A platform-independent Braille translator

Alasdair King, Gareth Evans, Paul Blenkhorn.

Department of Computation, UMIST, PO Box 88, Manchester, M60 1QD, +44 161 200 5832.
{a.king, g.evans, p.blenkhorn}@co.umist.ac.uk

Abstract

This paper describes the production of a computer program in Java that translates text to and from Braille. It builds on previous work on an existing translation system written in C, BrailleTrans, which has been used to provide translation functionality to the Microsoft Word word processor. The Java programs extend the function of BrailleTrans to support a universal character set, Unicode, and are portable and platform-independent. Their performance supports their potential for adding Braille translation functionality to applications.

Keywords

Braille, translation.

1 Introduction

Computer translation of text into Braille and Braille into text has permitted Braille users to access much information previously denied to them because of the cost and difficulty of translation, and to apply their Braille skills to computer use. Although the number of Braille users is small as a proportion of blind and visually-impaired people (Bruce *et al*, 1991), Braille still performs an important role in allowing blind people to achieve literacy, especially where the most advanced technological solutions are too expensive. This is especially true in non-Western countries and in local and community applications.

Commercial computer translation programs have been successful. They are used both for large-scale commercial translation to produce Braille versions of textbooks and other works for sale and also for teachers and individuals for pedagogical and personal use. The market leader is the Duxbury company which provides the "MegaDots" program for volume production and the "Duxbury Braille Translator" program for personal use (Duxbury Systems, 2001). These support eighteen languages, bi-directional translation, six-key Braille input, contractions, and a range of input formats including Microsoft Word documents. The Index Braille company supplies an MS Windows text-to-Braille translator called "WinBraille" with its Braille embossers (Blomvquist *et al*, 2002). This supports eighteen languages, Braille contraction, and uses a rule-based translation system that can be amended by the user for local usage. A number of other programs are available, but as they, like the systems mentioned above, are commercial ventures the method of operation of these programs is not available for study or discussion.

Despite their commercial success, the available translators share similar limitations. Some arise from the difficulties of performing computer translation of human languages *per se* (Blenkhorn, 1995; Blenkhorn, 1997). However, other limitations reflect their design philosophy. They have been written for specific platforms – typically the prevalent Microsoft consumer operating systems – which prevents their use in different environments such as different operating systems, hand-held devices or network servers. Portability is also compromised by the different encoding strategies employed for the text to be translated on different platforms and operating systems. The most common are based on an 8-bit 256-character extended-ASCII set, but these are language or even country specific (Gschwind, 1994; Fowles, 1997).

The BrailleTrans translation program, developed at UMIST (Blenkhorn, 1995), was devised to provide a cost-effective translation system available for public study and deployment in areas not addressed by the commercial solutions. In a market dominated by expensive translation programs, it was intended to be a useful resource for users of low-specification machines. It uses a set of translation rules and a simple state table to match the left and right context of segments of text to be translated. This allows for flexibility in translation of idiosyncratic

language – for example, postcodes, or abbreviations – but is still simple enough to allow a set of translation rules to be developed for a language by a non-technical Braille user. This is designed to support the use of the system for a language not supported by the commercial translators. The single BrailleTrans program can translate any language which is regular enough in Braille for a rules table to be produced.

Further work used BrailleTrans to provide translation functionality to Microsoft Word (Blenkhorn & Evans, 2001). This provided the ability to translate documents to Braille from within a familiar user interface.

BrailleTrans is written in the C programming language and runs as native code, and works with text encoded in an 8-bit 256-character extended-ASCII format. BrailleTrans therefore shares the platform-specific limitations of the commercial translators. This paper describes a new implementation of the BrailleTrans system in Sun Microsystem's Java programming environment (Gosling *et al*, 2000) which aims to address these limitations (King, 2001).

Programs written in Java, called "classes", can be run on any platform for which a Java Virtual Machine (JVM) has been developed. The JVM is written in native code and acts as an interpreter of the compiled Java classes. It handles memory management, errors, and other platform-specific features. The classes can, therefore, be independent of the local operating system, and should operate identically on every platform that is equipped with a JVM. Since every common operating system has a JVM, this makes them very portable and platform-independent. This allows developers to avoid having to recode software components for different operating systems.

Because Java classes are designed to run on any machine, they have also been adopted by the manufacturers of small consumer devices, such as personal digital assistants or mobile telephones. The differing hardware and operating systems of consumer devices have been a disincentive to application development. Java offers a partial solution. If the manufacturer can produce a JVM for the device, developers can leverage existing Java skills to deliver applications rapidly. A Java version of BrailleTrans might therefore be suitable for consumer devices, opening up many possibilities for assistive technology applications.

However, for these small consumer devices to be useful for translation, their JVMs must first be able to run the BrailleTrans class. Because of the wide variation in consumer device hardware (most importantly memory) some deviation from the language standard has been permitted in consumer device JVMs for different devices (Sun Microsystems, 2001) The consequence of this is that Java classes cannot be trusted to run without alteration on any consumer device as they should on any server or PC. Despite this limitation, the adoption of JVMs by consumer device manufacturers still supports the development of a Java BrailleTrans for consumer devices. The variation in JVMs is limited enough to suggest that only minimal amendments to a BrailleTrans class for a given consumer device would be required, and some device JVMs will require none at all if the classes restrict themselves to the more basic language features.

Another issue is performance. The classes must translate at a usable rate. As a part-interpreted, non-native language, Java has a reputation for poor speed, especially where system resources, and most especially memory, are scarce (Shirazi, 2000; Eckel, 2001). This is a significant factor on low-powered consumer devices. To some extent responsibility for performance in Java lies with the JVM, which is beyond the control of the developer. For example, many C techniques used by developers to optimise memory management are irrelevant in Java, where the JVM manages memory allocation. However, programming techniques are still very important in performance, from simple optimisation of control loops to more strategic choices of data types and object-oriented design. For the Java version of BrailleTrans, therefore, a number of different strategies were employed and tested to determine the most efficient approach.

The problem of non-portable text encoding was also addressed by the development of one Java class that uses the 16-bit Unicode character set for language rules tables and translation text. Unicode is used internally in Java and is designed to allow the representation of the great majority of human alphabets without duplication or conflicts (Unicode Incorporated, 2001). This allows the translation rules to be platform-independent, and removes the problem of variable non-standard character sets. It also supports languages where the commercial imperative to produce a translator is minimal. Unicode has been adopted as the basis for modern operating systems and is supported in a range of standards and text editors even on older platforms, so it is widely supported in applications outside of Java.

2 Development

The operation of the UMIST translation algorithm has already been detailed elsewhere (Blenkhorn, 1995). The Java classes developed followed this algorithm. A common translation engine utilises a set of rules to perform translation on the input text and deliver translated text back to the application using the class. A language translation rules file contains a set of translation rules and a decision table. It is specific for a particular language

and direction of translation, Braille-to-text or text-to-Braille. Translation rules files currently exist for English, Hungarian and Welsh (Blenkhorn, 1995 and 1997). The input text is worked through by the program, which tries to match the characters at the current position in the input text with a translation rule. The translation rule must match the characters at the current position in the input and the text to either side of this text. Wildcards are used to allow for matching situations as well as specific sequences, such as punctuation characters or whitespace. Finally, the translation engine is a finite state machine and uses the decision table and the current state to determine which rule can be applied. The rules allow the system to translate a wide range of context-specific syntactic situations, and the decision table allows different types of translation – typically different contraction grades – to be supported in the same language table.

Three different translation classes were developed, representing different strategies for performance and platform-independence. The first class, "C-style", was a porting of the existing highly-optimised BrailleTrans C code to a Java class. The similar syntax of C and Java required only limited changes to the C code. This served to some extent as an experimental control, being very similar to the existing, working, code. In addition, the overhead of managing an object-oriented system is one of the major factors in Java's poor performance when compared to functional languages like C, so using a non-object-oriented C design was intended to produce the least object-oriented and potentially fastest Java class possible. The second class, "Java-style", used Java objects internally for a more object-oriented approach, intending to capitalise on software engineering benefits such as easier maintenance and faster development that are supposed to derive from an object-oriented approach.

Both classes used the same 256-character translation rule files as the original BrailleTrans, loading and parsing them from disk. The C-style class held the translation rules in a single large array of integers, manipulating a position counter in the array and accessing the contents integer by integer. This was simple to load into memory and required the smallest processing overhead, but required the writing of more code. The Java-style class parsed the translation rules into an array of objects of a Java translation rule class. Within an instance of this translation rule class the character information was stored in separate arrays of integers. This required more processing when the translation rule file was loaded but allowed for simpler code that was easier to develop. A wider range of Java utility functions could be employed: for example, the use of the Hashtable class to index the translation rule object array saved writing more code to perform this function.

The third Java class, "Unicode", combined a similar Java-style object-oriented design with the use of Unicode and Java String objects instead of arrays of integers to represent text. The native Java string comparison and manipulation methods and the use of multiple objects to store the decision table and translation rules made this a simpler class to code. In addition the translation rules files for this class were stored as serialized instances of the class with rules fully loaded and parsed, ready to begin translation, obviating the need for parsing a translation rules file every time the class was loaded.

Method signature	**Use**
Boolean setState(int state)	Sets the state of the finite state machine, typically used to control the grade of Braille contraction.
Int getPermittedStates()	Returns the states permitted for this language.
Int getState()	Returns the current state.
String translate(String toConvert)	Returns a translated Java String object.
Int[] translate(int[] toConvert)	Returns a translated Java array of integers.

Table 1: the Java Language interface implemented by the three classes.

Every class implemented a common Java interface, allowing their translation functions to be defined consistently for future use by developers. The interface defined the methods detailed in Table 1. They allow for the control of the finite state machine and for two translate functions, one acting on arrays of integers and one on Java String objects. These suit the two types of translation; of 256-character sets as arrays of integers, and of Unicode strings. Every Unicode characters maps to a Java integer value.

The three classes differ more in obtaining an instance of each class to use to perform translation. Since the C-style and Java-style classes load a translation rules file from disk they are instantiated when needed and their constructor takes an argument that indicates the path and filename of the translation rules file to use. In comparison, the Unicode translation objects are constructed and populated before use and serialised to disk, so an instance is obtained by casting from an instance of the ObjectInputStream class that reads the Unicode translation object from another input file stream.

The classes developed are suitable components for future development with a Java application or from a native application using the Java Native Interface. They are not, however, stand-alone translation applications with functional user interfaces. Simple graphical and text interfaces were developed for testing, but the classes themselves, although developed as production candidates, could not be used by an end user.

3 Testing and results

For testing, the original C BrailleTrans program was amended to permit speed comparisons with the Java classes and provide sample translation output. Verification was performed first by comparing the output of the three classes and the original C program, and second by devising a set of translation rules and sample text to be translated, which when used together test every logical rule available. A number of different operating systems, JVMs, and machines of varying power were used. Validation compared the performance of the three classes, the original C program, published results for the BrailleTrans-based Word application, and the Duxbury commercial translation program. The optimisation of the code for the classes utilised high-resolution analysis of the internal operation of the classes, obtained from the JVM (Dittmer & White, 2001). This allowed methods that were rate limiting steps to be identified and amended if possible.

The classes were all shown to perform translation correctly, which means that any of them could be used to provide translation in a future application. The value of using Java as a component architecture was demonstrated by using the classes first to prototype an application that accessed translation rules files on demand over the Internet, and second by using them in an online translation application on a Java web server.

The focus on performance produced fast classes, achieving rates of up to 45,000 words per second, with the slowest speed recorded at 100 words per second on a low-powered older machine. Tables 2 and 3 show results on a range of platforms, JVMs and input text sizes. The four machines were: highest-specification: 866MHz, 262Mb; high-specification: 700MHz, 384Mb; mid-specification: 333MHz, 128Mb; low-specification: 66MHz, 16Mb (processor speed and memory respectively).

Rate of translation in words per second (ratio to LanguageUnicode)			
Environment	**C-style**	**Java-style**	**Unicode**
Highest-spec MS Windows 2000, mid-age JVM	45734 (14)	42844 (13)	3233 (1)
High-spec MS-DOS, newest JVM	33793 (4)	33569 (4)	8120 (1)
High-spec MS-DOS, oldest JVM	31541 (13)	18813 (8)	2441 (1)
Mid-spec Red Hat Linux	1213 (3)	1039 (3)	390 (1)
Low-spec MS-DOS	935 (7)	674 (5)	139 (1)

Table 2: Rate of translation in words per second of Java classes.

The C-style class was the fastest of the three classes because it operates on low-level, simple routines and memory structures rather than the complex object-oriented structures more common in Java and used in the Java-style class, which was the next-best performer. The small lead maintained by the C-style class over the Java-style class narrowed on the more powerful machines, where the JVM was able to make use of the increased resources to optimise speed. The difference also narrowed with more recent JVMs, which have been developed and optimised with performance as a particular consideration. On a faster machine these two classes performed better than the existing C program, while on slower machines the C program was the fastest of the three. (It is possible that the amendments necessary to BrailleTrans to allow the C program's speed to be compared to the Java classes may have been measured as detracting from its performance although controls were used to allow the results to be compensated for this possibility). The results also compared well with the performance of the BrailleTrans-based Word translation system, although this system is limited by factors in Word rather than BrailleTrans. A comparison with the popular commercial translator Duxbury [dux2001], indicated that the BrailleTrans system was not as fast as this native code application but the C-style class was of a similar order of performance, a respectable result.

The Unicode version was considerably slower than the other two classes, though it still compared well with the C BrailleTrans program on powerful machines and with the Word system. Analysis of JVM output confirmed that this was due to the representation of text in the classes as Java Unicode String objects. Manipulation and

use of these objects with the Java String functions was far less efficient, though simpler for a developer, than the lower-order processing coded manually in the C-style and Java-style classes. The majority of processing time was taken up with the Java library functions that perform String comparison and manipulation functions, which cannot be optimised, only replaced.

Platform-independence was tested across variants of three operating systems: Microsoft's DOS-based consumer-market Windows '95; Microsoft's NT-based enterprise-market Windows 2000, and the RedHat 5.1 distribution of the free GNU operating system based on the Linux kernel. The classes ran successfully and consistently on every platform. These three platforms between them command the great majority of the desktop market (Gerald, 2001). (The other major consumer platform is the Apple Macintosh, for which a JVM is available, so the classes should operate effectively there, but this was not tested). Platform independence for the C-style and Java-style may be hindered, however, by the different 256-character sets used on different systems, just as with the original BrailleTrans. Care must be taken by any developer using the class to ensure that the encoding of the input will be appropriate to the translation rules file used. This problem will not affect the Unicode class.

4 Conclusions and further work

Depending on the performance criteria of the application to be constructed, any of the classes can be used as-is for production of a text-and-Braille translation service. The favourable performance compared to the Word system suggests that any of the three Java classes could be used in a high-powered environment. However, the great difference in rates seen across the different machines suggests that the classes might not perform well enough in low-powered consumer applications. The minor advantages in using an object-oriented design in maintenance in the Java-style class do not justify its poorer performance compared to the C-style class, so the latter should form the basis for future development with consumer devices. Its performance, after any necessary amendments for the device, may well be adequate, but only testing on a specific platform will demonstrate this. Investigation of JVM performance output and the known origins of the class as highly-optimised C code suggest that the performance shortcomings are due to the nature of Java rather than a failure to produce adequate code, and this represents as best a performance as can be expected from a Java implementation using this translation algorithm.

The Java-style and C-style classes can be used with the existing translation rules files developed at UMIST. However, the Unicode class will require the conversion of these translation rules files into Unicode – a relatively trivial task already performed for English Braille - or the development of future translation rules files in Unicode.

The Unicode solution has scope for improvement in its performance. The Java Strings used to represent text internally should be replaced with arrays of 16-bit Unicode Java character types. This could be achieved simply by amending the Java-style class, which already uses arrays of integers internally to represent text and performs the necessary translation functions using these arrays. The performance improvement should allow the Unicode class to operate at the same speed as the Java-style class. This should be usable for many applications and of benefit where a Unicode solution is required. One potential application is the addition of translation functionality to Sun's free StarOffice or OpenOffice office application suite in a parallel development to the use of the original BrailleTrans to provide translation functionality to Microsoft Word. This would provide a user-friendly front end to this free office application suite.

References

Blenkhorn, P. (1995) "A System for Converting Braille into Print", *IEEE Transactions on Rehabilitation Engineering*, Vol 3, No 2, pp 215 – 221, June 1995.

Blenkhorn, P. (1997) "A System for Converting Print into Braille", *IEEE Transactions on Rehabilitation Engineering*, Vol 5, No 2, pp121 - 129, June 1997.

Blenkhorn, P. & Evans, G. (2001) "Automated Braille Production from Word-Processed Documents", *IEEE Transactions on Rehabilitation Engineering* Vol 9, No 1, pp 81 – 85 March 2001.

Blomvquist, M., Burman, P. & Blenkhorn, P. (2002) "Emboss contracted Braille directly from your word-processor using WinBraille", *Proceedings of the 2002 CSUN 17th Annual Conference*, 18-23 March 2002, Los Angeles, United States.

Bruce, I., McKennell, A. & Walker, E. (1991) "Blind and partially-sighted adults in Britain: the RNIB survey", Volume 1, H.M.S.O, London, 1991.

Dittmer, U. & White, G. (2001) “ProfileViewer”, application website, April 2001, <http://www.capital.net/~dittmer/profileviewer/index.html>

Duxbury Systems (2001) April, <http://www.duxburysystems.com>

Eckel, B. (2001) “Comparing C++ and Java”, JavaCoffeeBreak website, June 2001, <http://www.javacoffeebreak.com/articles/thinkinginjava/comparingc++andjava.html>

Fowles, K. (1997) “Character sets”, Microsoft websites, June 1997, <http://www.microsoft.com/typography/unicode/cs.htm>

Geralds, J. (2001) “Windows’ dominance unswayed”, VNUNET website, 1 March 2001, < http://www.vnunet.com/News/1118373>

Gosling, J., Joy, B., Steele, G. & Bracha, G. (2000) "The Java Language Specification, Second Edition", Sun Microsystems website, July 2000. <http://java.sun.com/docs/books/jls/second_edition/html/j.title.doc.html>

Gschwind, M. (1994) "ISO 8859-1 National Character Set FAQ", Verein der Informatik Studierenden website, 1994, <http://www.vis.ethz.ch/manuals/charsets/FAQ-ISO-8859-1.html>

King, A. (2001) “Text and Braille Computer Translation”, Masters thesis, UMIST, Manchester, 2001.

Shirazi, J. (2000) “’Any Java program can run fast’ says author”, O’Reilly Publishing website, October 2000, <http://press.oreilly.com/javapt.html>

Sun Microsystems (2001) "Java 2 Platform, Micro Edition (J2ME)", Sun Microsystems website, April 2001, <http://java.sun.com/j2me/>

Unicode, Incorporated. (2001) "What is Unicode?", Unicode Incorporated website, September 2001, <http://www.unicode.org/unicode/faq/utf_bom.html>