GATE: An Environment to Support Research and Development in Natural Language Engineering

Robert Gaizauskas, Hamish Cunningham, Yorick Wilks, Peter Rodgers, Kevin Humphreys
Department of Computer Science
University of Sheffield
{robertg,hamish,yorick,peterr,kwh}@dcs.shef.ac.uk

Abstract

We describe a software environment to support research and development in natural language (NL) engineering. This environment – GATE (General Architecture for Text Engineering) – aims to advance research in the area of machine processing of natural languages by providing a software infrastructure on top of which heterogeneous NL component modules may be evaluated and refined individually or may be combined into larger application systems. Thus, GATE aims to support both researchers and developers working on component technologies (e.g. parsing, tagging, morphological analysis) and those working on developing end-user applications (e.g. information extraction, text summarisation, document generation, machine translation, and second language learning). GATE will promote reuse of component technology, permit specialisation and collaboration in large-scale projects, and allow for the comparison and evaluation of alternative technologies. The first release of GATE is now available.

1. Introduction

For a variety of reasons, that subfield of artificial intelligence known as *natural language processing* (NLP) has, over the past few years, spawned a related engineering discipline called *language engineering* (LE), whose orientation is towards the application of NL techniques to solving large-scale, real-world language processing problems in a robust and predictable way. These problems include information extraction, text summarisation, document generation, machine translation, second language learning, amongst others. In many cases, the technologies being developed are assistive, rather than fully automatic, aiming to enhance or supplement a human's expertise rather than attempting to replace it.

The reasons for the growth of language engineering include:

- computer hardware advances which have increased processor speeds and memory capacity, while reducing prices;
- the increasing availability of large-scale, language-related, on-line resources, such as dictionaries, thesauri, and 'designer' corpora – corpora selected for representativeness and perhaps annotated with descriptive information;
- the demand for applications in a world where electronic text has grown exponentially in volume and availability, and where electronic communications and mobility have increased the importance of multilingual communication;
- maturing NL technology which is now able, for some tasks, to achieve high levels of accuracy repeatedly on real data.

Aside from the host of fundamental theoretical problems that remain to be answered in NLP, language engineering faces a variety of problems of its own. Two features of the current situation are of prime importance; they constrain how the field can develop and must be acknowledged and addressed. First, there is no theory of language which is universally accepted, and no computational model of even a part of the process of language understanding which stands uncontested. Second, building intelligent application systems, systems which model or reproduce enough human language processing capability to be useful, 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.

The first point means that any attempt to push researchers into a theoretical or representational straightjacket is premature, unhealthy and doomed to failure. The second means that no research team alone is likely to have the resources to build from scratch an entire state-of-the-art LE application system. Note the tension here: the first point

identifies a centrifugal tendency, pushing researchers into ever greater theoretical diversity; the second, a centripetal tendency forcing them together.

Given this state of affairs, what is the best practical support that can be given to advance the field? Clearly, the pressure to build on the efforts of others demands that LE tools or component technologies – parsers, taggers, morphological analysers, discourse planning modules, etc, – be readily available for experimentation and reuse. But the pressure towards theoretical diversity means that there is no point 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.

Our response to these considerations has been to design and implement a software environment called GATE – General Architecture for Text Engineering [5] – which attempts to meet the following objectives:

1. support information interchange between LE modules at the highest common level possible without prescribing theoretical approach (though it allows modules which share theoretical presuppositions to pass data in a mutually accepted common form);
2. support the integration of modules written in any source language, available either in source or binary form, and be available on any common platform;
3. 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.

The rest of this paper motivates and describes the design of GATE. Section 2 describes related work and motivates the underlying data model chosen for GATE. In section 3 we detail the design of GATE. Section 4 illustrates how GATE can be used by describing how we have taken a pre-existing information extraction system and embedded it in GATE. In Section 5 we discuss future work and make some concluding remarks.

2. Managing Information about Text

Research in NLP and computational linguistics has led to the development of algorithms for various tasks which are viewed as component tasks in the overall project of building a computational model of language processing. Part-of-speech (POS) tagging, morphological analysis, and parsing are prototypes of such subtasks. The information these algorithms consume and produce depends on a theoretical conception of the nature of the task they are trying to perform. For example, parsing is the process of determining the syntactic structure of a sentence; but there are many views of what structural relations should be sought (e.g. dependency relations vs. phrase structure) and even given broad agreement about these, there are myriads of grammatical theories each with its own set of categories and features which may or may not map in any simple way onto any other's.

One recent approach to providing a general environment for research and development in LE is ALEP – the Advanced Language Engineering Platform [15] – a project sponsored by the Commission of the European Community (CEC). ALEP aims to provide "the NL research and engineering community in Europe with an open, versatile, and general-purpose development environment" – superficially a similar goal to ours. The approaches are quite different, however. ALEP, while in principle open, is primarily an advanced system for developing and manipulating feature structure knowledge-bases under unification. Also provided are several parsing algorithms, algorithms for transfer, synthesis and generation [14]. As such, it is a system for developing particular types of data resource (e.g. grammars, lexicons) and for *doing* a particular set of tasks in LE in a particular way. ALEP does not aim for complete genericity (or it would need also to supply algorithms for Baum-Welch estimation, fast regular expression matching, etc.). Supplying a generic system to do every LE task is clearly impossible, and prone to instant obsolescence in a rapidly changing field.

In our view ALEP, despite claiming to use a theory-neutral formalism (an HPSG-like formalism), is still too committed to a particular approach to linguistic analysis and representation. It is clearly of high utility to those in the LE community to whom these theories and formalisms are relevant; but it excludes, or at least does not actively support, all those who are not, including an increasing number of researchers committed to statistical, corpus-based approaches. GATE, as will be seen below, is more like a shell, a backplane into which the whole spectrum of LE modules and databases can be plugged. Components used within GATE will typically exist already – our emphasis is reuse, not reimplementation. Our project is to provide a flexible and efficient way to combine LE components to make LE systems (whether experimental or for delivered applications) – not to provide 'the one true system', or even 'the one true development environment'. Indeed, ALEP-based systems might well provide components operating within GATE. Seen this way, the ALEP enterprise is orthogonal to ours – there is no significant overlap or conflict.

In our view the level at which we can assume commonality of information, or of representation of information, between LE modules is very low, if we are to build an environment which is broad enough to support the full range of LE tools and accept that we cannot impose standards

on a research community in flux. What *does* seem to be a highest common denominator is this: modules that process text, or process the output of other modules that process text, produce further information about the text or portions of it. For example, part-of-speech tags, phrase structure trees, logical forms, discourse models can all be seen in this light. It would seem, therefore, that we are on safe common ground if we start *only* by committing to provide a mechanism which manages arbitrary information about text. There are two methods by which this may be done. First, one may embed the information in the text at the relevant points. Second, one may associate the information with the text by building a separate database which stores this information and relates it to the text using pointers into the text. The next two subsections discuss systems that have adopted these two approaches respectively. In the final subsection we compare the two and indicate why we have chosen the second.

2.1. SGML/MULTEXT

MULTEXT [2, 17] is another EC project, whose objective is to produce tools for multilingual corpus annotation and sample corpora marked-up according to the same standards used to drive the tool development. Annotation tools under development perform text segmentation, POS tagging, morphological analysis and parallel text alignment. The project has defined an architecture centred on a model of the data passed between the various phases of processing implemented by the tools.

MULTEXT is based on SGML, the Standard Generalised Markup Language [9]. SGML works by adding extra information to texts in a standard format – the first of the two methods we mentioned above. For example, the (rather short) news article

```
Reuter
Dog bites man. Newshound implicated.
```

might appear in SGML as

```
<DOC>
<HEADERS>Reuter</HEADERS>
<SENT>Dog bites man.</SENT>
<SENT>Newshound implicated.</SENT>
</DOC>
```

Markup is between chevrons, '<' and '>'; slashes signify the end of a marked-up entity. The language is information-neutral (the tags 'DOC', 'SENT' etc. are not part of the language definition) and is encoded in the character set of the source text (e.g. ASCII). Of course arbitrarily more detailed information may be stored about characters, words, or sentences in the text using these conventions. For example we might code part-of-speech information as `<POS type=VBZ>bites</POS>`.

The MULTEXT architecture is based on a commitment to TEI-style (the Text Encoding Initiative [16]) SGML encoding of information about text. The TEI defines standard tag sets for a range of purposes including many relevant to LE systems. Tools in a MULTEXT system communicate via interfaces specified as SGML document type definitions (DTDs – essentially tag set descriptions), using character streams on pipes – an arrangement modelled after UNIX-style shell programming.

MULTEXT endorses the view that SGML is an appropriate and flexible language for the splitting and recombination of text analysis elements. A tool selects what information it requires from its input SGML stream and adds information as new SGML markup. An advantage here is a degree of data-structure independence: so long as the necessary information is present in its input, a tool can ignore changes to other markup that inhabits the same stream – unknown SGML is simply passed through unchanged (so, for example, a semantic interpretation module might examine phrase structure markup, but ignore POS tags). A disadvantage is that although graph-structured data may be expressed in SGML, doing so is complex (either via concurrent markup, the specification of multiple legal markup trees in the DTD, or by rather ugly nesting tricks to cope with overlapping, so-called "milestone tags"). Graph-structured information might be present in the output of a parser, for example, representing competing analyses of areas of text.

Another feature of MULTEXT is a set of abstract data types (ADTs) for all tool I/O [2] supported by a single shared API (Application Program(mers') Interface) for access to the types. An executive (the *tool shell*) glues tools together in particular configurations according to user specifications. The shell may extract sub-trees from SGML documents to reduce the I/O load where tools only require a subset of a marked-up document.

The ADT set forms an object-oriented model, in the sense of using inheritance and data encapsulation, of the data present in a marked-up document. Example classes include **Sentence**, **SentenceBlock** (sequence of sentences), **LexicalWord** (word plus definition from a lexicon). The ADT model reflects the type of processing available in the tool set – there is a type **TaggedSentence**, for example, but not a **ParsedSentence**.

MULTEXT is implemented for the UNIX platform. Access to tools is as unitary programs and via the tool shell; the SGML query language is supported by a C API.

2.2. TIPSTER

The ARPA-sponsored TIPSTER programme in the US, now entering its third phase, has also produced a data-driven architecture for NLP systems [10]. Like MULTEXT, TIPSTER addresses specific forms of language processing, in this case information extraction and document detection (or

information retrieval – IR). As will become clear below, however, TIPSTER's approach is not restricted to particular NL tasks.

Whereas in MULTEXT all information about a text is encoded in SGML, which is added by the tools, in TIPSTER a text remains unchanged while information is stored in a separate database – this is the second of the two methods for storing information about texts we mentioned in the introduction to this section. Information is stored in the database in the form of *annotations*. Annotations associate arbitrary information (*attributes*), with portions of documents (identified by sets of start/end byte offsets or *spans*). Attributes may be the result of linguistic analysis, e.g. POS tags or textual unit type. In this way the information built up about a text by NLP (or IR) modules is kept separate from the texts themselves. In place of an SGML DTD an *annotation type declaration* defines the information present in annotation sets. Figure 1 shows an example from [10].

Text

```
Sarah savored the soup.
0...|5...|10..|15..|20
```

Annotations

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

Figure 1. TIPSTER annotations example

The definition of annotations in TIPSTER forms part of an object-oriented model that deals with inter-textual information as well as single texts. Documents are grouped into *collections*, each with a database storing annotations and document attributes such as identifiers, headlines etc. Collections are the first-class entities in the architecture. The model also describes elements of IE and IR systems relating to their use, with classes representing queries and information needs.

The TIPSTER architecture is designed to be portable to a range of operating environments, so it does not define implementation technologies. Particular implementations make their own decisions regarding issues such as parallelism, user interface, or delivery platform. An implementation in C and Tcl [13] from CRL (the Computing Research Lab, New Mexico State University) implements client-server operation (using Tcl-dp), a server database manager fielding requests from client modules. This implementation is available now and includes both C and Tcl APIs. It is not currently portable beyond UNIX, though Tcl/Tk is becoming available on Windows and Macintosh. GATE uses a simpler, file-based implementation written in C++ and Tcl/Tk. A version for 32-bit PC Windows is planned.

2.3. Comparison of MULTEXT and TIPSTER

Both projects propose architectures appropriate for LE, but there are a number of significant differences. We discuss four here, then note the possibility of complimentary interoperation of the two.

1. MULTEXT adds new information to documents by augmenting an SGML stream; TIPSTER stores information remotely in a dedicated database. This has several implications. Firstly, TIPSTER can support documents on read-only media (e.g. Internet material, or CD-ROMs, which may be used for bulk storage by organisations with large archiving needs, even though access will then be slower than from hard disk). Secondly, TIPSTER avoids the difficulties referred to earlier of representing graph-structured information in SGML. From the point of view of efficiency, the original MULTEXT model of interposing SGML between all modules implies a generation and parsing overhead in each module. Later versions have replaced this model with a pre-parsed representation of SGML to reduce this overhead. This representation will presumably be stored in intermediate files, which implies an overhead from the I/O involved in continually reading and writing all the data associated with a document to file. There would seem no reason why these files should not be replaced by a database implementation, however, with potential performance benefits from the ability to do I/O on subsets of information about documents (and from the high level of optimisation present in modern database technology).
2. A related issue is storage overhead. TIPSTER is minimal in this respect, as there is no inherent need to duplicate the source text. MULTEXT potentially has to duplicate the source text at each intermediary stage, although this might be ameliorated by shifting to a database implementation.
3. There is no easy way in an SGML-based system to differentiate sets of results (i.e. sets of markup) by e.g. the program or user that originated them. In general, storing information about the information present in an SGML system (or *meta-information*) is messy. This is a problem for MULTEXT but not for TIPSTER. A related point is that TIPSTER can easily support multi-level access control via a database's protection mechanisms – this is again not straightforward in SGML.

4. Distributed control is easy to implement in a database-centred system like TIPSTER – the DB can act as a blackboard, and implementations can take advantage of well-understood access control (locking) technology. How to do distributed control in MULTEXT is not obvious.

Interestingly, a TIPSTER system could function as a module in a MULTEXT system, or vice-versa. A TIPSTER storage system could write data in SGML for processing by MULTEXT tools, and convert the SGML results back into native format.

We believe the above comparison demonstrates that there are significant advantages to the TIPSTER model and it is this model that we have chosen for GATE. Note that we believe that SGML and the TEI must remain central to any serious text processing strategy. The points above do not contradict this view, but indicate that SGML should not form the central representation format of every text processing system. Input from SGML text and TEI conformant output are becoming increasingly necessary for LE applications as more and more publishing adopts these standards. This does not mean, however, that flat-file SGML is an appropriate format for an architecture for LE systems. This observation is born out by the fact that TIPSTER started with an SGML/TEI architecture but rejected it in favour of the current database model, and that MULTEXT has gone halfway to this style by passing pre-parsed SGML between components.

3. GATE Design

Corresponding to the three key objectives identified at the end of section 1, GATE comprises three principal elements (see figure 2): GDM, the GATE Document Manager, based on the TIPSTER document manager; CREOLE, a Collection of REusable Objects for Language Engineering: a set of LE modules integrated with the system; and GGI, the GATE Graphical Interface, a development tool for LE R&D, providing integrated access to the services of the other components and adding visualisation and debugging tools.

Working with GATE the researcher will from the outset reuse existing components, and the common APIs of GDM and CREOLE mean only one integration mechanism must be learnt. And as CREOLE expands, more and more modules will be available from external sources.

3.1. GDM

The GDM provides a central repository or server that stores all information an LE system generates about the texts it processes. All communication between the components of an LE system goes through GDM, which insulates

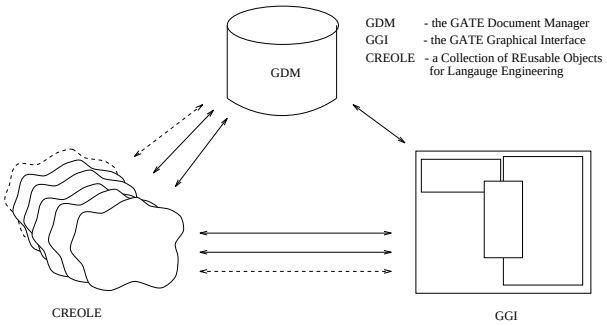

Figure 2. The three elements of GATE

these components from direct contact with each other and provides them with a uniform API for manipulating the data they produce and consume.

The basic concepts of the data model underlying the GDM have been explained in the discussion of the Tipster model in section 2.2 above. The TIPSTER architecture has been fully specified [10] and its specification should be consulted for further details, in particular for definitions of the API. The GDM is fully conformant with a core subset of this specification

3.2. CREOLE

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

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

There are two ways to provide the CREOLE wrapper functions. Packages written in C, or in languages which obey C linkage conventions, can be compiled into GATE directly as a Tcl package (see [Ous94]). This is *tight coupling*. Alternatively, the underlying implementation of services can be via an executable (*loose coupling*). This executable is then called by the CREOLE wrapper code. In ei-

ther case the implementation of CREOLE services is completely transparent to GATE. Note that the loose coupling route means modules supplied either in source form or binary form can be integrated into GATE, the latter possibility reducing problems of redistributing LE software to third parties.

CREOLE wrappers encapsulate information about the preconditions for a module to run (data that must be present in the GDM database) and post-conditions (data that will result). This information is needed by GGI – see below. Aside from the information needed for GGI to provide access to a module, GATE compatibility equals TIPSTER compatibility – i.e. there will be very little overhead in making any TIPSTER module run in GATE.

In addition to the macro requirements on CREOLE integration described above, GDM imposes constraints on the I/O format of CREOLE objects, namely that all information must be associated with byte offsets and conform to the annotations model of the TIPSTER architecture. The principal overhead in this process is making the components being integrated use byte offsets, if they do not already do so.

As we noted above, CREOLE objects may be data-orientated. It is our intention to integrate as large a set of LE data resources as possible within GATE in order to reduce the overhead of installing and understanding the software interfaces of these resources. For example, the Wordnet thesaurus [6] will be given a CREOLE wrapper encapsulating the C API as a GATE service. Grammars, lexica, gazetteers – all are candidates for CREOLE integration, and as the set expands GATE can become a standard resource repository for LE data as well as LE processing modules.

3.3. GGI

The GGI is a graphical tool that encapsulates the GDM and CREOLE resources in a fashion suitable for interactive building and testing of LE components and systems. The philosophy is to provide a rich set of tools including, but not limited to, the CREOLE modules. So, for example, access to a KWIC (keyword in context) tool and an interface to WordNet is included, as well as taggers, parsers, etc. from CREOLE. The GGI also has functions for creating, viewing and editing the collections of documents which are managed by the GDM and that form the corpora which LE modules and systems in GATE use as input data. Finally, the GGI has facilities to display the results of module or system execution – new annotations associated with the document. These annotations can be viewed either in raw form using a generic annotation viewer or in an annotation-specific way, if special annotation viewers are available. For example, named entity annotations which identify and classify proper names (e.g. organization names, person names, location names) can be shown by colour-coded highlighting of relevant words; phrase structure annotations could be shown by graphical presentation of parse trees.

The main function of the GGI is to provide a graphical launchpad for the various LE subsystems available in GATE. To that end, the main panel of the GGI top-level display shows the particular tasks which may be performed by modules or systems within the GATE system (e.g. parsing). Having chosen a task, an intermediate level display appears, presenting the user with a selection of icons, one for each of the one or more specific modules or systems capable of performing the selected task (e.g. a specific chart parser, LR parser, etc.). Once a particular module or system is selected, a final window appears displaying a connected graph of the one or more modules that need to be run for the selected module or system to achieve the task. In this graph, the boxes denoting modules are actually active buttons: clicking on them will, if conditions are right, cause the module to be executed.

Figure 3 is an example of such a display for the VIE information extraction system. The paths through the graph indicate the dependencies amongst the various modules making up this application. Execution takes place from left to right and dependencies may be read from right to left. Each nodule at the left of a module box indicates a dependency on results of previous processing, results which may be generated by running any one of the modules connected to the nodule. Thus, in the example, the final `buChart` module may only be run if the results of the `List Lookup` module and the `Brill Tagger` module and either the `tagged Morph` module or the `Morph` module are available. They in turn have earlier dependencies.

At any point in time, the state of execution of the system, or, more accurately, the availability of data from various modules, is depicted through colour-coding of the module boxes. Figure 3 shows the module window with the buChart task half complete. Light grey modules (green, in the real display) can be executed. Modules that require input from others not yet executed, and so cannot be executed yet, are shown with a white background (amber, in reality). The modules that have already been executed (or those whose output data is available – possibly through earlier execution, or even through direct import into the GDM) are shown in dark grey (red). After execution, the results of completed modules are available for viewing, via mouse button operations over the module box area, and are displayed using either the default raw annotation viewer, or an annotation-specific viewer if available, as described above. In addition, modules can be 'reset', i.e. their results removed from the GDM, to allow the user to pick another path through the graph, or re-execute having altered some tailorable data-resource (such as a grammar or lexicon) interpreted by the module at run-time. (Modules running as external executables might also be recompiled between runs.)

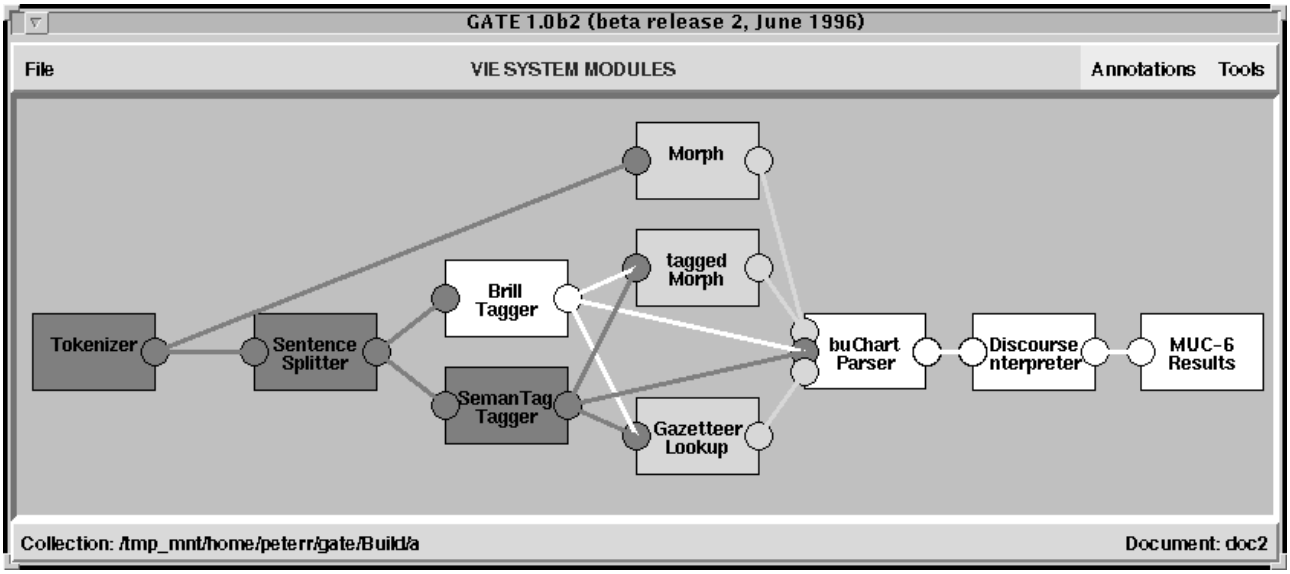

Figure 3. The GATE VIE System

Of course, before any processing can be done a document collection and, optionally, a document have to be selected (it is not mandatory to select a document, as tasks can be performed on entire collections). This is done through the drop-down `File` menu on the menu-bar. The collection and document selected are displayed on a bar at the bottom of the module graph window.

The GGI is implemented in Tcl/Tk, with access to CREOLE objects and to collections and documents via C++-implemented Tcl commands (using the GDM API where appropriate).

4. VIE: An Application In GATE

To illustrate the process of converting pre-existing LE systems into GATE compatible systems we use as an example the creation of the VIE (Vanilla Information Extraction) system from the LaSIE (Large-Scale Information Extraction) system [8], Sheffield's entry in the ARPA-sponsored Message Understanding Conference 6 (MUC-6) system evaluations. LaSIE module interfaces were not standardised when originally produced and its CREOLEization gives a good indication of the ease of integrating other LE tools into GATE. The resulting system, VIE, is distributed with the GATE system.

4.1. LaSIE

LaSIE was designed as a research system for investigating approaches to information extraction[1] and to be entered into the MUC-6 conference [1]. As such it is a standalone system that is aimed at specific tasks and while based on a modular design, none of its modules were specifically designed with reuse in mind, nor was there any attempt to standardise data formats passed between modules. Modules were written in a variety of programming languages, including C, C++, Flex, Perl and Prolog. In this regard LaSIE is probably typical of existing LE systems and modules.

The high-level tasks which LaSIE performs include the four MUC-6 tasks (carried out on *Wall Street Journal* articles):

- named entity (NE) recognition, the recognition and classification of definite entities such as names, dates, places;
- coreference (CO) resolution, the identification of identity relations between entities (including anaphoric references to them);
- template element (TE) construction, a fixed-format, database-like enumeration of organisations and persons;
- scenario template (ST) construction, the detection of specific relations holding between template elements relevant to a particular information need (in this case personnel joining and leaving companies) and con-

[1] *Information extraction (IE)* is a term which has come to be applied to the activity of automatically extracting pre-specified sorts of information from short, natural language texts – typically newswire articles (see, e.g., [12]). For instance, one might scan business newswire texts for announcements of joint ventures and extract the names and nationalities of the participating companies, the activity of the venture, the start date of the venture, its capitalisation, and so on. Put another way, IE may be seen as the activity of populating a structured information source (or database) from an unstructured, or free text, information source.

struction of a fixed-format structure recording the entities and details of the relation.

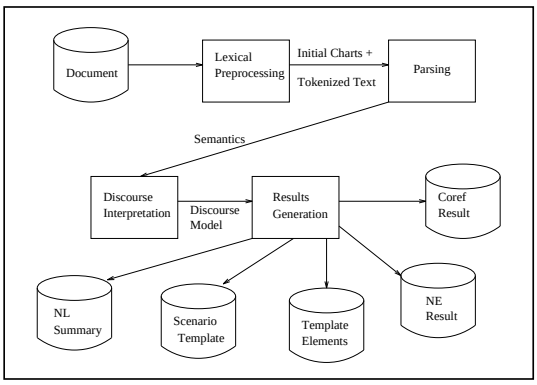

Figure 4. LaSIE architecture

The high level structure of LaSIE is illustrated in Figure 4. The system is a pipelined architecture which processes a text sentence-at-a-time and consists of three principal processing stages: lexical preprocessing, parsing plus semantic interpretation, and discourse interpretation. For further details of the system see [8, 7].

4.2. The CREOLEisation of LaSIE

Each of the stages depicted in the high-level LaSIE architecture diagram is implemented through a collection of modules, each of which must be CREOLEised in order to be integrated into the GATE system. As described in section 3.2, CREOLEisation of existing LE modules involves providing them with a wrapper so that the modules communicate via the GDM, by accessing TIPSTER-compliant document annotations and updating them with new information.

The major work in converting LaSIE to VIE involved defining useful module boundaries, unpicking the connections between them, and then writing functions to convert module output into annotations relating to text spans and to convert GDM input from annotations relating to text spans back into the module's native input format.

The complete VIE system comprises ten modules, each of which is a CREOLE object integrated into GATE. The CREOLEisation took approximately two person months, though an accurate estimate is difficult, because GATE functionality was itself changing during this time. The resulting system has all the functionality of the original LaSIE system and comparable performance. However, the interface makes it much easier to use. And, of course, it is now possible to swap in modules, such as a different parser, with significantly less effort than would have been the case before.

5. Concluding Remarks

5.1. Current State

In addition to the CREOLEisation of LaSIE into GATE, other modules have recently been integrated, including the LDOCE lexical database, the Plink shift-reduce parser [11], and the SemanTag part-of-speech tagger [4]. The effort required to integrate an existing module varies depending on the modifications needed to communicate in terms of byte offset annotations. In most cases this simply involves the addition of byte offset positions to the module's input and their preservation during the module's operation. The current CREOLE wrappers share significant functionality, allowing the rapid production of new wrappers, and a higher level of reusable (Tcl) wrapper code is planned. Modules' input/output requirements are specified in a formal `module registration' language, and inclusion in the interface's module dependency graphs is achieved via declarations in the GGI.

Version 1.0 of GATE is now available as a beta release. We are carrying out further testing of the system ourselves and are expecting feedback from those sites which have agreed to experiment with the beta release. The result will be a general release of version 1 by autumn 1996. We are already planning version 2 which will incorporate some of the ideas mentioned in the next paragraph. At this time GATE will only run on UNIX platforms. GATE itself requires only public domain software, but VIE currently requires Quintus or Sicstus Prolog.

5.2. Future Work

There are a vast number of extensions and refinements that could be undertaken. Amongst the higher priorities are:

- enhancing the underlying GDM database technology to use an SQL-accessible relational database package;
- porting the system to PC platforms;
- extending the range and functionality of annotation viewers;
- extending the range of auxiliary tools and data resources available;
- automatically generating the module dependency graphs from configuration information supplied with each CREOLE wrapper – i.e. there should be no information hard-coded into GATE regarding different modules. This can be achieved, for example, by each object registering its name, version, input requirements, result type, results viewer, and so on.

But of course the highest priority will be extending the CREOLE set, by CREOLEising modules which are available and for which there is a high demand. At first we will

carry out this work ourselves, as resources permit. Eventually the system must become simple enough for users to configure themselves.

5.3. Final Remarks

The recent completion of this work means a full assessment of the strengths and weaknesses of GATE is not yet possible. The implementation of VIE in GATE, however, provides an existence proof that the original conception is workable. We believe that the environment provided by GATE now will allow us to make significant strides in assessing alternative LE technologies and in rapidly assembling LE prototype systems. Thus, to return to the themes of our introduction, GATE will not commit us to a particular linguistic theory or formalism, but it will enable us, and anyone who wishes to make use of it, to build, in a pragmatic way, on the diverse efforts of others.

Of course, GATE does not solve all the problems involved in plugging diverse LE modules together. There are two barriers to such integration: incompatibility of *representation* of information about text; and incompatibility of *type* of information used and produced by different modules.

GATE enforces a separation between these two and provides a solution to the former based on the work of the TIPSTER architecture group. Because GATE places no constraints on the linguistic formalisms or information content used by CREOLE modules, the latter problem must be solved by dedicated translation functions – e.g. tagset-to-tagset mapping – and, in some cases, by extra processing – e.g. adding a semantic processor to complement a bracketing parser. As more of this work is done we can expect the overhead involved to fall, as all results will be available as CREOLE modules. We are confident that integration *is* possible (partly because we believe that differences between representation formalisms tend to be exaggerated) – and others share this view, e.g. the MICROKOSMOS project [3], which seeks to integrate many types of knowledge source in a usable whole, as well as the LexiCadCam experience at New Mexico [18] which sought to provide core lexical information as needed in a range of user-specified formats.

Acknowledgements

The research reported here has been supported by grants from the U.K. Department of Trade and Industry (Grant Ref. YAE/8/5/1002) and the Engineering and Physical Science Research Council (Grant # GR/K25267). The authors would like to thank: Jim Cowie, Remi Zajac, and Ted Dunning of the Computer Research Laboratory (CRL), New Mexico State University for making the CRL Tipster Document Manager available to us and for discussions and advice; Afzal Ballim of the Polytechnic of Lausanne for help with MULTEXT and SGML.

References

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

[2] A. Ballim. Abstract Data Types for MULTEXT Tool I/O. LRE 62-050 Deliverable 1.2.1, 1995.

[3] S. Beale, S. Nirenburg, and K. Mahesh. Semantic Analysis in the Mikrokosmos Machine Translation Project. In *Proceedings of the Second Symposium on Natural Language Processing (SNLP-95)*, 1995.

[4] G. Cooke. The SemanTag project. Further details at `http://www.rt66.com/gcooke/SemanTag`, 1996.

[5] H. Cunningham, R. Gaizauskas, and Y. Wilks. A General Architecture for Text Engineering (GATE) – a new approach to Language Engineering R&D. Technical Report CS – 95 – 21, Department of Computer Science, University of Sheffield, 1995. Also available as `http://xxx.lanl.gov/cmp-lg/9601009`.

[6] M. G.A., R. Beckwith, C. Fellbaum, D. Gross, and K. Miller. Introduction to WordNet: On-line. Distributed with the WordNet Software., 1993.

[7] R. Gaizauskas and K. Humphreys. Quantative Evaluation of Coreference Algorithms in an Information Extraction System. In *DAARC96 - Discourse Anaphora and Anaphor Resolution Colloquium*. Lancaster University, 1996.

[8] R. Gaizauskas, T. Wakao, K. Humphreys, H. Cunningham, and Y. Wilks. Description of the LaSIE system as used for MUC-6. In *Proceedings of the Sixth Message Understanding Conference (MUC-6)*. Morgan Kaufmann, 1995.

[9] C. F. Goldfarb. *The SGML Handbook*. Oxford University Press, 1990.

[10] R. Grishman. TIPSTER Architecture Design Document Version 1.52 (Tinman Architecture). Technical report, Department of Computer Science, New York University, 1995. Available at `http://www.cs.nyu.edu/tipster`.

[11] C. Huyck. Plink: An intelligent natural language parser (cse-tr-218-94). Technical report, Computer Science and Engineering Division, The University of Michigan, 1994.

[12] P. Jacobs, editor. *Text-Based Intelligent Systems: Current Research and Practice in Information Extraction and Retrieval*. Lawrence Erlbaum, Hillsdale, NJ, 1992.

[13] J. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[14] J. Schütz. Developing Lingware in ALEP. *ALEP User Group News, CEC Luxemburg*, 1(1), Oct. 1994.

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

[16] C. Sperberg-McQueen and L. Burnard. *Guidelines for Electronic Text Encoding and Interchange (TEI P3)*. ACH, ACL, ALLC, 1994.

[17] H. Thompson. MULTEXT Workpackage 2 Milestone B Deliverable Overview. LRE 62-050 Develiverable 2, 1995.

[18] Y. Wilks, L. Guthrie, and B. Slator. *Electric Words*. MIT Press, Cambridge, MA, 1996.