as a vehicle to explore visual aspects of NLP. The original motivation for GATE was to support software reuse within the NLP research community and in particular to permit small research groups to build large prototype systems by combining software components developed elsewhere. However, to make the platform easy to use and useful for the sort of challenges that face NLP system developers, we found ourselves increasingly moving towards a fundamentally visual environment. This paper reports the design that has resulted, concentrating on the visual aspects of the system; it reports extensions to the work presented in [14]. GATE as a platform for NLP software reuse is discussed in detail elsewhere—see, e.g. [15, 16].

The rest of the paper is organised as follows. In Section 2, we provide the background context and motivation for GATE and discuss the overall structure of GATE. In Section 3, we describe the underlying model of modularisation and data and control flow within GATE. This abstract discussion sets the stage for how data flow graphs are generated and manipulated in GATE, described in detail in Section 4. Section 5 describes the data visualisation tools available in GATE, for both viewing and comparing module results. These tools are based on a classification of types of information about texts that allows the tools to be reused by modules producing output in the same class. This classification is defined and its significance for the design of visualisation tools explored. Section 6 concludes the paper, summarising the principal results.

2. Background: GATE and Natural Language Engineering

GATE—the General Architecture for Text Engineering—is a software environment that aims to support researchers and educators who are working in natural language processing (NLP) and computational linguistics (CL), and developers who are producing and delivering language engineering (LE) systems.

This area is characterised by a fundamental tension. On the one hand, there is considerable theoretical diversity about what the component subprocesses in language processing are and about the informational content of the inputs and outputs of these subprocesses. This diversity tends to lead to every researcher building his or her own system. On the other hand, building robust systems which model or reproduce enough human language processing capability to be useful (either as applications or as tests of theories) when run on real-world data, is a large-scale engineering effort which, given political and economic realities, must rely on the efforts of many small groups of researchers, spatially and temporally distributed, with no collaborative master plan.

Given this state of affairs, designing a practical support environment to assist in advancing the field is no easy task. Clearly, the pressure to build on the efforts of others demands that LE tools or component technologies—parsers, taggers, morphological analysers, discourse interpretation modules, etc.—be readily available for experimentation and reuse. But the pressure towards theoretical diversity means that there is no point in attempting to gain agreement, in the short term, on what set of component technologies should be developed or on the informational content or syntax of representations that these components should require or produce.

GATE is a software environment created in response to these considerations. It attempts to meet the following objectives:

(1) to support information interchange between LE modules at the highest common level possible without prescribing any specific theoretical approach (though it allows

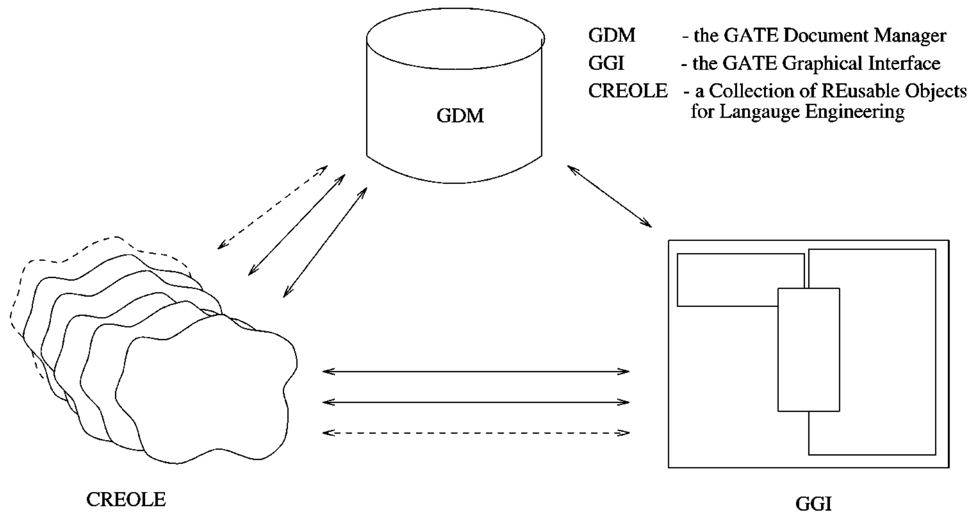

GDM - the GATE Document Manager
GGI - the GATE Graphical Interface
CREOLE - a Collection of REusable Objects for Langauge Engineering
GDM
CREOLE
GGI

Figure 2. The three elements of GATE

modules which share theoretical presuppositions to pass data in a mutually accepted common form);

(2) to support the integration of modules written in any source language, available either in source or binary form, and be available on common platforms;

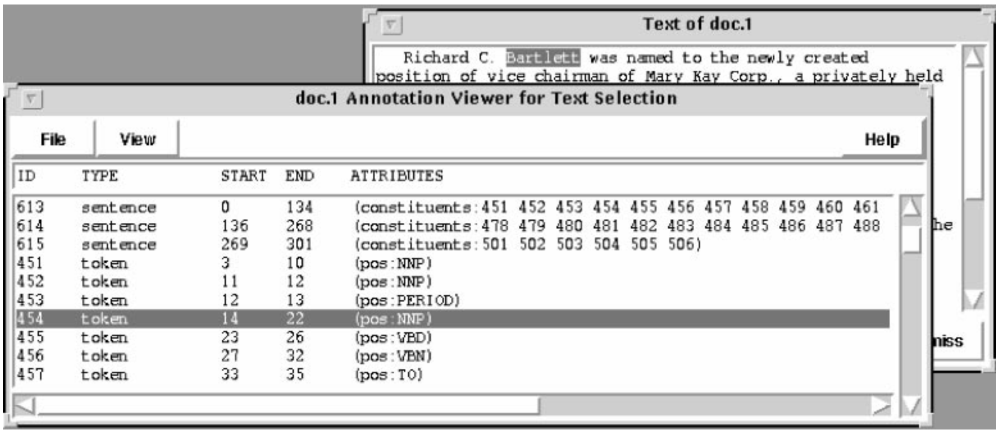
(3) to support the evaluation and refinement of LE component modules, and of systems built from them, via a uniform, easy-to-use graphical interface which in addition offers facilities for managing test corpora and ancillary linguistic resources.

GATE comprises three principal elements (see Figure 2), GDM, the GATE Document Manager, a central data management service that stores and provides access to texts and to information about texts; 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 rest of this section briefly considers each of these components in turn. The GATE WWW home page—http://www.dcs.shef.ac.uk/nlp/gate—contains more general information about GATE, including user and developer documentation.

2.1. GDM: The GATE Document Manager

The GATE document manager is based on the DARPA TIPSTER architecture specification [17] which defines an object-oriented data model whose central class is the *document*. In this model, documents are not texts; rather, a document is a repository of information about a text. A document data structure may contain a copy of the text, or it may contain only a pointer to the text, such as a URL. Documents are gathered into



```
Richard C. Bartlett was named to the newly created
position of vice chairman of Mary Kay Corp., a privately held
```

```
ID    TYPE        START  END   ATTRIBUTES
613   sentence    0      134   (constituents:451 452 453 454 455 456 457 458 459 460 461
614   sentence    136    268   (constituents:478 479 480 481 482 483 484 485 486 487 488
615   sentence    269    301   (constituents:501 502 503 504 505 506)
451   token       3      10    (pos:NNP)
452   token       11     12    (pos:NNP)
453   token       12     13    (pos:PERIOD)
454   token       14     22    (pos:NNP)
455   token       23     26    (pos:VBD)
456   token       27     32    (pos:VBN)
457   token       33     35    (pos:TO)
```

Figure 3. Some annotations and associated attributes

collections (the same source text may be the subject of more than one document, provided the documents belong to different collections). Information about a text is stored in a document either in the form of *document attributes*, if it pertains to the whole text (e.g. language), or as records called *annotations*, if it pertains to a portion or portions of a text. Annotations have a unique *id number*, a *type* (for example token) and a set of *spans* which identify one or more pairs of byte-offset positions in the text with which the annotation is associated. Each annotation has associated with it a set of *attributes* (attribute–value pairs, really) which record information about the annotation (for example, an annotation of type token might have associated with it a POS attribute recording the token's part-of-speech). Aside from allowing attribute values to be many standard data types—booleans, integers, strings, and lists—the model allows them to be the annotation ids of other annotations, hence permitting arbitrarily complex recursive data structures. Figure 3 shows a simple example of a set of annotations associated with a text.

The TIPSTER/GATE data model may be described as an indexed annotation scheme: the data about a text is stored in a separate data repository from the original text and is indexed to the relevant portion of the source text using byte offsets. Such schemes may be contrasted with markup annotation schemes, such as SGML [18], in which the data about a text are embedded in the text itself using a special syntax (e.g. the begin/end tags such as ⟨tag⟩ … ⟨/tag⟩ in SGML) to differentiate meta-data from the text source. While broadly equivalent in terms of the information which may be conveyed using them, these two approaches to annotation do have different implications for text processing architectures. For a variety of reasons including processing and data storage efficiency, we believe that the indexed annotation model is preferable for a tool such as GATE, and hence we have adopted it (see [15, 16] for more details). Of course, data marked up according to a markup annotation scheme may be imported to and exported from an indexed annotation scheme.

The GDM implements the TIPSTER data model, and provides an application program interface (API) that supports standard operations on the data model. In particular, GATE provides APIs that implement the document management subset of the TIPSTER architecture in Tcl, Java, and C++. For more information about the TIPSTER architecture and GATE's relation to it see [15–17].

2.2. CREOLE: Language Processing Modules in GATE

For most users of GATE, the central interest is the language processing that can be carried out within it. Such processing is accomplished by one or more modules organised into systems for performing some specific task. Such tasks can range from those of interest exclusively to the language engineer or researcher, such as part-of-speech tagging or parsing, to application-level tasks, such as information extraction or translation. It is worth stressing that GATE is not specific to any theoretical approach nor to any application area within NLP or CL or LE: GATE is an architecture within which specific approaches and applications may be investigated and developed.

The entire collection of modules available within GATE is referred to as CREOLE—Collection of REusable Objects for Language Engineering. These modules communicate with each other entirely through the GDM; that is, they take their input from the GDM and they write their output, in the form of annotations and attributes on annotations, to the GDM. This approach enforces a regime of modularisation that promotes algorithmic resource sharing, since developers need concern themselves only with the GDM API and not with the peculiarities of potentially dozens of module interfaces. It also permits the superimposition of a uniform graphical interface and standard viewing tools. In the future it is hoped that it will also support a full client–server model, permitting distributed and asynchronous processing.

The particular set of CREOLE modules that is available in any given GATE installation may vary. GATE is distributed with a base set of CREOLE modules that, for historical reasons, form an information extraction system (approximately, the modules that made up the Sheffield MUC-6 system [19]). These modules provide some initial building blocks that serve to demonstrate the approach and may or may not prove useful to others. Numerous sites have created their own CREOLE modules and we are in the process of devising a centralised module registry scheme that will permit GATE users to publicise and exchange modules in a straightforward fashion.

It is one of the functions of GATE to facilitate the addition of new user-developed NLP modules to the CREOLE set, and a well-defined API, a straightforward configuration procedure, an easy-to-use user interface, and extensive documentation attempt to realise this goal. It is also possible to select subsets of the CREOLE set and define new systems (see Section 4.2). Together, these features make GATE a 'plug-and-play' architecture, supporting the straightforward addition, swapping, and reconfiguration of modules.

2.3. GGI: The GATE Graphical Interface

The casual user of GATE sees only the user interface. It is designed to provide users with simple mechanisms to:

- assemble and access multiple document collections;
- run multi-component NLP systems against single documents or against document collections;

- view the results of NLP module processing in an easy-to-understand fashion;
- alter the parameters of individual NLP modules, so as to experiment with their behaviour;
- add new NLP modules to the CREOLE set;
- create new NLP systems from the set of CREOLE modules available;
- manually add, remove or edit annotations on individual documents, to support the creation of manually annotated document collections, for evaluation or machine learning purposes;
- view the results of comparing annotations added to the same document by different annotators (human or machine).

The rest of this paper addresses the novel features which the GGI brings to support the visual exploration of language processing. However, before we can present these features directly, we must explain the notions of modularisation and data and control flow that underpin them.

3. Modularisation and the Flow of Data and Control in GATE

Much recent work in language engineering has concentrated on developing specific technologies to address subproblems within the area. Perhaps the most notably successful of these component technologies has been part-of-speech tagging, where stand-alone taggers are now capable of achieving tagging accuracies of up to 99%. Morphological analysis, sentence boundary detection, parsing, word sense disambiguation and coreference resolution are examples of other 'single issue' technologies in which significant progress is actively being made in a current divide-and-conquer climate within the language processing community. It is within this climate and with these sorts of component technologies in mind that the TIPSTER data model, and the GATE architecture were developed. High-performance components are admirable, but unless they can be assembled into working systems that carry out higher level application tasks that users actually want, such as information extraction, machine translation, or summarisation, they remain a curiosity.

In order to support the assembly of existing component technology into larger systems in an effective way, a platform such as GATE must pay particular attention to the underlying model of modularisation and flow of data and control it assumes. This model will determine the type of system that can be constructed within the 'meta-system'; and it will underlie the design of any visual model that is presented to a system developer or researcher using the platform.

The TIPSTER architecture specification does not place many constraints on the nature of the modules, or annotators as they are known in TIPSTER terminology, which access the data model it defines. Nor does it specify a set of control mechanisms available for constructing systems from collections of TIPSTER-compliant modules. The only constraint is to insist that all module-to-module communication must take place through the database. Thus, in any multi-module system the database acts as a global state which is accessed and updated by various modules each of whose internal state is of no consequence to its fellows. This is a very weak constraint, and various classes of system, built out of modules that conform to this constraint, can be distinguished. In building GATE we have made some additional assumptions that constrain both what constitutes a module and what sort of control mechanisms are available.

3.1. Modules

In the least prescriptive model, a module is any code that processes all or part of a document object (raw text plus associated annotations) or a collection object (set of document objects plus collection attributes). We have found it useful to constrain what constitutes a module in two further ways. First, we insist that modules be clearly typed in terms of their input–output annotations and attributes; second, and more significantly, a module must have a clearly defined quantum of information that it processes in an atomic execution step. The second issue is related both to the first requirement, and to the following discussion of control, since the grain-size of execution unit affects what are possible models of data and control flow.

3.1.1. Typing

In order for code which accesses the GDM to count as a module for possible inclusion in GATE we require that it be explicitly typed. As is generally acknowledged, typing promotes good design and programming practice. Further, as we shall see below, declarative-type definitions permit data flow dependencies to be determined largely automatically. Thus, each module is assumed to have a signature—the type of annotations and attributes it requires as input, and the type of annotations and attributes it produces as output. Several remarks should be made here. First, for document annotators the raw text of the document is presumed always available as an additional input. Second, since attribute values may be references to other annotations, any module whose input annotations include attributes of this type is assumed to have access to all of the data referenced by its input annotations (recursively). For example, a parser might be designed to process an annotation of type sentence where the sentence annotation has as an attribute a list of constituents each of which is a pointer to a token annotation with associated attributes such as part-of-speech and morphological data.

3.1.2. Granularity of Module Processing

Taking part-of-speech (POS) tagging as a canonical example of the sort of module to be made available within a platform like GATE, there is an issue as to the grain size of data the ‘module’ processes in one execution step. Most language processing modules, such as POS taggers, have a logical unit of input data that they require in order to carry out their function. For most POS taggers this unit is the sentence. For, while the unit which gets tagged is the word, the process can only be carried out given some context and for POS taggers the unit of required context is generally taken to be the sentence (e.g. *study* is a both a noun and a verb and its word class is only revealed, for any particular instance, by the context in which it appears). However, many POS taggers are written as modules which can take a text as input and tag the whole text. The issue, therefore, is whether the POS tagger ‘module’ processes a sentence, or a text.

This distinction is important for it affects how larger systems can be assembled out of modules. If, by definition, a module processes a whole text before any further module which depends on its output processes any of the text, then much more limited systems may be assembled than if control may move between modules after portions of the text have been processed. This topic is discussed at greater length in the next section. Here, we note only that we shall assume that one of a module's inputs is a distinguished control input, which is the 'natural' or 'logical' unit of processing for the module. That is, while the module may need access to more of the text, or to the output of other module's processing on more of the text, it processes the text one control input at a time. Further, the control unit is atomic for the module in that it is the smallest unit of data the module can process. We shall also assume that for each language-processing module the control inputs are totally ordered, and thus present an input stream to the module, within which it is uniquely positioned at any time. Typically, the control inputs will be natural units of the underlying sequentially ordered document text: characters, words, sentences, paragraphs or sections. However, they could be multispan annotations, such as coreference chains which may overlap. In such cases we are assuming that a total order can always be defined, likely by reference to the underlying text.

3.2. Data Flow and Control Flow

3.2.1. Module-by-module vs. Text-unit-by-text-unit Strategies

As noted in the previous section, we are assuming a fine-grained notion of what constitutes a module. This view is not the one that appears to have been assumed in the TIPSTER specification, though it is not explicitly ruled out. There the view seems to have been that an annotator is a program unit that processes an entire text, and the issue of whether control may shift between annotators before an entire text is processed is simply not addressed.

The issue may appear irrelevant in designing an underlying data model for text processing that facilitates modularity and 'plug and play' capability. But in designing an environment such as GATE which must provide effective and flexible system building capabilities it is crucial. We distinguish two general strategies for multi-module processing of multi-unit texts as follows (see Figure 4). In the module-by-module processing strategy each module processes every unit in the text before the next module is invoked. In the text-unit-by-text-unit processing strategy the first text unit is processed by each module in turn, then the second text unit is processed by each module, and so on (we ignore here the possible difference in size of control input between modules—a more sophisticated statement of the text-unit-by-text-unit strategy is that just as many control input units of the upstream module(s) that are necessary to yield one control input for a downstream module would be executed before control moved downstream).

The appeal of supporting a module-by-module processing only strategy is that in a multi-module system no facility need be provided to return to any earlier module in the processing sequence and no memory is required, at the system control level, of how far through a particular document any module has progressed. A further practical advantage is that since loading a module may form the bulk of the processing overhead, dwarfing execution time for short to medium-length texts, the module-by-module strategy, where

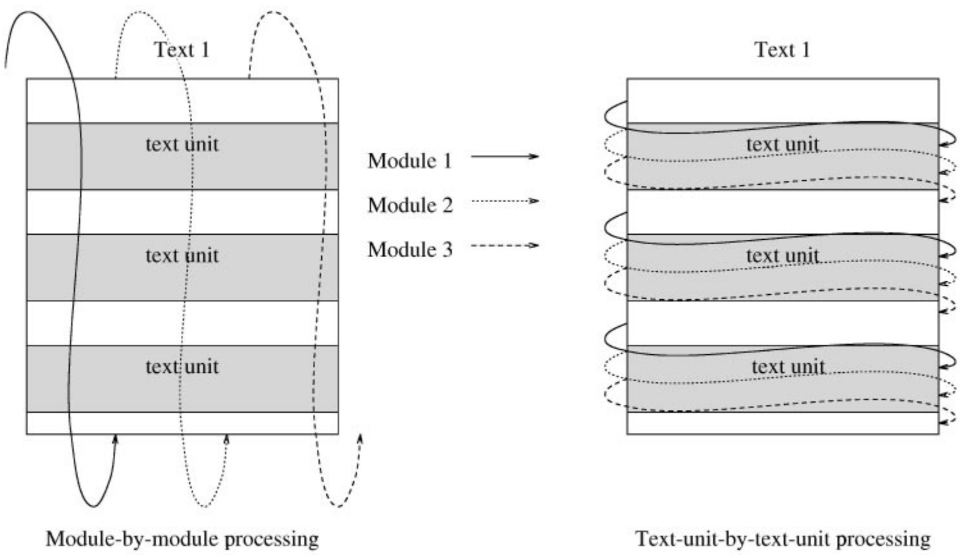


Figure 4. Different text processing strategies for multi-module systems

each module is loaded once only, can be much more efficient than the text-unit-by-text-unit strategy where each module is loaded as many times as there are text units to be processed.

The chief drawback of the module-by-module strategy is that it precludes the design of systems where the output of a downstream module on an earlier text unit becomes available as input to an upstream module when processing a later text unit. This severely constrains the type of systems that can be designed, or at least severely constrains how they may be modularised. For example, the semantic interpretation of a sentence and its integration into a discourse model typically takes place after parsing. During this process, some discourse theories nominate an entity as the local focus of the discourse. The focus of one sentence can affect how the following sentence is parsed. Such a model cannot be implemented in a system which adopts a strict module-by-module processing strategy unless the parsing, semantic interpretation and discourse integration activities are all rolled into a single, large module, so that all the requisite information is available in the local state of the 'mega-module'. But, forcing developers to produce unnaturally bundled modules like this defeats the purpose of a modular development environment with visual access to information at module interfaces.

A second drawback of the module-by-module strategy is that it embodies a very unnatural model of text processing. There is something odd about, for example, tokenising an entire text, then tagging it, then parsing it, and so on. Humans appear to do full language processing on a chunk of the text, such as a clause or sentence, and then move on. While few language engineers are committed to building cognitively plausible models of language processing, there seems to be no good reason to produce a development environment which actively precludes such models being constructed. Further, there is the practical consideration that a developer may be interested in how a later module in a system processes an early sentence in a document and may not have the time or patience, in the case of longer texts, to run earlier modules over the whole document.

Due to its simplicity, the module-by-module strategy is the model that was originally implemented in GATE. However, because of its limitations, we have attempted to support a text-unit-by-text-unit strategy as well, so that developers may choose the most appropriate strategy for the system under development.

3.2.2. Centralised vs. Distributed Control

A set of modules, each of which is specified to process one control input from the global state and possibly update the global state may be thought of as a set of primitive operations. Given, in addition, a set of control operations, one can then think of building systems out of the primitive operations together with the control operations as a form of procedural programming. A system builder when given a set of modules which comply with the TIPSTER/GATE data model could specify a system as a series of steps in a control language which might take the form, e.g. DO tokenise *text*; DO tag1 *text* OR tag2 *text*; DO parse *text*, etc. If the modules are explicitly typed, such a program could be compile- or run-time checked to ensure that appropriate annotations would reside in the global database prior to a dependent module being executed. Further, from such a program a graphical representation of control flow through a system could be abstracted and a visual program presented to a user to use in processing texts.

Alternatively, given the relatively limited number of modules likely to be available for integration and their strict typing, the type constraints can be treated as implicitly specifying a set of possible systems. That is, by explicitly expressing the type of the input and output for a module, and assuming that the inputs must be present for the module to run, and that the outputs are guaranteed to be present after it has run, data dependencies can be determined and the set of possible systems synthesised and presented to a system builder visually in the form of data dependency graphs which specify control flow. The system builder could then select and save the desired system visually and, as before, a visual program for text processing would then be available for a user to process texts.

This latter model bears strong resemblances to the data flow model [20] in which the computational units are functions which are executed when their inputs are available and in which control flow is not explicitly expressed, but rather only data dependencies. This model has several advantages for the GATE scenario. First, it fits well with the ‘plug and play’ philosophy underlying GATE: a module developer need only supply a declarative specification of the module’s data dependencies and data products, and the larger environment determines how these can be satisfied and used, respectively. This relieves the module developer of any need to worry about learning the syntax of some command language. More importantly, this means that there need be no specification of which modules get run before or after the module in question, so that if other modules supplying the inputs or using the outputs replace existing ones, or are added to the set of available modules, no higher level control specification needs to be altered. Secondly, the data flow approach naturally lends itself to a graphical interpretation which can be presented to the user in the form of a visual program.

For these reasons we have adopted a distributed/data flow model of control in GATE. The next section discusses the types of data dependencies that may be expressed in the module-by-module approach and the text-unit-by-text-unit approach.

3.2.3. Specifying Control Flow via Data Dependencies

In the data flow model, the sequence of modules to be executed is not specified explicitly. Instead, only the data preconditions and postconditions of each module are specified. These dependencies are then interpreted and a module execution sequence is determined.

Specifying data dependencies requires some language for doing so. In the context of the TIPSTER/GATE data model (Section 2.1), such a language must allow a module to require that:

(1) one or more document attributes be present, perhaps with specific values;
(2) one or more annotations be present, each with zero or more attributes, where each attribute may have a specified value;
(3) the annotations and attributes pertain to 'previous' input—this allows benign cycles to be created, where a module *A*, which produces results required by a module *B*, may in turn require results from *B*, as long as it is the output of *B* for inputs previously processed by *A*.

Further, such a language must allow a module to advertise its products, stating that it produces:

(1) one or more document attributes, perhaps with an enumerated set of possible output values;
(2) one or more annotations, each with zero or more attributes, where each attribute has an associated enumerated set of possible output values.

A schema indicating the form that declarations of pre- and postconditions take for each module is shown in Figure 5 (square brackets indicate lists, curly brackets optionality). A concrete example of these declarations, as specified in a (slightly simplified) configuration file for a module currently integrated into GATE—a chart parser, is shown in Figure 6. The preconditions indicate that a document whose language is English is required, and that annotations of type token with attributes of type pos (part-of-speech, produced by a tagger) and root (root form produced by a morphological analyser), sentence (produced by a sentence splitter) and lookup (produced by a gazetteer lookup module) must be available before the module can run. The postconditions indicate that a document attribute signalling that the parser has run will be set, and that annotations of type name (indicating classes of proper name found by the parser), syntax (phrase structure found by the parser) and semantics (semantic interpretation of the sentence constructed by the parser) will be produced. The title and viewer sections of this configuration file are irrelevant to the current discussion (the viewer section is discussed further below in Section 5.2).

A set of modules, each with a data dependency declaration of this form, implicitly defines a set of possible control flows. Each set of modules whose outputs satisfy the

```
preconditions [
    document_attributes [[doc_att1 {doc_att1_val}] ...
                         [doc_attn {doc_attn_val}]]
    annotations [ [ann1 {[ann1_att1 {ann1_att1_val} {'previous'}] ...
                         [ann1_attm {ann1_attm_val} {'previous'}]}
                        {'previous'}]
                               ...
                  [annj {[annj_att1 {annj_att1_val} {'previous'}] ...
                         [annj_attp {annj_attp_val} {'previous'}]}
                        {'previous'}]
                ]
postconditions [
    document_attributes [[doc_att1 {[doc_att1_val1 ... doc_att1_valu]}] ...
                         [doc_attq {[doc_attq_val1 ... doc_attq_valv]}]]
    annotations [ [ann1 {[ann1_att1 {[ann1_att1_val1 ... ann1_att1_valw]}] ...
                         [ann1_attr {[ann1_attr_val1 ... ann1_attr_valx]}]}]
                               ...
                  [annk {[annk_att1 {[annk_att1_val1 ... annk_att1_valy]}] ...
                         [annk_atts {[annk_atts_val1 ... annk_atts_valz]}]}]
                ]
```

Figure 5. Data dependency declarations

```
creole_config(buchart) [
    title [buChart Parser]
    pre_conditions [
        document_attributes [language english]
        annotations [[token pos] [token root] sentence lookup]
    ]
    post_conditions [
        document_attributes [buchart true]
        annotations [name syntax semantics]
    ]
    viewers [
        [name tag name]
        [syntax tree syntax]
        [semantics raw semantics]
    ]
]
```

Figure 6. A sample module configuration file

inputs of another module become one possible set of modules which, if run, will enable the latter module to be run. These dependencies define the sequence in which modules must be run. If the outputs of more than one set of modules may satisfy another then there are alternative sequences in which modules may be run.

To the extent that a module requires annotations or attributes of a certain type the precise module sequence, or set of sequences if there are alternatives, can be determined prior to execution. However, if a particular attribute value is required then this may only become known at run time. This facility permits conditional branching depending on data being processed. For example, a particular module might serve as a language identifier, capable of recognizing English, French, German, etc. documents. Its postconditions would contain a document attribute (say language) with an enumerated set of values (english, french, german, ...). A module designed to tokenise English text would specify [language english] as a document attribute in its preconditions. Control flow would then pass to this module at run time if an English document is detected; otherwise it passes to an appropriate module for the language that was recognised or simply did not proceed if no such module was available.

Dependency cycles are prohibited in module-by-module processing. If module A requires annotations of type ann2 produced by module B and B requires annotations of type ann1 produced by A, then there is no acceptable module ordering that can meet these constraints. However, if a sequence of text units $U_1, U_2, \ldots$ is being processed then it is quite reasonable when processing U_i that A requires annotations of type ann2 pertaining to any $U_j, j < i$, produced by B, while B requires annotations of type ann1 produced by A on U_i in order to process that same U_i. Note that creating sets of modules with this sort of dependency requires a text-unit-by-text-unit processing strategy (the converse is not the case: a text-unit-by-text-unit processing strategy may be adopted without there being any cyclical dependencies).

It is important to note that specifying dependency declarations does not necessarily fully determine the order in which modules must be run or even that they must be run. If a module simply updates the annotations andıor attributes on which it depends—i.e. if its postconditions match its preconditions then it is 'optional' in the sense that any module depending on it could omit it and be attached directly to its inputs instead. Further, given two or more modules whose postconditions and preconditions all match it is clear that the data dependency declarations do not define a unique order. In such cases, the order of execution of such modules cannot be determined solely on the basis of module data dependency declarations and some further input is required to determine control flow fully.

Finally, note that the language of data dependencies sketched above is very limited in that it does not allow more complex conditional or disjunctive dependencies to be specified. For example, there is no way to assert that either annotation ann1 or annotation ann2 is sufficient; or that if ann2 is present then ann1 must also be. Such expressive power would greatly expand the sophistication of the systems that could be specified in this fashion, but at the cost of significant extra complexity in the language itself and in algorithms that work out control from these specifications. We have not perceived a need to extend the expressive power of data dependencies in this way as yet.

3.2.4. Residual Control Issues

While specifying data dependencies for modules implicitly constrains control flow, it does not strictly determine it. At least two issues remain.

(1) The timing of module execution in some models of data flow computing, there is no overall control of execution and modules are assumed to execute as soon as sufficient data are available at their inputs to enable them. This model leads to both the so-called pipeline parallelism as well as parallelism between modules with no dependencies and the apparent ease with which parallelism can be exploited in data flow models is one of their strong appeals. High performance throughput, however, is not one of the objectives of an environment like GATE. Instead, control needs to be firmly in the hands of the system developer using GATE. To this end, and because it is a simpler model to implement, GATE adopts an execution model where just one module runs at a time and module execution is initiated from the

interface by the user (see Section 4.3). However, should high throughput become a requirement, the implementation could be modified to take advantage of the inherent parallelism of data flow systems.

(2) Control facilities to support a module-by-module or text-unit-by-text-unit processing strategy. Since a module is, by definition, assumed to execute one control input each time it runs, control facilities to support a module-by-module or text-unit-by-text-unit processing strategy need to be provided. These facilities are provided through the user interface. The default is that each module when invoked from the interface is run iteratively on all available inputs, i.e. the module-by-module strategy is the default. At present, the text-unit-by-text-unit strategy is not fully supported, and may only be used:
    (a) by creating a named 'super' module which contains as submodules all of the modules to be run iteratively in a text-unit-by-text-unit mode and which itself can be run as a single module against the whole input (i.e. no module outside the super module on which it depends also depends on it);
    (b) with modules whose control input is the same (this avoids complex issues of determining for each module in the chain to be iterated how many control inputs that module must minimally consume to ensure that every other module in the chain can run at least once).

4. Building and Executing NLP Systems via Data Flow Graphs

One of the appeals of the data flow model is that it lends itself straightforwardly to a graphical interpretation [11, 12]. In such an interpretation, modules form the nodes in a directed graph, and arcs represent data flows. We have implemented this graphical model in GATE. This has involved developing an algorithm for automatically generating data flow graphs from data dependency specifications and then providing the user with facilities to assemble a new NLP system from a graph of available language-processing modules and to execute the resulting graph. These topics are addressed in turn in the following sections.

4.1. Deriving Data Flow Graphs from Data Dependencies

One of the steps in adding a module to GATE is to provide a module configuration file which declares the module's data dependencies. A simplified configuration file for a chart parser module has been shown above in Figure 6. From configuration files such as this the data flow graph generation algorithm attempts to generate a graph of the sort shown in Figure 7.

The standard box-line notation is used with boxes representing modules and directed lines (arrows) representing data flows. Each required input for a given module is indicated by a simple arrow originating at the module which generates the required data and terminates at the given module; alternative sources for a given input are indicated using a cross-tie between the arrows originating from the alternative source modules. Conditional branching can be indicated by a specially shaped node (cf. the 'distributor' nodes of [11]). Cyclical dependencies, where the processing of an 'earlier' module on a given input depends on the output of a 'later' module processing previous input,

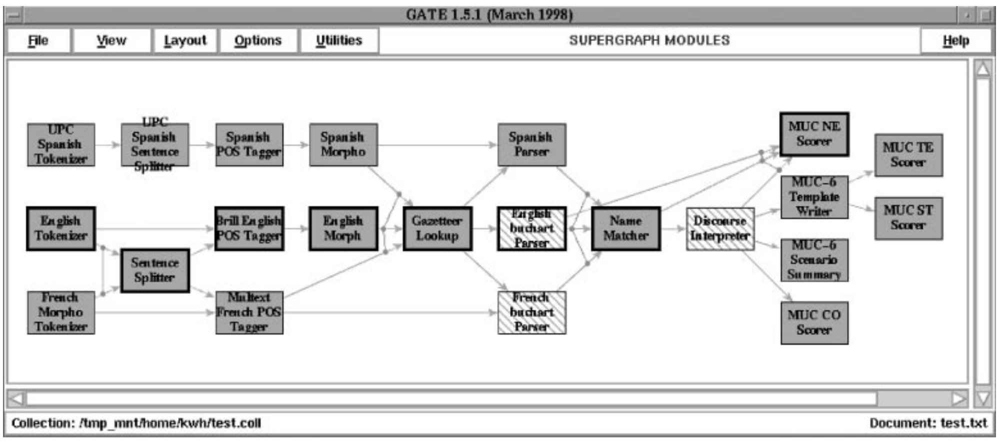


Figure 7. The GATE Supergraph

can be indicated using a cycle in the graph whose arc is visually distinct from the normal arc.

Since the generated graphs are chiefly intended to be used as executable programs and since depicting all data dependencies can lead to extremely complex graphs, the autogeneration algorithm discards 'redundant' arcs. Redundant arcs are those that connect an upstream module to a downstream module where it can be deduced that the preconditions of modules between the two given modules cover the annotation types that the arc represents. For example, suppose module *A* produces annotation ann1, module *B* requires ann1 and produces ann2 and module *C* requires ann1 and ann2. Then this scenario would be drawn with one arrow going from *A* to *B* and one arrow going from *B* to *C*. The arrow from *A* to *C*, representing *C*'s dependence on ann1 is omitted since there is no way *C* could be executed without *B* being executed, which in turn assures the execution of *A* and the presence of ann1.

As discussed in Section 3.2.3, data dependencies do not always provide sufficient information to permit modules to be uniquely placed in a graph. This occurs if one or more modules do not add new annotations, attributes, or attribute values, but instead simply update existing annotations, attributes, or attribute values in their input. Such modules we call filter modules. Since they cannot necessarily be automatically placed in a unique position in the graph, and since proposing all logically possible places in the graph where they could be added leads to confused and complex graphs, we have adopted the approach of presenting these modules as unconnected nodes along one side of the graph and allowing the user to drag and drop them onto the arc on which they should act. Once a filter is selected for placement, the possible arcs upon which it may be placed are highlighted. On adding the filter to the arc, an alternate arc not including the filter is also added to the graph, so that systems with and without the filter may be constructed.

Circular dependencies amongst modules can quite easily be specified in the data dependency declarations. Some of these dependencies are pernicious and some are not. If, as discussed in Section 3.2.3, an upstream module depends on downstream output pertaining to previous input data (in the text-unit-by-text-unit processing strategy), the situation is quite benign. If, however, a module *A* requires as input for a given text unit the output solely produced by another module *B* for the same text unit and *B* requires as input an output solely produced by *A*, the dependency is malign and needs to be ruled out. This sort of case may be relatively easily detected automatically and an error message presented to the user indicating that unacceptable circular dependencies are present in the data dependency declarations.

Between these two extreme cases, there are others which are more difficult to assess automatically. For example, a module *A* could produce annotation ann1 required by *B* which produces ann2 required by *C* which also produces ann1. Superficially, this creates a circular dependency between *B* and *C*. However, a situation could arise where *C* produces more information of the sort that *A* produces following information produced by *B* given *A*'s output. There is nothing pernicious about this—the intention could be that *B* require *A*'s output, but not *C*'s. One solution to this problem is to eliminate such dependencies by forcing the user to name the outputs of *A* and *C* distinctly, and then modify modules downstream of *A*, *B* and *C* to take into account the outputs of both *A* and *C*. Another solution is to identify circular dependencies automatically and then invite the user to confirm dependencies between the modules involved before adding them to the graph. We have adopted the latter solution, in order to constrain as little as possible annotation and attribute naming conventions. The negative aspect of this solution is that nonsense graphs may be created by the user, and the system relies on sensible input from the graph creator to work properly.

Once a graph has been generated, it is automatically laid out with a graph drawing algorithm based on that of Sugiyama *et al.* [21], but the layout can be further manually altered by dragging modules to preferred positions and adding curve points to arcs. Graph layout and editing facilities are implemented in Tcl/Tk using our own graph display tool.

4.2. Assembling New Systems in GATE

New language-processing systems are assembled visually in GATE via the Supergraph. The Supergraph is a data flow graph of the sort described in Section 4.1 and contains all the language-processing modules known to the local GATE installation. Specialised systems are produced by selecting, then naming and saving, a subgraph from the Supergraph. The named system is subsequently available from the top level of the GGI as an iconified application which can be run directly. Figure 7 shows a Supergraph; Figure 1 is an example of a specialised system produced by selecting, naming, and saving a subgraph from such a Supergraph.

More specifically, a new system is created as follows. The user selects a Create system function from the GATE top-level menu at which point the data flow graph generation algorithm attempts to derive a data flow graph as indicated in Section 4.1. If necessary, the user is prompted to resolve any cyclical dependencies that cannot be automatically resolved. When the Supergraph has been constructed and displayed the user may place filter modules on arcs as desired and may further manually add or remove dependencies (though at the risk of creating systems that cannot run). He may then select the subgraph he wishes to become a named system, either by 'rubber-banding' the subgraph, if this is possible (this depends on the graph layout), or by individually selecting the component modules by mouse click operations. When he has specified the system, he saves it and provides a name when doing so.

Typically, before defining a new system, a user will want to introduce one or more new modules to GATE. This requires the user to select the Create Module option from the GATE top-level menu and specify the name of the directory in which the module code and any wrapper code reside (the user must also write a wrapper for each new module, if the module has not been written specifically for GATE; this wrapper reads or writes the annotation information the module requires or produces from or to the global database, as well as invoking the module itself in its native form). On doing this, the shell of a configuration file for the new module is automatically generated, in which the module's data dependencies must be specified, as well as the viewing requirements for its results (see Section 5.2). This configuration file is opened in a text editor and the user must complete and save it before the new module may be loaded and added to any system. Once the configuration file has been completed, the user selects Load Module from the GATE top-level menu, selects the new module by name, and the module is then dynamically loaded. Following this step, a new system may be created, as described above, via the Supergraph in which the new module will now appear suitably connected, provided modules which supply its data dependencies have already been created and loaded (otherwise the module will appear, but completely unconnected).

At present there is a very limited facility for what has been termed visual procedural abstraction [12]—introducing a single node or supermodule which represents a subgraph of a system graph. Abstraction is carried out in the Supergraph. A subgraph may be selected and a name provided for an abstract node which will stand for that subgraph. Such nodes are drawn with cross-hatching to distinguish them from normal nodes and after creation become available in the Supergraph for inclusion in systems derived from the Supergraph. During execution, as described in the next section, such nodes may either be executed as a single step, or may be expanded into the subgraph and the component modules executed individually. There are constraints on the sort of subgraph that may be abstracted. To be eligible for abstraction a subgraph must:

(1) be a connected graph;
(2) contain no module which is dependent both on another module in the subgraph and on one outside the subgraph (i.e. there is a set of one or more mutually independent 'start' modules);
(3) contain no module on which both another module in the subgraph and one outside the subgraph depend (i.e. there is a set of one or more mutually independent 'end' modules).

Only one level of abstraction is possible at present. So, supermodules may not contain supermodules as components.

In the case of benign cyclical dependencies, where an earlier module depends on the output of a later module on a previous input, it does not make sense to include only a part of the minimal graph containing the loop in any system. Therefore, in such cases a supermodule is automatically created after all of the component modules which it comprises have been created and loaded. This supermodule is given a default name which the user can later edit, and only the supermodule, flagged as such by its visually distinct—cross-hatched—module symbol appears in the Supergraph. It is this facility which the user must employ to implement a text-unit-by-text-unit processing strategy across two or more modules, by adding a dependency between the last and first modules of the sequence of modules to be executed in this manner (see Section 3.2.4).

4.2.1. Discussion

Allowing users to assemble customised systems easily from a wider collection of language-processing modules is a key feature of a language engineering environment. In any serious research or development environment the set of usable modules will grow rapidly and only some of these will be relevant for any given task. Users need to know what the available modules are, what their data dependencies are, be able to select the modules they want for a specific application, and be able to integrate new modules straightforwardly.

The visual approach we have taken to implementing this functionality has a number of strengths:

(1) the Supergraph makes immediately apparent what the available modules are and what their data dependencies are;
(2) creating a subsystem is easily achieved by simple mouse operations and yields a system that can be run immediately;
(3) systems with non-valid data dependencies cannot be created, since the user is always selecting a system from a graph constructed from module data dependencies;
(4) since a new module is integrated visually into the existing graph of modules, mistakes in specifying the module's data dependencies are immediately apparent from its incorrect positioning.

However, there are problems with this approach as well:

(1) as the number of modules grows to be very large, the Supergraph can become unmanageably large and complex (the scaling problem [22]);
(2) as the number of modules grows to be very large, computing and laying out of the Supergraph can become very time-consuming.

These difficulties are mitigated to some extent by the limited procedural abstraction capabilities described above. However, a more thorough solution would require presenting the system builder with both atomic modules and abstracted modules in the Supergraph. Providing this functionality visually in an effective fashion is a challenge we have not yet addressed. It may prove, ultimately, that some method for visually assembling executable graphs from component modules other than by selection from an automatically generated Supergraph is more effective. For example, a user could be allowed to drag and drop modules onto a canvas from palettes of related modules (e.g. taggers, parsers, etc.), connecting them into a graph as they are assembled. Attempts to connect modules in a manner that violated data dependencies could be automatically detected and the user alerted. This approach loses the advantages of allowing the user to see all the data dependencies in advance of creating a system; but it also avoids the confusion that can arise from overly large graphs. Further investigation of such alternatives is necessary.

4.3. Executing Systems in GATE

Once a system graph, which may be thought of as a partially specified program, has been assembled and saved it may be executed visually. Its input may be any document or document collection created using the document collection management facilities in GATE. An example system graph has been shown above in Figure 1.

The modules in the graph are colour coded according to a traffic light metaphor to indicate their current execution state: green (shown in light grey in the figure) modules can be executed immediately; red (dark grey) modules have already been executed, and so cannot be executed again; and amber (white) modules have not had the modules on which they depend executed, and so cannot be executed yet.

To execute a 'ready' module (a green module), the user selects it with one mouse click (at which point it turns white), then executes it with a second click. When processing is complete the module turns red and the downstream modules whose input dependencies are now satisfied automatically turn green. The user may proceed to select and execute them, or may view the results of the module just executed, by clicking on it and selecting from one or more results viewers presented in a pop-up menu. The visualisation of module results is further discussed in Section 5.2. In the case of 'supermodules' which represent a subgraph the user may either run the module as a whole, or select to expand it into the subgraph in which case a separate window appears containing the subgraph and the individual modules in it may be run in precisely the same way as modules are run in the main graph.

Separating the operations of module selection (first mouse click) and execution (second mouse click) allows the user to select a path through the graph and then execute all modules in the path at once with a single second click on the final module. The selected modules in the dataflow graph are executed in a data-driven manner, wherein modules are executed as soon as their input data are available. The modules in the selected path turn red to indicate completion as execution proceeds. This facility allows the user to 'batch' the modules to be executed, which is useful if he or she wants to run the system up to a given point before examining any output.

A more interactive mode is also possible. Since each module's output is known from its data dependency specification, it is possible to reset a module by deleting from the database the data it outputs. The user accomplishes this by clicking on a 'red' module (an executed module) at which point he or she is prompted as to whether to reset the module. When a module is reset all dependent modules are also reset in order to ensure that no downstream modules are executed with out-of-date data. In the case of modules which modify data in the database, as opposed to adding new data, resetting the module causes the earliest module which creates the data (and any intervening dependent modules) to be reset.

This is a useful feature for NLP module developers, as it allows for a development cycle where it is possible to gain immediate feedback on changes made to the underlying code or resource (e.g. a declarative grammar) on which the module depends. A module can be executed, the results observed, changes to the module made, the module reset and then executed again.

At any point in executing a system graph the user may choose to exit the system, saving results obtained so far if this is desired. If results are saved then restarting the system on the same document at a later time restores the visual execution state as it was when the system was exited, i.e. modules will colour (red, amber, green) as appropriate.

The default processing strategy is the module-by-module strategy—each module is always executed on all of the text units in the input data available to it (document or document collection) before the next module is executed.

However, there are two exceptions to this. First, if a supermodule which represents a benign cycle is encountered and the user chooses to expand the supermodule, then the component modules within this supermodule, which by definition form a minimal loop, will execute text-unit-by-text-unit. Second, it is possible for the user to choose to execute any single module in a more flexibly controlled manner. In this mode, which is similar to the execution model of many debuggers, the user may choose either to single step through the input annotations, or may select a particular input annotation as a breakpoint, in which case the module is run iteratively over all input annotations up to and including the breakpoint annotation, at which point it halts. The user effects this behaviour by first setting a step parameter for the module and then optionally setting a breakpoint annotation (default iteration behaviour is single step). Breakpoint annotations are set by presenting the user with a view of the relevant annotations using the generic annotation viewer (Figure 3), from which the user selects one with a mouse operation.

4.3.1. Discussion

Our experience with the executable graph and feedback from other users has been extremely positive. In particular, the executable graph has these advantages:

(1) it supports fine-grained control of execution throughout system and allows focused intervention to view intermediate results and to edit, reload and retest in an easy manner;
(2) visual execution means no need to learn, remember or type complex sequences of commands—execution is controlled through a simple series of mouse clicks;
(3) the overall data dependencies and state of execution of the system are visible at all times in a synoptic fashion; if the execution state is saved and restarted later the execution state is immediately apparent.

These and other advantages of the executable graph may be seen as specific instances of the well-known advantages afforded by direct manipulation interfaces in general [23].

5. Viewing and Analysing Results in GATE

While the capabilities to build and execute NLP systems visually are of considerable value, the most important aspect of the GATE environment is the capability to visualise the results of NLP modules. NLP systems are complex, in both their individual components and their interaction effects, and their development mostly consists of iterative refinement on test data. The chief problems in examining results are the volume of output data, viewing information about text and the text itself simultaneously, and detection of differences between one run of a system and another, following changes to code or data resources, such as grammars or lexicons.

Writing separate results viewers for each module in a system is a time-consuming activity and should be avoided if possible. In many cases results viewers are reusable for any suitably similar types of module output. GATE makes available a number of generic results viewers, each appropriate for a specific type of output, which require little or no reconfiguration for use with new modules generating similar types of data. Comparing results between two runs of a module is also an activity the details of which are largely dependent on the type of the results. Some reusability can be achieved by having a generic tool for comparing results between two sets of output from a module for a document, with specialised procedures for comparing any two instances of an output data type. If new modules are written whose output types correspond sufficiently to those for which comparison procedures already exist, then no new work needs to be undertaken to supply a tool for comparing results across multiple runs.

Given the importance of output result types in partitioning the space of results viewers and comparison procedures, we start this section with a classification of common types of text annotation and discuss how these annotations are likely to be represented in an indexed annotation scheme like the TIPSTER/GATE data model. In Sections 5.2 and 5.3 we show how this classification is used to develop generic visual tools for viewing and comparing output respectively.

5.1. A Classification of Text Annotations

This classification makes no attempt to cover every possible form of information that is likely to be associated with a text by a language-processing module (see [24] for a current survey of the field of corpus annotation). In the general case, any sort of information whatsoever, represented using any data type, might be associated with a text. The TIPSTER/GATE data model effectively supports this general case by allowing attributes of documents or document spans to be arbitrary strings; these can be used to encode any sort of information.

However, there are a number of activities which regularly recur in language processing, and provided the output from these activities is represented in a regular fashion, generic tools which display or compare this output can be designed. The types of annotation we have identified as resulting from common text processing activities are: segmentation, tagging, simple and abstract relations. We discuss these in turn.

(1) *Segmentation.* At its most basic level a text is simply a sequence of characters. Most language processing manipulates higher level units, and presupposes a segmentation of the text into these units. Typical higher level units are: words or tokens, sentences, paragraphs, etc. Processes that carry out this segmentation vary depending on the type of unit, the language, and the text type and markup, and may be far from trivial. They have in common that they divide the undifferentiated sequence of text units at a lower level into units at a higher level. In the TIPSTER scheme segments are typically represented simply by means of a specific single-span

annotation type for that segment; e.g. token segments are represented by a beginning and ending byte offset and an annotation type token. Segments at any given level are assumed to be non-overlapping sequences of contiguous characters in the underlying character stream.

(2) *Tagging.* Taggers classify contiguous text units, typically one or more segments identified by a segmentation process, into one of a finite number of enumerated classes. Typical examples of this process are part-of-speech taggers and named-entity taggers (e.g. taggers that tag names into classes such as person, organisation, location, etc.). In the TIPSTER model tag annotations are naturally represented as attributes on annotations of the type of unit being tagged (e.g. a part-of-speech attribute on a word (segment) annotation).

(3) *Simple relations.* Simple relational annotations indicate that some relation holds between two or more spans of text. These spans may or may not be contiguous or overlapping. Simple relations should be contrasted with abstract relations, discussed next, where the relation may hold between entities which are themselves defined by a relational annotation. Examples of simple relations include columns of data in textual tables, anaphoric reference (e.g. relations between pronouns and their antecedents), dependency relations in dependency grammars, semantic relations (e.g. the employee_of relation holding between persons and organisations as instantiated in newswire texts). Such relations can be represented in the TIPSTER/GATE model either as a single multiple-span annotation, one span for each related text segment, or as multiple single-span annotations, where each annotation has an attribute containing pointer to one or more other annotations in the relation. The latter representation is more general in that in the case of multi-link relations (relations involving three or more text spans) it allows arbitrary information to be stored with each link, such as direction, thus supporting arbitrary graph data structures linking text spans.

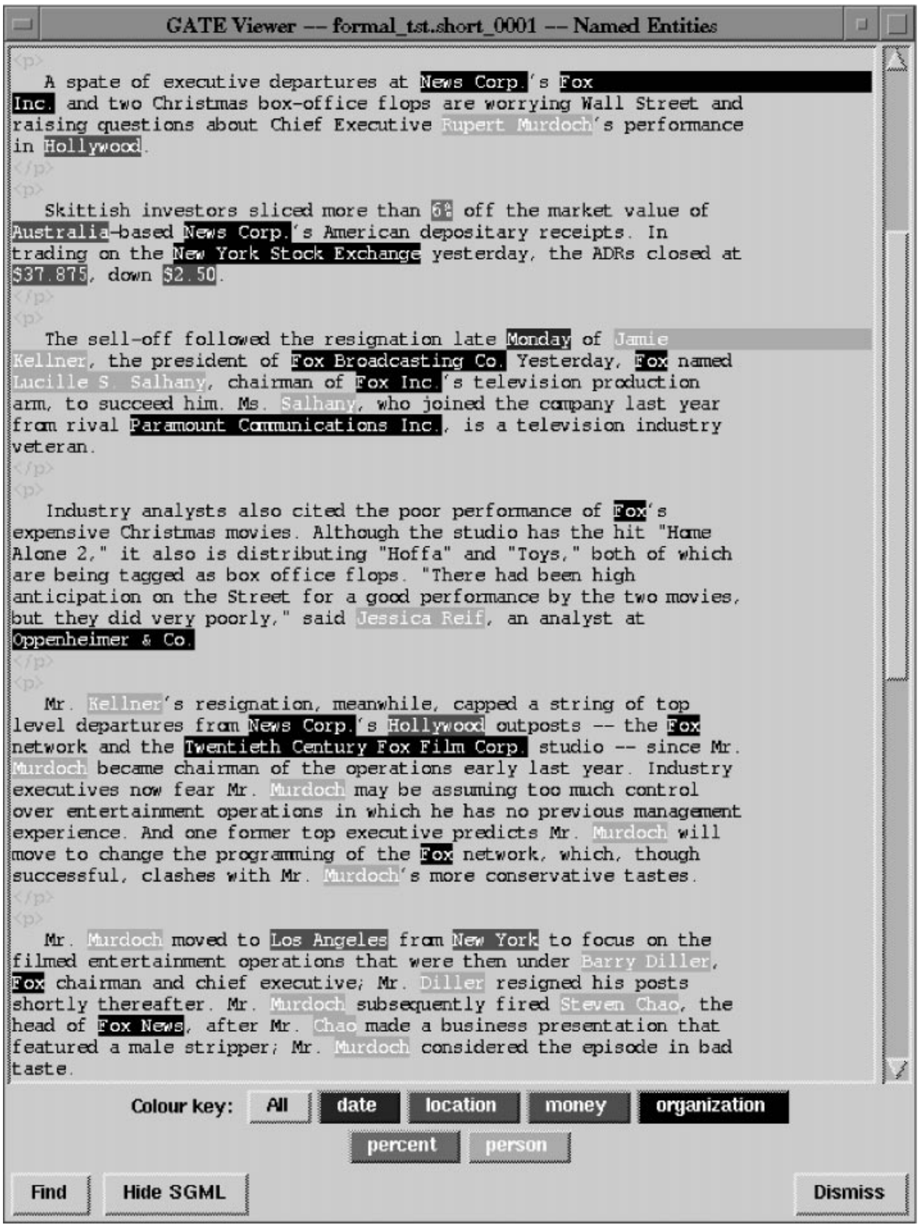
(4) *Abstract relations.* An abstract relational annotation is one that relates textual entities that themselves are defined via a relation annotation. They thus abstract away from the underlying text, though since they are annotations they retain a textual referent — the set of spans in the text related by their component relations, or by their component relations' component relations, and so on. The most common sort of abstract relation seen in analysing text is constituency. Constituency analysers determine that one sequence of text units is a constituent of a larger unit, which in turn may be part of a larger unit, and so on. Typical examples of constituent analysers are parsers for phrase structure grammars or text grammars and the simplest and most common kind of constituency analysis is a tree-structured analysis. Tree structures may be represented in the TIPSTER/GATE data model by one annotation per node with an attribute specifying the constituents (e.g. to specify that a particular sentence consists of a noun phrase and a verb phrase we create a syntax annotation that spans the character sequence of the whole sentence and possesses two attributes, a category attribute, with value sentence in this case, and a constituents attribute whose value in this case would be a list containing pointers (the annotation ids) to the syntax annotations of the noun phrase and the verb phrase). In a similar fashion arbitrarily complex abstract relations can be represented in the TIPSTER/GATE model.

5.2. Viewing Module Results

The data visualisation tools or viewers described here are designed with generality in mind, so that the output of many different modules can be used with them. To this end they follow the annotation classification system, defined in Section 5.1, and rely on a conventional representation of each type of annotation in the TIPSTER/GATE data model. Segmentation, tagging, simple relation and tree viewers are all present within GATE, as well as a raw annotation viewer. We first describe how these viewers work and then discuss the benefits they convey.

As indicated in Section 4.3, a viewer is generally invoked from the execution graph by selecting it from a pop-up menu that can be obtained by clicking on a module in the graph once it has been run. Several viewers may be associated with each module. This association is defined in the module configuration file which contains a list of annotation type-viewer-label triples—the annotation type serves as a parameter to the viewer which instructs it, when invoked, as to which annotations to retrieve from the database (see Figure 6 for an example); the label is the name for the viewer which appears in the pop-up menu in the interface. For new modules which produce annotation results of one of the conventional types discussed above, adding this viewer declaration to the configuration file is all the user needs to do to be able to use the viewer. Of course, custom viewers may be specified as well.

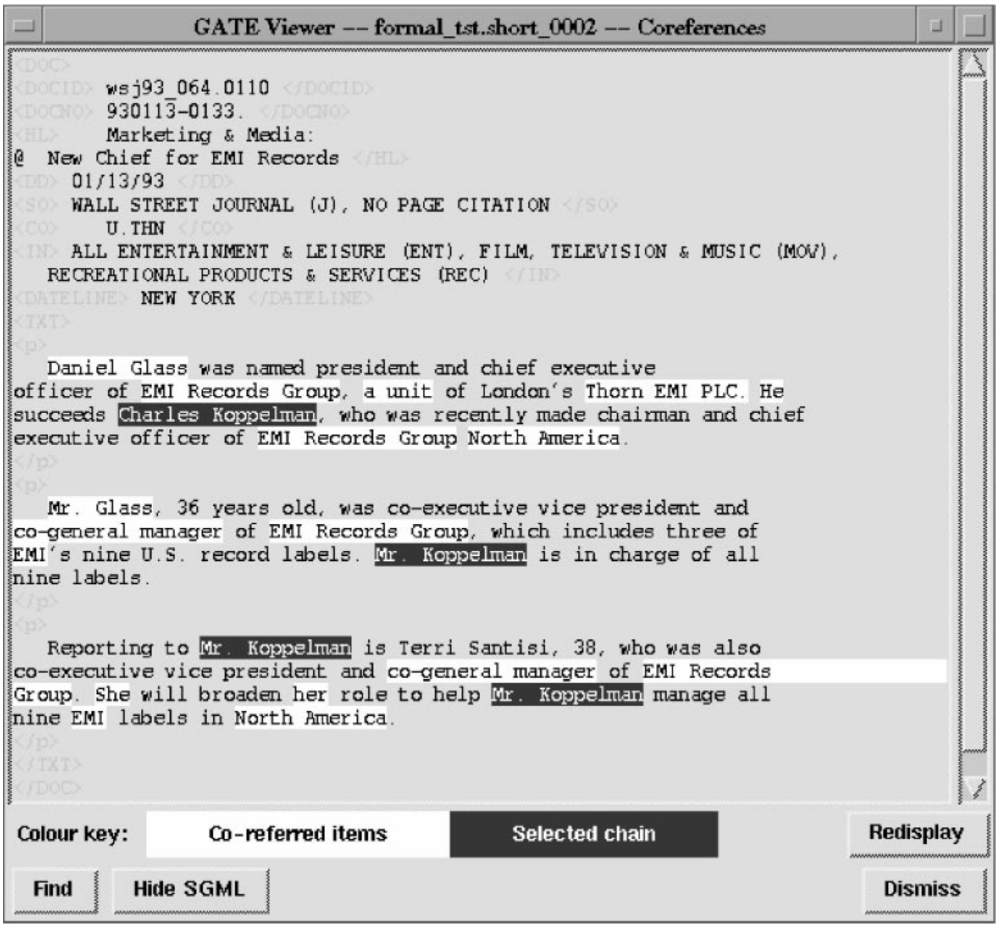
(1) *Segmentation viewer.* The segmentation viewer simply displays the source text, highlighting in it each segment of the type it is instructed to display (word, sentence, paragraph, etc.). As the user moves the mouse over a highlighted segment, it reverses the colour of the highlighting so that the segment stands out from adjacent segments.
(2) *Tag viewer.* The tag viewer highlights each tagged text span of a given tag class in the same colour, using distinct colours for each tag class in the document. Buttons are provided at the bottom of the display window, each one labelled with a tag label and coloured in the same colour as the highlighted members of its class. Clicking on the button turns off highlighting on all tag classes other than the one selected. An ALL button permits full highlighting to be restored. The number of buttons, their labels and colours are dynamically determined by the number of tag classes found in the data and do not need to be specified in advance by the user. See Figure 8.
(3) *Simple relations.* At present, only a restricted form of simple relation viewer is implemented, one that works on multiple span annotations only. Each span within each of the multiple span annotations of the selected type is highlighted throughout the text. Clicking on any highlighted span in the text causes all the other spans in the same annotation to change the colour in which they are highlighted. So for example, suppose coreference relations are represented using multi-span annotations (i.e. each coreference chain is represented as a single annotation with one start–end byte offset pair for each element in the chain). The viewer shows all elements in all chains coloured yellow; clicking on any element turns all elements in the selected chain blue. Clicking on an element in another chain turns that other chain blue and the first chain reverts to yellow. See Figure 9.
(4) *Tree viewer.* The only sort of abstract relation viewer currently available is a viewer for tree structure relations. The tree viewer works in two stages. First, a viewer similar to



<p>
A spate of executive departures at News Corp.'s Fox Inc. and two Christmas box-office flops are worrying Wall Street and raising questions about Chief Executive Rupert Murdoch's performance in Hollywood.
</p>
<p>
Skittish investors sliced more than 6% off the market value of Australia-based News Corp.'s American depositary receipts. In trading on the New York Stock Exchange yesterday, the ADRs closed at $37.875, down $2.50.
</p>
<p>
The sell-off followed the resignation late Monday of Jamie Kellner, the president of Fox Broadcasting Co. Yesterday, Fox named Lucille S. Salhany, chairman of Fox Inc.'s television production arm, to succeed him. Ms. Salhany, who joined the company last year from rival Paramount Communications Inc., is a television industry veteran.
</p>
<p>
Industry analysts also cited the poor performance of Fox's expensive Christmas movies. Although the studio has the hit "Home Alone 2," it also is distributing "Hoffa" and "Toys," both of which are being tagged as box office flops. "There had been high anticipation on the Street for a good performance by the two movies, but they did very poorly," said Jessica Reif, an analyst at Oppenheimer & Co.
</p>
<p>
Mr. Kellner's resignation, meanwhile, capped a string of top level departures from News Corp.'s Hollywood outposts -- the Fox network and the Twentieth Century Fox Film Corp. studio -- since Mr. Murdoch became chairman of the operations early last year. Industry executives now fear Mr. Murdoch may be assuming too much control over entertainment operations in which he has no previous management experience. And one former top executive predicts Mr. Murdoch will move to change the programming of the Fox network, which, though successful, clashes with Mr. Murdoch's more conservative tastes.
</p>
<p>
Mr. Murdoch moved to Los Angeles from New York to focus on the filmed entertainment operations that were then under Barry Diller, Fox chairman and chief executive; Mr. Diller resigned his posts shortly thereafter. Mr. Murdoch subsequently fired Steven Chao, the head of Fox News, after Mr. Chao made a business presentation that featured a male stripper; Mr. Murdoch considered the episode in bad taste.

Figure 8. Tag viewer

the segmentation viewer is invoked, using highlighting in the text to show which spans are covered by a tree. The user then clicks on one of these spans and the tree covering that span is displayed in another window (see Figure 10). The tree display window is horizontally and vertically scrollable to allow the user to move about large trees. Each node is labelled with the value of a designated attribute from the



GATE Viewer -- formal_tst.short_0002 -- Coreferences

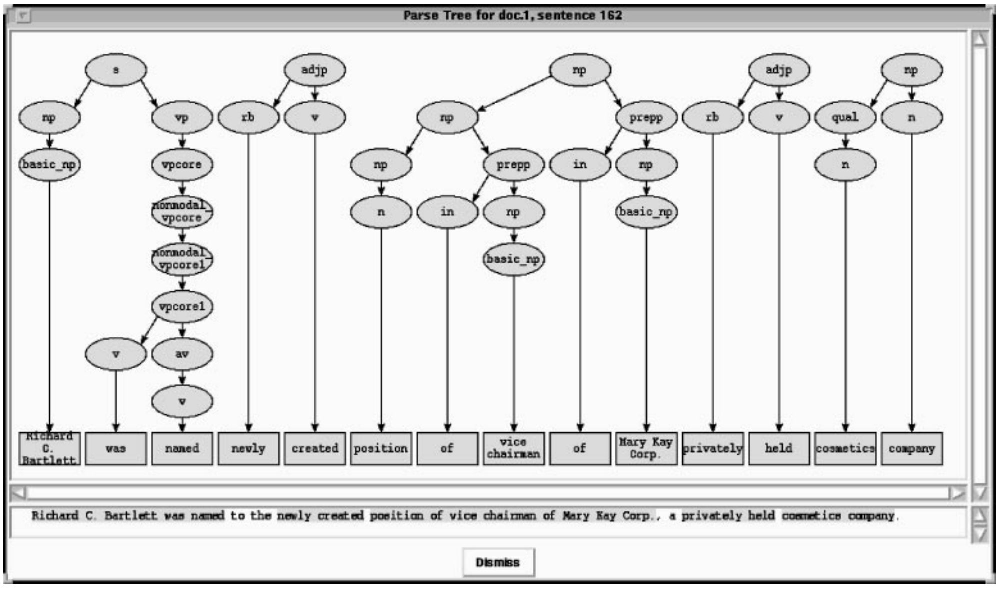
```
<DOC>
<DOCID> wsj93_064.0110 </DOCID>
<DOCNO> 930113-0133. </DOCNO>
<HL>    Marketing & Media:
@  New Chief for EMI Records </HL>
<DD>  01/13/93 </DD>
<SO>  WALL STREET JOURNAL (J), NO PAGE CITATION </SO>
<CO>    U.THN </CO>
<IN>  ALL ENTERTAINMENT & LEISURE (ENT), FILM, TELEVISION & MUSIC (MOV),
   RECREATIONAL PRODUCTS & SERVICES (REC) </IN>
<DATELINE>  NEW YORK </DATELINE>
<TXT>
<p>
   Daniel Glass was named president and chief executive
officer of EMI Records Group, a unit of London's Thorn EMI PLC. He
succeeds Charles Koppelman, who was recently made chairman and chief
executive officer of EMI Records Group North America.
</p>
<p>
   Mr. Glass, 36 years old, was co-executive vice president and
co-general manager of EMI Records Group, which includes three of
EMI's nine U.S. record labels. Mr. Koppelman is in charge of all
nine labels.
</p>
<p>
   Reporting to Mr. Koppelman is Terri Santisi, 38, who was also
co-executive vice president and co-general manager of EMI Records
Group. She will broaden her role to help Mr. Koppelman manage all
nine EMI labels in North America.
</p>
</TXT>
</DOC>
```

Figure 9. Relation viewer

annotation representing that node (such as syntactic category—e.g. NP for noun phrase). In a separate canvas at the bottom of the window the text spanned by the tree is displayed (this text is fixed so that scrolling about the tree always leaves the text in view). Selecting any node in the tree highlights the text that it spans in the bottom canvas and pops up a display containing any additional attribute information associated with that node (e.g. in a feature-based syntactic analysis environment this can be used to display feature structures associated with nodes).

(5) *Raw annotation viewer*. The raw annotation viewer simply displays all annotations of the requested type in raw form, as they occur in the database of annotations (see Figure 3). Clicking on any one annotation shows the source text in another window with the span(s) of the annotation highlighted in the text.

Annotations may also be viewed without being invoked via a module results viewer. A drop-down menu option (View) appearing with each executable graph allows several viewing options.



Richard C. Bartlett was named to the newly created position of vice chairman of Mary Kay Corp., a privately held cosmetics company.

Figure 10. Tree viewer

(1) By selecting Annotations the raw annotation viewer described above is loaded with all annotations currently in the database. A filter facility allows all annotations not of a selected type to be discarded; a find facility allows simple regular expression matching across the annotations. As before, clicking on an annotation brings up another window with the annotation span highlighted in the source text.
(2) By selecting Text the text of the currently open document is displayed and the user may sweep the mouse across a region of text and then request that the annotations for this text region be displayed. Annotations of all types, whose spans are contained within this region are then displayed using the raw annotation viewer.

5.2.1. Advantages of the Visualisation Tools and Environment

Both the visualisation tools and the visual environment into which they are integrated (the executable data flow graph) offer advantages to the system developer. Considering first the generic tools described above simply on their own, as visualisation tools for data about texts, the following strengths may be noted.

(1) *Relating annotations and text.* By definition, text-processing modules, or annotators, produce information about texts. A key problem for anyone attempting to ascertain whether the module is working correctly (a developer) or what it is doing (a student) is to be able to relate a module's outputs for a text portion back to that text portion. In both the indexed annotation model, where the annotations are stored separately from the text, or in the markup model (e.g. SGML) where the annotations are embedded in the text there are cognitive problems for the user since he now has two separate textual information sources to attend to—the text being processed and the

annotations in textual form. In the indexed annotation approach two physically distinct textual sources need to be attended to and co-ordinated simultaneously. This leads first to arduous alignment difficulties and then requires switching focus back and forth between two textual sources (the negative impact of this sort of 'split attention' on learning has been empirically verified by [25]). In the markup approach, where the source text and textual annotations are interleaved, the original text quickly becomes imperceptible in a jumble of interwoven markup.

Relating annotations and text is best done from a user's perspective if the text remains in an accessible and relatively stable form (i.e. is not altered through the addition of textual markup) and yet by directing attention to different parts of the text the information conveyed by the annotations becomes viewable at the same time in a non-textual manner. Clearly, a sensible way to do this, where the information to be conveyed is simple enough to be expressed in this way, is through the use of colour, font or simple layout changes—what Green and Petre [26] call 'secondary notations' and Waller [27] refers to as 'diagramming a text'. We have made extensive use of colour, as the descriptions above make clear. This eliminates the problem of relating two sources of textual information, since there is now only one source of textual information to attend to; the other is presented in a different perceptual mode and alignment comes for free.

(2) *Searching for results.* A related point concerns searching for the relevant output. Typically, following a change to a module, either its effects on a particular section of text or its effect on a given phenomenon across the entire text will be of interest. For example one might want to know whether the second sentence in the third paragraph is now being tagged correctly, or one might want to know whether all adjectives are being tagged correctly across a text. In the former case the text serves as the best possible index to itself—the user simply wants to go to the relevant place in the text and see if it has been correctly processed. Again, secondary notations provide a mechanism where the original text can be searched and the information then read off it without further ado. In the latter case, colour coding of the relevant phenomenon allows all instances across the text to be picked out easily. Of course, while this allows false positives to be easily identified it does not help in detecting false negatives.

(3) *Context.* Yet another reason for treating the original source text as a fixed object in relation to which the user can see or explore added information as transparently as possible is that the context of any phenomenon being attended to remains accessible just as in the original. Since so much of language processing depends on context, any scheme for presenting results which did not allow easy access to results for neighbouring text elements (words, phrases, sentences) would inhibit an investigator in exploring how treatment of nearby text may have had bearing on the phenomenon being directly investigated. Again, being able to use the source text as a direct visual index to these results is much preferable either to searching a separate and perhaps differently organised database of information about the text, or searching through a marked up version of the original text in which the original spatial relations between text elements have been arbitrarily disturbed.

(4) *Colour vs. links.* An alternative to the use of colour plus dynamic highlighting to indicate simple relations and particular relation instances, is to use links explicitly drawn on the text. This approach was used in a tool for displaying coreference

chains developed by SRA International and supplied to participants in the MUC-6 coreference evaluation [28]. While this approach has the advantage that links can indicate directionality, its chief disadvantage is the visual clutter that results from large numbers of crossing links participating in separate relations. Of course, a technique could be adopted, similar to the one we have used, of showing all chains initially, but then allowing a single chain to be selected so that it stands out. However, since the extent of the related text elements must still be indicated (e.g. by colouring or underlining), it is not clear that explicit links add any further information (except directionality), and do visually obscure the text, even when they are not overlapping.

(5) *Abstraction.* Of course not all annotation information can be displayed using colouring on text. This is particularly true when dealing with abstract relations, since some symbol for each abstracted relation needs to be introduced so that it can take part in further relations. The only viewer we have introduced for abstract relations is the tree viewer described above, which deals with a very simple type of abstract relations. Two principles underlying the design of this viewer are worth pointing out. First, the trees are still visually linked directly to the text via the first stage in the viewing process wherein the user sees highlighted spans of text for which there are associated tree annotations—thus, as in the case of colouring techniques described above, search for the annotation associated with an input is minimised. Second, the tree is presented graphically as a tree, rather than textually as, e.g. a bracketed string. This diagrammatic representation brings with it all of the well-known advantages of diagrammatic over sentential representations [29]. Allowing related information to be grouped proximately in a two-dimensional plane (i) reduces search in problem solving, (ii) eliminates the need for symbolic labels linking related information and the associated label-matching overhead, (iii) exploits the human strengths for drawing perceptual inferences.

In addition to their individual strengths as data visualisation aids, these tools acquire greater value because of their close coupling with the execution model in GATE.

(1) *Linking execution and results viewing.* Access to the viewers directly from module nodes in the executable data flow graph allows users quick and intuitive access to the output from a module after running it. Resetting a module, rerunning it and viewing the output, after editing the module, can be accomplished in a few mouse clicks, all in the same screen area.

(2) *Viewing output from multiple modules.* Access to the results of all the modules in an execution graph allows the user to explore rapidly hypotheses about errors through the sequence of modules which lead up to the current one. For example, questions such as 'Was this parsing error the result of a part-of-speech tagging error?' can be answered quickly and easily.

5.2.2. Residual Problems

While the approach to designing viewers described above has led both to reusable viewers and to viewers which have proven very useful in practice, there are some areas where further work needs to be done.

(1) *Ambiguous tagging*. In many cases text annotators (human or machine) may want to produce multiple annotations for the same text unit. At present our viewers, with the exception of the raw annotation viewer do not cope with this: one annotation alone, the first found in the database, is displayed. Various strategies could be adopted to circumvent this problem. For tag viewers, if the number of ambiguities is small and of low order (i.e. two or three readings as opposed to dozens) then new tags representing the ambiguity classes could be introduced and included in the key at the bottom of the viewer. Alternately, a single 'ambiguous' tag colour could be used to indicate that there are multiple readings which could be then be displayed via a popup when the tagged item is selected. In the case of multiple abstract relational annotations for one text span, e.g. multiple parses, a similar technique could be used to indicate the presence of multiple analyses. Displaying them all, however, may be less useful in this case simply because of the complexity of these annotations. Typically, the user wants to become aware of the differences in analysis, and some visual way to support this process needs to be found.

(2) *Overlapping tagging*. A related problem is that of overlapping or nested tags. This may or may not be the result of ambiguous tagging. For example, if a place name is embedded in a company name (e.g. *New York Times*), then a named entity tagger might be designed to tag both. For the case of the simple viewers based on text colouring, similar solutions to those proposed for ambiguous tagging may be adopted. If both ambiguity and overlapping may be present simultaneously, however, care needs to be taken not to confuse the conventions for flagging them, or to make the flagging scheme too complex.

5.3. Comparing Module Results

In addition to viewing the output of a module, it is frequently necessary to compare or evaluate the output of one version of a module with another (during development), or against a manually produced 'gold standard' output (during benchmarking). There are typically two requirements in comparing one set of results with another. First, one wants to get an assessment or score—an overall numerical measure of how close one set is to the other. Second, one wants to get a diagnosis—an indication of where the differences are between the two sets of results. While there is not much need for visual techniques in presentation of scores—these are usually a small set of numbers or a numerical table—there is a real need, and considerable scope, for using visual techniques to help to pin-point differences between result sets. Even the simplest module running on a modest-sized 'real' text generates more textual output, frequently representing complex data structures, than can be readily comprehended. Comparing two such sets for differences manually is time-consuming, exhausting and error-prone. Using text comparison tools such as the Unix utility diff leads to output which may still be too voluminous to digest and which is difficult to comprehend, since such tools capture differences at the byte level, rather than in terms of objects in the problem domain. Successful visualisation tools therefore can massively speed up the process of development by focusing developers' attention on the problem area and by helping developers literally 'see' a problem in the domain, as opposed to seeing differences in symbol sequences that need to be interpreted in terms of the domain.

As with the viewing of results, one might hope to achieve reusability of comparison modules by developing them with respect to the annotation types distinguished in Section 5.1. We are in the process of pursuing this programme, but so far have only created a generic comparison tool for comparing segmentation and tag-type annotations. Designing useful visual comparison tools for complex data structures such as trees is extremely difficult. However, we have also integrated several existing special-purpose scoring tools developed by the NLP community into the GATE environment. While not generic, these tools still contribute significantly to the utility of GATE as an environment for the testing and development of language-processing modules, and demonstrate the ease with which externally developed code can be 'plugged in' to GATE.

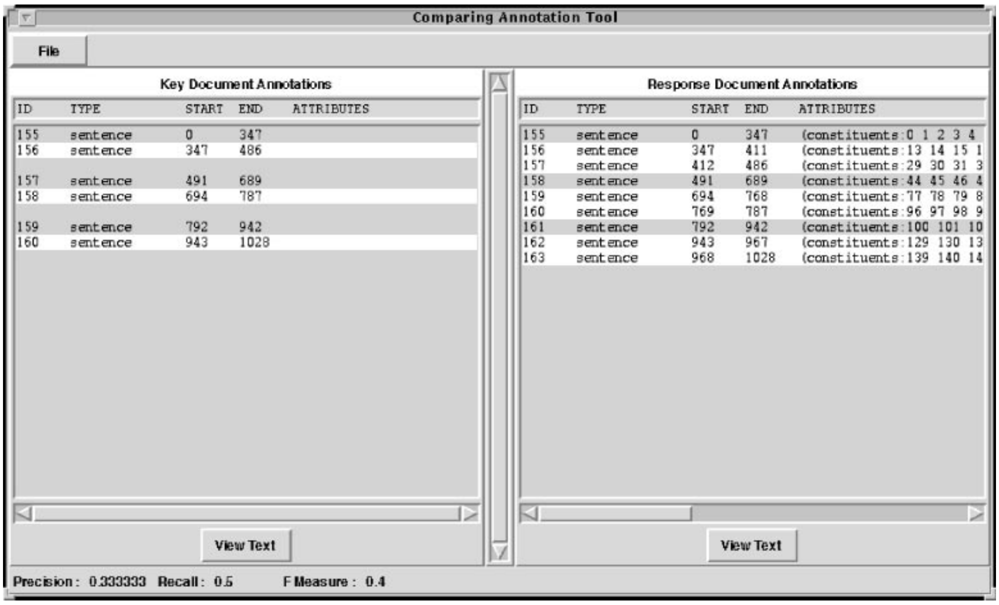
As indicated above, there are two modes in which comparison is usually carried out: the incremental comparison of one module under development with earlier versions of itself (regression testing) and the comparison of a module against either another module performing the same task or against a manually created correct data set. In order to allow a manually marked-up set of data to be created, some sort of annotation editor, generally more powerful than a text editor is required. We have developed a general-purpose annotation tool called MAT—Manual Annotation Tool.

In the rest of this section we first describe CAT, a prototype Comparing Annotations Tool, then describe MAT, and finally briefly describe contributory evaluation tools which have been integrated in GATE.

5.3.1. A Tool for Comparing Annotations

Segmentation- or tag-type annotations may be compared using the comparing annotations tool (CAT), shown in Figure 11. To use it, the user enters into a dialogue to select two versions of the same document which are already annotated. They may have been annotated using different versions of the same module, by two different modules, or the annotations of one may have been created using MAT, or some other manual annotator. One of these is designated the key, or 'truth', and the other the response, the hypothesis to be scored against the truth. The user must also select a predicate on which to compare the data in the two documents. This requires specifying the annotation type which is to be compared, and which attribute values of that type will be considered in the comparison.

Once the user is satisfied with the definition of the comparison, the comparison is carried out. In a manner similar to the *tkdiff* utility [30], the selected annotation type and attributes of both versions of the document are then displayed in parallel in two aligned viewing panels, each panel being an instance of the raw annotation viewer. Annotations which differ are shown in a different colour compared to those which are the same. Annotations differ either because their span cannot be matched in the other data set or their spans match but they differ in the specified attributes. As ever, clicking on an annotation in either raw annotation viewer displays the text spanned by that annotation, highlighted in another window. A single slider in the middle of the two display panels lets the user scroll up and down in the two viewing panels, which stay aligned, as best as possible, on annotation byte offsets (since both the number of annotations and the identification numbers assigned to annotations may differ between the two annotation sets).



Key Document Annotations

ID	TYPE	START	END	ATTRIBUTES
155	sentence	0	347	
156	sentence	347	486	
157	sentence	491	689	
158	sentence	694	787	
159	sentence	792	942	
160	sentence	943	1028	

Response Document Annotations

ID	TYPE	START	END	ATTRIBUTES
155	sentence	0	347	(constituents:0 1 2 3 4
156	sentence	347	411	(constituents:13 14 15 1
157	sentence	412	486	(constituents:29 30 31 3
158	sentence	491	689	(constituents:44 45 46 4
159	sentence	694	768	(constituents:77 78 79 8
160	sentence	769	787	(constituents:96 97 98 9
161	sentence	792	942	(constituents:100 101 10
162	sentence	943	967	(constituents:129 130 13
163	sentence	968	1028	(constituents:139 140 14

Precision: 0.333333 Recall: 0.5 F Measure: 0.4

Figure 11. CAT—the Comparing Annotations Tool

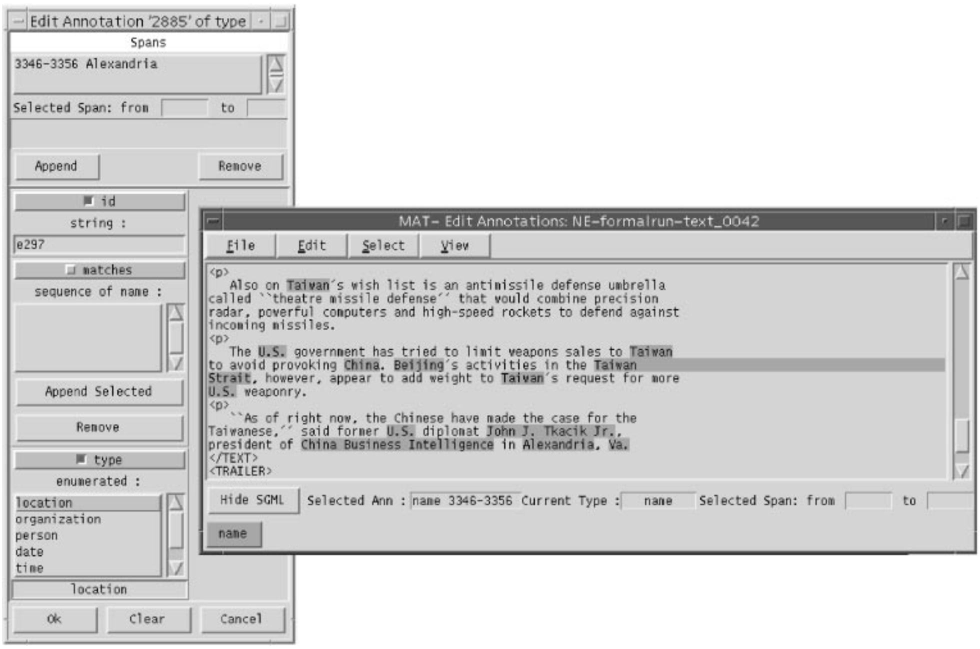
In addition to highlighting differences in the two annotation sets, numerical scores are calculated using the well-known measures of recall, precision and *F*-measure [31, 32]. Recall is the percentage of key results found in the response; precision is calculated as the percentage of response results found in the key. The *F*-measure is a combination of recall and precision that rewards maximising both over maximising one at the expense of the other and is frequently used by NLP researchers when a single-figure measure is required for assessing systems.

5.3.2. A Manual Annotation Tool

MAT, the Manual Annotation Tool, is a prototype general tool with which it is possible to add, modify and delete instances of all the forms of annotation described in Section 5.1. The user may also define new annotation types, provided they fall within the general class of annotation types supported in the TIPSTER data model.

While MAT is extremely general in terms of the annotation types that can be manipulated with it, it is most appropriate for use with non-abstract annotation types—segmentation, tag and simple relation annotations. This is because the only visual support it offers, at present, is colouring on text to display annotations and mouse-based selection of text (sweeping the mouse over a text span, highlighting the selected span) to create annotations. All information added to text spans (attributes) is done through a separate text-based dialogue box; this includes references to other annotations, which occur, by definition, in all abstract annotations. Visual support for manipulation of abstract annotations remains for the future.

Figure 12 illustrates the use of MAT. The general pattern of use of MAT is as follows. The user first selects an annotation type to work on; existing annotations of this type are



Spans: 3346-3356 Alexandria
Selected Span: from to
id — string : e297
matches — sequence of name :
type — enumerated : location, organization, person, date, time
location

MAT- Edit Annotations: NE-formalrun-text_0042

```
<p>
   Also on Taiwan's wish list is an antimissile defense umbrella
called ``theatre missile defense'' that would combine precision
radar, powerful computers and high-speed rockets to defend against
incoming missiles.
<p>
   The U.S. government has tried to limit weapons sales to Taiwan
to avoid provoking China. Beijing's activities in the Taiwan
Strait, however, appear to add weight to Taiwan's request for more
U.S. weaponry.
<p>
   ``As of right now, the Chinese have made the case for the
Taiwanese,'' said former U.S. diplomat John J. Tkacik Jr.,
president of China Business Intelligence in Alexandria, Va.
</TEXT>
<TRAILER>
```

Selected Ann : name 3346-3356 Current Type : name Selected Span: from to

Figure 12. MAT—the Manual Annotations Tool

then shown via text colouring. An existing annotation may be selected for editing or removal by simply clicking on it; a new annotation span is defined by sweeping the mouse over the text. The type of edit operation is chosen next—add, edit or remove. A dialogue box appears which displays the span or spans of which the annotation is composed (both byte offsets and text) and an area pertaining to each attribute which is part of the type definition for this annotation type. The user may then edit the span(s) associated with the annotation and the annotation's attributes. If the attribute has been defined as an enumerated type then the set of possible values is displayed and the user simply clicks on the desired value. In the example shown in Figure 12, the user has chosen to edit an existing name annotation and is in the process of altering its type attribute from the value person to location, to correct a value incorrectly proposed by a proper name tagging module.

MAT's generality means that for any high-volume annotation task (e.g. annotating hundreds or thousands of documents to create a 'gold standard' corpus) it would almost certainly need specialisation and optimisation, to support maximum throughput for the given task. Thus, MAT is best conceived as a tool for small-scale annotation exercises, or for post-editing machine-created annotations to correct them, or as a prototype to help in thinking about the design issues involved with annotation tools.

5.3.3. Contributed Evaluation Tools

5.3.3.1. MUC scoring software. One major influence in evaluating NLP systems has been the DARPA Message Understanding Conference (MUC) series of evaluations

[28, 33, 34]. The most recent MUC, MUC-7, involved the competitive analysis of information extraction (IE) software performing five tasks—named entity recognition and classification, coreference resolution, and three template-filling tasks. Scorers for all of these tasks were developed by DARPA and have subsequently been integrated into GATE, so that they can be run as modules in a chain of modules making up an IE system, and the results viewed using viewers linked to the GATE interface, as with other modules. Each scorer produces scoring results in the form of a table and a score report which lists differences between key and response. Each of these is displayed verbatim in a text window. The score reports are difficult to read because of the standard difficulty of relating the differences printed in the report back to the original text. For one of these reports, the coreference report, we have produced another viewer by post-processing the report to extract data concerning missing, spurious and correct coreference chains. The standard tag annotation viewer is then used to display text spans falling into these three classes in separate colours on the original text. The result is a viewer that enables coreference errors to be understood more rapidly and in context.

5.3.3.2. Parsing evaluation tools. With the recent availability of large-scale syntactically annotated corpora such as the Penn Treebank [35] and the SUSANNE corpus [36] there has been a surge of interest in metrics for evaluating parsers with respect to a 'gold standard'. We have integrated a number of parsing evaluation algorithms into GATE that compare a parser represented according to the Penn Treebank standard—as labelled, bracketed strings. These include the original parseval algorithm [37], the evalb [38] algorithm, and the treeval algorithm [39]. An evaluation algorithm for parsers based on dependency grammars has also been incorporated [40] which enables them to be evaluated with respect to the SUSANNE corpus.

At present, all of these evaluation algorithms produce numeric scores only, sometimes in tabular format. While these scores may be on a sentence-by-sentence basis, enabling a developer to see where a system is particularly weak, there are no tools to support moving from the score reports to the original text, nor to use visual techniques to highlight where in the tree or dependency structures errors are occurring. Clearly, there is scope for much more work in this area.

6. Concluding Remarks

We have described an integrated visual development environment called GATE which supports the visual assembly and execution of modularised text-based natural language-processing systems and the analysis of results produced by such systems. Such a visual development environment is motivated by (i) the naturalness of assembling complex modular systems via diagrammatic manipulation (ii) the advantages of direct manipulation in executing and viewing the results of interdependent modules in complex systems (iii) the utility of powerful visualisation tools in assisting users to comprehend system output.

The GATE visual development environment is based around the idea that specific language-processing systems may be represented as executable data flow graphs. These graphs are user-selected subgraphs of a 'supergraph', representing all possible data flows, automatically generated from data dependency declarations which a user associates with each module in a global pool of available modules. Users execute the graphs by clicking on nodes representing language-processing modules which then run and allow the data they produce to 'flow' on to subsequent modules in the system. GATE demonstrates how the data flow program graph idea can be usefully deployed in the area of language-processing systems, and shows how various control strategies appropriate for text processing can be supported in such systems, as well as variable levels of user interaction. While such an approach is particularly useful for research and development of language-processing systems, it is by no means restricted to them. It is appropriate for any area where multiple modules process a complex underlying data source, communicate via a shared, global data store, and where the order of processing and the content of intermediate representations is not fixed and needs to be investigated.

GATE also contains various visualisation tools which are specialised for analysing types of information frequently added to texts by language-processing modules, but generic in that the viewers can be immediately reused by any module producing information of one of the supported types. These tools support both viewing the output of individual modules and comparing the output of two versions of a given module, or of a module and a human-prepared 'gold-standard' output. We have argued that these tools lighten 'cognitive load' by facilitating the activity of relating text and information about text, by reducing the amount of search in output, and by displaying results in context. Since the programming of language models in NLP systems largely consists of iterative refinement of rule-bases or statistical models based on analysis of results, powerful visualisation tools enhance this process significantly.

GATE is in a stable, general release and while no systematic evaluation of it has been carried out, the take-up of the system and feedback we have had indicates that it is proving to be an extremely useful tool for NLP researchers and system developers, and even for teaching NLP. Further work, some suggestions for which have been made above, can be carried out on various aspects of the system. However, enough has been done to demonstrate the utility of visual programming and visualisation techniques for natural language processing.

Acknowledgements

The research reported here has been supported by a grant from the UK Engineering and Physical Science Research Council (Grant #GR/K25267). The authors would like to acknowledge the work of Hamish Cunningham on the design and implementation of many aspects of the GATE system. They would also like to thank Shaaron Ainsworth, Hamish Cunningham, and Yorick Wilks for comments on drafts of this paper.

References

1. S.-K. Chang (Ed.) (1990) *Principles of Visual Programming Systems.* Prentice-Hall, Englewood Cliffs, NJ.
2. J. Landauer & M. Hirakawa (1995) Visual AWK: a model for text processing by demonstration. In: *Proceedings VL'95 11th International IEEE Symposium on Visual Languages*, Darmstadt. IEEE Computer Society Press, Silver Spring, MD.

3. J. McWhirter (1995) VGrep: a graphical tool for the exploration of textual documents. In: *CHI '95: Proceedings 1995 Conference on Human Factors in Computing Systems*. ACM, New York.
4. R. K. Larson, D. S. Warren, J. Freire de Lima e Silva & K. Sagonas (1996) Syntactica. MIT Press, Cambridge, MA.
5. R. K. Larson, D. S. Warren, J. Freire de Lima e Silva, O. P. Gomez & K. Sagonas (1997) Semantica. MIT Press, Cambridge, MA.
6. N. K. Simpkins (1994) An open architecture for language engineering. In: *First Language Engineering Convention*, Paris.
7. D. Day, J. Aberdeen, S. Caskey, L. Hirschman, P. Robinson & M. Vilain (1998) Alembic workbench corpus development tool. In: *Proceedings of the 1st International Conference on Language Resources and Evaluation* (*LREC'98*) [12], pp. 1021–1028.
8. W. Skut, T. Brants, B. Krenn & H. Uszkoreit (1998) A linguistically interpreted corpus of german newspaper texts. In: *Proceedings of the 1st International Conference on Language Resources and Evaluation* (*LREC'98*), Granada, pp. 705–711.
9. H.-H. Chen & M.-S. Shaw (1998) A treebank development tool. In: *Proceedings of the 1st International Conference on Language Resources and Evaluation* (*LREC'98*), pp. 725–729, Granada.
10. O. Christ (1994) 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*), pp. 23–32, Budapest, Hungary.
11. A. L. Davis & R. M. Keller (1982) Data flow program graphs. *IEEE Computer* **15,** 26–41.
12. D. D. Hils (1992) Visual languages and computing survey: data flow visual programming languages. *Journal of Visual Languages and Computing* **3,** 69–101.
13. S. B. Steinman & K. G. Carver (1996) Visual Programming With Prograph CPX. Manning Publication.
14. P. J. Rodgers, R. J. Gaizauskas, K. W. Humphreys & H. Cunningham (1997) Visual execution and data visualisation in natural language processing. In: *VL'97 IEEE Symposium on Visual Languages*, Capri, Italy. IEEE Computer Society. Silver Spring, MD. pp. 342–347.
15. H. Cunningham, K. Humphreys, R. Gaizauskas & Y. Wilks (1997) Software infrastructure for natural language processing. In: *Proceedings of the 5th Conference on Applied Natural Language Processing* (*ANLP-97*), pp. 237–244, March 1997. Available as http://xxx.lanl.gov/ps/9702005.
16. R. Gaizauskas, H. Cunningham, Y. Wilks, P. Rodgers & K. Humphreys (1996) GATE—an environment to support research and development in natural language engineering. In: *Proceedings of the 8th IEEE International Conference on Tools with Artificial Intelligence* (*ICTAI-96*), Toulouse, France, pp. 58–66.
17. R. Grishman (1997) TIPSTER Architecture Design Document Version 2.3. Technical report, DARPA. Available at http://www.nist.gov/itl/div894/894.02/related_projects/tipster.
18. C. F. Goldfarb (1990) The SGML Handbook. Oxford University Press, Oxford.
19. R. Gaizauskas, T. Wakao, K. Humphreys, H. Cunningham & Y. Wilks (1995) Description of the LaSIE system as used for MUC-6. In: *Proceedings of the 6th Message Understanding Conference* (*MUC-6*) [12], pp. 207–220.
20. T. Agerwalak & Arvind (1982) Data flow systems: guest editor's introduction. *IEEE Computer* **15,** 10–13.
21. K. Sugiyama, S. Tagawa & M. Toda (1981) Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man & Cybernetics* **11,** 109–125.
22. M. Burnett, M. J. Baker, C. Bohus, P. Carlson, S. Yang & P. van Zee (1987) Scaling up visual languages. *IEEE Computer* **28,** 45–54.
23. B. Shneiderman (1982) The future of interaction systems and the emergence of direct manipulation. *Behaviour and Information Technology* **1,** 237–256.
24. R. Garside, G. Leech & A. McEnery (Eds) (1997) Corpus Annotation: Linguistic Information from Computer Text Corpora. Addison-Wesley, Longman, Reading, MA, New York.
25. P. Chandler & J. Sweller (1992) The split-attention effect as a factor in the design of instruction. *British Journal of Educational Psychology* **62,** 233–246.
26. T. R. G. Green & M. Petre (1996) Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *Journal of Visual Languages and Visual Computing* **7,** 134–174.

27. R. Waller (1982) Text as diagram: using typography to improve access and understanding. In: *The Technology of Text* (D. H. Jonassen, ed.). Educational Technology Publications, Englewood Cliffs, NJ, U.S.A., pp. 137–166.
28. Defense Advanced Research Projects Agency (1995) *Proceedings of the 6th Message Understanding Conference* (*MUC-6*). Morgan Kaufmann, Los Altos, CA.
29. J. H. Larkin & H. A. Simon (1987) Why a diagram is (sometimes) worth then thousand words. *Cognitive Science* **11,** 65–99.
30. J. Klassa (1998) The official tkdiff home page. http://www.ipass.net/klassa/tkdiff/. Site visited 11/11/98.
31. C. J. van Rijsbergen (1979) Information Retrieval. Butterworths, London.
32. N. Chinchor (1995) Four scorers and seven years ago: the scoring method for MUC-6. In: *Proceedings of the 6th Message Understanding Conference* (*MUC-6*) Morgan Kaufmann, Los Altos, CA, pp. 33–38.
33. Advanced Research Projects Agency (1993) *Proceedings of the 5th Message Understanding Conference* (*MUC-5*). Morgan Kaufmann, Los Altos, CA.
34. Defense Advanced Research Projects Agency (1998) *Proceedings of the 7th Message Understanding Conference* (*MUC-7*): Available at http://www.saic.com.
35. M. P. Marcus, B. Santorini & M. A. Marcinkiewicz (1993) Building a large annotated corpus of english: the Penn treebank. *Computational Linguistics* **19,** 313–330.
36. G. Sampson (1995) English for the Computer: The SUSANNE Corpus and Analytic Scheme. Clarendon Press, Oxford.
37. P. Harrison, S. Abney, E. Black, D. Flickinger, C. Gdaniec, R. Grishman, D. Hindle, R. Ingria, M. Marcus, B. Santorini & T. Strzalkowski (1991) Evaluating syntax performance of parser/grammars of english. In: *Proceedings of the Workshop On Evaluating Natural Language Processing Systems.* Association For Computational Linguistics, Connecticut, U.S.A.
38. S. Sekine & M. Collins (1998) Evalb: A bracket scoring program. http://cs.nyu.edu/cs/projects/proteus/evalb/. Site visited 11/11/98.
39. R. Gaizauskas, M. Hepple & C. Huyck (1998) A scheme for comparative evaluation of diverse parsing systems. In: *Proceedings of the 1st International Conference on Language Resources and Evaluation* (*LREC'98*), Granada, pp. 143–149.
40. D. Lin (1995) A dependency-based method for evaluating broad coverage parsers. In: *Proceedings of IJCAI-95*, pp. 1420–1425.