A Unicode-based Environment for Creation and Use of Language Resources

Valentin Tablan*, **Cristian Ursu***, **Kalina Bontcheva***, **Hamish Cunningham***,
Diana Maynard*, **Oana Hamza***, **Tony McEnery**†, **Paul Baker**†, **Mark Leisher**‡

*Dept. of Computer Science
University of Sheffield
Regent Court, 211 Portobello St
Sheffield, S1 4DP, UK
[V.Tablan, C.Ursu, K.Bontcheva, H.Cunningham, D.Maynard, O.Hamza]@dcs.shef.ac.uk

†Department of Linguistics and Modern English Language
Bowland College, Lancaster University
Lancaster, LA1 4YT, UK
A.McEnery@lancs.ac.uk, bakerjp@exchange.lancs.ac.uk

‡Computing Research Laboratory
New Mexico State University
Box 30001/MSC 3CRL, Las Cruces, NM 88003-8001, USA
mleisher@crl.nmsu.edu

Abstract

GATE is a Unicode-aware architecture, development environment and framework for building systems that process human language. It is often thought that the character sets problem has been solved by the arrival of the Unicode standard. This standard is an important advance, but in practice the ability to process text in a large number of the World's languages is still limited. This paper describes work done in the context of the GATE project that makes use of Unicode and plugs some of the gaps for language processing R&D.
First we look at storing and decoding of Unicode compliant linguistic resources. The new capabilities for processing textual data and taking advantage of the Unicode standard are detailed next. Finally, the solutions used to add Unicode displaying and editing capabilities for the graphical interface are described.

1. Introduction

GATE(**?**)[1] is an architecture, development environment and framework for building systems that process human language. It has been in development at the University of Sheffield since 1995, and has been used for many R&D projects, including Information Extraction in multiple languages and for multiple tasks and clients.

It is often thought that the character sets problem has been solved by the arrival of the Unicode standard. This standard is an important advance, but in practice the ability to process text in a large number of the World's languages is still limited by

- incomplete support for Unicode in operating systems and applications software
- languages missing from the standard
- difficulties in converting non-Unicode character encodings to Unicode

This paper describes work done in the context of the GATE project[2] that makes use of Unicode and plugs some of the gaps for language processing R&D.

[1]GATE is implemented in Java and is freely available from http://gate.ac.uk as open-source free software under the GNU library licence.

[2]Partly funded in this case by the EMILLE project (**?**).

The GATE architecture defines almost everything in terms of components - reusable units of code that are specialised for a specific task. There are three main types of components:

- *Language Resources (LRs)* store some kind of linguistic data such as documents, corpora, ontologies and provide services for accessing it. At the moment all the predefined LRs are text based but the model doesn't constrict the data format so the framework could be extended to handle multimedia documents as well.
- *Processing Resources (PRs)* are resources whose character is principally programatic or algorithmic such as a POS tagger or a parser. In most cases PRs are used to process the data provided by one or more LRs but that is not a requirement.
- *Visual Resources (VRs)* are graphical components that are displayed by the user interface and allow the visualisation and editing of other types of resources or the control of the execution flow.

The GATE framework defines some basic language resources such as documents and corpora, provides resource discovery and loading facilities and supports various kinds of input output operations such as format decoding, file or database persistence.

GATE uses a single unified model of *annotation* - a modified form of the TIPSTER format (**?**) which has been made largely compatible with the Atlas format (**?**), and uses the now standard mechanism of 'stand-off markup' (**?**). The TIPSTER format was always based on 'stand-off', i.e. using pointers into texts instead of adding markup to them; the SGML/XML community later adopted the same style, perhaps in response to (**?**), or because no other option is possible in an environment where documents come in many different formats. Annotations are characterised by a *type* and a set of *features* represented as attribute-value pairs. The annotations are stored in structures called *annotation sets* which constitute independent layers of annotation over the text content.

The advantage of converting all formatting information and corpus markup into a unified representation, i.e. the annotations, is that NLP applications do not need to be adapted for the different formats of each of the documents, which are catered for by the GATE format filters (e.g. some corpora such as BNC come as SGML/XML files, while others come as email folders, HTML pages, news wires, or Word documents).

The work for the second version of GATE started in 1999 and led to a complete redesign of the system and a 100% Java implementation. One of the additions brought by version 2 is full support for Unicode data allowing the users to open, visualise and process documents in languages different from the default one for the underlying platform.

2. Unicode Compliant Language Resources

The most important types of Language Resources that are predefined in GATE are the *documents* and the *corpora*. A corpus is defined in GATE as a list of documents which makes the the representation problem only applicable with regard to documents.

Documents in GATE are typically created starting from an external resource such as a file situated either on a local disk or at an arbitrary location on the Internet. Text needs to be converted to and from binary data, using an *encoding* (or *charset*), in order to be saved into or read from a file. There are many different encodings used worldwide, some of them designed for a particular language, others covering the entire range of characters defined by Unicode. GATE uses the facilities provided by Java and so it has access to over 100 different encodings including the most popular local ones, such as ISO 8859-1 in Western countries or ISO-8859-9 in Eastern Europe, and some Unicode ones e.g. UTF-8 or UTF-16.

Apart from being able to read several character encodings, GATE supports a range of popular file formats such as HTML, XML, email, some types of SGML and RTF.

After being processed in GATE, the documents can be stored for later use (or made *persistent*) by using one of three options: as a text file, a binary file or as entries in a database. A document saved as text becomes an XML file using the UTF-8 character encoding which is an 8-bit Unicode Transformation Format. Another option is to save the documents as the binary serialisation of the memory image of the document which also preserves the internal Unicode representation. If database storage is preferred, GATE makes use of the support for Unicode documents provided by the database engine which in the case of most modern databases is available (ORACLE for instance provides full support for Unicode data).

3. Processing Resources and Unicode

The use of the Java platform implies that all processing resources that access textual data will internally use Unicode to represent data, which means that all PRs can virtually be used for text in any Unicode supported language. Most PRs, however, need some kind of linguistic data in order to perform their tasks (e.g. a parser will need a grammar) which in most cases is language specific. In order to make the algorithms provided with GATE (in the form of PRs) as language-independent as possible, and as a good design principle, there is always a clear distinction between the algorithms - presented in the form of machine executable code - and their linguistic resources which are typically external files. All PRs use the textual data decoding mechanisms when loading the external resources so these resources can be represented in any supported encoding which allows for instance a gazetteer list to contain localised names. This design made it possible to port our information extraction system ANNIE from English to other languages by simply creating the required linguistic resources: GATE has been used on a variety of Slavic, Germanic, Romance, and Indic languages (**?**; **?**; **?**; **?**).

One of the PRs provided with GATE is the tokeniser which not only handles Unicode data but is actually built around the Unicode standard, hence its name of "*GATE Unicode Tokeniser*". The role of the tokeniser is to split the text into simple tokens and to provide some basic information about the type of the generated tokens. It classifies tokens into numbers, punctuation, symbols or words and, in the case of words, provides some information about their orthography (e.g. with an initial capital, all upper case, all lower case, etc.).

Running the tokeniser over a document will generate a group of annotations of type *Token* or *SpaceToken* which will never overlap and will cover the entire content of the document (see table **??**).

Like many other GATE PRs, the tokeniser is based on a finite state machine (*FSM*) which is an efficient way of processing text. In order to provide a language independent solution, the tokeniser doesn't use the actual text characters as input symbols, but rather their categories as defined by the Unicode standard.

The underlying FSM of a tokeniser is defined starting from a set of rules, each of them being composed from a regular expression on the left hand side and an annotation template on the right hand side. The regular expression defines a pattern of characters, named by their respective Unicode categories, while the annotation template is used to generate new annotations when the pattern described on the left hand side matches the input. See figure **??** for an example of a tokeniser rule which detects words containing only letters and dashes and that start with a capital letter. Each such rule is converted into a FSM designed to recognise the regular expression pattern, and then the full FSM for the tokeniser is constructed as a disjunction of all the FSMs defined by the rules. We are currently using a tokeniser of 23

Text			
`On March the 3rd 2002...`			
`|0.|3.....|9...|13.|17...`			
Annotations			
Type	SpanStart	Span End	Features
Token	0	2	`kind=word, orth=upperInitial, string="On"`
SpaceToken	2	3	`kind=space, length=1 string=" "`
Token	3	8	`kind=word, orth=upperInitial, string="March"`
SpaceToken	8	9	`kind=space, length=1 string=" "`
Token	9	12	`kind=word, orth=lowercase, string="the"`
SpaceToken	12	13	`kind=space, length=1 string=" "`
Token	13	14	`kind=number, string="3"`
Token	14	16	`kind=word, orth=lowercase, string="rd"`
SpaceToken	16	17	`kind=space, length=1 string=" "`
Token	17	21	`kind=number, string="2002"`
Token	21	22	`kind=puctuation, string="."`
Token	22	23	`kind=puctuation, string="."`
Token	23	24	`kind=puctuation, string="."`

Table 1: A tokenisation example

rules that recognises many types of words, whitespace patterns, numbers, symbols and punctuation and which should handle any language from the Indo-European group without any modifications.

```
UPPERCASE_LETTER
(LOWERCASE_LETTER|DASH_PUNCTUATION)*
>
Token;orth=upperInitial;kind=word;
```

Figure 1: A tokeniser rule.

For situations not handled by the current tokeniser new rules can easily be added extending it for new languages.

The modular architecture of GATE allows for the localisation of the tokeniser when a particular language requires additional processing, by simply adding another PR after the tokeniser in the execution chain. For instance, in the case of English text, we are using a JAPE transducer (**?**) which adapts the output of the tokeniser for the needs of the part-of-speech tagger provided with GATE.

4. The GATE Unicode Kit

The third type of resource provided with GATE are the Visual resources which are used in constructing user interfaces for the visualisation and editing of data as well as for the control of the execution flow. A Unicode enabled graphical user interface (*GUI*) needs to address two main issues: the capability to display text and the ability to enter text in other languages than the default one.

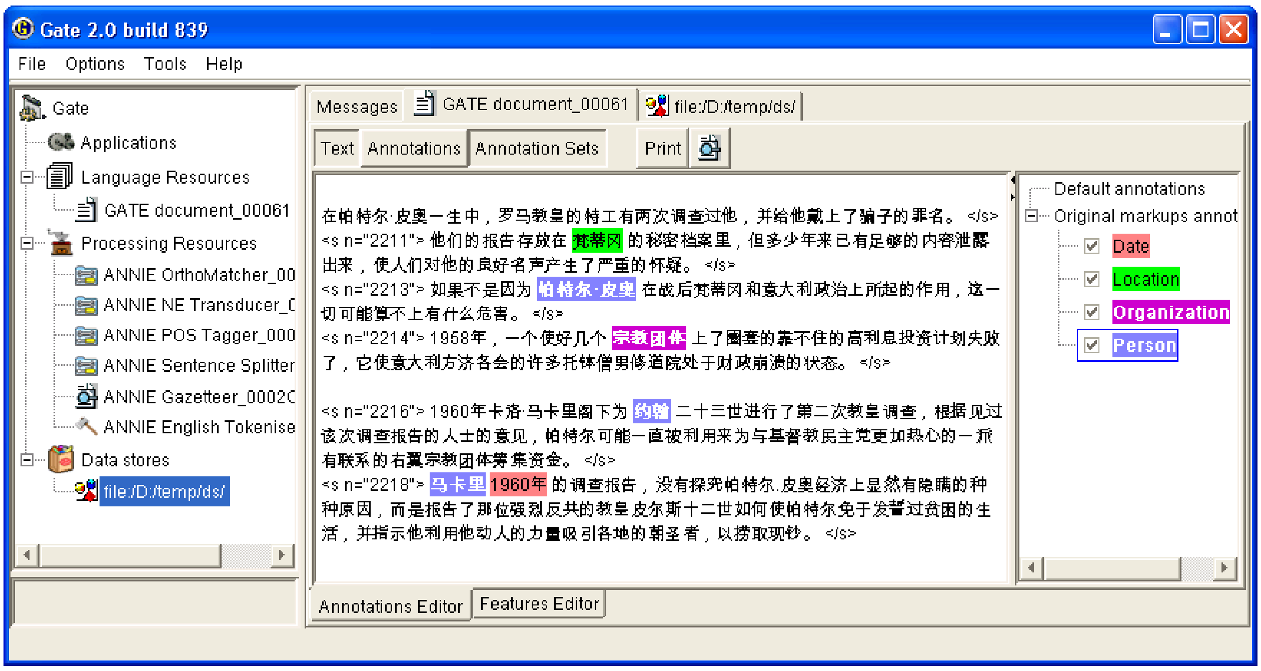
4.1. Displaying Unicode Data

The support for displaying text using different languages and scripts in Java is quite comprehensive and covers many languages supported by the Unicode standard. Sun Microsystems, the authors of the Java programming language, are actively working on extending the support to new languages and we can expect the coverage to get better with every new released version. One problem is the provision of fonts that cover character blocks that are less mainstream, which are hard to find and expensive to create. However the situation is improving and for example Microsoft provides a comprehensive font that covers all the character blocks defined by Unicode for the Windows platform with new versions of their *Microsoft Office* product. The support for localisation is improving for other operating systems as well, and the new versions of the Java virtual machine are able to make use of the fonts provided by the underlying platform which means that if a particular language can be used on a given platform it is more than probable that it will be available in GATE as well. Figure **??** shows a Chinese document loaded and annotated in GATE on an operating system localised for United Kingdom using the Arial Microsoft Unicode font without having installed any support for Asian languages.

4.2. Unicode Input Methods

While the support for displaying Unicode text is provided to a large extent by the underlying platform, the same is not true with respect to editing Unicode text. Many platforms provide some support for localisation but that is not always very comprehensive and is often not Unicode compliant, which makes it difficult for Java to make use of it. Also the level of support provided varies largely between platforms and in order to address this problem we have provided our own dedicated solution in the form of the *GATE Unicode Kit (*GUK*).*

The recommended way of adding new text input facilities to a Java application is to define so called *input methods (*IM*).* An IM allows users to enter text in other languages than the default one by intercepting the events generated by the input hardware such as the system keyboard or mouse and mapping them to a different output from the one normally obtained.



在帕特尔·皮奥一生中，罗马教皇的特工有两次调查过他，并给他戴上了骗子的罪名。 </s>
<s n="2211"> 他们的报告存放在 梵蒂冈 的秘密档案里，但多少年来已有足够的内容泄露出来，使人们对他的良好名声产生了严重的怀疑。 </s>
<s n="2213"> 如果不是因为 帕特尔·皮奥 在战后梵蒂冈和意大利政治上所起的作用，这一切可能算不上有什么危害。 </s>
<s n="2214"> 1958年，一个使好几个 宗教团体 上了圈套的靠不住的高利息投资计划失败了，它使意大利方济各会的许多托钵僧男修道院处于财政崩溃的状态。 </s>

<s n="2216"> 1960年卡洛·马卡里阁下为 约翰 二十三世进行了第二次教皇调查，根据见过该次调查报告的人士的意见，帕特尔可能一直被利用来为与基督教民主党更加热心的一派有联系的右翼宗教团体筹集资金。 </s>
<s n="2218"> 马卡里 1960年 的调查报告，没有探究帕特尔.皮奥经济上显然有隐瞒的种种原因，而是报告了那位强烈反共的教皇皮尔斯十二世如何使帕特尔免于发誓过贫困的生活，并指示他利用他动人的力量吸引各地的朝圣者，以捞取现钞。 </s>

Figure 2: A Chinese document annotated in GATE.

GUK consists of two main components: a set of IM definitions and the Java code that handles the communication with the system, the decoding of the IM definitions and that does the actual input mapping. At present GUK provides input methods for 17 different languages some of them with more than one version, amounting to a total of 30 different input methods (see appendix **??** for a full list).

Although distributed with GATE, GUK is effectively a separate entity that can be used outside of GATE as well. To this end, GUK is packaged as a separate library that can either be registered with the Java virtual machine at startup time or can be added to a Java installation as an *installed extension* thus making it available to all applications that run within that particular virtual machine. GUK integrates seamlessly with the platform and there are no special requirements an application needs to fulfill to be able to use its facilities. The GATE distribution contains GUK and makes use of it as a runtime library.

The IMs defined by GUK provide support for text input by means of virtual keyboards. Because of restrictions imposed by the platform independent manner in which Java treats the input hardware, there is no reliable way to determine the actual layout of the physical system keyboard: only the characters generated by a key stroke (e.g "E") can be obtained but not the actual position of the pressed key (e.g. third from the left in the top row of keys). The virtual keyboards defined in GUK are based on the assumption that the system keyboard uses a British layout which could cause some keys to be misplaced when the system keyboard has a different layout. However, the layout of the currently active virtual keyboard can be displayed on the screen in order to assist the user in finding the right key (see figure **??**). As long as the virtual keyboard map is visible on screen it can also be used for entering text directly by clicking with the mouse on the virtual keys, which might be more efficient, particularly when using a keyboard type the user is not accustomed to.

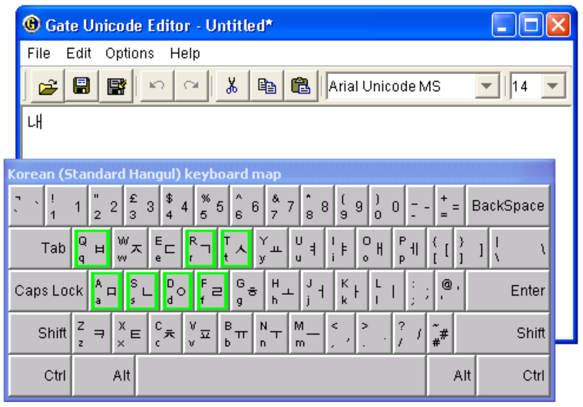


Figure 3: The GUK Unicode editor using a Korean virtual keyboard.

Each input method maps an input consisting of keystrokes onto an output consisting of characters in the target language according to the IM definition file. When a GUK input method is activated, its definition file is read and used to construct a finite state transducer that starts to "listen" for events from the keyboard. When the user presses a key the character normally generated by it will be passed on to the transducer and as soon as the transducer starts generating output it will be sent to the Java virtual machine as if it came directly from the keyboard. Because GUK intercepts the keyboard events at a very low level in the input hierarchy, it is unlikely that its actions will cause any conflicts with the actual application that receives the translated input. The client application will probably not even be aware of the presence of an intermediate step in the handling of the user input.

A transducer was required rather than a simple mapping table because there is no direct correspondence between the number of keystrokes read and the amount of generated text, in some cases (e.g. for the TCode Japanese keyboard) more keystrokes are required in order to generate one output character while in other cases (e.g. the Bengali keyboard) a single keystroke can generate up to three different characters.

There are also situations, particularly for some modifier characters, when the output character (or group of characters) is different from the symbol that needs to be displayed on the key of the virtual keyboard.

The IM definition file is a plain ASCII text file which lists all the mappings, one for each line (see figure **??** for an example).

A line from an IM definition file contains up to three values labelled *bind*, *send* and *keycap*:

- the *bind* value is a list of one or more ASCII characters which represent the input coming from the system keyboard.
- the *send* value is a list of Unicode characters written as 16 bit hexadecimal values which will form the Unicode output of the input method when the *bind* value has been keyed.
- the *keycap* value is also a list of Unicode characters in hexadecimal and represents the text to be displayed on the virtual key. The *keycap* is only required when the *bind* value contains a single character as it wouldn't make sense otherwise.

From the "Standard Hangul Korean" IM:
```
bind "r" send 0x3131 keycap 0x3131
bind "rk" send 0xAC00
bind "rkr" send 0xAC01
```

From the "Inscript Bengali" IM:
```
bind "=" send 0x09CD0x098B
keycap 0x25CC0x200D0x09CD0x098B
```

Figure 4: Input Method Mappings.

For input methods that require for instance double keystrokes to generate characters, not all pairs of keystrokes are valid combinations. In such cases the keyboard map (if displayed) will highlight the keys that will lead to a valid input (see keys "Q", "R", "T", "A", "S", "D" and "F" in figure **??**).

Using plain text definition files for input methods allows users without prior knowledge of Java or the inner workings of GUK to extend it by adding new input methods simply by creating the appropriate files and placing them where GUK can find them.

Apart from the input methods GUK also provides a simple Unicode-aware text editor which is important because not all platforms provide one by default or the users may not know which one of the already installed editors is Unicode-aware. Besides providing text visualisation and editing facilities, the GUK editor also performs encoding conversion operations. The editor has proved a useful tool during the development and testing of GATE in a crossplatform environment.

5. Conclusion and Future Work

GATE has proven its worth as a development tool for language processing R&D in a number of languages. The new facilities for Unicode support have extended the range of the system to many more languages, and this support will improve as the Java environment continues to develop, and as GATE users contribute new input methods and character encoding conversions. Work in progress includes the provision of a number of converters for Indic languages.

Other planned developments include the localisation of the GATE graphical interface and the ability to adapt the GUK virtual keyboards according to the actual layout of the system keyboard.

We also plan to extend GATE to support multimedia language resources (e.g. audio/video corpora). Experience from an early prototype has shown that this is a feasible step (**?**).

6. Acknowledgements

Work on GATE has been funded by EPSRC grants GR/K25267 and GR/M31699 (GATE, GATE 2). Support for non-indigenous minority language writing systems within GATE is also funded by the EPSRC on grant GR/N19106 (EMILLE). Several smaller grants supported prototyping work in 1998 and 1999, including two from the US TIPSTER programme and one from the Max Planck Institute in Nijmegen, Holland.

We would like to thank Markus Kramer from Max Planck Institute, Nijmegen, for providing us with the IPA and Chinese input methods.

A Appendix: GUK Input Methods

Target Language	Keyboard Layout
Arabic	ATeX MLT Arabic Windows
Armenian	Standard
Bengali	Inscript
Chinese	China* Taiwan*
English	ASCII
Georgian	Heinecke Imnaishvili Arrangement MLT
German	Windows
Greek	XTerm Windows
Hebrew	Standard
Hindi	Inscript VOA
Japanese	TCode
Korean	Standard Hangul
Persian	ISIRI 2901-94 CRL Phonetic
Russian	YAWERTY (Phonetic) CRL YAWERTY (Phonetic)
Serbo-Croatian	Cyrillic MLT Latin MLT
Urdu	CRL Urdu
Vietnamese	VIQR Implicit VIQR Implicit (Decomposed) Telex
	IPA-96*

* Provided by Markus Kramer, Max Planck Institute, Nijmegen, Holland.

Table 2: Languages currently supported by GUK