Working Paper Series
ISSN 1170-487X

Bi-level Document Image Compression using Layout Information

by Stuart J. Inglis and Ian H. Witten

Working Paper 96/1
January 1996

Department of Computer Science
The University of Waikato
Private Bag 3105
Hamilton, New Zealand

Bi-level Document Image Compression using Layout Information

Stuart J. Inglis and Ian H. Witten

Computer Science, University of Waikato, Hamilton, New Zealand
email: {singlis, ihw}@cs.waikato.ac.nz

1 Introduction

Most bi-level images stored on computers today comprise scanned text, and their number is escalating because of the drive to archive large volumes of paper-based material electronically. These documents are stored using generic bi-level image technology, based either on classical run-length coding, such as the CCITT Group 4 method, or on modern schemes such as JBIG that predict pixels from their local image context. However, image compression methods that are tailored specifically for images known to contain printed text can provide noticeably superior performance because they effectively enlarge the context to the character level, at least for those predictions for which such a context is relevant. To deal effectively with general documents that contain text and pictures, it is necessary to detect layout and structural information from the image, and employ different compression techniques for different parts of the image. Such techniques are called *document image compression* methods.

A scheme that extracts characters from document images and uses them as the basis for compression was first proposed over twenty years ago by Ascher and Nagy (1974), and many other systems have followed (Pratt *et al.*, 1980; Johnsen *et al.*, 1983; Holt, 1988; Witten *et al.*, 1992, 1994). The best of these methods break the compression process into a series of stages and use different techniques for each. Whereas the images of the actual characters must necessarily be compressed using an image compression model, their *sequence* can be better compressed using a standard text compression scheme such as PPMC (Cleary and Witten, 1984; Bell *et al.*, 1990).

The present project extends these systems to include an automatic discrimination between text and non-text regions in an image. If a particular area of an image is known to be a photograph, for example, a more appropriate compression model can be used. In order to do so, the document must first be divided into apparently homogeneous regions or *zones*; in order to do that, it is necessary to make estimates of the inter-line and inter-character spacing. The analysis of document layout has been investigated previously by several researchers (e.g. Casey *et al.*, 1992; Nagy *et al.*, 1992; Pavlidis and Zhou, 1992; O'Gorman, 1993); but the methods often depend on particular numeric values or rules that relate to the resolution, print size and typographical conventions of the documents being analyzed, and on hand-tailored heuristics to differentiate blocks of text from drawings and halftones. Our emphasis is on completely automatic processing that does not require any manual intervention or tuning, except for the fact that it may be trained on representative examples of documents that have been segmented and classified manually.

This paper describes the document image compression process, and gives an account of how the zones in a document are located and how they are classified into text and non-text. We provide an extensive comparison of the compression achieved by this new system with JBIG (CCITT, 1993),

the standard for bi-level image compression, on one thousand document images.[1] We begin by mentioning a large publicly available corpus of scanned images that was used for the work, in order that it can be replicated. The resulting document image compression system achieves lossless compression ratios 25% greater than JBIG, and very slightly lossy ratios that are nearly three times as great as JBIG.

2 Image Corpus

Previous researchers on document image compression have obtained their results using the eight standard CCITT facsimile test images (Hunter and Robinson, 1980), or with unavailable private image collections. We have selected a large publicly-available database of 979 bi-level images, published by the University of Washington, as the image corpus for the experiments (University of Washington, 1993).

The images in the UW ENGLISH DOCUMENT IMAGE DATABASE I are scanned at 300 dpi from 160 different journals. Many have small non-zero skew angles, and in some cases text is oriented vertically instead of horizontally. Each one has been manually segmented into zones; there are a total 13,831 zones in the database or an average of 14 per image. For instance, a particular image may be divided into six text zones, one halftone zone, and one table zone. The corpus contains a total of twelve zone classes. Text zones, which are by far the most frequent (88% of all zones), represent paragraphs, bibliography sections, page numbers and document titles; other zone types include line drawings, halftones, math, and rules.

3 Document Image Compression

Document image compression works by building a dictionary of repeated patterns, usually representing characters, from an image and replacing them by pointers into the pattern dictionary. Because most English characters contain only one connected area, we define a *component* as an eight-connected set of black pixels and use the component in all stages of the system.

The first stage is to extract the components from the image. The dictionary is built from these using a template-matching procedure. As the components are processed, the patterns belonging to each dictionary entry—that is, all instances of that character—are averaged. When all components have been processed, the dictionary contains an average, supposedly representative, example of each different component, and the components in the image can be represented as indexes into the dictionary.

This kind of compression, where a dictionary is created and indexes into it are coded, can be viewed as a type of (LZ78-style) macro compression. However, during the matching process that determines whether or not two components are similar, matches are accepted even if the components are not completely identical. Thus when the image is reconstructed from the dictionary, there is noise around the edges of components. The difference between the reconstructed image and the original is known as the *halo* image. Of course, if exact matching were enforced, the dictionary would become almost as large as the number of components in the image, because the noise makes each component unique.

[1]This present version of the paper contains preliminary results for a subset of 200 images.

A simple model of the process consists of five steps:

- extract all components from the image;
- build a dictionary based on the components;
- compress the dictionary;
- compress the components using the dictionary;
- compress the halo to form a lossless image.

This model is described by Witten *et al.* (1992). In this paper we extend it to cater for zones of different types. Components are clustered into zones, and the zones classified into the types *text, non-text* and *noise*. Only the text components are used in the dictionary building stage. The halo of errors around the reconstructed characters is compressed as before. Now, however, there is a further image, called the *remainder*, which comprises the non-text regions (including noise).

The new model contains eight steps:

- extract all components from the image;
- group the components into zones;
- classify each zone into text, non-text or noise;
- build a dictionary based on the text zones;
- compress the dictionary;
- compress the text components using the dictionary;
- compress the remainder—the non-text and noise regions;
- compress the halo to form a lossless image.

We briefly describe each step. Figure 2 gives an example image and shows how it is broken down into parts.

COMPONENT EXTRACTION

Components are located by processing the image in raster-scan order until the first black pixel is found. Then a boundary tracing procedure is initiated to determine the width and height of the component containing that pixel. A standard flood-fill routine is called to extract the component from the image. As each component is extracted, several features, including the centroid and area, are determined and used later for comparing and classifying components.

DOCUMENT ZONE CLUSTERING

Components are grouped into zones by clustering them using a bottom-up technique known as the *docstrum* (O'Gorman, 1993). This involves finding the k nearest neighbors of each component, using the Euclidean distance between the components' centroids. Each component is linked with its neighbors to form a graph. In our experiments k=5 is used, because the five closest neighbors usually include two or three components in the same word, or the same line, and two components in adjacent lines.

Once the nearest neighbors are found, links in the graph between components that exceed a threshold are broken. This is a dynamic threshold, calculated directly from the docstrum, and is defined as the smaller of three times the most common component-within-line distance, and $\sqrt{2}$ times the average between-line distance. After the links whose lengths exceed this threshold have been broken, the transitive closure of the graph is formed. The resulting sub-graphs represent document zones.

ZONE CLASSIFICATION

To use the appropriate compression model, each zone's classification must be determined. Instead of developing an *ad hoc* heuristic to distinguish the zones, we use a machine learning technique to generate rules automatically for each possible type of zone. Machine learning methods learn how to classify instances into known categories. The process involves a training stage, which operates on pre-classified instances, followed by a testing stage, which uses the rules obtained during training on instances whose class is unknown. We use the well-known and robust C4.5 method (Quinlan, 1992), having determined in preliminary tests that it achieves consistently good results on this problem compared with other machine learning methods (Inglis and Witten, 1995).

Six features are calculated for each zone:

- zone density—number of components per unit area;
- component density—mean density of black pixels in the components' bounding boxes;
- aspect ratio—mean ratio of height to width of the components' bounding boxes;
- circularity—the square of the perimeter divided by the area, averaged over all components;
- separation—mode distance between a component and its nearest neighbor;
- run length—mean horizontal run length of the components' black pixels.

The 13,831 zones in the database images have been classified manually and comprise a total of 12,216 text and 1615 non-text zones. They were divided randomly in half to form training and test sets. The features for the training set, along with the manually-assigned class, text or non-text, were given to C4.5, which produced a decision tree. When this tree was used to classify the zones in the test set, 96.7% accuracy was achieved in replicating the manually-assigned classifications. (This result is averaged over 25 runs, using different random splits each time.) Figure 1 shows a simple 10-node decision tree that is generated for the simpler task of discriminating text regions from ones containing halftone. Trees for the text *vs.* non-text discrimination are three or four times as large as this.

For example, Figure 2(a) shows an image in its original form, and Figure 2(b) shows the text zones that are identified by segmenting it into zones as described above and classifying them automatically. The training data for the classifier comprises image zones that were determined manually: these are not the same as the zones generated automatically. In general, the automated procedure finds more zones, although this does depend on the distance threshold.

DICTIONARY GENERATION

The dictionary is built using only the zones that are classified as text. The dictionary-building operation relies on the ability to compare two different components to see whether they represent the same character. This comparison operation is known as *template matching*. The template matcher aligns the components by their centroids and processes them pixel by pixel. The method that we use rejects a match if the images differ at any point by a cluster of pixels of greater than a certain size; in order to detect mismatches due to the presence of a thin stroke or gap in one image but not the other, another heuristic is used (Holt, 1988).

The easiest way to build the dictionary is to store the first example of each different component as a reference example for subsequent comparisons. However, compression is adversely effected if the first component is a poor representative of its class. Consequently a running average is made of all examples of each different component during the dictionary-building process. As components are

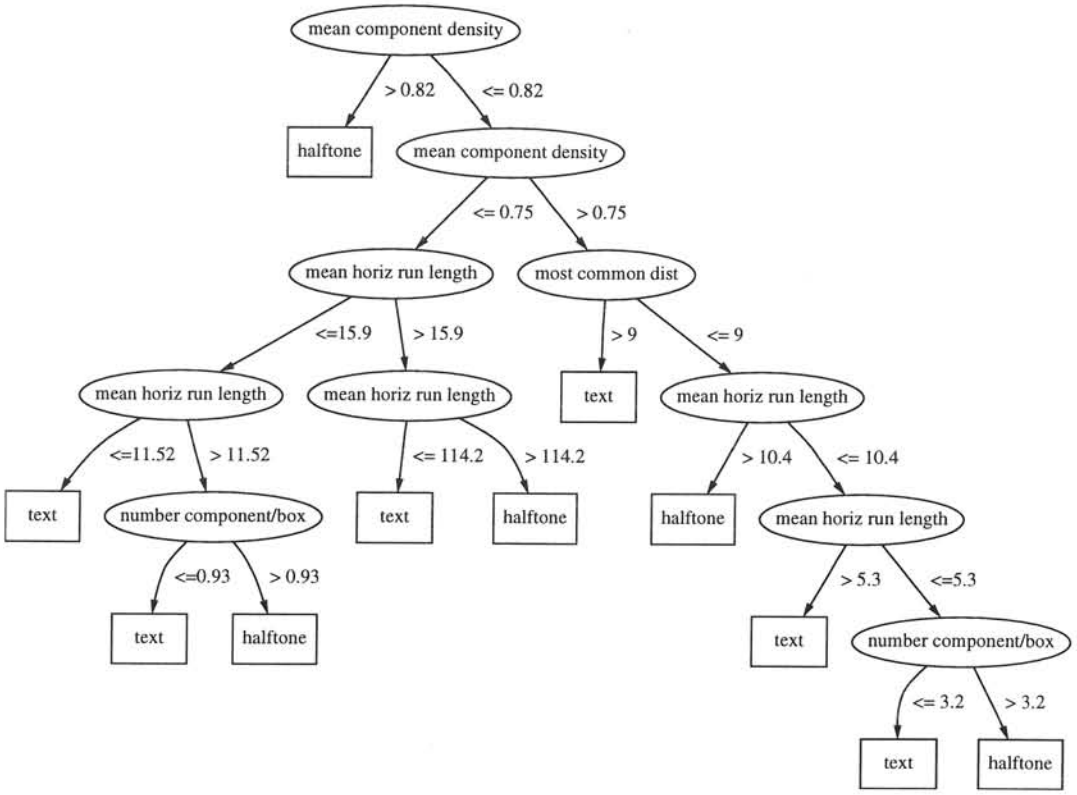


Figure 1: Decision tree discriminating text from halftones

extracted from the image, they are checked against the dictionary to determine whether a sufficiently similar component has been seen before. If not, the new component is added to the dictionary. Otherwise, if the component did match a dictionary element, that element is updated by incorporating the new component into the running average.

Because of this averaging process, it is possible that once the dictionary has been finalized a component might not match any of the elements in the dictionary. Therefore after the building process, each component is re-checked against the dictionary and added if necessary. This happens rarely: it accounts for less than 2% of dictionary entries.

DICTIONARY COMPRESSION

The dictionary elements are compressed using Moffat's (1991) two-level context method. This scheme gives the highest compression of all pixel-context based methods. The compression model is initialized at the start of compression, and the statistics are carried over from one dictionary element to the next.

TEXT COMPONENT COMPRESSION

The components in each text zone are processed sequentially. First, their indexes into the dictionary are compressed using PPMC (Bell *et al.*, 1990). Then the relative position of each component with respect to its predecessor in the zone is calculated. These offsets are compressed using a three-tier model, using the first model if possible, the second one if the statistics required by the first are not available, and the third as a last resort. The first model contains counts of offset

values with respect to a dictionary element; that is, the offsets are character-specific. The second accumulates all offset information for every dictionary element; a generic model for all characters. Finally, if neither of these first two models can be used, the offset is gamma encoded (Elias, 1975).

At this stage the dictionary has been compressed, the component order is known, and the component positions in the image are known. Thus the recipient can form a *reconstructed* image using the average components from the dictionary.

REMAINDER COMPRESSION

The *remainder image* consists of the non-text and noise components. An example is shown in Figure 2(d). As the remainder is independent of the previous stages, the other images are of no help in coding it. Therefore it is compressed using the two-level context method. If the system is processing an image in lossy mode, this is the end of the compressed stream.

HALO COMPRESSION

As there is only one example of each component in the dictionary, pixel errors occur around the borders of characters when the reconstructed image is compared with the original. These errors constitute the halo image, and Figure 2(c) shows an example. It is possible from the halo image to make out the outlines of characters, and in some places actually to read them. Because the reconstructed image has already been coded, it makes sense to encode the halo with respect to it. In fact, since the entire reconstructed image is available, a context that is centered on the pixel to be coded can be used. This has been coined "clairvoyant" compression (Witten *et al.*, 1992). In practice, it turns out that coding the original image using a clairvoyant context in the reconstructed image leads to higher compression than if the halo is coded using the same context, because the halo is complicated by its exclusive-or nature. The compressed stream is now lossless.

4 Experiments

In this section, the document image compression scheme described above is compared with two pixel-context models: JBIG[2] and Moffat's (1991) two-level compression scheme.

The JBIG implementation gave an overall compression ratio of 17.5:1 over the corpus of images, while the two-level scheme achieved 19.1:1. The document image compression scheme was run in two different modes, with and without inclusion of the halo. The lossless mode, with the halo, achieved 22.1:1 and the lossy mode 49.4:1. These results are summarized in Table 1. It should be noted that the lossy mode guarantees a very good approximation to the original image: it is impossible for a block of more than four incorrect pixels to occur. Arguably, the lossy image is actually cleaner than the original because all examples of each letter are exactly the same.

The times taken to compress and decompress, measured on a SUN SparcCenter 1000, are shown in Table 2.[3] The document compression scheme is asymmetrical in that dictionary building and zone classification are not required when decompressing. This contrasts with JBIG and the two-

[2]We have used an implementation of JBIG written by Markus Kuhn (1995), because it is the most complete of the two publicly available implementations.

[3]The times quoted for the document compression scheme are somewhat pessimistic because no serious attempt has been made to optimize the code.

Scheme	Total size	Compression ratio
JBIG	11 841 564	17.5
Two-level	10 854 655	19.1
Lossless	9 394 914	22.1
Lossy	4 202 827	49.4

Table 1: Compression figures for the different schemes

Scheme	Compression time (min:sec)	Decompression time (min:sec)
JBIG	0:22	0:23
Two-level	1:00	1:04
Lossless	5:00	2:00
Lossy	3:50	0:20

Table 2: Compression time per image for each scheme

Header	Dictionary	Symbols	Offsets	Halo	Remainder	Footer
0.1%	5.8%	5%	4%	60%	25%	0.1%

Table 3: Relative component sizes for document image compression

level scheme, for which compression and decompression take about the same time. Lossless compression in the document-based scheme is fifteen times slower than JBIG, decompression is six times slower. In lossy mode, compression is ten times slower and decompression is slightly faster. Table 2 shows the full results for all schemes.

Due to the asymmetric nature of the document image compression system, it is most applicable to situations where lossy decompression is time-critical, such as on-line searches, or when compression time is unimportant, such as in CD-ROM mastering.

Table 3 shows the contribution of the different kinds of information in lossless document image compression to the total compressed size, for a typical example image. The halo is over half the file size, which explains why the lossy compression ratio is more than twice as great the lossless one.

5 Conclusion

We have extended a document image compression system to include layout and zone classification information, and made it robust and suitable for general-purpose use by removing the necessity for manual intervention or tuning to particular typographical conventions. The resulting compression scheme yields a 25% improvement over JBIG. Using the system in lossy mode where the halo around the characters is not encoded, the compression ratio becomes nearly three times that for JBIG. The new scheme is asymmetric in that decompression is much faster than compression, but both are considerably slower than JBIG—except for lossy decompression, which is slightly faster.

References

Ascher, R.N. and Nagy, G. (1974) "A means for achieving a high degree of compaction on scan-digitized printed text." *IEEE Transactions of Computers*, vol. 23, no. 11, pp. 1174–1179, November.

Bell, T.C., Cleary, J.G. and Witten, I.H. (1990) "Text compression", Englewood Cliffs, NJ, USA: Prentice Hall

Casey, R. Ferguson, D., Mohiuddin, K. and Walach, E. (1992) "Intelligent forms processing system." *Machine Vision and Applications*, vol. 5, pp. 143–155.

CCITT (1993) "Progressive bi-level image compression." *Recommendation* T.82; also appears as ISO/IEC standard 11544.

Cleary, J.G. and Witten, I.H. (1984) "Data compression using adaptive coding and partial string matching," *IEEE Trans. on Communications*, COM-32, pp. 396–402.

Elias, P. (1975) "Universal codeword sets and representations of the integers." *IEEE Trans Information Theory*, vol. IT-21, pp. 194–203.

Holt, M.J. (1988) "A fast binary template matching algorithm for document image data compression." in *Pattern Recognition*, J. Kittler Ed. Berlin, Germany: Spring Verlag.

Hunter, R. and Robinson, A.H. (1980) "International digital facsimile coding standard." *Proc IEEE*, vol. 68, no. 7, pp. 854–867.

Inglis, S. and Witten, I.H. (1995) "Document zone classification using machine learning." *Proc. Digital Image Computing: Techniques and Applications*, Brisbane, Australia, December.

Johnsen, O., Segen, J. and Cash, G.L. (1983) "Coding of two-level pictures by pattern matching and substitution." *Bell System Tech. Jour.*, vol. 62, no. 8, pp. 2513–2545, May.

Kuhn, M. (1995) URL: http://wwwcip.informatik.uni-erlangen.de/user/mskuhn

Moffat, A. (1991) "Two level context based compression of binary images." in *Proc. IEEE Data Compression Conference*, Los Alamitos, CA, USA: IEEE Computer Society Press, pp. 328–391, April.

Nagy, G., Seth, S. and Viswanathan, M. (1992) "A prototype document image analysis system for technical journals." *IEEE Computer*, pp. 10–22; July.

O'Gorman, L. (1993) "The document spectrum for page layout analysis." *IEEE Trans Pattern Analysis and Machine Intelligence*, vol. PAMI-15, no. 11, pp. 1162–1173, November.

Pavlidis, T. and Zhou, J. (1992) "Page segmentation and classification." *CVGIP: Graphical Models and Image Processing*, vol. 54, no. 6, pp. 484–496.

Pratt, W.K., Capitant, P.J., Chen, W.H., Hamilton, E.R. and Wallis, R.H. (1980) "Combined symbol matching facsimile data compression system." *Proc. IEEE*, vol. 68, no. 7, pp. 786–796, July.

Quinlan, J.R. (1992) C4.5: *Programs for Machine Learning*, Morgan Kaufmann.

University of Washington English Document Image Database I, (1993) Seattle, WA, USA.

Witten, I.H., Bell, T.C., Harrison, M.H., James, M.L. and Moffat. A. (1992) "Textual image compression." *Proc. DCC*, edited by J.A. Storer and M. Cohn, pp. 42–51. IEEE Computer Society Press, Los Alamitos, CA.

Witten, I.H., Bell, T.C., Emberson, H., Inglis, S. and Moffat, A. (1994) "Textual image compression: Two-stage lossy/lossless encoding of textual images." *Proc. IEEE*, vol. 82, no. 6, pp. 878–888, June.

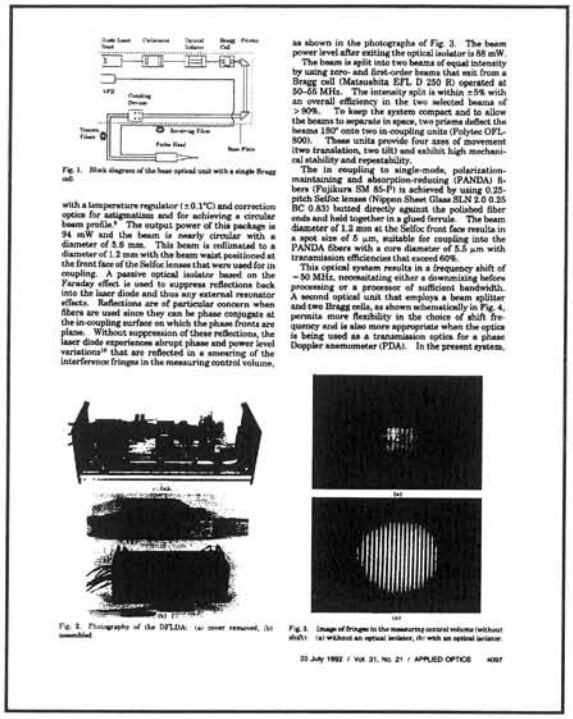


Figure 2(a): Original Image

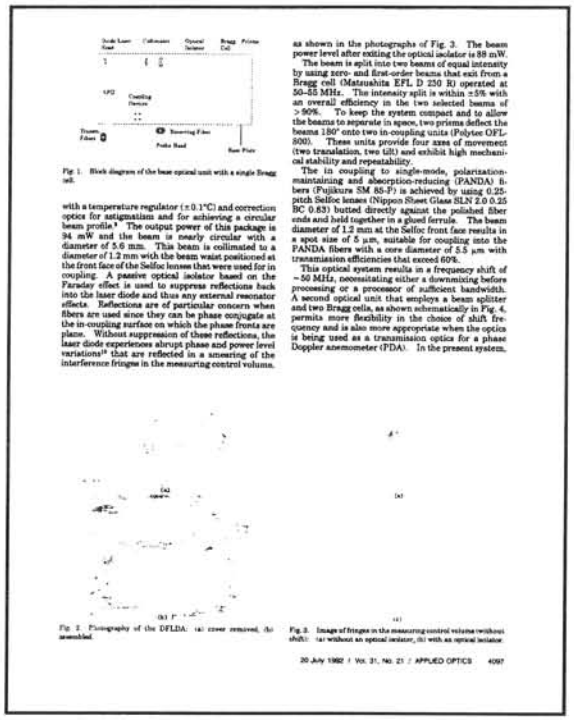


Figure 2(b): Text components

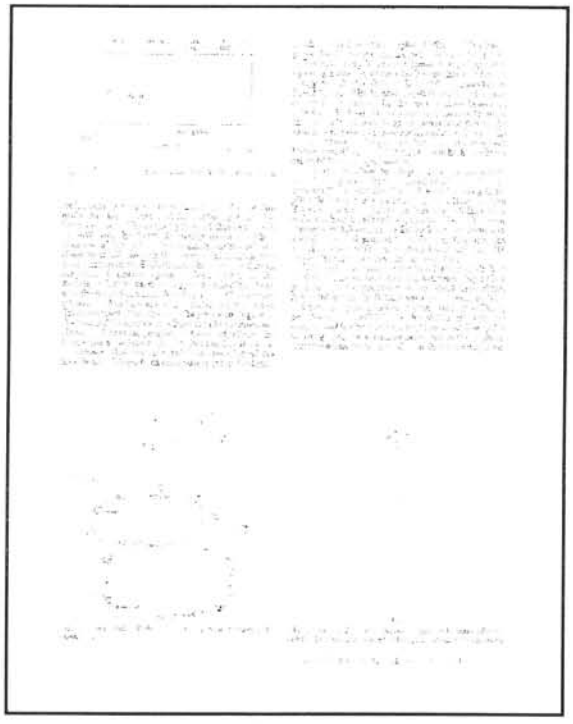


Figure 2(c): Halo image

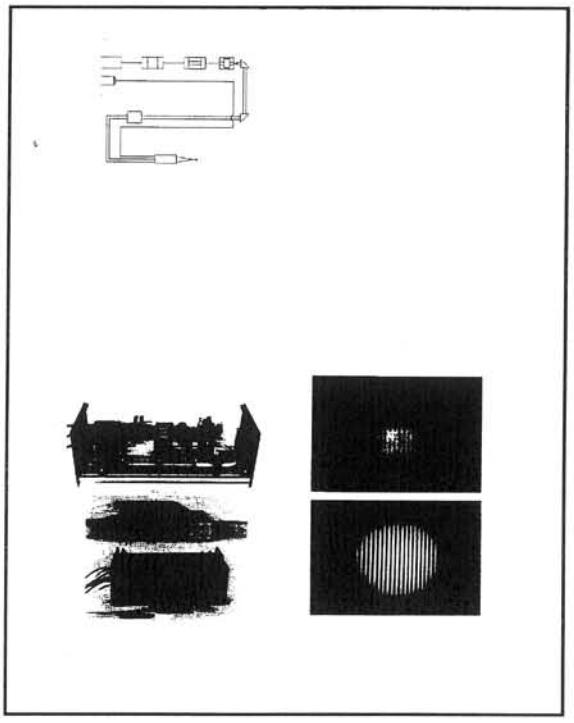


Figure 2(d): Remainder image

Working Paper Series
ISSN 1170-487X

Software for the Rest of the World

by Alvin Yeo and Robert H. Barbour

Working Paper 96/2

February 1996

Department of Computer Science
The University of Waikato
Private Bag 3105
Hamilton, New Zealand

Software for the Rest of the World

Alvin Yeo and Robert H Barbour

Department of Computer Science, University of Waikato, Hamilton New Zealand

Email: r.barbour@waikato.ac.nz

Abstract

A survey is made of facets of the process of providing software across national and cultural boundaries. Internationalisation (i18n), localisation (l 10n) and globalisation (g11n) are identified as three descriptors for recent Information Technology developments in this field. Current practice and advice for successfully providing software in places other than countries and cultures of origin is reported. Suggestions for further work are made in the light of the survey.

1. Introduction

Information Technology (IT) is being adopted by users all over the world. Software users are not limited to the English speaking West but include people from all language groups and cultures. Kay (1994) reports "international sales make up more that half of the revenues for the top 100 US software companies". Reduced demand in Western nations has provoked software developers to widen markets and amortise development costs by attempting to meet the needs of all users from diverse cultures at the scale of major language groups. Software developers have also attempted to customise their products to meet the needs of specific communities and cultures. In this paper we describe the current strategies and approaches used in the design of software intended for use in global markets. A check-list of cultural attributes that need to be addressed during the software development process is provided. Possible future approaches to the globalisation of software will also be discussed.

2. Key concepts and Definitions

Software that is designed to accommodate the needs of many cultures meets design criteria through internationalisation or the process of making a system or application software independent of or transparent to natural language. If a system can support any language, then it is fully internationalised: if it supports only a limited subset languages, then it is partially internationalised (Unicode and Internationalisation Glossary, 1995). Preparing software in this way is usually carried out in the country of origin of the software (Russo and Boor, 1993). The modifications consisting of the removal or replacement of culture-specific features in the original software (Russo and Boor, 1993). Internationalisation is also known as i18n: there being 18 letters between the "i" and "n". It is also related to the process known as globalisation in which 'the global corporation will seek sensibly to force suitably standardised products and practises on the entire globe (Levitt, in Russo and Boor, 1983). The processes of globalisation falls outside of the scope of this paper.

Contrasting Levitt's view, others promote the idea of diversity in product marketing. Software *localisation* refers to the modification of software to allow its use in a *locale* other than the software's origin. A locale is a geographical region defined by a country, language or set of cultural conventions (adapted from Hall, 1994). Locales are not only characterised by country alone because in some countries, three different languages are spoken in a particular country, e.g. French, German, and Italian are spoken in Switzerland (del Galdo, 1990); Malay, English, Mandarin, Tamil are used in Malaysia. Software localisation is,

therefore, not limited to "translating applications menus, dialogue boxes, alert boxes and content areas into a language or regional dialect" as defined in the Human Computer Interface Guidelines (Apple Computer, 1992). Localisation is also known as l10n; there being 10 letters between "l" and "n" in the word "localisation".
Both the software localisation and internationalisation processes should reflect consideration of the language issue and other user interface design issues such as the provision of icons, metaphors, navigational techniques and mental models, input and output mechanisms..
Culture is defined as behaviour typical of a group or class (Webster, 1995). Culture is conceptualised as a "system of meaning that underlies routine and behaviour in everyday working life" (Bødker, 1991). "Culture includes race and ethnicity as well as other variables and is manifested in customary behaviours, assumptions and values, patterns of thinking and communicative style" (Borgman, 1992). Thus, using the phrase "supporting a locale" means supporting that locale's language, cultural conventions and interface design preferences; an internationalised software package supports many locales simultaneously while localised software supports only one particular locale. An example of partially internationalised is Microsoft Windows in which the user can employ the Internationalisation section in the Control Panel to select a particular locale.

3. **Internationalisation and Localisation**

Most of the early software was developed in the United States (US) by Americans for Americans. There was little demand for software outside the US. As the number of computers increased around the world, demand for software also grew. Economic growth, especially in Asia, probably contributed most significantly to this demand. (See Section 5). With this growth of the overseas markets for software, software developers began to adapt their original software to other locales. The first attempts at localisation would have involved direct translation of the software into the language of the target community with modifications to format or display of numbers, date, time as well as character sets. This localisation process demanded time and effort. Furthermore, the early software developers did not bear in mind "problems of translation or effects of differences in culture and language on usability of software interfaces" del Galdo (1990).
The efforts required to localise software were magnified given that particular software needed to be localised to more than one language. Software companies found it was easier internationalise software - initially design the software such that the software provides the necessary support for a number of languages and cultural conventions (Chris Miller, 1994). Internationalisation thus brought about huge savings, increased revenues and profits, and shorter time to market, as little or no modification was required to localise an internationalised software (Chris Miller, 1994). The internationalisation process is, in a sense, a means to an end; the end being localisation.

4. **Approaches of Internationalisation and Localisation**

The current approach to preparing a product for world-wide use is to first internationalise and then localise the product (Russo and Boor, 1993) (Belge, 1995). There are a number of approaches to internationalisation.
Chris Miller (1994) reported these methods of internationalisation:

"1. Compile Time Internationalisation
Programmers change the files that contain the source code and algorithms
2. Link-Time Internationalisation
Programmers extract all the text strings along with algorithms that are dependent on language or culture, from the program code.
3. Run-time Internationalisation
End users select locale. The software contains text files for more than one locale."

Chris Miller (1994) also quotes another strategy by Bill Tuthill's four levels of internationalisation from the Solaris Internationalisation Guide: Global Product Design (Prentice-Hall, 1992):

> Level 1: software with texts and code sets that are adaptable internationally or are considered "8-bit clean" and that support the Latin-1 code set.
> Level 2: Software with formatting and collation methods that are locale sensitive. Includes sorting order for alphabets and formats for date, time and so on.
> Level 3: Software that allows easy translation of user-visible text, usually by placing such text into a separate file form the executable code,
> Level 4: Software that supports East Asian languages, which frequently require multi byte code sets.

Abramson (1994) believes that "the best design approach to internationalise software is to maintain strict language independence throughout the code".

Russo and Boor (1993) provided these guidelines:

"1. A product team needs to think about each culture the product will be sold in from the beginning of the product development cycle.
2. Are the product requirements different? Do the features need to be modified? How are the users different? The team needs to identify the differences, understand the design implications and design for them before the original product is released.
3. Establish A Relationship Within The Target Market
   A close working relationship or partnership with natives from the target cultures is needed.
   During the various stage of the product development cycle, information must be reviewed and feedback must be provided from the representatives of the target cultures.
   Early in the development cycle when features are defined, review for errors of omission and commission must be carried out by the target cultures.
   If additional features are identified, the original product development team can make appropriate adjustments at the product's architecture to allow for the change during the localisation process.
4. International Usability Testing
   Usability testing should be performed at the same time as domestic usability tests; before the product is released."

Belge (1995) provides the following steps to "more thoroughly serve international markets":

"1. Identify all areas of all products that change when going from one country to another. (A list of these areas are provided in Section 7).
2. Gain a better understanding of users.
   How users in the US and abroad think about, and deal with international issues, At this stage, one should investigate actual user work habits, and propose a conceptual, or user model that best describes how users think about their work.
3. Leverage technology to facilitate design suitability and consistency.
   Take advantage of the best and most appropriate technologies to test out your conceptual and design ideas.
4. Decide a common strategy that can accommodate the many ways in which international users work.
   Once ideas have been tested create guidelines and design specifications. When working on very large projects this will ensure that everyone is going in the same direction.
5. Unify the user interface design of the products by using this strategy".

The most direct approach localising approach to a product is translating the text in the interface, as well as that embedded to the language, to that of the target community. The translation models used to translate software in Aston and Dolden (1994) are similar to these techniques mentioned by Chris Miller (1994). For instance, the translation of "text external from the executable file (extracted either manually or with software tools" and "text external to the source code files but embedded in the executable file at compilation" is similar to Chris Miller's (1994) methods 1 and 2 above. If the product to be localised is an internationalised product, and if the internationalisation has been carried out effectively, localising the product will require little effort. The localisation process may only require the selection of desired locale.

5. Historical marketing of Software

5.1 Money Matters

It was only in the early 1990's that there has been indication of growth in research into localisation of software (Russo and Boor, 1993),(del Galdo, 1990),(Nielsen, 1990). One reason that may have catalysed this development has been the economic boom in the Asia as well as the opening up of markets in China and Vietnam, thus allowing a greater *economy of scale*. With greater economies of scale, more firms may find it more financially feasible to localise software. A large component of computer corporation's revenue have been derived from outside US , for example in IBM's 1994 Annual Report, 67% of IBM's revenue were derived from outside US. Furthermore, more and more countries are adapting Information Technology . The software market has become more culturally diverse with more countries becoming more technologically adept. The natives of many countries are able to develop software or localise software for their own consumption. Such as for example, the development of input and display of Chinese, Japanese and Korean characters.

5.2 Ethnocentrism

Another reason why software was not localised was because of the *ethnocentrism* of the software developers in the US. The Oxford Dictionary defines ethnocentrism as the egotistical opinion that their own race is the most important. This ethnocentric behaviour could be explained by this abstract from Gannon and Associates (1994):

> "Consequently, US economic success and military might often induces egotistical reactions to international events among its citizens. These narcissistic feelings are frequently evoked by news media's persistent portrayal of and fascination with the world disasters and mishaps. To the American media, "no news is good news," and thus the outside world is often reported as volatile, violent, and miserable. The positive features of foreign countries are rarely highlighted by US media, because the average American is constantly bombarded with news of famines, wars, violence, and political turmoil from the outside world. He or she is frequently not aware of the pleasant and fascinating features of other nations. This ethnocentrism is so extreme that many Americans believe that the United states is the safest and most prosperous country in the world, even when the facts do not support such a conclusion."

It is possible that such ethnocentric behaviour may have prevented the development of localised software. Other examples of this ethnocentric behaviour: rationalisation "they can understand English" was given for not translating a product's documentation and on-line help (Sprung, 1990), an American international marketing manager when asked whether translation was at all necessary was quoted "because isn't 1992 when everyone in Europe is going to speak English?" (Sprung, 1990).

6. Why localise software?

Increasing profits should not be the only goal for companies internationalising and localising their software. This section describes other reasons to globally marketing software.

From a social standpoint, localisation of software can serve as a medium to bring nations together by increasing the potential for communication between the countries involved. For example, software written in English and translated to Chinese may require the co-operation of both countries. Software, such as CAL, require large investments of time and

programming talent. With localisation of such software, there would be increased world-wide availability of state-of-the-art software as software companies increased the distribution of software, thus meeting the needs of a wider range of people. Many projects fail due to lack of acceptance by target cultures. With localisation, such projects would also stand a better chance of success. Furthermore, by localising software, communities less adept in software technology are exposed to state of the art software; thus, localisation of software can also be an effective mechanism for the transfer of technology.

People who can interact with computers in their own language learn and progress faster than those forced to use foreign software (Griffiths, 1994). For example, Bulgarian children only painted and drew but did not write when using software with an English language interface. However, in software that was localised, the children attempted to write creatively and explore all potential of the software (Griffiths, 1994).

Many people identify strongly with the culture they were brought up in. A particular community may feel "personally rejected and second rate if their language is not accepted into the magic circle of technology" (Griffiths, 1994). These communities may never develop the confidence to adopt IT. As such, significant contributions where IT is used would not materialise if these people cannot use the technology (Griffiths, 1994).

Furthermore, focusing on providing for "indigenous" cultures, preserving some of the diversity of human experience. By doing so "a wider variety of perspectives, potential insights and solutions to the world's problems" are preserved (Griffiths, 1994). This argument is also supported by a study carried out by the American Management Association (AMA) that "heterogeneous work groups create solutions to work and business problems that are more innovative and more effective than those developed by homogeneous groups" (HR Focus, 1993). It is clearly in the interest of humanity to maintain the diversity of human cultures.

7. Factors in internationalising and localising software.

Internationalising and localising software is not just a matter of translating the software. Many issues and factors are included in the partial list below. Software developers interested in developing software outside their locales need to be aware of these issues.

7.1 Date

Date formats differ in sequence from one locale to another for example: day-month-year, month-day-year, year-month-day. The dates also differ in the use of separators (full-stop/period, comma, hyphen, space, slash or no separator). Some dates use either numbers or alphanumeric characters (del Galdo, 1990)(Russo and Boor, 1993). Del Galdo suggests using alphanumeric characters for the months to avoid confusion especially if the application has an audience of more than one nationality. Dates can be written in a number of ways: May 12, 1959 (US), 5 June 1995, 5-Jun-95, 12/5/59 or 12.5.59 (UK), 12/5/59(Denmark), 1959-05-12 (Sweden) (Chris Miller, 1994).

7.2 Calendars

Different countries use different types of calendars such as Gregorian, Chinese, Chinese Lunar, Islamic, Hebrew and Japanese Imperial era. The US and UK and most the Western world use the Gregorian calendar. The Chinese use a Lunar calendar that is synchronised with the solar calendar by adding months, and the Muslims use another version of lunar calendar (Belge, 1995). Some countries use two calendars. Israel uses the Gregorian and Hebrew calendars while the Arab countries, the Gregorian and Islamic calendar (Chris Miller, 1994). Although the Gregorian calendar is used, the Japanese also count years using the Japanese imperial era based on the date the Japanese emperor ascended the throne (Chris Miller, 1994) (Belge, 1995). However, it is impolite to create a calendar that allows the user to reset the year to the beginning as this resetting questions the emperor's immortality. Even changing the date has hidden consequences for the unwary.

7.3 Weekends

Countries differ in the days the "weekend" or "Holy Day" is on. Most of the West have Saturday or Sunday as "weekends" because Sunday is the Christian's day of rest or Sabbath. In some Islamic countries, the "weekend" is Thursday and Friday, not Saturday and Sunday (Belge, 1995) as Friday is Prayer day. Some states in Malaysia have weekends on Saturday and Sunday whereas some other states, Thursday and Friday. In some Middle East countries, the "weekend" days are also not necessarily contiguous (Belge, 1995).

7.4 Day Turnovers

In most countries, date changes at midnight, e.g. US and UK. However, in Islamic countries, date changes at sunset (Belge, 1995).

7.5 Time

Designers should be aware of the 12 or 24 hour notation and valid separators used e.g. colon, period, space and no separators. Time are denoted as 8:32 pm. (US), 20:32 (Canada), 20,32,00 (Switzerland), 20.32 Uhr (Germany), KI 20.32 (Norway)(Chris Miller, 1994). Del Galdo (1990) suggests that time should cater for 12 and 24 hour clock notation and for international time, use of time zones to indicate the variation from the Greenwich Mean Time for example +12 hours (Hamilton, New Zealand). It would be useful to include the country in which the city is located to avoid confusion. For example, there are a number of Hamiltons in the world. Besides the one in New Zealand, there is one in Canada and another in Bermuda (Crystal, 1994) among others. Another feature that should be incorporated is the allowance for daylight savings, especially in the temperate countries which experience longer hours of daylight in summer.

7.6 Telephone Numbers

Different countries have telephone numbers of different lengths, formats and separators (space, hyphen, period, comma, square brackets, parenthesis, and plus sign). The telephone numbers may differ depending on whether the numbers are for national or international use(del Galdo, 1990). Del Galdo suggests a design for arbitrary formats for national numbers. For example, in New Zealand, local telephone numbers are written as 856 2889 or 856-2889 and (7) 856-2889 for international numbers, (7) being the area code for Hamilton. In the United States, the numbers are written as +1-212-626-0500, 1 is the country code and 212 the area code. In the UK, (01222) 584396 or 1222 584396, where (01222) is the area code for Cardiff region in UK. Abbreviations of telephone may also differ e.g. "Ph:" (NZ), "Phone:" (US), "Tel:" (Malaysia). The abbreviation for Facsimile, "Fax", seems consistent for UK, US and Europe.

7.7 Personal and Business Addresses

Address formats differ from country to country. House numbers are generally placed before street names. For example, in America but not in Latin America and Europe, the street name is written first (Chris Miller, 1994). The postcode may also appear either before or after the city, state or province, or suburb. Americans would write their house number before the street address, postcode after the state:

John Doe
456 Del Mar Avenue
Santa Barbara California **93140**
USA.

An example of an address in Europe, with the house number after the street and the postcode before the city:

European Service Center
Avenue Marcel Thiry **204**
1200 Brussels, Belgium.

7.8 Character Sets

"Every language has its own alphabet or script" (Chris Miller, 1994). English uses the 26 Roman characters while languages like Chinese requires practical modern dictionaries covering about 7000 characters. European languages contains some characters with diacritics

such as ë, â and á. Software in any language would require the support of the correct alphabet and other characters such as currency symbols.
Thus, 7-bit ASCII code set which supports 128 characters was invented to support American English, the language of the earliest programs. The 128 characters that ASCII supported include control characters, accents, control codes and the upper and lowercase of the Roman alphabet (Chris Miller, 1994). To include characters for European languages, diacritics and umlauts, the code set was extended to use 8-bit extended ASCII, which supports 256 characters. The Windows character set is based on ISO standard 8859-1 also known as ECMA-94 and Latin 1 (Hall, 1991). Latin 1 includes character sets that support the main languages of Western and Eastern Europe, such as Cyrillic, Greek and Turkish.
Languages with more than 256 characters such as Chinese, Japanese and Korean use double-byte, multi-byte or wide character sets. "A double-byte character uses 16 bits. A multi-byte can mix single and multi-byte characters. A wide character set typically contains 16- and 32-bit characters" (Chris Miller, 1994). Given the multitude of single- and multi byte-characters, a consortium developed a 16-bit character code standard known as Unicode. Unicode, also known as ISO 10646, is being promoted to replace ASCII and the other multi-byte character sets.

Unicode is a fixed-width, 16-bit character set which can support up to 65,536 characters. This support allows the inclusion the written text of almost all known modern languages as well as some ancient languages (Abramson, 1994). Windows NT has embraced Unicode while Apple intends to do so in future releases of its system software (Chris Miller, 1994). Any software developer intending to internationalise software products should ensure that support is provided for the character sets of their target community's languages. .Problems that may arise from use of 2-byte representation include font data storage and handling (Peterson, 1990) as well as a requirement for significant amounts of additional memory to run the eventual applications (Chris Miller, 1994).

7.8.1 Input and Output

Although input of Roman alphabets may be straightforward on the Qwerty keyboard, input problems arise in other languages. For example, there are more than 50,000 Hanzi. There exists more than 100 methods to enter these characters (Huang, 1985). Some methods to input Chinese, Japanese and Korean characters include the use of coding schemes, arrangement of symbols and phonetic schemes which are covered in (Huang, 1985), (Becker, 1985),(Walters, 1990) and (Hsu, 1991). Software developers entering the Chinese market would need to be aware that the input methods of the Windows 3.1 version for Taiwan include Chang Jei, Phonetic, Quick/simplified, Internal code, DA-YI, and Array (Chris Miller, 1994). Output of languages also differ with certain languages such as Arabic and Hanzi being more graphical than other languages such as English. A minimum resolution for screen display is required to ensure readability of the script (Sukaviriya and Moran, 1990). For example, in Walters (1990): "According to Huang and Huang, dot matrix techniques using a 16 x 16 matrix cannot produce (Chinese) characters of acceptable quality but storage of the more acceptance 24 x 24 matrix requires additional memory".

7.9 Collating/Sorting Order sequence

As different countries have different alphabets, so sorting or collating sequence also differ (Zobel-Pocock, 1990)(del Galdo, 1990)(Belge, 1995). Sorting sequences affect address lists and telephone directories. In Japan alone sorting sequences can be according to Ascending Lead Byte, Code Order, Pitch Casing, Pronunciation, and Stroke (Belge, 1995).
In certain cases, different rules govern the way the same sets of characters are sorted (del Galdo, 1990). Del Galdo (1990) describes sorting of letter with umlauts (diacritics) e.g. ä,ö,ü: in the German collating sequence, letters with umlauts are sorted as the same letters without umlauts; the Swede sorts the "ü" with "y" and places the other letters with diacritics at the end of the sequence. Walters (1990) reports that sorting sequence in the same language may be different, e.g. the German telephone directory is sorted slightly differently from a German dictionary.

7.10 Flow

Reading and writing directions differ from language to language. Chinese characters are written top to bottom with each new line of characters appearing to the left of the old line i.e. top to bottom, right to left. Arabic is written from right to left with each new line of script appearing below the previous line i.e. right to left, top to bottom. However, the numbers in Arabic are still written from the left to right (Tayli, 1990). Other languages follow the English language, left to right, top to bottom flow. Software developers must bear in mind the effects of language direction on user interface design, particularly with menus, dialogue boxes and error message windows.

7.11 Punctuation Marks

In Greek, interrogation is expressed not by the American English question mark but by what looks like a semicolon "; " (Chris Miller, 1994). Other punctuation marks include an inverted exclamation and question mark (Hall, 1992). In Hanzi, Greek and French "« »" are used as quotation marks (Chris Miller, 1994).

7.12 Translation

The translation of a user interface is not as easy as the translation of a book (Nielsen, 1990). There are many pitfalls to avoid when translating the user interface as "an interface introduces many additional subtleties such as computer-human interaction which can complicate a translation" (Russo and Boor, 1993). Translation problems are exacerbated when "the translator is unfamiliar with the application content or human factors principles" (Russo and Boor, 1993). An Indonesian exchange student, hired by a US company, translated "software" as "underwear". Sun Microsystems translated the word "menu" as "list of food items" in Cantonese (Russo and Boor, 1993). Examples of translation problem "page" in MS Word is translated to "side" Danish. In MacPaint, "Goodies" (English) to "godter" (Danish), a proper translation, but "godter" refers more to candies than to a variety of functionalities. Del Galdo (1990) also warns about the use of acronyms or abbreviations since the acronyms or abbreviations may have negative associations in other cultures. Software developers need to be aware of the following translation issues.

7.12.1 Words that don't exist

It is very common to find that computer-related words or terms do not exist in some languages. After all, the present computer vocabulary was invented when the need arose or when the item or concept was created. Terms like "Diskette", "Disk drive", "Zooming" and "Panning" do not have a direct translation in the Thai language (Sukaviriya and Moran, 1990) and probably in many other languages as well. Thus, new words are created and new phrases are invented to convey the meaning. However, even if new words are invented, the new word may not encapsulate the meaning intended. For example, in Microsoft Word, a direct translation of "ruler" in Thai refers to an object rather than a tool in Word (Sukaviriya and Moran, 1990). New phrases created also tend to be lengthy and sometimes users return to the original word. Nielsen (1990) found the use of "undo" command in English while all other commands were in Kanji because it was really impossible to translate "undo". Lengthy translations would also increase the space required for the commands in menus as well as dialogue boxes thus "cluttering" up the user interface.

Misinterpretation of translations also exists. For example the translation of "eject" in English to "aflever" which means "hand over" in Danish; users misinterpreted "aflever" as hand over the file from the computer to diskette (Nielsen, 1990). A better translation was found, "skub ud" which means "push out" (Nielsen, 1990). Another example is "Quit" in English translated to "slut" in Danish where "slut" means "the end"; naive novice users were scared of the word but not the more confident new users or the experienced users (Nielsen, 1990). The word "slut" was interpreted as too much of a termination, such as shut down of the computer (Nielsen, 1990).

In countries that need to create new words, it is important that there be an organisation that invents these new words. This organisation should publicise the words created to ensure there is a consistency in the terminology used.

7.12.2 Menu accelerators

According to Nielsen (1990), the problem users hate most is the inconsistency in control/function-keys/menu-accelerators. Nielsen illustrated the control-keys problem with this example: As the Danes used both English and where possible Danish software,

"MacDraw 1 retained control keys such as R for right-justify and L for left-justify (instead of H for "højre" and V for "venstre" thereby destroying their mnemonic value)". "Save" and "Exit" in French are "Sauver" and "Sortie" respectively. There is a confusing overlap if "*S" was also used for "Exit" (del Galdo, 1990).

7.12.3 **Names**

The names for everyday objects vary from country to country. Some common examples includes the trash can (US) vs. a dustbin (UK) or a rubbish bin (NZ); truck (US) vs. lorry (UK), a period (US) vs. a full-stop (UK). Some product names do not translate well. The Italian automobile manufacturer, Fiat, had to change the name of their car "Uno" because "Uno" means garbage in Finnish (Russo and Boor, 1993). Similarly, Rolls Royce had to alter the "Silver Mist" in Germany because mist means manure (Russo and Boor, 1993). The brand name of a New Zealand line of clothes, Tahi (one in Maori) needs to be changed if the manufacturers plan to export this line of clothes to Malaysia; "Tahi" in Malay means manure. Terms preferred by male software engineers in past such as kill, trash, abort should be avoided as some women find these terms "brutal" (Marcus, 1993).

7.12.4 **Translation of Text**

When translating text from English to another language, Chris Miller (1994) suggests leaving enough space the for text expansion. Chris Miller (1990) reports that Microsoft Software Development Kit recommends an allowance of 200 percent extra space from 1 to 10 English characters, 100 percent extra space for 11 to 20 English characters and 30 percent for 71 or more English characters. Hall (1991) describes Machine Readable Information (MRI) as "IBM's term for the language-sensitive information that passes between the user and the program". MRI can comprise commands, responses, messages, menus, dialogues, help animation audio, tutorials, clip art, and icon (Hall, 1990). Thus when translating MRI from one language to another, enough space must be provided for MRI expansion (Hall, 1990). The table below gives recommended minimum MRI Expansion space (Hall, 1990).

English text length	**Expansion space recommended**
0 to 10 characters	100 percent to 200 percent
11 to 20	100 percent to 200 percent
21 to 30	80 percent to 100 percent
31 to 40	60 percent to 80 percent
41 to 50	40 percent to 60 percent
51 to 70	31 percent to 40 percent
Over 70	30 percent

An extreme example of MRI expansion : "message pop-up" in German is; "Nachrichtenuberlagerrungsfenster" and in Portuguese "janela de sobrepopsiçao de mensagem, Danish pop-op-meddelelse" (Hall, 1991). For example, "Bildschirmeinstellungen" is the German translation of the Preferences menu item from the Windows menu (Chris Miller, 1991) Certain situations may arise that with the space allowed for dialogue boxes may become too "crowded". To solve this situation, boxes that contain text could be self sizing or movable (Chris Miller, 1994). Chris Miller (1994) warns "terminology consistency is crucial, since there can be hundreds of cross-references between the interface, the documentation, the text files and the filenames". One must be thus aware of the effects of translated items on its linked or related components.

7.12.5 **Documentation Translation**

Sprung (1990) reports that a marketing manager will often decide not to translate his product's user interface (documentation, on-line help) with the rationalisation: "they can

understand English". Companies do not to translate documentation because the translation of such documentation is effort-intensive and costly. However, documentation can serve as a marketing and public-relations tool, a means of showing that the company stands behind its product (Sprung, 1990).

7.13 Paper Size

Different countries also have different paper standards. Eg. American paper standards are legal, letter or ledger, elsewhere in the world are A3, A4 and A5 (Chris Miller, 1994). Japanese paper size are JIS-B4 and JIS-B5(Chris Miller, 1994). All these paper sizes need to be provided for WYSIWYG software (Belge, 1995).

7.14 Units of Measure

Although many countries have converted to the Metric System, some countries like US still use the English imperial units e.g. pounds, inches, gallons, miles, Fahrenheit, instead of kilograms, centimetres, litres, kilometres, Celsius or Centigrade. These units need to be reflected in software such word processors where units such as inches, centimetres, picas are used or in scientific software requiring correct quantitative measurements.

7.15 Numbers

Separators used in numbers (Arabic numerals) vary from culture to culture. In Europe, a comma "," is used as a decimal point and a period or full-stop "." is used to separate thousands. This conversion is the opposite of the UK and US convention where a full stop "." is used as a decimal indicator and a comma separates thousands. In Europe, the number 1,234,567.89 in US is represented as 1.234.567,89 (del Galdo, 1990). Valid separators include comma, period, space and apostrophe. Thus 3,912.45 can be 3.912,45 or 3 912,45. (Chris Miller, 1994). Positive and negative numbers can be indicate by either using the plus "+" and minus "-" prefix or suffix. Negative numbers can be denoted by enclosing the number in parenthesis (del Galdo, 1990). A billion in the US refers to one thousand million i.e. a one followed by nine zeros whereas in Latin America and Europe a billion is one million million, a one followed by 12 zeros (Chris Miller, 1994).

7.16 Currency

Different countries would have different currency symbol; British pound (£) or Japanese yen (¥), and the location of the symbol would also differ (Belge, 1995). In some countries, currency symbol may appear at the end of the number, for example, 4.567,89 DM in Germany or in Portugal, where the currency symbol is used as the decimal "symbol" 4.567$89Esc (Belge, 1995)(Esc is the abbreviation of the Portuguese escudo (World Fact Book, 1995)).

7.17 Visuals

In this section, images, icons and symbols are classified as visuals. Visuals used in the user interface impart certain information to the users. However, like other attributes described, visuals can be perceived differently by different cultures. An example of a visual that may not be recognisable is the trash can icon that appears in the Macintosh operating system. To British users, the trash can looks like a postal box (Russo and Boor, 1993).

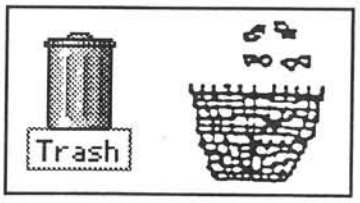


Figure 1 Icons for containers for refuse in the United States and Thailand (Sukaviriya and Moran, 1990).

The American trash can may not even be recognised by the Thais as in Thailand the "trash can" is actually a wicker basket (Sukaviriya and Moran, 1990).

Figure 2 Icons for container for mail in a number of nations (Apple Computer 1992).

The appearance and shape of mailboxes vary from country to country as shown in Figure 2 above (Apple Computer, 1992). Ironically, some urban American dwellers also do not recognise rural mail box icon (Russo and Boor, 1993). In the US and UK, red and blue ribbons are used awarded in recognition for performance or achievements in competitions, but ribbons as symbols of recognition for achievements might not hold true in other countries (del Galdo, 1990). Some visuals are recognisable in another culture but they convey a totally different meaning. In the United States, the owl is a symbol of knowledge but in Central America, the owl is a symbol of witchcraft and black magic (Apple Computer, 1992). A black cat is considered bad luck in US but good luck in the UK (del Galdo, 1990). The use of a cocktail glass icon in a calendar application to represent after work appointments may be appropriate in societies that socialise after work at the local bar or tavern, but the icon is inappropriate in countries where alcohol is not associated with social activities (del Galdo, 1990) such as Islamic countries. An image of the sun shining may conjure up feelings of pleasure and satisfaction to British and New Zealand residents, but to residents of arid countries, the same image may be associated with the threat of pain or death (Griffiths, 1994).

Figure 3 An ambiguous hand gesture (Pease, 1981).

Images of hand gestures such as the ring or OK gesture (Figure 3) may be understandable to most English speaking countries, but in France the same gesture means "zero", "nothing" or "worthless". In some Mediterranean countries the gesture implies a man is a homosexual (Pease, 1981). In the Christian world, symbols like the archetypal cross is associated with prohibition but not in Egypt (Russo and Boor, 1993). Crosses or checks that are commonly perceived as signs or disapproval and acceptance may not have universal meaning (Russo and Boor, 1993). The number 13 is considered unlucky in America as is number 4 in Japan, 7 in Ghana, Kenya and Singapore (Russo and Boor, 1993). To most Cantonese (a Chinese dialect), 8 is considered lucky as 8 is pronounced as "fatt" which means prosperity and good fortune.

Certain visuals may be comprehensible but unacceptable. Care needs to be taken when using religious symbols such as crosses and stars: that's the reason why we have a Red Cross in Western countries but a Red Crescent is adopted in the Islamic countries.

Figure 5 Universally recognised icons for Olympic Games events .

However, there are globally recognisable icons such as those depicting the various Olympic sports shown in Figure 5.

7.18 Colours

Colours hold different meanings for different cultures. For example, red could be associated with happiness or prosperity in China, anger or danger in Japan, life or creativity in India, death in China, aristocracy in France, and danger or stop in the United States (Russo and Boor, 1993). Russo and Moran (1990) also reports the association of green with go. A red "X" was found to be ineffective as a symbol for forbiddance (sic) in Egypt (Russo and Boor, 1993). In the UK, a red and blue ribbon signifies best or first and second in a given class respectively. In the US, the opposite is true, blue for first and red for second (del Galdo, 1990). Workers at a nuclear plant may have different colour associations compared to forest rangers because of the work settings.

7.19 Functionality

Developers must realise that the functionality embodied in software that works for one country may not work for another. Groupware is a useful and effective tool. However, groupware loses some of its value in use in Japan. According to Nakakoji (1994), Japanese businessmen often conduct brainstorming sessions in a social settings after work and they usually conduct negotiations before the meeting. It is "socially unacceptable to challenge your manager's idea in public". During the actual meeting time, employees just agree with any proposal put forward by their manager. Using groupware to record decisions during a meeting would be pointless.

Nielsen (1990) described another example of different cultures' preference of functionality. LYRE, a French hypertext system for teaching poetry, allows the students to see the poem from various "viewpoints". LYRE allows the teacher, but not the students, to add new viewpoints. This design is acceptable to Southern European tradition because if students were allowed to make changes, the teacher's authority would be undermined. However, people in Denmark where Scandinavian attitudes are prevalent would view the current design of LYRE as unsociably unacceptable as "it limits the students' potential for independent discovery" (Nielsen, 1990). Lotus research revealed that methods of amortisation in accounting varied between Germany and America. Thus, the financial functions in Lotus 1-2-3 had to incorporate these differences (Sprung, 1990).

7.20 Sound

Sound affects different nationalities differently. Most error messages are accompanied by a "beep" sound to draw the users' attention to the problem. In Japan, it is impolite to "Beep" as this calls attention to a possible error on the part of the user. Other users may hear the beep and become aware of the person's error, causing embarrassment to all involved (Belge, 1995).

Also, a pig may "oink" but to Hungarians a pig goes "gruff". Thus, if children were required to pair off animals with the sound the animals made, in a software originating from the UK, the children in Hungary may have problems trying to identify the correct "sound".

7.21 Metaphors

A typewriter metaphor would be quite useless for a word processor if you don't know about typewriters. " The typical Japanese has had little exposure to the typewriter because, before computers, documents were formatted using writing pads that typically used a 20-by-20 grid for each character. To judge the length of a document in Japanese, we count characters

instead of words. Japanese users have had considerable trouble understanding the concept so cursor movement embodied in a typical work processor application. Nakakoiji (1994) states in Usage Styles that it is "Advisable to get to know the culture if one is to get a feel ' for what metaphors will be acceptable.

8. **Critique and Further Work**

Internationalisation and localisation have gained more attention today when compared with the late 80s. This trend is evident given more papers being published since 1990. The book *Designing User Interfaces for International* edited by Jakob Nielsen, without a doubt, contains the earliest relevant discussion of internationalisation and localisation. More papers have been published since then, the creation and promotion of Unicode, and the setting up of mailing lists like INSOFT-L and SIGCHI's (Special Interest Group in Computer Human Interaction's) intercultural.CHI and INSOFT-L to encourage active discussions on these topics. Also, more books on the internationalisation and localisation have also appeared. Examples of such books include *Programming for the Whole World: A Guide to Internationalisation* by Sandra O'Donnel (1994), *Global Software: Developing Applications for the International Market* by David Taylor (1993) and *Software Internationalisation and Localisation: An Introduction* by Emmanuel Uren, Robert Howard and Tiziana Preinotti (1993). Furthermore, large software companies including Apple, Hewlett Packard, Sun Microsystems and Unix International also have books internationalising and localising their companies' systems.
Despite the recommendations of the more recent papers like Russo and Boor (1993),Chris Miller (1994), Belge (1995) most of the software existing now still emphasise character sets, time, date, collation formats. For example, Windows NT is "internationalised" and has a choice of language, keyboard, date, time, currency and number formats. Windows NT also supports Unicode. These factors such as character sets, time date, collation format can be termed as the "basics" or covert attributes which are straight forward and intuitive. On the other hand, with the overt attributes such as visuals, colours, sound; problems with these attributes can only be isolated through usability tests with target users. Covering the "basics" is not enough. Russo and Boor (1993) quoted Taylor: "properly localised software applications, just like properly localised automobiles, toasters, beverages and magazines, reflect the values, ethics, morals and language (or languages) of the nation in question". Thus, there is a need to look into the covert attributes to find out how these factors affect the user interface and the software as a whole, and how these factors can be integrated into the interface. For example, if the software supports ten locales, there should be at most ten interfaces with each locale's own cultural conventions.
There is also a need to develop techniques for incorporating the list of factors into the internationalised software development process (Russo and Boor, 1993). Another critical question is how to reduce the effort involved in localising software. A possible solution would be to separate the user interface from the functionality of the application software. With a separate user interface, changes to the user interface is less likely to affect the application code and vice versa. Thus, every locale can have its own user interface. User interface software tools can also speed up the development of the different user interfaces by converting overt and covert attributes to meet the requirements of a different locales.

8.1 **Globalising Software Products**

It can be argued that the whole endeavour of i18n is misinformed. A better solution for non-western nations seeking to exploit the potential of Information Technology (IT) would be to develop software for their own needs within their own the cultural context. It is clear for each culture to recapitulate the IT experiences of western nations represents a huge investment in reinvention of an industry with a wealth of experience in building software. Furthermore, the underlying issue of cross-cultural communication within geographical areas would not be addressed by the separate development of multiple software products. Somewhere between IT Apartheid (each country develops only indigenous software) and Globalisation (only software localisable to any culture is produced) is a balance that takes

into account the desire of people to both take advantage of IT and have those advantages provided in a culturally appropriate way.

9. **Conclusion**

Two processes are involved in developing internationalised software, internationalisation and localisation. There are a number of factors that need to be addressed before any software is internationalised or localised. These factors include identifying overt factors such as date, time collation, number and currency formats specific to the target cultures. Covert factors such as visuals, colours and metaphors also need taken into consideration. Even though there is an increase in publicity of internationalisation and localisation processes, research on these topics is still sparse. The technique of separating the functionality of software from the application interface has been demonstrated (Barbour, 1996) and is advocated because it appears to speed up the development of some localised software.

Successful i18n processes may not resolve issues of cultural independence or enhance the preservation of cultural variety that enriches human experience. I18n suits multi-national companies and ensures the hegemony of western nations over information technology. Cultures with little global influence will have to work very hard to develop information technology in the face of the current lead of western technologies. A first step in that process is the development of a consensus, in the global community, that software for multicultural (i18n) contexts reflects in design the separation of functionality from interface. This step would facilitate the growth of software industries in all cultures which wish to particpate in data exchange. The continued involvement of culturally specific providers would ensure that the issues of identified in i18n documents at United Nations level are accommodated for in localised software. Clearly, further and ongoing research is required to monitor and evaluate new algorithmic techniques and interface designs for specific cultures. Urgent work is also required in monitoring and evaluating the progress the IT industry makes towards enhancing international and intercultural communication supported by software.

References

1. Abramson D. 1994: Globalization of Windows. *BYTE.* Vol 19. No 6. p177-184.
2. Apple Computer. 1992: *Human Computer Interface Guidelines.* Addison-Wesley.
3. Aston M, Dolden B. 1990: Logiciel Sans Frontiers. *Computers in Education.* Vol 22. No 1/2.
4 Barbour R.H. 1996: Te Ripanga: the spreadsheet. *Proceedings of the 6th Biennial New Zealand Computers in Education Society.* Hamilton pp66-76.
5. Becker J D. 1985: Typing Chinese, Japanese and Korean. *IEEE.* January 1985. p27-34.
6. Belge M. 1995: The Next Step in Software Internationalisation. *interactions.* ACM Publishers. Vol 2. No 1. p21-25.
7. Bødker K, Pedersen J S. Workplace Cultures. 1991: *Looking at Artifacts, Symbols and Practices.* In Greenbaum J, Kung M (ed.). Design at Work: Cooperative Design of Computer Systems. Lawrence Erlbaum Associates. p121-136.
8. Borgman C L. 1992: Cultural Diversity in Interface Design. *SIGCHI Bulletin.* Vol. 24. No. 4. p31.
9. Chris Miller L. 1994: Transborders Tips and Traps. *BYTE.* Vol 19. No 6. p93-102.
10. Farid M. 1990: Software Engineering: Globalization and Localization. *IEEE (CDROM version).* p491-495. **
11. del Gardo, E. 1990: *Internationalization and Translation: Some Guidelines for the Design on Human Computer Interfaces.* In Jakob Nielsen (Ed.) Designing User Interfaces for International Use. Elsevier, New York. p1-10.
12. Gannon M J and Associates. 1994: *Understanding Global Cultures: Metaphorical Journeys through 17 countries.* Sage Publications.
13. Griffiths D , Heppell S, Millwood R, Mladenova G. 1994: Translating Software: What It Means and What It Costs for Small Cultures and Large Cultures. *Computers in Education.* Vol. 22. No. 1/2. P9-17.
14. Hall W S. 1991: Adapt Your program for Worldwide Use With Windows Internationalization Support. *Microsoft Systems Journal.* Nov-Dec 1991.p29-58.
15. Hall W S. 1994: Internationalisatioon in Windows NT, Part 1: Programming with Unicode. Microsoft Systems Journal" June, p57-71.
16 Hoecklin, L. 1995: *Managing Cultural Difference.. Strategies for Competitive Advantage.* Addison-Wesley Publishing Company, Wokingham, England.
17. HR Focus. 1993: More Companies Are Drawing Strength from Diversity. *HR Focus.* Vol. 70. No 6. p2.
18. Hsu S C. 1991: *A Flexible Chinese Character Input Scheme.* UIST '91 (Hilton Head, South Carolina 11-13 November). p195-200.
19. Huang J K. 1985: The Input and Output of Chinese and Japanese Characters. *IEEE.* January 1985. P18-24.
20. Kay R. 1994: Software Goes Global. *BYTE.* Vol. 19. No. 6. p90-91.
21. Marcus A. Human Communication Issues in Advanced User Interfaces. *Communications of the ACM.* April 1993. Vol. 4. No. 4. p101-109.
22. Microsoft Annual Report, 1994.
23. Nakakoji K. 1994; Crossing the Cultural Boundary. *BYTE.* Vol. 19. No. 6. p107-109.
24. Nielsen J. 1990:*Usability Testing of International Interfaces.* In Jakob Nielsen (Ed.) Designing User interfaces for International Use. Elsevier, New York. p39-44.

25. Nielsen J (Ed.). 1990: Designing User Interfaces for International Use. Elsevier, New York.

26. _ OECD. 1989: The Internationalisation of Software and Computer Services. OECD.

27. Pease A. 1981:*Body Language: How to Read Others' Thoughts by Their Gestures.* Camel Publishing Company, Avalon Beach, Australia.

28. Pugh J, LaLonde W. 1992: Internationalizing your application. *Journal of Object oriented Programming.* Vol. 4 No. 8. January 1992. p59-62.*

29. Russo P, Boor S. 1993: *How Fluent is your Interface? Designing for International Users.* INTERCHI '93 Conference on Human Factors in Computing Systems: INTERACT '93 and CHI'93. (Amsterdam, 24-29 April). ACM Press. p342-347.

30. Sprung R C. 1990: *Two faces of America: Polyglot and Tongue-tied.* In Jakob Nielsen (Ed.) Designing User interfaces for International Use. Elsevier, New York. p71-102.

31. Sukaviriya P, Moran L. 1990: *User Interface for Asia.* In Nielsen J (Ed.). Designing User Interfaces for International Use. Elsevier, New York. p189-218.

32. Tayli M, Al-Salamah A I. 1990: Building Bilingual Microcomputer Systems. *Communications of the ACM.* Vol. 33. No. 5. May 1990.

33. Walters R F. 1990: Design of a Bitmapped Multilingual Workstation. *IEEE.* February 1990.

34. Weitz, 1995: Posting in newsgroup comp.std.internat.

35. Webster HyperText Interface. 1995: http://c.gp.cs.cmu.edu:5103/prog/webster

36. World Fact Book. 1995: http://www.odci.gov/cia/publications/95fact/.

37. Zobel-Pocock R A.1990:*International User Interfaces.* In Jakob Nielsen (Ed.) Designing User interfaces for International Use. Elsevier, New York. p219-227.

Working Paper Series
ISSN 1170-487X

An MDL Estimate of the Significance of Rules

**by John G. Cleary,
Shane Legg, and
Ian H. Witten**

Working Paper 96/3

March 1996

Department of Computer Science
The University of Waikato
Private Bag 3105
Hamilton, New Zealand

An MDL Estimate of the Significance of Rules

John G. Cleary, Shane Legg, and Ian H. Witten

Department of Computer Science,
University of Waikato,
Private Bag 3015,
Hamilton, New Zealand
email: jcleary@cs.waikato.ac.nz
Submitted to ISIS'96 - Melbourne Australia.

Keywords: Machine learning, MDL, compression, evaluation of models, adaptive complexity
Area of Interest: Minimum Encoding Length Inference Methods

Abstract

This paper proposes a new method for measuring the performance of models—whether decision trees or sets of rules—inferred by machine learning methods. Inspired by the minimum description length (MDL) philosophy and theoretically rooted in information theory, the new method measures the complexity of test data with respect to the model. It has been evaluated on rule sets produced by several different machine learning schemes on a large number of standard data sets. When compared with the usual percentage correct measure, it is shown to agree with it in restricted cases. However, in other more general cases taken from real data sets—for example, when rule sets make multiple or no predictions—it disagrees substantially. It is argued that the MDL measure is more reasonable in these cases. and represents a better way of assessing the significance of a rule set's performance. The question of the complexity of the rule set itself is not addressed in the paper.

1. Introduction

So many different learning algorithms have been proposed that evaluating and comparing them presents a significant challenge. This paper restricts attention to "supervised" learning schemes which operate in the following standard way. They are given *training data* to be learned, which is divided into separate *instances*. Each instance comprises a fixed set of *attributes* and a *classification*. Each attribute has a range of values that it can assume—typically a finite enumerated set, an integer, or a real number. After learning from the training data, a scheme is presented with a set of *test data*. This is just like the original input data except that now the classification must be computed from the remaining attributes.

The result of learning is a model or "theory" of the data. Given an instance from the training or test data, the theory generates a classification which it attributes to that instance. This paper contributes to a common framework which is being developed for comparing and evaluating the various supervised learning schemes, the ultimate aim being to estimate how well a theory will perform when presented with a new set of test data. This aim is often diluted to that of providing an estimate of which rules will perform best on unseen data.

The easiest way to estimate performance is to observe how well the theory does on the original training set and assume that it will do just as well on test sets. This has the obvious disadvantage that a theory which merely replicates the data instances with one rule per instance does very well when evaluated against the training set but is likely to perform poorly on a new data set because it has not generalized the original data. Such a theory is said to be *overfitted* to the training set. This situation is normally avoided by measuring performance on a test data set which is different from the training data used to form the rules.

There are other problems with measuring a theory's performance. The obvious metric is the frequency of correct classifications. But even such a simple measure raises practical problems. Theories may generate no classification for a particular instance, or multiple classifications; or the instance may have missing data values. Different schemes resolve these ambiguities in different ways, making it difficult to replicate results reported in the literature and rendering comparisons between schemes dubious, if not meaningless. Another difficulty is encountered with highly-skewed data sets. Consider, for example, a situation in which one classification occurs 99% of the time. A trivial theory which always predicts that classification has a 99% success rate—apparently a very good result. The frequency of correct classifications is not necessarily a useful overall measure of performance.

The "minimum description length" (MDL) measure, proposed by Rissanen and others, is an alternative way of assessing inferred theories (Rissanen 1985, 1989). The MDL principle deems the best theory for a set of data to be one which requires the smallest amount of information to specify both the theory and the original data given the theory. Information is measured in bits, assuming an efficient coding scheme for theory and data. In order to apply the minimum description length principle to the output of machine learning schemes we have had to come up with general ways of computing complexity, which are described below.

This paper shows the performance of rule sets generated by different learning schemes can be measured in a uniform way, and describes a system for doing so. Whereas most previous

work (eg. Quinlan and Rivest, 1989; Wallace and Patrick, 1993) concentrates on the question of evaluating the complexity of theories, this paper examines how to evaluate the complexity of a data set given a theory. This is normally dismissed as a fairly trivial part of the problem, but in fact we argue that it involves some crucial issues that are not generally considered. We back up this claim by examining the results of several commonly-used machine learning schemes on actual datasets—both the standard datasets and ones encountered in a project on the agricultural applications of machine learning (Garner *et al.*, 1995)—and showing that the new metric does capture subtleties that are hidden by the usual percent correct measure.

The next section discusses how the notion of complexity can be quantified, and introduces the key distinction between static and adaptive estimation methods. It also briefly discusses measurement of the complexity of theories, which is not further addressed in this paper. Following that we introduce two different ways of measuring the complexity of data with respect to a theory. One takes account of the frequency of different items, and the other does not. Next we use these metrics to examine the significance of actual rule sets generated by machine learning schemes on a wide variety of different datasets. In a certain special case it is possible to relate the new measures to the standard "percentage correct" measure theoretically; this confirms the correctness of the results. Finally we summarize our findings and draw conclusions from the work.

2. Complexity

According to the minimum description length principle, the "best" theory is the one that represents the data in the simplest, most economical, way. The complexity of a data set relative to a theory is defined as the length of the shortest program necessary to reconstruct it from that theory. The total information required to represent the data is the amount of information necessary to specify the theory, plus the information necessary to specify the data given the theory. This can be written

$$\mathrm{I}(D) = \mathrm{I}(T) + \mathrm{I}(D|T)$$

where $\mathrm{I}(X)$ is the information necessary to specify X, and $\mathrm{I}(X|Y)$ the amount of information necessary to specify X when Y is known. This formulation encapsulates a trade off between a complex, over-fitted, theory where $\mathrm{I}(T)$ is large and $\mathrm{I}(D|T)$ small, and a simple, over-general, one where $\mathrm{I}(T)$ is small and $\mathrm{I}(D|T)$ large.

Our purpose here is to compare different theories on the same data. The measure above includes information that is irrelevant for this purpose. The regenerated data constitutes not only the classification that is assigned by the theory, but also all other attributes of the test instances. One might view the attributes being presented as a question posed to the theory, and the returned classification as the answer to this question. In these terms, the complexity of the questions is being included along with that of the answers. It can be eliminated by splitting the complexity measure into the information required to specify the attributes in the test instances, and that required to specify the class given both the theory and the attributes. That is,

$$\mathrm{I}(D|T) = \mathrm{I}(Q) + \mathrm{I}(D|T,Q),$$

where $\mathrm{I}(Q)$ is the complexity of the attributes (the questions) and $\mathrm{I}(D|T,Q)$ is the complexity of the data given both the theory and the questions. The total complexity of interest is then:

$$\mathrm{I}(D) = \mathrm{I}(T) + \mathrm{I}(Q) + \mathrm{I}(D|T,Q).$$

$\mathrm{I}(Q)$ does not depend on the theory T and can be suppressed for comparison purposes. To compare different theories it is sufficient to compute the sum $\mathrm{I}(T) + \mathrm{I}(D|T,Q)$, in other words, the complexity of the theory plus the complexity of the answers. We show below one way of estimating these two quantities; first, however, we discuss the general philosophy that underlies our approach.

Static versus adaptive complexity estimation

The complexity $\mathrm{I}(T)$ of a theory is defined as the smallest program needed to generate it, and the complexity $\mathrm{I}(D|T)$ of a set of data relative to a theory is the smallest program needed to generate that data, given the theory. In the theory of complexity, a standard machine is assumed on which these programs are to be executed. The particular machine chosen is deemed to be unimportant because in the limit for large theories and data sets it makes no difference—any machine can be interpreted on any other using a program of fixed size. Even if a particular machine is assumed, complexity theory does not prescribe how to compute these measures: indeed, the question of whether any particular program is in fact the smallest is undecidable in general. This makes it necessary

to devise a methodology for estimating the complexities.

Let us consider, by way of example, the complexity of a theory T. One way to estimate I(T) is to specify T by a sequence of decisions that build the theory up. Representing T as a sequence of binary digits, a decision would be made for each digit about whether it was a 0 or 1. If the probability of each decision's outcome were known, a complexity equal to the entropy of the outcomes could be attained. For example, if it were known *a priori* that zeros would occur three times as often as ones in the theory's binary representation, the sequence could be encoded at $-0.25 \log_2 0.25 - 0.75 \log_2 0.25 = 0.81$ bits per binary digit, while if they occurred equally often the theory could be encoded no more efficiently than one bit per binary digit. Using the technique of arithmetic coding, this entropy can be approximated arbitrarily closely (Witten *et al.*, 1987).

Summing the entropy of each outcome only gives a good estimate of the entropy of the sequence if the individual bits are chosen independently. In practice, this will not be the case and a more complex model would be used to encode the theory. However, the real disadvantage of this coding regime is much more serious. If the *a priori* probabilities are not chosen correctly—that is, if they do not reflect the frequencies with which elements of the sequence actually occur—the coding will be inefficient. And the inefficiency will take the form of a constant overhead *per element of the sequence*, so that there is no limit to the inefficiency with which large theories are coded. Contrast this with the choice of standard machine in the alternative, program-oriented, view of complexity, which can only affect the complexity measure by an additive constant.

The problem with the method we have described is that it uses a *static* model to encode the theory, and any deficiency in the model results in inefficient coding of each element of the sequence. An alternative is to use *adaptive* coding techniques instead. The idea is that, as a sequence is processed, statistics are gathered about the decisions that are made. Each prediction about what will occur next is informed by the accumulated statistics so far. Thus if it is thought *a priori* that zeros will occur three times as often as ones, but this turns out to be incorrect, the adaptive model will not suffer the same penalty as a static one.

Adaptive techniques have been used very successfully in many applications. For lossless compression of text, all the best methods are adaptive (Bell *et al.*, 1990). An important property of adaptive techniques is that they are very robust. In the limit of long messages, they can guarantee to achieve the entropy of the sequence plus a term that is logarithmic in message size (Bell *et al.*, 1990; Cleary and Witten, 1984). In contrast, the use of predetermined statistics can cause the data to be expanded by an unbounded constant factor. For any particular problem, adaptive algorithms can be constructed in many different ways for computing the complexity, depending on what statistics are accumulated and how they are used to estimate predictions. The main advantages of adaptive techniques are their robustness and flexibility: these allow them to be applied easily to different problems.

Theory Complexity

As part of our overall system an adaptive modeller to compute the complexity of general Prolog programs was developed. However, the output from supervised learning schemes are very stereotyped: for example, many of them can be represented by decision trees. The result is that the general complexities computed are significantly greater than the apparent complexity of the theory—so much so that the resultant values for I(T) appears not to be useful for comparing different theories. We are currently re-implementing a system that recognizes special subtypes of theories (decision trees, rules using a single attribute, sets of rules, trees of rules, and so on) and computes the adaptive complexity from these.

3 Data Complexity

We now describe a sequence of more refined adaptive algorithms for computing I($D|T,Q$). The classification of the members of the data set D are each specified in turn. For each one, the theory generates a set of predictions (say m of them) out of a universe of possible classifications (say n of them); our task is to encode which classification actually occurs.

This can be specified in two steps. The first determines whether or not the actual class appears as one of the predictions. If so, it is only necessary to specify which of the m predictions it is. The theory provides no way to distinguish the different predictions, so this requires $\log_2 m$ bits. If the actual class is not one of the predictions, $\log_2(n-m)$ bits are

needed to specify it. It remains only to determine how to specify whether or not the actual class is in the prediction set. A simple adaptive approach is to count how often the actual class has been predicted (say c_p) and how often it was not in the prediction set (say c_n). The probability of the class being predicted can be estimated as $p = (c_p+1)/(c_p+c_n+2)$: this is Laplace's law of succession (the 1 and 2 in numerator and denominator allow for the case where c_p or c_n is 0 because the class has not yet occurred). The number of bits needed for encoding is $-\log_2 p$.

This measure yields some qualitatively correct results. For example, suppose the theory is perfectly correct and makes a single correct prediction for each instance in the test data. Then p will approach 1 (and $-\log_2 p$ will approach 0). Because there is only a single prediction, no bits are needed to specify the class ($\log_2 m = \log_2 1 = 0$), and so in the limit the data requires zero bits per instance to specify it. Now consider a situation where the theory always specifies two classifications. On average, one bit will be required per instance: zero bits for selecting whether there will be a correct prediction, and one to identify it from two predictions.

Nevertheless, the estimator is unsatisfactory for three reasons. If there are no predictions (m=0), the class cannot possibly be in the prediction set and yet it is predicted with non-zero probability—thus wasting output bits. Similarly, if m=n it is impossible for the class to occur, yet it is predicted with non-zero probability. A third, more subtle, problem arises when considering the maximum average complexity per instance. Ignoring the predictions and specifying the class requires $\log_2 n$ bits per instance, and adaptive coding should never do worse than this. However, suppose the behavior of the predictions alternates. First, n–1 classes are predicted and the actual class is among them. Then just one class is predicted, which is incorrect. If these two cases continue to alternate, the probability of a class being predicted correctly will tend to 1/2. In both cases the actual class is selected from among n–1 alternatives, so the complexity per instance is $1 + \log_2(n-1)$ bits, which exceeds $\log_2 n$ whenever n>2. In this case, the adaptive technique is strictly worse than ignoring the theory altogether.

Constant Weighted Complexity

These problems can be solved with a more sophisticated probability estimation technique. This uses two weights α and β. The former applies to the case when some prediction is correct, the latter when they are all incorrect. The probability p is computed as $p = \alpha m/(\alpha m+\beta(n-m))$. This deals correctly with the cases where nothing is predicted (m=0) because then p=0 and no bits are wasted predicting that the class is in the prediction set. Similarly, when all possible classes are predicted (m=n), p=1.

It is not obvious how best to update α and β as evaluation proceeds. The procedure used is that as each instance is evaluated, α is replaced by $\alpha + 1/m$ if the actual class occurred amongst the m predictions, otherwise β is replaced by $\beta + 1/(n-m)$. Note that division by zero can never occur, for if m=0 the actual class can never be predicted so α will not be incremented; similarly if m=n the actual class must be in the prediction set so β will not be incremented. This procedure passes two important tests for reasonableness.

1 In the special case where the theory always generates the same number of predictions (m is constant), then the probability p is, in the limit, the probability that the theory will make a correct prediction. Moreover, the term αm counts the number of times that the theory predicted correctly, while $\beta(n-m)$ counts of the number of times it predicted incorrectly.

2 In the limit, the complexity per instance is less then $\log_2 n$ bits per instance no matter what sequence of correct and incorrect predictions are made by the theory.

These two properties hold no matter what initial values are used for α and β. Choosing the "best" value is one form of the zero-frequency problem (Witten and Bell, 1991) which has no optimum solution unless a prior for α and β is assumed. The current scheme initializes α and β to $1/n$, the smallest amount by which either can be incremented. This complexity calculation is summarized in Table 1.

The complexity measure just described will be referred to as the "constant weight" complexity of the data and written $I_c(D|T,Q)$. However, there is a problem that has arisen when dealing with data where one class has a

frequency close to 1—which, in our experience, often occurs in practical applications of machine learning. The measure takes no account of frequency information about the individual classes themselves. A simple system that codes the data adaptively by accumulating the frequency of each class (without using the theory predictions) will often compress the data more than by using the theory as above. This suggests that both the frequency of the classes and the theory predictions should be taken into account.

Frequency Weighted complexity

First consider how to adaptively code the data using just the frequency of the classes (but not the theory). Let the number of times that class i has occurred so far during the scan of the data be c_i. Let the total number of instances be

$$N = \sum_i c_i .$$

The probability of the i'th class is then estimated as $p_i = (c_i + 1)/(N + n)$. The addition of the 1 and n deals with the zero-frequency case when a class has never been seen before but must be predicted with a non-zero probability. In the limit, the complexity of this scheme will be at worst $\log_2 n$ bits per instance (this worst case will occur when all the classes have equal probability $1/n$). The complexity computed in this way is independent of the theory and is written $I_f(D|Q)$.

To incorporate both predictions and frequency information, the two-step procedure described earlier is used—that is, weights α and β are used to predict whether the actual class is in the predicted set or not, whereupon the class is predicted from within the appropriate set. Let R be the set of classes predicted by the theory. The probability that the actual class is in the predicted set is estimated as:

$$p = \frac{r\alpha}{r\alpha + (1-r)\beta} .$$

Here, r is the probability that a class will be in the predicted set,

$$r = \sum_{i \in R} p_i .$$

Here α and β again function as weighting factors, estimating the probability that the theory predictions will be correct. As each instance is evaluated, α is replaced by $\alpha + p_i / r$ if the actual class occurred amongst the m predictions, otherwise β is replaced by $\beta + p_i / (1-r)$.

This new form of complexity is referred to as the "frequency weighted" complexity and written $I_f(D|T,Q)$. The complexity calculation is summarized in Table 1. It gives reasonable results in special cases. For example, if all the classes occur with equal frequency these formulae reduce to just those used above for the constant weight complexity. Also, in the limit, it cannot exceed the simple frequency based complexity which ignores the theory.

Comparison With Other MDL Techniques

Muggleton (1992) proposed using the MDL principle for evaluating learned rules, following closely the seminal ideas proposed by Rissanen (1985).

The major difference from our work is that Muggleton used proof complexity when computing the complexity of the data. Proof complexity encodes the result of executing a theory in terms of the sequence of choices made by a Prolog interpreter while generating the actual class. In contrast, answer complexity, as described in this paper, first collects all the answers and then determines the complexity of the actual class from the answer set.

As has been noted in (Kovacic,1994; Conklin and Witten 1994; and Srinivasan *et al*, 1992) proof complexity can be a very inefficient way of specifying the class: the average complexity of a proof is always larger than the answer complexity and can be unboundedly larger. Think of the proof tree (ie. the SLD tree): every answer must appear once on some leaf, but may appear many times on different leaves, and there may be failure or infinite branches that contain no answer. To specify the actual answer in a proof tree must always involve more choices on average than to specify it in the set of answers.

4. Significance of Theories

We use the complexity measures to evaluate theories, in other words to assess whether a theory is "significant" or not, or to answer the question "does this theory provide significantly better predictions than the null hypothesis".

Our procedure is to take a learning algorithm and apply it to a training set, generating a theory. The theory is then applied to a different test set and the various complexity measures computed. $I_f(D|Q)$ is a

measure of the data complexity without using any theory and can be taken as the complexity of the data with respect to the null hypothesis. $I_f(D|T,Q)$ is a measure of the complexity of the data with respect to the generated theory and if the theory has actually captured some regularity or information from the data should be less than $I_f(D|Q)$. The difference $S_f=I_f(D|Q)-I_f(D|T,Q)$ indicates the extent to which the theory has departed from the null hypothesis and can be used as a measure of the significance of the theory. The larger the difference, the more likely it is that the theory has captured some regularity in the data.

In order to fully determine whether the theory provides a worthwhile improvement over the null hypothesis, it would be necessary to take into account the complexity of the theory itself, and see whether it could be represented in less than S_f bits. This is beyond the scope of the present paper.

Experiments

To assess this significance measure a number of experiments were run. In each one a data set was split randomly into two equal training and test subsets 25 times. Equal splits were made because the complexity measures are not linear on small numbers of instances: keeping the test and training sets equal allows easier comparison of the complexities on the two sets. Several different learning schemes were applied to these splits:

1-R (Holte, 1993)
C4.5 (Quinlan, 1986, 1993), generating
  a pruned decision tree
  an unpruned decision tree
  decision rules;
FOIL (Quinlan, 1990)
INDUCT (Gaines, 1991), generating
  ripple-down rules
  DNF rules.

In each case, S_f was computed. The data sets used were those used by Holte (1993) and a number of agricultural data sets supplied by the Weka project (Garner *et al.*, 1995).

Relating complexity to percent correct

To assess the reasonableness of S_f as a significance measure we relate it to the standard percentage correct measure. The usual way that the latter is used is to compare the percentage of correct predictions made by the theory against the percentage correct if just the most likely class had been predicted. At first sight it is not clear what relationship there might be between the two measures. However, in some special cases it is possible to analytically compute the expected S_f as a function of the percentage correct.

Consider a data set where there are exactly two classes, and assume that the theory always predicts exactly one class. Let the number of instances of the most likely class be i and the corresponding probability be $p = i / N$. Let the number of instances correctly predicted by the theory be j, with corresponding probability $q = j / N$—in our experimental design q is the percentage correct measure. We now compute S_f as a function of p and q.

It is easily shown that

$$I_f(D|Q) = \log_2 \frac{(N+1)!}{i!(N-i)!}$$

and

$$I_f(D|T,Q) = \log_2 \frac{(N+1)!}{j!(N-j)!}.$$

Applying Stirling's approximation to these expressions, simplifying, and neglecting terms of $O\left(N^{-1}\right)$ and less gives:

$$S_f = N(E(p)-E(q)) - \frac{1}{2}\left[\log_2 \frac{p}{q} + \log_2 \frac{1-p}{1-q}\right]$$

where

$$E(p) = -p\log_2 p - (1-p)\log_2(1-p).$$

Note that S_f is 0 whenever $p=q$. Also the second term, which is constant in N, is small on most actual examples.

$E(q)$ is decreasing function of q (for $q \geq 1/2$) so S_f increases monotonically as q increases beyond the default probability p. Thus the percentage correct and S_f measures agree on the ordering of theories in such cases.

Results

Figure 1a shows the result of one experiment. S_f and percentage correct are plotted together for the dataset CH taken from (Holte, 1993), in which the class is binary. The learning schemes C4.5-pruned, C4.5-unpruned, C4-rule and Induct-ripple all produce just one prediction for each instance in this case. The theoretical value for S_f derived above is also shown and is seen to lie very close to the data points for these schemes—the expanded view in Figure 1b emphasizes this point.

The schemes FOIL and Induct-DNF depart markedly from single predictions. The former has between 40% – 80% multiply classified instances and approximately 10% unclassified instances; the latter has between 3% – 8% multiply classified instances and 0% – 1% unclassified instances. Their points are scattered well off the theoretical curve.

Interestingly, while the percentage correct figures for the Induct-ripple scheme lie clearly outside the range for the majority of the schemes, the value for S_f is not correspondingly reduced. In the presence of multiple and unclassified instances the predictions of percentage correct and S_f may depart markedly.

This point is emphasized by the results in Figure 2. This shows the plot of percentage correct versus S_f on dataset "wcr", one dataset from an agricultural problem involving cow-culling (McQueen *et al.*, 1995). The data consists of some 2000 instances of data about a single farmer's cows over a period of five years. As well as data about the cows milk production, breeding and other attributes, it also records which cows were culled from the herd and which died from other causes. The goal is to determine what rules the farmer used for deciding which cows to cull.

There are several problems with this data. The frequency of culling is very low—only 6.1% of the cows were culled. The frequency of cows which died was also low—1.6%. These instances were essentially random with respect to the given data. In work using the standard percentage correct measure to evaluate the results, we have found that standard learning schemes seem to be unable to develop any significant theories to explain the data.

Although strictly speaking the conditions for the analytic curve derived above do not hold in this case, for there are more than two classes, nevertheless the curve has been plotted using the probability of the most frequent class for p. Some of the schemes (C4.5-rule, and C4.5-pruned) lie close to it, while others depart markedly.

Of all the schemes only Induct-DNF has a consistently positive S_f, and thus seems to have captured some of the structure of the problem. Closer investigation shows that it has a high level of multiply classified instances (60% - 90%). Even more remarkably, this is accompanied by a very low percentage correct score. Using a traditional analysis the fact that these rules had captured a significant part of the data's structure would have been missed entirely.

Preliminary results from other agricultural data sets highlight the importance of rules which make multiple (or no predictions) and consequently of having an evaluation methodology which takes these into account. For example, in a study on the grading of venison the class is essentially a discretisation of a continuous variable. Thus it is often difficult to separate which of two adjacent grades should be chosen. The most successful rules seem to predict two possible classes in these cases.

In another set of experiments, the 16 data sets used by Holte (1993) were tested against seven different learning schemes, including the 1R and C4.5-pruned schemes that Holte used. The values for percentage correct and S_f averaged over the 25 random splits are listed in Table 2, which also includes the frequency of the default class for each data set (the percentage correct values are all shown in bold face). The Sf values generally corroborate Holte's conclusions about the relative performance of 1R and C4.5. That is, 1R generally does well compared to C4.5 except on two data sets CH and SO. It is notable, however, that on the data set GL S_f indicates 1R performing worse than the percentage values would otherwise indicate. There is little difference between pruned and unpruned C4.5 (this is to be expected as generally both these schemes generate rules which make single predictions and so the percentage correct and S_f should give comparable orderings). The major differences between the two measures occur for FOIL on G2 and LY and Induct-DNF on LY. In each of these cases, the S_f value indicates higher significance than percentage correct. This seems to be a consequence of these schemes generating rules with multiple and unclassified instances which may still perform well in these cases.

6. Summary and Conclusions

An adaptive MDL measure of the performance of rule sets has been proposed. It has been argued that it:

- provides an unbiased measure of different learning schemes performance by evaluating theories independent of the original software;
- is theoretically well founded in complexity theory;

- deals in a principled way with theories that sometimes make multiple or no predictions;
- provides a measure of the significance of a theory
- agrees in simple cases with the "percentage correct" measure;
- deals in a satisfactory way with data where one class has a probability close to 1.0.

It has been applied to several large agricultural data sets, and found to provide an intuitively reasonable account of the performance of different rule sets on this data. The work underlines the fact that it is important for rule sets to be able to make ambiguous predictions in some cases: evaluation metrics must deal with this gracefully and correctly.

The work would be further enhanced by a complexity measure for theories, in order to detect overfitting and provide estimates of performance on unseen test data. An attempt to compute theory complexity by using general Prolog rules failed in practice because actual rules tend to be very specialized. In future work we intend to implement a more specialized measure of rule complexity, assess its ability to measure overfitting, and use it to predict performance on unseen data.

Acknowledgments

We would like to thank all members of the WEKA team, particularly Stephen Garner and Luke Guyton, for help in implementation and generating the results.

References

Bell, T.C., Cleary, J.G. & Witten, I.H. (1990) *Text compression.* Prentice Hall, Englewood Cliffs, NJ.

Cleary, J.G. and Witten, I.H. (1984) "A comparison of enumerative and adaptive codes." *IEEE Trans Information Theory IT-30*(2): 306–315; March.

Conklin D., and Witten I.H(1994) "*Complexity-Based Induction*," Machine Learning, vol. 16.

Gaines, B.R. (1991) "The tradeoff between knowledge and data in knowledge acquisition." In *Knowledge discovery in databases*, edited by G. Piatetsky-Shapiro and W.J. Frawley. AAAI Press, Menlo Park, CA, pp. 491–505.

Garner, S.R., Cunningham, S.J., Holmes, G., Nevill-Manning, C.G. and Witten, I.H. (1995) "Machine learning in practice: experience with agricultural databases." *Workshop on Applications of Machine Learning*, Tahoe City, California; July.

Holte, R.C. (1993) "Very simple classification rules perform well on most commonly-used datasets." *Machine Learning 11*, 63–91.

Kovacic, M.(1994) "*MDL-Heuristics in ILP Revisited*," Workshop on Applications of Descriptional Complexity to Inductive, Statistical, and Visual Inference.

McQueen, R.J., Garner, S.R., Nevill-Manning, C.G. and Witten, I.H. (1995) "Applying machine learning to agricultural data." *J Computing and Electronics in Agriculture*, *12*(4), 275–293.

Muggleton, S., Srinivasan, A., and Bain, M. (1992) "Compression, Significance and Accuracy" Proc Machine Learning Conference, pp. 338–347.

Quinlan, J.R. (1986) "Induction of decision trees." *Machine Learning 5*, 239–266.

Quinlan, J.R. and Rivest, R.L. (1989) "Inferring decision trees using the minimum description length principle." *Information and Computation 80*: 227–248.

Quinlan, J.R. (1990) "Learning logical definitions from relations." *Machine Learning 1*, 81–106.

Quinlan, J.R. (1993) *C4.5: programs for machine learning*. San Mateo, CA.

Rissanen, J. (1985) "Minimum description length principle," *Encyclopedia of the Statistical Sciences, Vol. 5*, edited by S. Kotz and N.L. Johnson. Wiley, NY, 523–527.

Rissanen, J. (1989) *Stochastic Complexity in Statistical Inquiry*. World Scientific, Singapore.

Srinivasan A., Muggleton S., and Bain M.(1992) "*Distinguishing Exceptions from Noise in Non-Monotonic Learning*," Proc. of the 2nd Int. Workshop on Inductive Logic Programming, Tokyo.

Wallace, C.S. and Patrick, J.D. (1993) "Coming decision trees." *Machine Learning 11*: 7–22.

Witten, I.H., Neal R., and Cleary, J.G. (1987) "Arithmetic coding for data compression." *Communications of the Association for Computing Machinery 30* (6) 520–540, June.

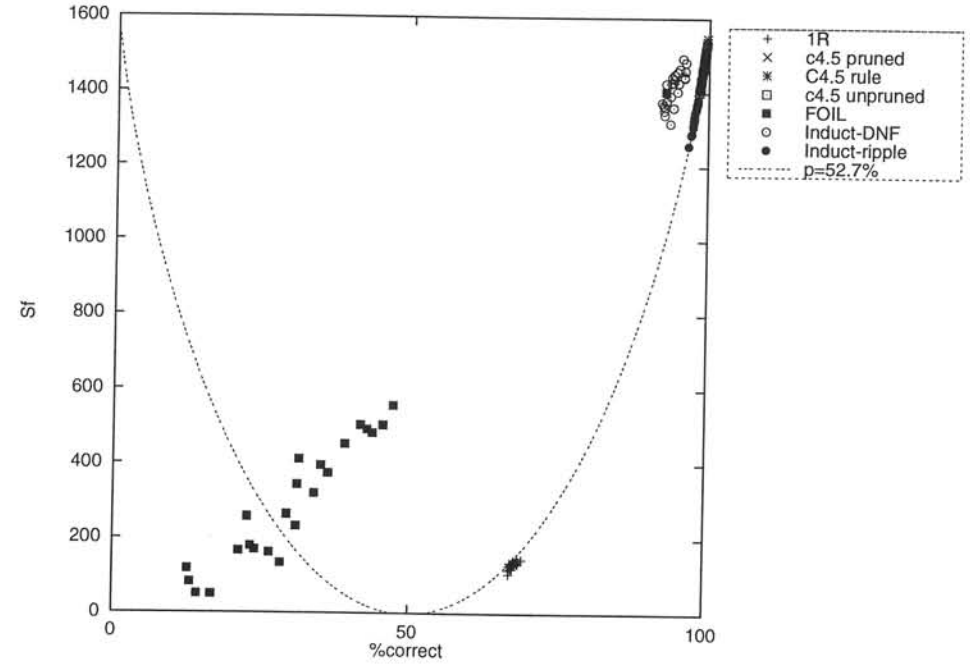
Witten, I.H. and Bell, T.C. (1991) "The zero-frequency problem: estimating the probabilities of novel events in adaptive text compression." *IEEE Trans Info Theory 37*(4): 1085–1094.

	Probability of Actual Class	Updates
$I_c(D\|Q)$	$1/n$	
$I_c(D\|T,Q)$ if in predicted set R if not in R Initialization	$\frac{\alpha m}{\alpha m+\beta(n-m)}\times\frac{1}{m}$ $\frac{\beta(n-m)}{\alpha m+\beta(n-m)}\times\frac{1}{n-m}$ $\alpha\leftarrow 1/n \quad \beta\leftarrow 1/n$	$\alpha\leftarrow\alpha+1/m$ $\beta\leftarrow\beta+1/n-m$
$I_f(D\|Q)$	$c_i+1/N+n$	
$I_f(D\|T,Q)$ if in predicted set R if not in R Initialization	$\frac{\alpha r}{\alpha r+\beta(1-r)}\times\frac{p_i}{r}$ $\frac{\beta(1-r)}{\alpha r+\beta(1-r)}\times\frac{p_i}{1-r}$ $\alpha\leftarrow 1/n \quad \beta\leftarrow 1/n$	$\alpha\leftarrow\alpha+p_i/r$ $\beta\leftarrow\beta+\frac{p_i}{1-r}$
Where	n is the number of classes c_i is the number of occurrences of class i R is the set of classes predicted by the rule set $m=\|R\|$ $N=\sum c_i$ $p_i=c_i+1/N+n$ $r=\sum_{i\in R} p_i$	

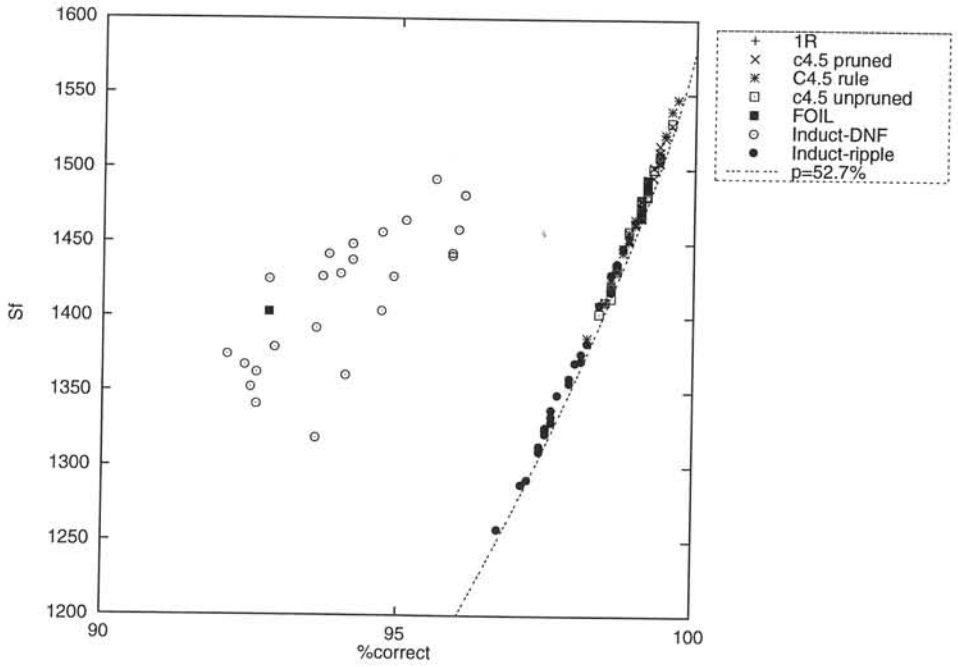
Table 1. Algorithms for Computing $I(D|T,Q)$.

Scheme	BC	CH	G2	GL	HD	HE	HO	HY	IR	LA	LY	MU	SE	SO	V1	VO
Baseline accuracy	**70.3**	**52.7**	**53.4**	**35.5**	**54.5**	**79.4**	**63.0**	**95.2**	**33.3**	**64.9**	**54.7**	**51.8**	**90.7**	**36.2**	**61.4**	**61.4**
1R	**68.4**	**67.9**	**72.0**	**49.6**	**70.5**	**71.4**	**81.1**	**89.5**	**93.1**	**45.6**	**71.4**	**98.5**	**64.5**	**78.5**	**85.7**	**93.7**
	-4.1	132.2	9.0	14.3	23.6	1.3	50.6	181.3	85.3	-0.8	13.9	3603.9	119.2	19.4	91.0	153.4
C4.5 pruned	**70.6**	**99.2**	**74.2**	**63.7**	**73.0**	**69.7**	**78.2**	**91.0**	**94.1**	**65.7**	**74.8**	**100.0**	**75.4**	**95.8**	**84.4**	**92.9**
	-3.5	1484.0	12.0	56.0	31.0	1.7	47.8	251.4	89.2	1.9	18.8	4053.4	246.6	36.7	99.3	149.9
C4.5 rule	**68.6**	**99.0**	**75.8**	**62.6**	**75.8**	**81.0**	**81.2**	**99.1**	**94.1**	**82.4**	**74.4**	**99.9**	**97.4**	**95.3**	**89.2**	**94.7**
	-3.2	1469.9	14.4	48.4	35.5	0.5	45.2	318.7	89.5	6.3	17.4	3994.3	433.5	36.0	99.3	142.6
C4.5 unpruned	**66.9**	**99.0**	**74.4**	**63.3**	**72.3**	**60.8**	**44.7**	**88.1**	**94.1**	**53.3**	**74.4**	**100.0**	**73.8**	**95.8**	**81.1**	**91.1**
	-1.4	1468.9	12.4	54.6	29.5	1.8	6.4	222.4	89.4	0.3	18.2	4053.4	206.6	36.7	88.4	142.0
FOIL	**54.3**	**31.5**	**63.5**	**47.5**	**64.1**	**66.9**	**60.9**	**97.9**	**90.9**	**64.9**	**63.3**	**99.6**	**95.0**	**95.5**	**76.9**	**87.6**
	2.7	316.6	20.8	62.0	47.0	6.1	39.0	298.7	88.9	6.4	26.0	4037.4	398.7	39.8	94.4	142.2
Induct-DNF	**51.1**	**94.1**	**63.5**	**43.2**	**60.5**	**67.4**	**69.8**	**95.1**	**84.6**	**69.6**	**65.8**	**99.9**	**84.2**	**95.7**	**81.8**	**88.4**
	-1.6	1413.6	16.0	57.8	39.7	4.7	55.2	312.4	86.2	5.0	29.3	4051.0	364.0	39.9	114.9	152.2
Induct-ripple	**65.1**	**97.8**	**73.2**	**61.7**	**70.7**	**78.1**	**80.0**	**98.6**	**93.8**	**80.0**	**76.0**	**100.0**	**96.3**	**97.1**	**87.9**	**94.0**
	-1.3	1350.8	11.6	46.7	23.0	0.2	39.8	266.9	88.1	4.7	22.1	4048.6	338.6	38.0	90.8	136.6

Table 2. Average Percentage Correct and S_f for Datasets from (Holte, 1993).



a)



b)

Figure 1. Comparison of Percentage Correct and S_f for "CH" Dataset.

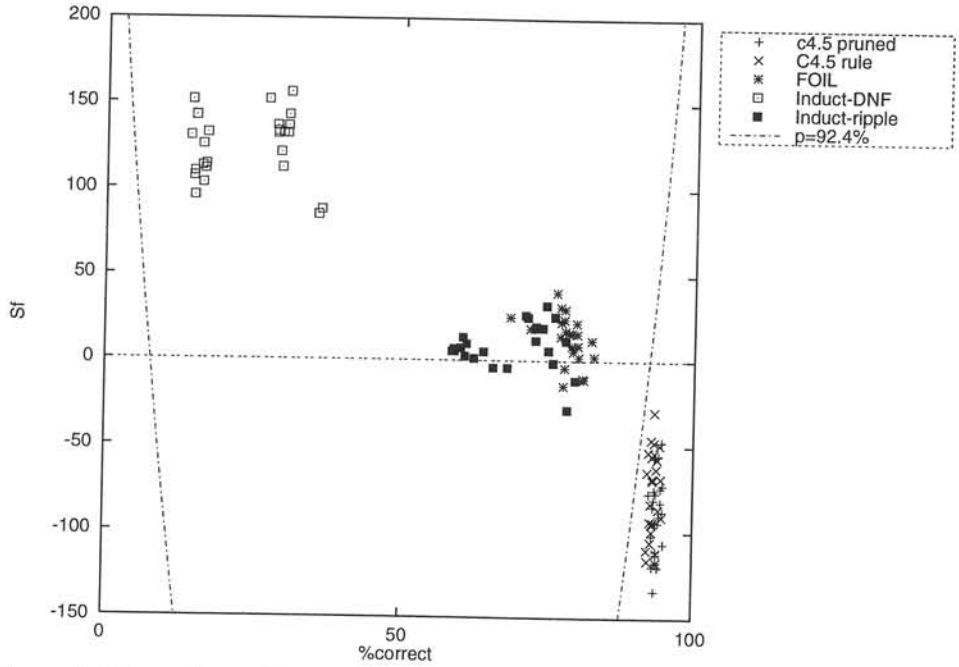


Figure 2. Comparison of Percentage Correct and S_f for "wcr" Dataset.

Working Paper Series
ISSN 1170-487X

Keeping Free-edited Textual and Graphical Views of Information Consistent

by John C Grundy and John G Hosking

Working Paper 96/4

March 1996

Department of Computer Science
The University of Waikato
Private Bag 3105
Hamilton, New Zealand

Keeping Free-edited Textual and Graphical Views of Information Consistent

John C. Grundy

Department of Computer Science
University of Waikato
Private Bag 3105, Hamilton, New Zealand
jgrundy@cs.waikato.ac.nz

John G. Hosking

Department of Computer Science
University of Auckland
Private Bag, Auckland, New Zealand
john@cs.aukuni.ac.nz

Abstract

Multi-view editing is useful in many situations where users of a software application want to see and interact with different representations of the same information. This paper describes a new approach to keeping free-edited multiple textual and graphical views of information consistent. Descriptions of changes to information items are displayed in various ways in the multiple views of these items. Users can request an editing tool to automatically apply changes to a view, select a change to make from a range of possible changes, or manually implement changes to maintain view consistency. Semantic errors, user-defined changes and hierarchical changes can be represented, and this technique also supports flexible view consistency for cooperative work systems. Experience with this technique in several diverse multi-view editing environments is described.

1. Introduction

Multi-view editing provides software application users with multiple, editable views (perspectives) of a common information model, as shown in figure 1 [4, 29]. This allows users to view and interact with the information in different ways. Some representations are better suited than others for the various tasks involved in software system specification, design and implementation.

One consequence of a shared information model is that users of multiple views usually want each view to be consistent with the other views [4, 29, 33]. When one view is updated, the other views which share the affected data must be updated to reflect the change. This involves propagating the effect of a view change to the common information model and then to all other affected views (i.e. view consistency management). Sometimes it is desirable to maintain view inconsistency, for short or long periods of time [8, 30, 20], and consistency management is further complicated when multiple users cooperate to edit multiple views of shared information [7, 31].

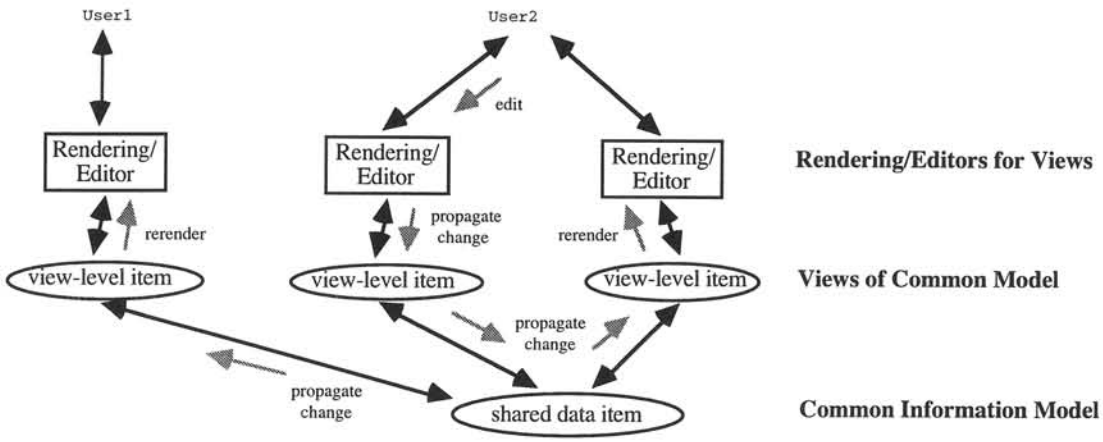


Figure 1. The concept of multi-view editing.

Many systems employ multi-view consistency, including CASE tools, programming environments, Integrated Software Development Environments (ISDEs), interface builders, and CSCW applications. Tools for building such multi-view editing applications include user interface toolkits, editor toolkits, program visualisation systems, environment generators and database

systems. However most existing multi-view editing systems support limited forms of consistency management, with only direct translations of editing changes supported between views. Many systems also require views to be structure-edited or employ ad-hoc and/or difficult to reuse techniques. This severely limits the degree of true inter-view consistency (and inconsistency) management that can be achieved.

This paper describes a new approach for supporting consistency management in multi-view editing environments. Multiple views are either interactively edited (graphical views) or free-edited (textual views), for maximum flexibility and suitability to user editing preferences [2, 43]. The kernel idea in our approach is to make view inconsistencies visible and interactible for environment users. Inconsistencies between views caused by a user editing a view are presented in affected views as descriptions of changes in dialogs and headers, view annotations and colouring, and option choices. Users view and interact with these representations of inconsistencies by asking the environment to automatically make a view update, manually making an update, or selecting a desired update option. Inconsistencies can be hierarchically grouped, ignored and deleted, or retained in modification history lists, and can be used to support flexible versioning and cooperative work facilities.

Section 2 reviews existing multi-view consistency management applications and models. Section 3 describes how our approach presents inconsistencies to users. Section 4 discusses interaction techniques with these inconsistency representations. Section 5 briefly presents the design and implementation of an architecture for building new multi-view editing environments utilising these techniques. Section 6 illustrates the versatility of our technique and architecture in relation to various applications we have built with it. Section 7 summarises the contributions of this research.

2. Existing Multi-view Editing Systems

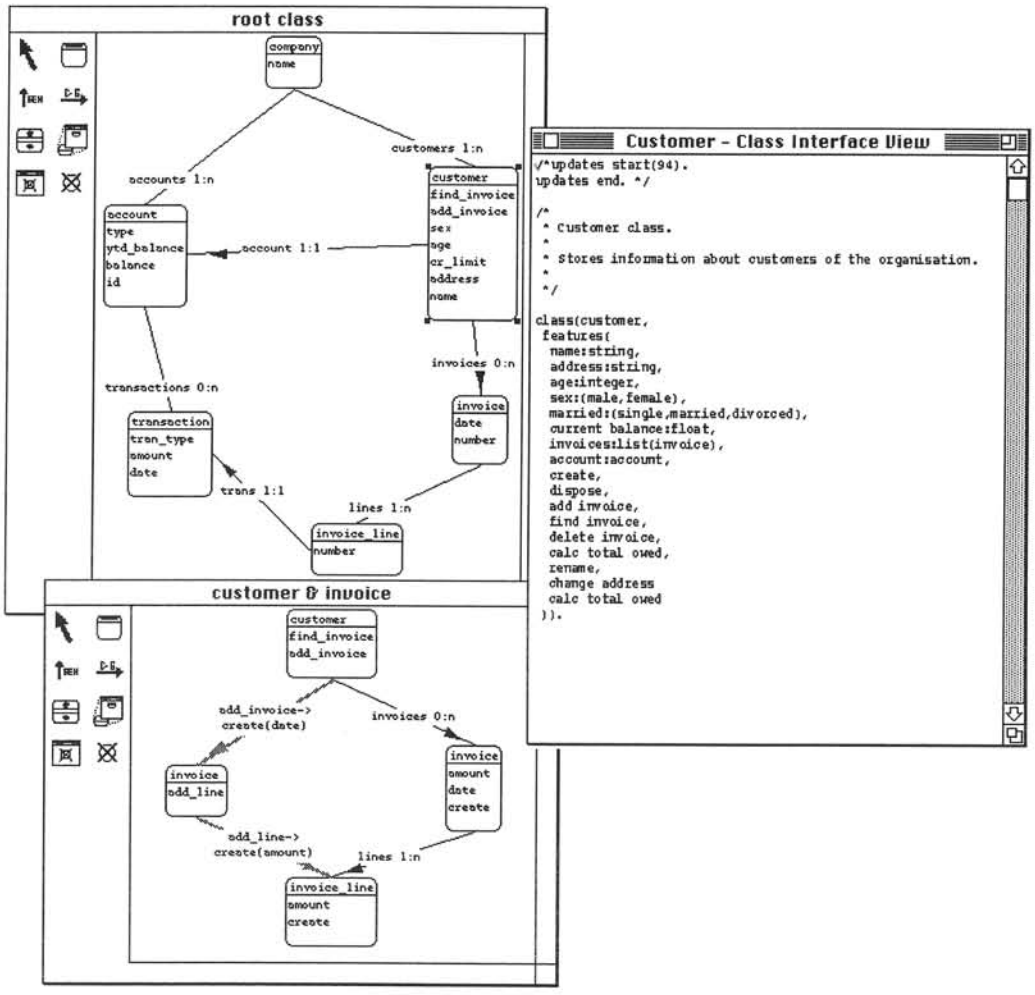


Figure 2. An example multi-view editing environment.

Figure 2 shows a screen dump from SPE (Snart Programming Environment), an ISDE for developing object-oriented programs using Snart, an object-oriented Prolog [11]. SPE supports multiple textual and graphical views of software development, including full bi-directional consistency management between all views.

We use SPE to illustrate a number of consistency management issues not well supported by current multi-view editing environments and models. Later, we show how our model can be used to provide consistency and inconsistency management for SPE and other ISDEs built using our model. One consistency requirement for SPE is that textual views need to be kept consistent with graphical views, and vice-versa. Many changes to graphical views, however, cannot be directly translated into textual view updates. An example is when a client-supplier (method call) relationship is added in a graphical design view. This cannot be automatically translated into a textual view update, as neither the call's arguments nor its position in the textual code are known. Similar problems occur when trying to keep graphical analysis and design views, textual class interface and method implementation, and graphical and textual documentation views consistent [11]. In addition to managing consistency between views supporting different phases of software development, SPE also needs to support consistency between multiple views of a single program description, with views sharing a common information model. Collaborative software development must also be supported, with multiple views being shared and kept consistent for multiple developers.

Many systems, such as those developed on top of Unix, provide no consistency management between multiple tools and views. Changing information in one view is not reflected in other views which share the information, and different tools are usually implemented as separate, disconnected entities [23, 29]. FIELD environments [35, 36] and succesors, such as DECFUSE [19], as an exception, use selective broadcasting to propagate information changes between multiple Unix tools which share a consistent user interface look-and-feel. Only limited forms of view consistency are supported by FIELD, however, and building such environments and integrating new tools into the environment requires much effort [29].

Visual programming environments, such as Prograph [5] and Garden [34], user interface builders, such as Unidraw [41], and CASE tools, such as Software thru Pictures™ [42] and TurboCASE™ [39], provide multiple views of programs and designs. Views are usually graphical, with only changes that can be automatically applied by the environments being propagated. Textual view "consistency" is usually limited to regenerating textual code, and reverse-engineering is used to partially update design views when text is modified. This results in disjointed environments for users, with many changes having to be manually propagated between views.

FormsVBT [3], implemented using the Zeus program visualisation system [4], provides integrated textual and graphical views for user interface specification. A free-edited textual view is kept consistent with an interactively edited graphical view. FormsVBT, however, requires graphical view updates to be locked out when the textual view is being updated, greatly constraining a user's choice of editing mode. Ad hoc techniques are used to keep the simple S-expression textual code consistent, which do not scale up to more complex infomation representations.

Many ISDEs provide multiple views of software development, including Dora [32], PECAN [33], and MELD [21]. These environments utilise restrictive structure-editing to keep their views consistent [2, 43], and only support the propagation of changes which can be automatically applied to views by the environment.

A few systems provide inconsistency management support, where multiple views need to be inconsistent for some time, but this inconsistency needs to be recorded and resolved at a later date. An example is [8], which uses logic predicates to record inconsistencies between different viewpoints on software designs. However, this approach utilises complex, cumbersome logic expressions to express the inconsistencies between multiple views.

Many environment generators, models and software architectures have been developed for building new applications supporting multi-view editing. Examples include the Smalltalk MVC framework [24], Interviews [25], the ItemList structure [6], and Zeus [4]. However, most of these systems provide no support for the kinds of flexible (in)consistency management required in environments like SPE, and are very difficult or impossible to extend to support such techniques.

3. Presentation of Inconsistencies

Any environment supporting multiple views should keep these views consistent under change, or at least inform users when a view is presenting inconsistent information [29]. It is useful in the latter case to have the environment describe the inconsistent nature of the view information. This can be by describing the update made to another view making this view inconsistent, presenting a range of operations on the view that need to be made to resolve the inconsistency, or by indicating the view is inconsistent and providing more detailed information elsewhere. Our approach utilises all of these techniques to present users with representations of view inconsistencies. As inconsistencies caused by multi-view editing often can not be automatically resolved by the environment, users need to be informed of view inconsistencies via presentation techniques, and then be allowed to interact with these representations to resolve inconsistencies.

We have built an architecture which automatically generates descriptions of changes made to view items, called *change descriptions*, whenever view items are updated. These are propagated to other views which share the updated information, and these views can respond to the change descriptions by updating their own state, presenting the change descriptions to users, or ignoring the change to the updated view. Presentation of change descriptions allows users to visualise inconsistencies between views. This architecture is described in more detail in Section 5. This section illustrates different ways of presenting view inconsistencies by using change descriptions. The example environments used have been built using our architecture, and all of the techniques described are general and not specific to each system.

3.1. Automatic Update of Views

Some view updates are easy for environments to automatically carry out. For example, in SPE renaming a class icon is easy to propagate and automatically implement in all other graphical views in which the class is shown. All graphical icon objects for the class are updated and rerendered to reflect the class rename. All of the existing multi-view editing models described previously support this simple change propagation mechanism. Most, however, only support this level of view consistency - if the update can be directly made to another affected view, then it is performed by the environment. If the update cannot be made, it is not performed and the user of the environment is usually never informed of the possible view inconsistency. In addition, many environments do not support translation of change between views supporting different notations, when this could be automatically made. For example, many CASE tools do not support change propagation between OOA and ER or NIAM views [40].

3.2. Presenting Descriptions of Inconsistencies

Some view updates can not be directly translated into appropriate updates on other views. This often occurs when the updated view and other affected views represent the shared information in quite different ways, or at different levels of abstraction. For example, SPE textual views show class interface definitions and method implementation code, which are at a lower level of abstraction than graphical analysis and design views. Keeping these views consistent, or at least informing programmers of inconsistencies between the views, is more difficult than keeping the design views consistent with each other.

Our multi-view editing approach solves this problem by presenting users with a description of any inter-view inconsistency that may be present. This is done by inserting *change descriptions* into the view. For example, figure 3 shows an SPE class interface view with several change descriptions expanded into a header region at the start of the view's text. These describe inconsistencies between this view and other, modified views of the program. These change descriptions can be used to present inconsistencies between any kinds of views.

In this example, change #32 indicates that the programmer has renamed the address attribute to address1, change #33 indicates a new attribute address2 has been added, change #36 indicates a client/supplier relationship between the customer::add_invoice method and the invoice::create method has been added, and change #44 indicates that a compilation (semantic) error has occurred, with duplicate calc_total_owed methods being defined for the customer class. The first three of these changes might have been made in a graphical view and thus the change descriptions are

informing the programmer of (possible) view inconsistencies between this textual view and the modified graphical view.

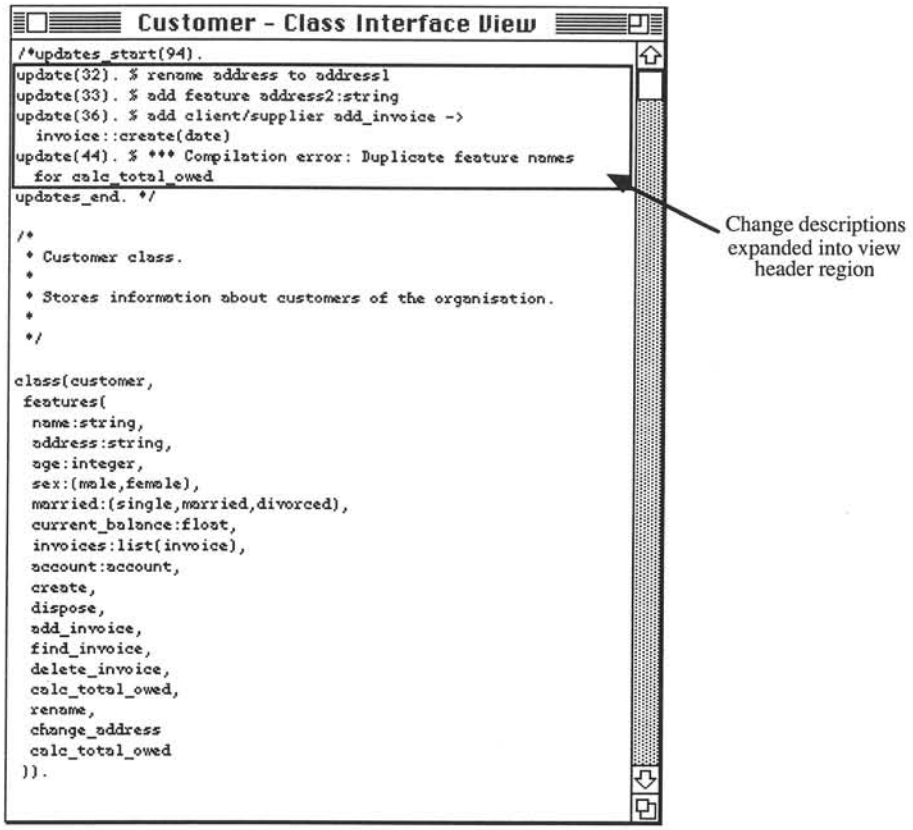


Figure 3. Indicating inconsistencies in textual views.

Changes 32 and 33 can be automatically applied by the environment to the textual view to update it, and, in SPE, the programmer can specify that the environment is to always automatically update the view's text to reflect such changes, rather than displaying the change descriptions. Changes 36 and 44 can not be automatically carried out by the environment. To resolve change 36, for example, the programmer must, at some later time, modify the textual view of the customer::add_invoice method to insert an appropriate method call with appropriate arguments, and possibly update the customer class textual view so that an appropriate reference to invoice class objects exists.

Changes from graphical view to graphical view, and from textual view to textual view, can also be presented in this way. For example, figure 4 shows an Object-Z-like view in SPE being kept consistent with a method implementation view. Both views are textual, and changes to one type of view are shown in the other type by change description presentation. This approach is also used to keep graphical analysis and design views consistent, with change descriptions shown in dialogs.

There are several other ways to present such change descriptions in views. Header regions in different parts of a view and hypertext buttons allow users to access groups of change descriptions in either the view itself or in pop-up menus or dialogs. We have found scrolling dialog menus to be most appropriate when a potentially large number of inconsistencies are, or may be, present. Pop-up menus work well when a smaller number of inconsistencies need documenting. View annotation, as used for the header section in SPE textual views, is useful when users want or need to see all relevant view inconsistencies at the same time.

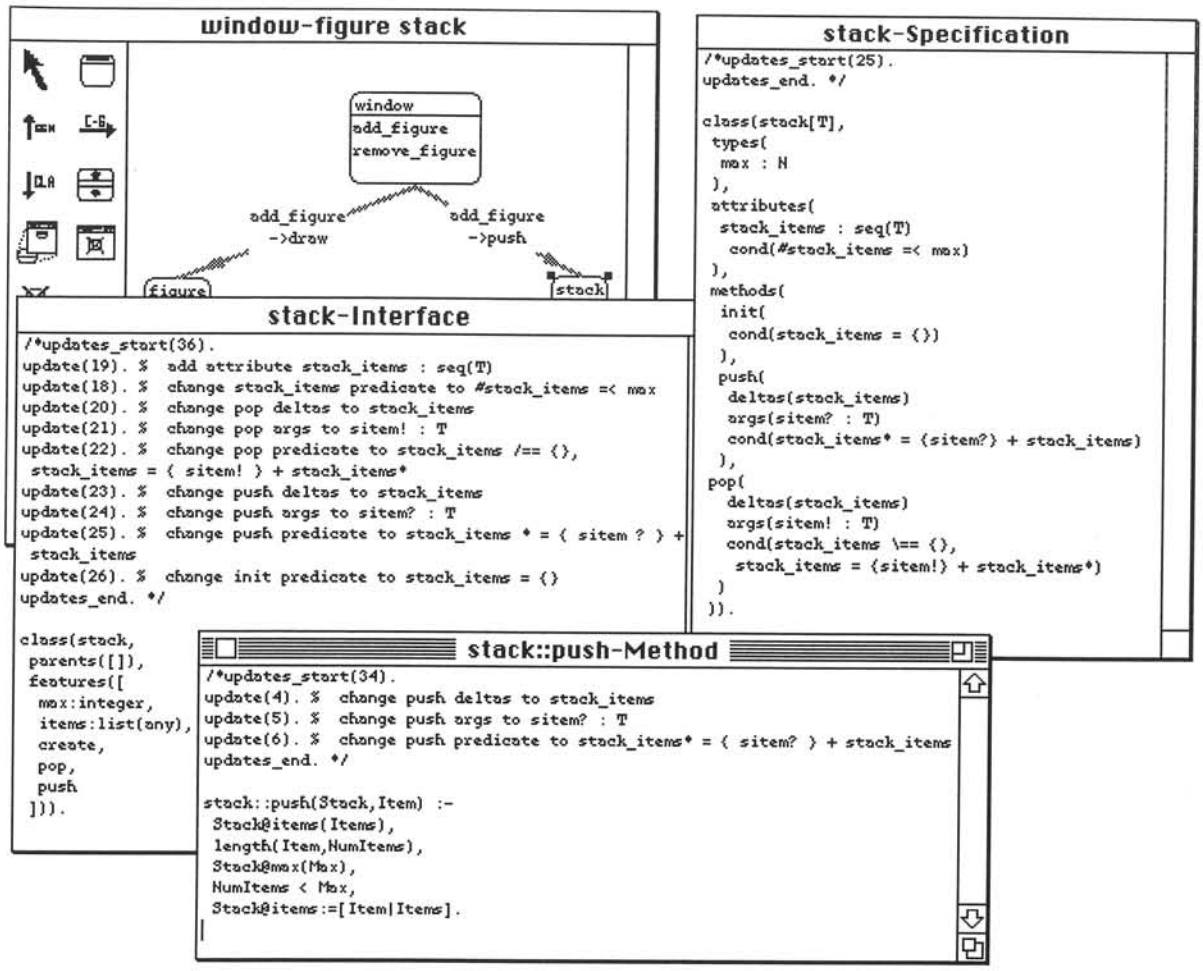


Figure 4. Keeping textual views consistent with one another.

3.3. Semantic and User-defined Inconsistencies

Change descriptions can present semantic and user-defined inconsistencies between views, in addition to the structural inconsistencies described above. Change 44 shown in figure 3 describes a semantic error which has occured as a result of editing a view. SPE provides an option to locate classes whose definition has semantic errors. These change descriptions can be viewed for each class, and views containing representations of each class can be highlighted to indicate the presence of inconsistencies (via graphical icon shading/colouring and textual name highlighting).

Semantic inconsistencies can be described both by highlighting affected view information and by presenting detailed change descriptions to users. It is often harder to fully describe the cause of semantic inconsistencies, and in most situations impossible to provide automatic resolution facilities. Often an environment can only inform users of the editing change(s) that caused the inconsistency and the semantic error(s) that now exist. SPE allows users to interact with semantic inconsistencies to find the source of the inconsistencies (usually individual view edits) and to obtain detailed descriptions of their causes.

Users can create their own "change descriptions" via modification history dialogs, as shown in figure 5. These do not directly represent any particular view inconsistency, but rather serve as user-defined view documentation. Such change descriptions are associated with specific information items, and, like structural and semantic change descriptions, are propagated to other views of these items. They also serve a useful purpose in supporting context-dependent communication in cooperative environments, allowing cooperating users to interact via "messages" sent via the change description mechanism [11, 13]. All change descriptions can be annotated via the dialog shown in figure 5, allowing users to specify additional reasons why changes have been made.

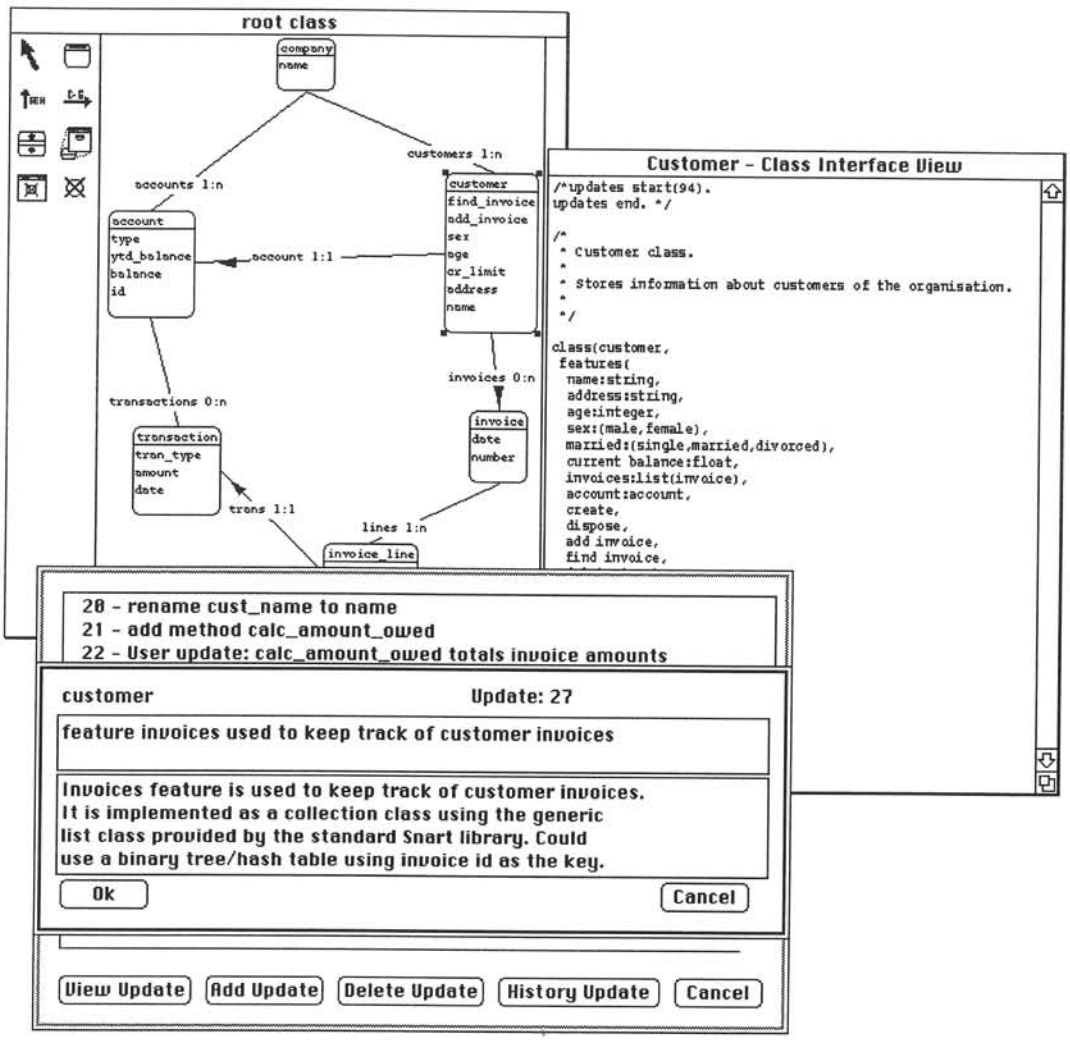


Figure 5. Example of a user-defined change description from SPE.

3.4. Highlighting Inconsistencies

It is often appropriate for an environment to present inconsistencies in a more context-dependent way than via textual change descriptions. Figure 6 shows a screen dump from MViewsDP, an interface builder which supports graphical and textual dialog specification. A dialog is under design, and the dialog control button 'Ok' in the graphical view has been shifted. This is reflected in the textual view header by a change description. This change description can also be shown in a pop-up menu associated with the graphical view.

There is, however, a semantic inconsistency resulting from the Ok button's border overlapping that of its enclosing dialog box. This must be resolved, as the specification is currently invalid. This inconsistency is indicated by shading the Ok button icon, so the user of the environment is drawn to this component. The change description is shown in a dialog if the user requests to see it, in order to describe the inconsistency.

A variety of presentation techniques can be utilised to highlight inconsistencies in a particular part of a view. These include shading and colouring graphical icons, changing font, style, size or colour of text, or more dynamic techniques, such as blinking affected icons. Highlighting inconsistencies and explicit description of inconsistency via change descriptions are complementary techniques. The presence of highlighting in a view can draw the user's attention to a particular aspect, which may then be interacted with to view detailed change descriptions.

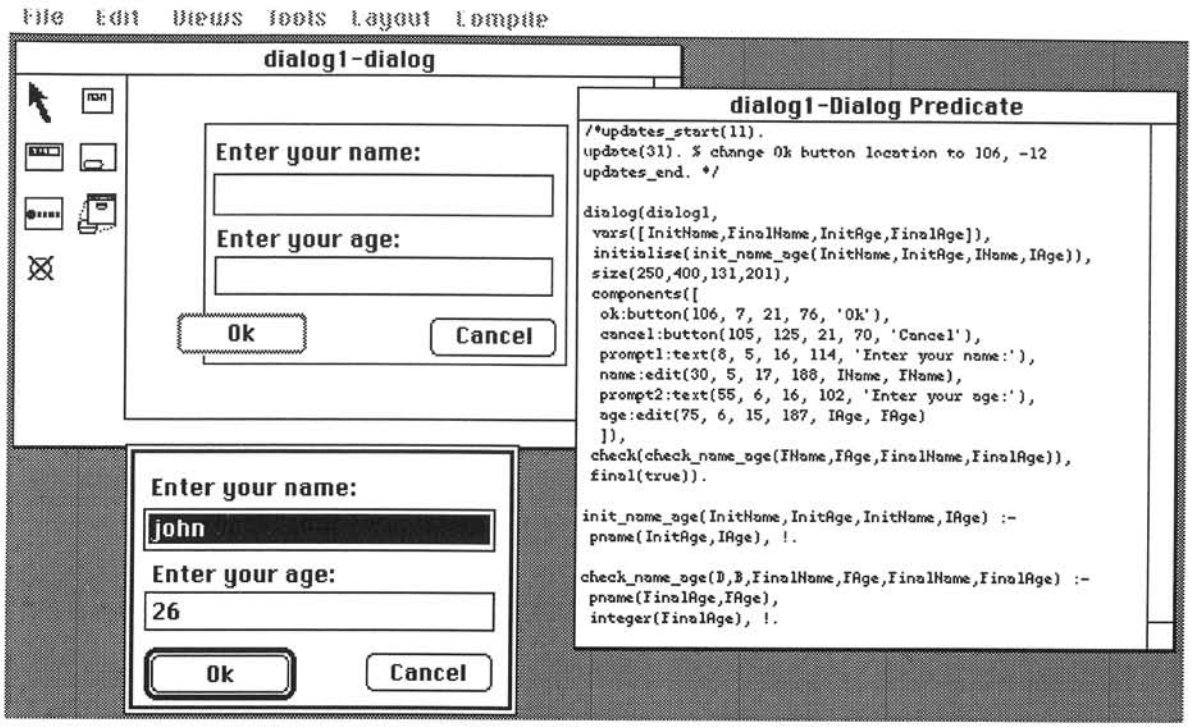


Figure 6. Indicating inconsistencies in graphical views.

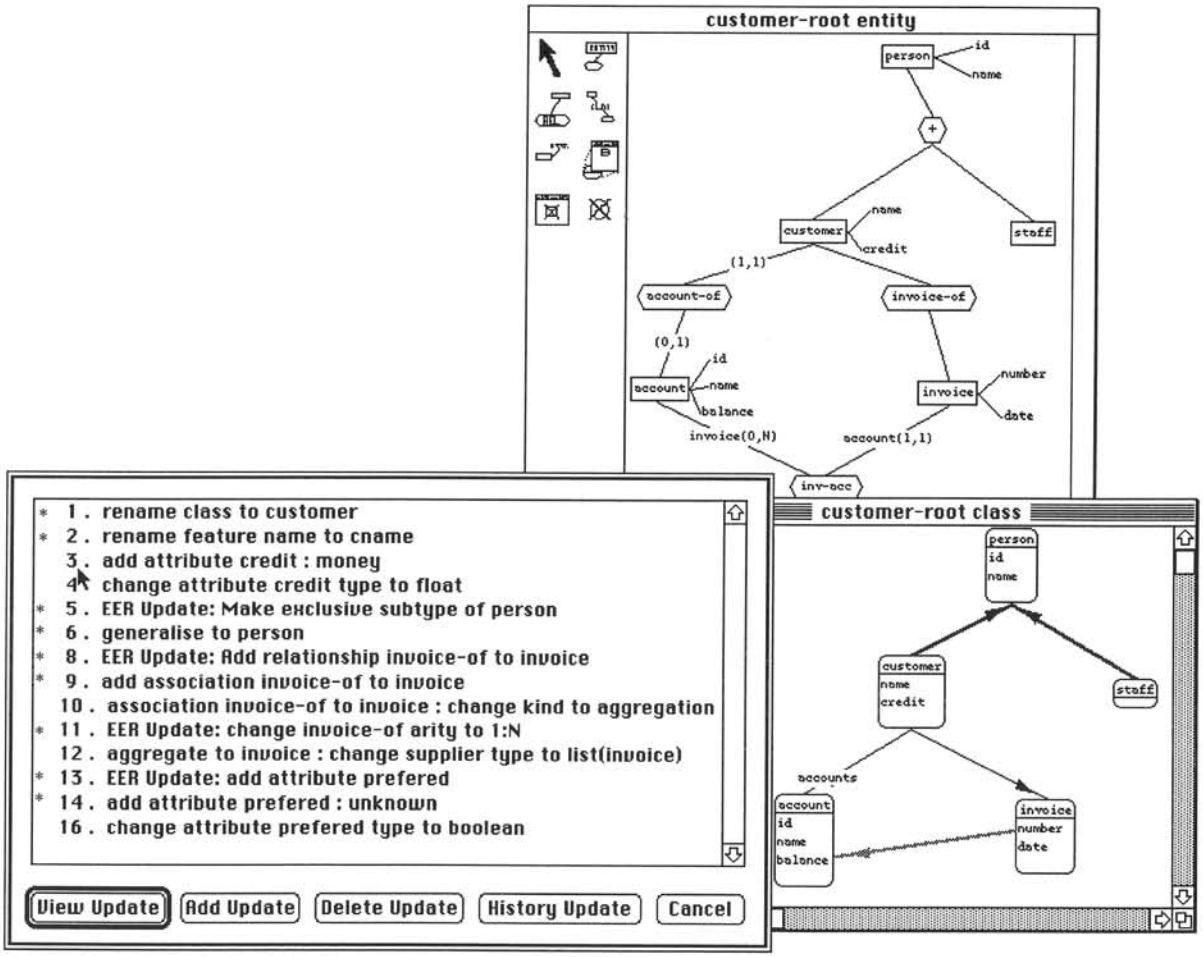


Figure 7. Graphical view inconsistencies.

3.5. Presenting Stored Inconsistencies

It is often useful to retain descriptions of inconsistencies and present these to users on request in many environments. Figure 7 shows a screen dump from OOEER, an environment supporting integrated OOA/D and EER modelling [14]. OOEER was produced by integrating SPE and MViewsER, an environment supporting integrated graphical EER and textual relational schema modelling. When an EER notation view is modified, the update is translated into an update on corresponding OOA/D views. Some translations can be automatically applied by OOEER, others result in partial translations with view inconsistency. These inconsistencies are presented by colouring affected icons and text. The user can then view a detailed description of the inconsistency in a dialog.

The dialog in figure 7 describes changes made in the EER view customer entity which affect the OOA/D view customer class. Items highlighted with a '*' were actually made in the EER view and translated into OOA/D view updates. The user can make further changes to the OOA/D view if the EER update could only partially be translated into an OOA/D view update. For example, adding a EER relationship (change #8) was translated into the addition of an OOA/D association relationship (change #9). The user then refined this relationship to an aggregation relationship (change #10). This couldn't automatically be done, as the EER notation does not support the distinction between different kinds of relationships.

OOEER retains inter-notation view change descriptions as a "modification history", documenting the history of updates a view or view component has undergone. OOEER provides menu options to locate inconsistent views or components, making it easier for users to determine what inconsistencies need their attention. On display, OOEER highlights which items in a view are (or may be) inconsistent.

4. Interaction with Inconsistencies

The presentation of inconsistencies is not sufficient to support flexible multi-view editing of shared information. Environments must allow users to interact with these inconsistencies to resolve them. Our approach provides a range of options for users and environment implementers to support interaction with inconsistency presentations. These include: allowing users to select presentations and request the environment to automatically resolve the inconsistency; presenting users with a list of optional updates which can resolve the inconsistency; and requiring users to manually update the affected view. Our techniques facilitate cooperative work by broadcasting and presenting change descriptions to other users and allowing them to interact with these to resolve interperson view inconsistencies.

4.1. Selection and Action of Descriptions

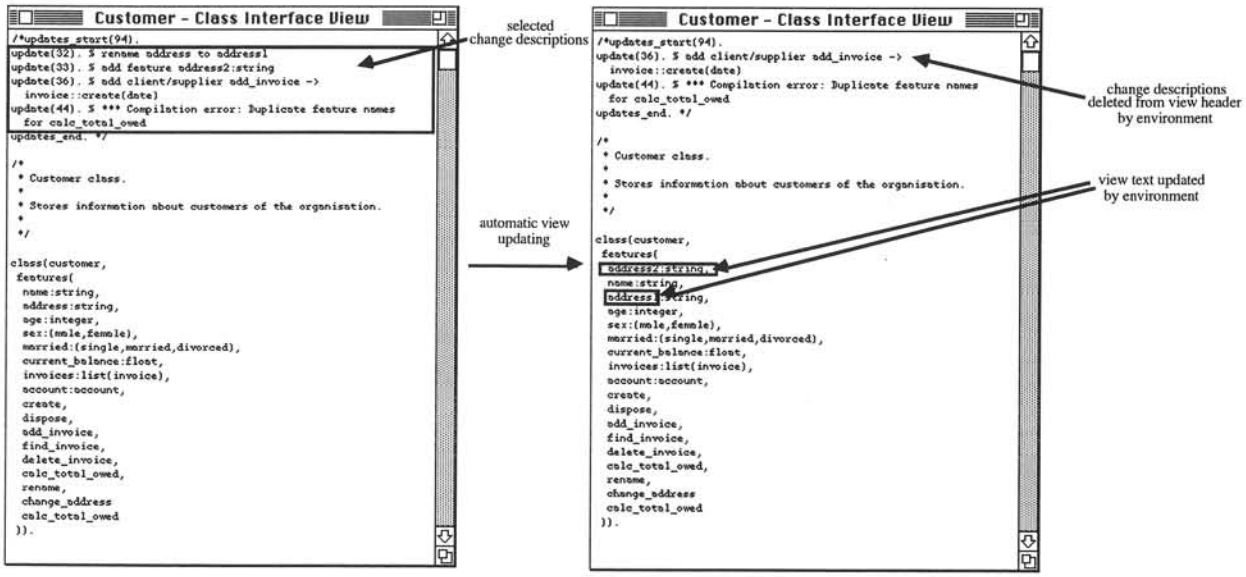


Figure 8. Automatic resolution of inconsistency.

Change descriptions can be selected by users and a request made to the environment to resolve the inconsistency described. Users interactively select change descriptions in a dialog or in a view, issue a request to implement the update, and are informed of the result of this request. Figure 8 shows an example of requested view updates in SPE. The user has selected all of the change descriptions in the textual view header and asked SPE to update the view's text. The first two changes can be automatically applied by the environment and results in the attribute address being renamed to address1, and the addition of attribute address2. Both of these change descriptions are deleted to indicate successful update of the view. The other two changes, however, can not be automatically applied, and the change descriptions are left in the view's text to indicate this. Updates selected in a dialog but which could not be successfully applied by the environment are highlighted or shown to the user in another dialog.

View changes can be animated in sequence, if desired, so the user can "see" the effects of a sequence of changes. Updated items in the view can also be distinguished, for example by shading, colouring and font characteristics, allowing users to identify aspects of the view which were updated. This helps users determine the effects of changes and, moreover, to ensure that the desired view update(s) were inferred correctly by the environment.

It is usually straightforward to update graphical view information by actioning change descriptions. Environment implementers specify change description patterns and action sequences to perform on the appropriate graphical view components. As our textual views are free-edited, and hence the structure of the view information is not represented except in the view's text, this information is automatically kept consistent via a technique we call incremental unparsing. A view is incrementally parsed to identify where text should be added, updated or deleted. Change descriptions are then unparsed, or "pretty printed", into the view by modifying the affected text as appropriate. Further details on the implementation of this technique can be found in [12].

4.2. Selection of Optional Update

Often an inconsistency arises when a multi-view editing environment can not choose between several possible options for resolving an inconsistency. This occurs in environments which translate changes between views of information of differing levels of abstraction or between views which only share part of the information being viewed. Examples are OOEER, which translates changes to and from EER and OOA/D views of the same software system, and SPE, with informal code views and formal Object-Z views. Often in these environments a change made to one notation view can be partially translated into a change in views using the other notation. The user must complete the translation by choosing from a range of possible view updates.

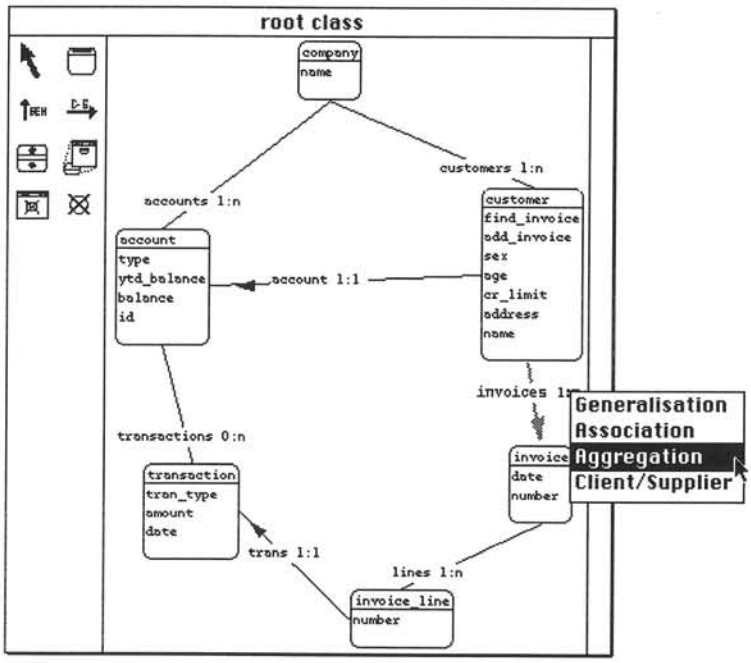


Figure 9. Selection of optional update to fully resolve an inconsistency.

For example, in OOEER if a relationship is added in an EER view between customer and invoice entities, then this is translated by the environment into the addition of a relationship between the customer and invoice classes in the OOA/D views. However, more information about the relationship is needed in the OOA/D views: the relationship might be an association or aggregation OOA relationship (not specified in the EER view), or if it is a OOD client/supplier relationship, it may specify a method call between objects and thus need to specify caller/called method names and arguments. As this information is not specified in the EER views, OOEER defaults the relationship to an OOA association relationship, and lets the user refine this further by changing it to an aggregation or client/supplier relationship and adding extra information about it. OOEER indicates the new relationship has been defaulted by greying it, and provides a pop-up menu to allow the user to easily change its type. The user can also optionally specify more information about the new relationship in the OOA/D views as appropriate. Figure 9 shows an example of such an interaction with an inconsistency via optional update.

4.3. Manual Resolution of Inconsistency

In this approach, the user must be prompted to manually update a view to resolve an inconsistency. The environment can assist the user in identifying parts of the view which are inconsistent and provide a description of the view updates causing the inconsistencies using the techniques of Section 3. Such a situation usually occurs between views which provide quite different representation, interaction or abstraction levels on shared information.

For example, many design-level updates in SPE can not be sensibly defaulted in textual code-level views. The design-level view updates are presented to users as change descriptions and it is left to the user to manually resolve the view inconsistency. The change descriptions can then be deleted by the user when they are no longer required. The addition of a design-level client/supplier relationship is usually implemented as a code-level method call. Such a change can not be automatically implemented in a code-level textual view, as the environment does not know the appropriate method arguments (variables or constants) and position of the method call within the method code. Semantic errors require users to correct the problem and can not usually be automatically corrected by the environment. Figure 10 shows an example of this manual inconsistency resolution.

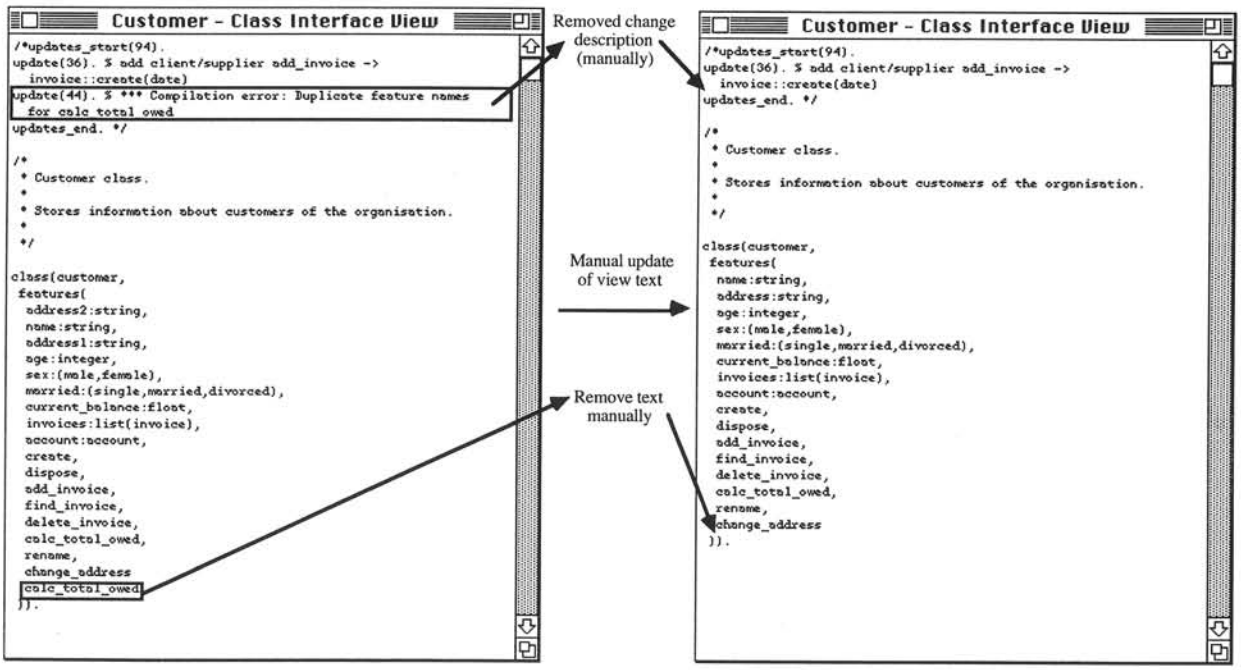


Figure 10. Manual resolution of inconsistency.

In addition to resolving inconsistencies, users can interact with inconsistency representations in other ways. An environment can add interaction "hot-spots" to highlighted, inconsistent view items or to change descriptions. Hypertext buttons can be clicked on to: display change

descriptions in dialogs or pop-up menus; display pop-up menus with a list of optional updates; or display of the view whose update caused the inconsistency.

4.4. Versioning and Cooperative Work via Inconsistency Presentation and Interaction

Computer-Supported Cooperative Work (CSCW) systems may utilise a multi-view editing approach to sharing and modifying information [32, 36, 29]. Inconsistencies between views become more difficult to resolve as different users of the views are affected. We have used our technique to provide support for asynchronous, semi-synchronous and synchronous multi-view editing for C-SPE, a collaborative form of SPE [11].

Change descriptions are used to describe changes between different versions of updated views i.e. together form a modification history for views. By using these modification histories as versions, a flexible multi-view version control system can be produced [11]. A non-sequential undo/redo facility can be provided by reversing or reapplying view changes, and this can be extended to supporting version merging and revision[11]. An example from C-SPE is shown in figure 11, with conflicts between two merged versions presented to the merging user as a sequence of merge conflict change descriptions. Semantic conflicts produced by merging are presented in the same manner.

Inconsistencies between multiple views shared by different users are presented in similar ways to those described previously. Asynchronous collaboration is supported by giving users their own versions of each view, with a single developer merging these versions at a later time. Semi-synchronous collaboration is supported by broadcasting change descriptions between different users' environments as they occur. These are presented to users by being stored or presented in a view or in a dialog. Figure 12 shows an example of semi-synchronous view editing in C-SPE. Synchronous collaboration requires users to share and edit the same version of a view, with fine-grained locking on view components [11].

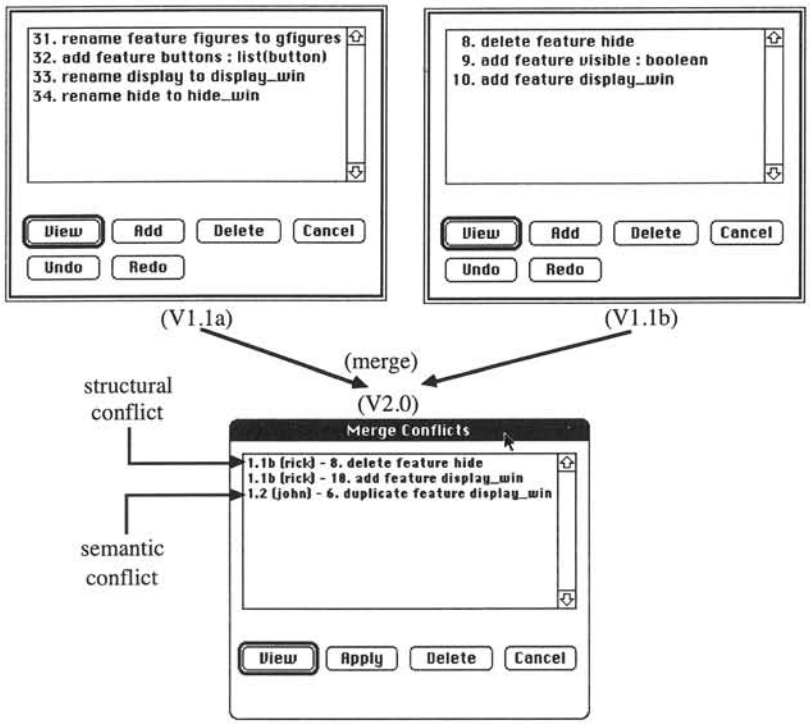


Figure 11. Version merging and presentation of merge conflicts via change descriptions.

We are currently experimenting with multi-view editing facilities which utilise inconsistency presentation and interaction to coordinate cooperative work and planning processes. This allows users to specify the action their environment should take when generating or receiving change descriptions. This includes specifying the method of obtaining and/or presenting extra information about the view update made, such as the work or planning context it was made in, and specifying action to take when processing different kinds of view edits. This allows users to build up highly

flexible and configurable cooperative systems on top of multi-view editing environments which support our view consistency model [15].

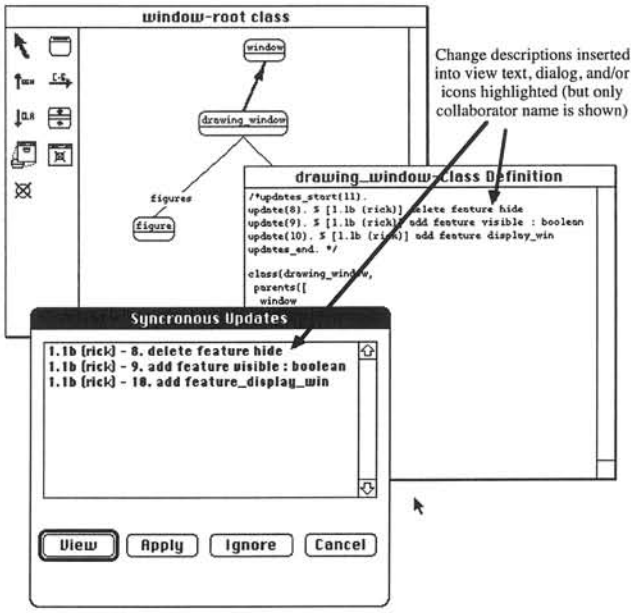


Figure 12. Semisynchronous view editing via change description broadcasting.

5. A Supporting Software Architecture

We describe a basic architecture for supporting our inconsistency presentation and interaction techniques. We have used this model as the basis for an object-oriented framework of classes, which are reused to construct new multi-view editing environments.

5.1. CPRGs

Our view consistency model requires descriptions of the changes made to a view to be generated and propagated to other views which share the updated information. We have developed a software architecture, called Change Propagation and Response Graphs (CPRGs), that supports a wide range of flexible consistency management mechanisms based on change description propagation [13]. The kernel idea of CPRGs is that information is structured as attributed *components* linked by inter-component *relationships* , forming a graph-based description of related information items. The state of a component is modified by *operations*, which generate *change descriptions* describing the modification the updated component has undergone.

Change descriptions are propagated to all relationships a component participates in. These relationships respond to change descriptions by: updating their own or related component states; passing on change descriptions to related components; or ignore the change to the updated component. This technique supports a wide variety of consistency management facilities used by ISDE environments, including flexible multiple view consistency, inter-component constraints, efficient incremental attribute recalculation, generic undo/redo facilities, and version control and collaborative facilities.

Figure 13 illustrates how consistency management in SPE is supported by the CPRG model: 1) a class component is renamed, generating a change description of the form update(class,name,customer,customer2); 2) this change description is propagated to all relationships the class component participates in; 3) the relationships determine if other components related to class need to be updated. If they do, they either apply operations to update these components or forward the change description to them; 4) the related components are updated to maintain view consistency, inform users of inconsistencies that can't be automatically resolved, or inform cooperating users of view changes.

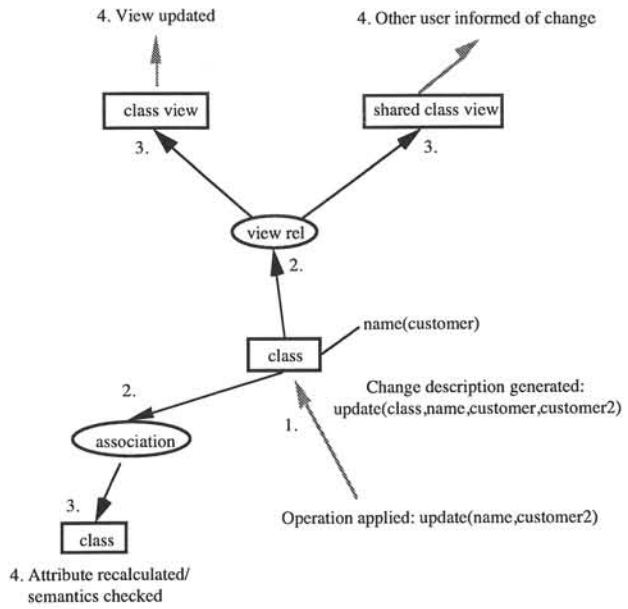


Figure 13. Example of consistency management using CPRGs.

In addition, our CPRG model supports the composition of discrete change descriptions into *macro* change descriptions [13]. It is often more useful to store and present to users the more abstract macro change description rather than all of its individual change descriptions, reducing the number of individual change descriptions presented. CPRG macro change descriptions can be presented to users in the same manner as simple change descriptions, but an environment can present or allow users to interact with a macro description in more abstract ways than its composite descriptions. As the components of macro change descriptions are still retained, the user can request to see them if necessary.

5.2. MViews

In addition to developing the multi-view editing inconsistency generation, propagation, presentation and interaction techniques described above, we have developed MViews, a reusable object-oriented framework for building environments using these mechanisms. MViews classes are specialised to build new multi-view editing environments [9, 12]. MViews is based on the CPRG model, and provides a variety of classes for implementing base information representations, multiple textual and graphical views, and multi-view editors. Figure 14 illustrates the structure of SPE, built using MViews. A *base layer* acts as the shared information repository for views. A *view layer* defines the contents of each view in terms of CPRG components and relationships i.e. views have the same structure as the base information items they represent. A *display layer* provides a graphical or textual rendering of the view layer, and provides suitble editing facilities for this representation. External tools not built using MViews can also be interfaced at the view and display layers.

MViews uses the inconsistency presentation and interaction techniques described previously to support flexible multi-view editing, as shown by the SPE examples described earlier. Change descriptions are retained so that when users interact with inconsistency presentations, MViews can determine the nature of the inconsistency. When a change description representation is selected and/or interacted with in a textual view or dialog, MViews can determine the CPRG change description being interacted with. The environment can then take action for the view inconsistency being interacted with. CPRG change descriptions, such as update(Class,class_name,OldName,NewName), are translated into readable forms, such as 'rename class Class to NewName', by environment implementers specifying change description patterns and resultant formatted representations. Change descriptions generated in other views which are not relevant to the user of another view sharing the updated information are filtered out during propagation to the affected view.

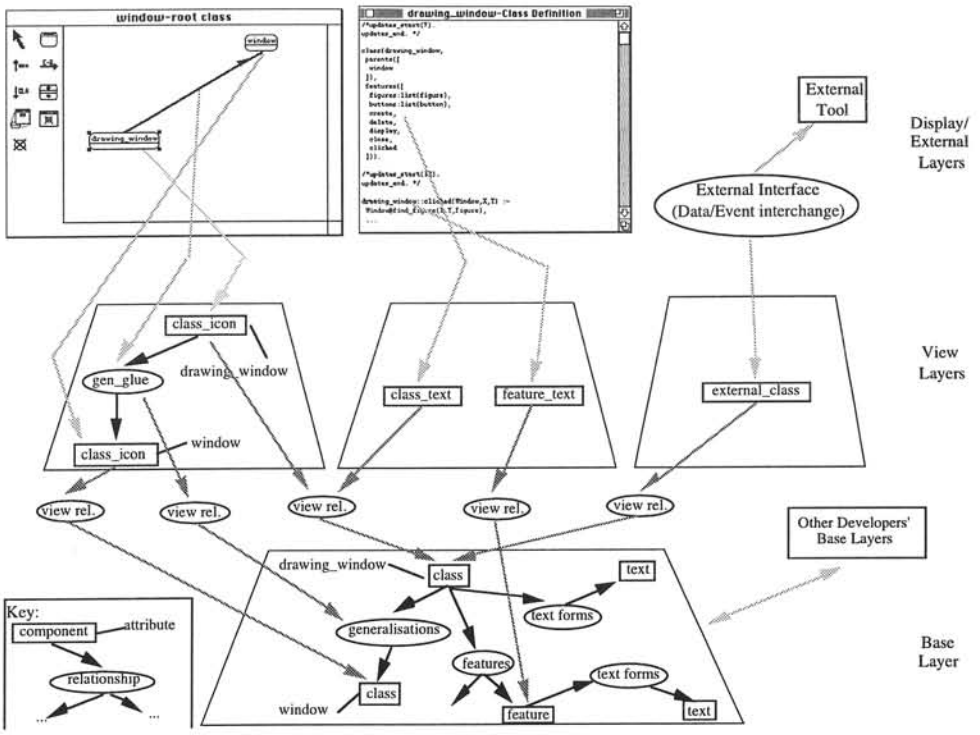


Figure 14. An architecture supporting change description propagation.

5.3. Implementation

MViews is implemented in Snart, an object-oriented extension to Prolog [10], running on Macintoshes using LPA MacProlog[1]. Environment implementers specialise Snart classes to define new environment information repositories, multiple views, and view renderings and editors. Snart is a persistent language, with objects dynamically saved and loaded to a persistent object store, making information repository and view persistency management transparent for environment implementers.

Change descriptions are represented as Prolog terms. This makes change description generation, propagation and storage very efficient, resulting in an architecture which performs well, even for large applications such as SPE. Representations of these terms are generated for display to users via declarative Prolog predicates. Being Prolog terms, change descriptions can be stored in Prolog lists to store inconsistency representations and form modification and version histories. They are converted to and from a textual representation for broadcasting to other users' environments to support cooperative work [16].

MViews textual view editors are standard Macintosh free-edited text windows. Definite Clause Grammar (DCG) parsers are used to convert textual view contents into Prolog terms, used to determine base view component updates. Graphical view components are rendered and edited using LPA MacProlog's Graphics Description Language (GDL) facilities. This allows icon appearances to be easily modified to incorporate shading and/or colouring, as well as scaling and various other translations. The font style of any text associated with items can also be modified for inconsistency presentation.

6. Experience

We have used the techniques described in this paper in several multi-view editing environments. Some of these environments are briefly described below. A brief discussion of users' perceptions of the techniques, some disadvantages we have encountered, and a comparison with existing environments and techniques are given.

---

[1] A C++ implementation of MViews is currently being developed

6.1. Environments

ISDEs built using the techniques described in this paper include SPE [11], EPE [1] and C-SPE [16]. EPE provides graphical EXPRESS-G views and textual EXPRESS views, and is used for product modelling design applications. C-SPE extends SPE to support collaborative software development using synchronous, semi-synchronous and asynchronous interaction techniques. SPE has recently been extended to incorporate textual Object-Z formal specification views [17], which are kept consistent with SPE implementation textual views and design graphical views by expanding change description representations.

Modelling environments and interface builders include MViewsER [14], MViewsDP [13], OOEER [14], and NIAMER [40]. MViewsER provides graphical views of ER data models and textual views of relational schema. MViewsDP provides graphical views of dialog specifications and textual views of dialog constraints. OOEER integrates SPE and MViewsER to provide an environment which supports truely integrated OOA/D and EER views. Changes to OOA/D views are propagated to EER views, and vice-versa, helping designers keep views of a common model consistent. NIAMER integrates MViewsER and MViewsNIAM, an environment which supports modelling using the NIAM notation. Changes to NIAM views are propagated to EER views, and vice-versa, helping designers keep views of a common model consistent.

Visual programming environments include ViTABaL [18], Skin [20], and HyperPascal [26]. These environments provide graphical views for specifying systems based on the tool abstraction paradigm (ViTABaL), flexible user interface components (Skin), and a visual Pascal (HyperPascal). Graphical views are kept consistent by representing inconsistencies via icon highlighting and change description representation in dialogs. Low-level textual views are kept consistent with aspects of the graphical views via change description expansion.

6.2. User Perception

Users of these environments like having view inconsistency representations to see and interact with [12, 11, 13]. When moving to a view, users are always informed of updates which potentially affect the view. Use of an appropriate way of presenting inconsistencies gives users immediate feedback on the accuracy and consistency of their views. For complex views, the presence of inconsistency presentations assists users in determing where in the view inconsistencies exist and how to go about resolving them. Hypertext facilities to navigate to related views (also supported by MViews) allow users to quickly navigate between inter-dependent, inconsistent views, allowing them to quickly resolve inconsistencies. Our techniques work equally well for small environments with relatively simple views, such as MViewsER and MViewsDP, and scale up for larger environments with many complex views, such as SPE, OOEER, NIAMER and EPE.

Environment implementers choose between using unobtrusive techniques to inform users of view inconsistencies, such as auto-expansion into textual views and highlighting, to presenting change description representations in dialogs immediately a view is selected. As automatic update of graphical and textual views is often possible, implementers can assist users by having an environment resolve inconsistencies whenever it can. We have found that it is often useful to still present representations of change descriptions, or highlight affected view components, even when inconsistencies have been sucessfully resolved. This allows users to confirm a correct modification has been done and these act as modification history documentation.

The main disadvantages of our approach have been encountered when environment implementers choose an inappropriate way of presenting or interacting with inconsistencies. For example, if a view is not used for some time, or a view is often affected by many other view updates, automatic expansion of change description representations into the view can result in large numbers of change descriptions, and users can become confused. It is more appropriate in these situations to highlight affected view components and indicate that change descriptions can be browsed in a dialog. This problem is compounded if an environment does not allow users to choose their own preferred mechanism for presenting or interacting with inconsistencies. Ideally environments should allow users to configure the view consistency management process.

It can be frustrating for users if some view inconsistencies can't be automatically resolved, when the user knows there is a simple resolution process the environment should carry out. This is a problem when the environment implementer does not know about this process, or the process is specific to a particular user. Once again users should be able to extend the environment's automatic resolution algorithms.

6.3. Comparison to Existing Approaches

Our inconsistency presentation and interaction techniques allow multiple, editable graphical and textual views of information to be automatically or partially kept consistent, with support for inconsistency management. Most other multi-view editing environments and models, such as the MVC [24], the ItemList [6], and Zeus [4], are not as flexible as they do not have inconsistency presentation. Our approach is useful where information shown at differing levels of abstraction must be kept consistent, as it supports "fuzzy" consistency management, where changes to one view impact on others so the environment can not automatically resolve inconsistencies. Other systems supporting multiple textual and graphical views of the same information, such as Dora [32] and PECAN [33], can not propagate such changes between views. Controlled inconsistency management between views is supported by retaining change descriptions denoting the inconsistencies, and users of the environment then determine when the inconsistencies are to be resolved. This a simple, elegant approach to inconsistency management, less cumbersome than the logical expressions denoting inconsistencies of [8].

Version control and cooperative work facilities are supported, as change description lists form a natural modification history for views and their components, and can be divided into versions. Broadcasting change descriptions and utilising appropriate presentation and interaction techniques facilitates synchronous and semi-synchronous cooperation that other cooperative environments, such as GroupKit [38], Mjølner [28], [31], and Mercury [22], do not support.

Our techniques are not confined to MViews-style free-edited view environments.The same problem of high-level view updates propagated to low-level views, and vice-versa, occurs in most multi-view editing environments. Structure-oriented environments, such as Dora [32], Mjølner [27], and the Cornel Program Synthesizer [37], could utilise these techniques when trying to maintain consistency between their views of information at differing levels of abstraction. In fact, automatic resolution of inconsistencies becomes easier in these environments, as the underlying structure of both graphical and textual views is readily accessible [2].

7. Summary

To achieve true consistency management in multi-view editing environments all view modifications must be propagated to other views which are affected by the change. Unlike existing environments and models, our approach supports full view consistency management by presenting users with representations of all view inconsistencies. Presentation mechanisms include descriptions of changes in dialogs, pop-up menus and textual views, view component highlighting, and lists of inconsistency resolution options. Users can interact with these inconsistencies by requesting automatic inconsistency resolution, choosing a resolution option, and manual resolution. Our technique also supports non-structural semantic and user-defined view inconsistencies, view versioning and cooperative work facilities, and the grouping of hierarchical, macro inconsistency descriptions. These techniques can all be efficiently implemented and packaged for reuse. Use of environments which exhibit these view consistency techniques indicates users prefer these environments to those which do not support inconsistency presentation and interaction.

We are extending many of our environments to support user-definable links between view components. These allow information items to be linked in arbitrary, user-defined ways, with the level of consistency management across these relationships mostly restricted to change description display and inconsistency highlighting. These will allow users to link information items in ways not conceived of by the environment designers, and yet allow the multi-view editing environment to maintain at least a basic level of (in)consistency management between them. In order to make multi-view editing environments more suitable for different users, environments need to support user-configurability. This includes the capability to allow users to specify their own preferred inconsistency presentation, interaction and automatic resolution techniques, both for single-user environments and multi-user, cooperative work environments.

References

1. Amor, R., Augenbroe, G., Hosking, J.G., Rombouts, W., and Grundy, J.C. Directions in modelling environments. *Automation in Construction* , 4 (1995), 173-187.

2. Arefi, F., Hughes, C.E., and Workman, D.A. Automatically Generating Visual Syntax-Directed Editors. *Communications of the ACM 33*, 3 (March 1990), 349-360.

3. Avrahami, G., Brooks, K.P., and Brown, M.H. A Two-View Approach to Constructing User Interfaces. *ACM Computer Graphics 23*, 3 (1990), 137-146.

4. Brown, M.H. Zeus: A System for Algorithm Animation and Multi-View Editing. In *Proceedings of the 1991 IEEE Workshop on Visual Languages*, IEEE Computer Society Press, 1991, pp. 4-9.

5. Cox, P.T., Giles, F.R., and Pietrzykowski, T. Prograph: a step towards liberating programming from textual conditioning, , IEEE Computer Society Press. In *Proceedings of the 1989 IEEE Workshop on Visual Languages*, 1989, pp. 150-156.

6. Dannenberg, R.B. A Structure for Efficient Update, Incremental Redisplay and Undo in Graphical Editors. *Software-Practice and Experience 20*, 2 (February 1990), 109-132.

7. Ellis, C.A., Gibbs, S.J., and Rein, G.L. Groupware: Some Issues and Experiences. *Communications of the ACM 34*, 1 (January 1991), 38-58.

8. Finkelstein, A., Gabbay, D., Hunter, A., Kramer, J., and Nuseibeh, B. Inconsistency Handling in Multiperspective Specifications. *IEEE Transactions on Ssoftware Engineering 2*, 8 (August 1994), 569-578.

9. Grundy, J.C. and Hosking, J.G. A framework for building visusal programming environments. In *Proceedings of the 1993 IEEE Symposium on Visual Languages*, IEEE Computer Society Press, 1993, pp. 220-224.

10. Grundy, J.C. *Multiple textual and graphical views for Interactive Software Development Environments*, Ph.D. dissertation, University of Auckland, Department of Computer Science, June 1993.

11. Grundy, J.C., Hosking, J.G., Fenwick, S., and Mugridge, W.B. Connecting the pieces, *Visual Object-Oriented Programming*, Prentice-Hall (1994), Chapter 11.

12. Grundy, J.C. and Hosking, J.G., "Constructing Integrated Software Development Environments with MViews," Working Paper, Department of Computer Science, University of Waikato, 1994.

13. Grundy, J.C., Hosking, J.G., and Mugridge, W.B. Supporting flexible consistency management via discrete change description propagation. to appear in *Software - Practice and Experience*.

14. Grundy, J.C. and Venable, J.R. Providing Integrated Support for Multiple Development Notations. In *Proceedings of CAiSE'95*, LNCS 932, Springer-Verlag, Finland, June 1995, pp. 255-268.

15. Grundy, J.C., Mugridge, W.B., Hosking, J.G., and Apperley, M.D. Coordinating, capturing and presenting work contexts in CSCW systems. In *Proceedings of OZCHI'95*, Wollongong, Australia, Nov 28-30 1995, pp. 146-151.

16. Grundy, J.C., Mugridge, W.B., Hosking, J.G., and Amor, R. Support for Collaborative, Integrated Software Development. In *Proceeding of the 7th Conference on Software Engineering Environments*, IEEE CS Press, April 5-7 1995, pp. 84-94.

17. Grundy, J.C., and Hosking, J.G. Support for Integrated Formal Software Development. In *Proceedings of APSEC'95*, Brisbane, Australia, December 1995, IEEE CS Press, pp. 264-273.

18. Grundy, J.C. and Hosking, J.G. ViTABaL: A Visual Language Supporting Design By Tool Abstraction. In *Proceedings of the 1995 IEEE Symposium on Visual Languages*, IEEE CS Press, 1995, pp. 53-60.

19. Hart, R.O. and Lupton, G. DECFUSE: Building a graphical software development environment from Unix tools. *Digital Tech Journal 7*, 2 (1995), 5-19.

20. Hosking, J.G., Fenwick, S., Mugridge, W.B., and Grundy, J.C. Cover yourself with Skin. In *Proceedings of OZCHI'95*, Wollongong, Australia, Nov 28-30, 1995, pp.

21. Kaiser, G.E. and Garlan, D. Melding Software Systems from Reusable Blocks. *IEEE Software 4*, 4 (July 1987), 17-24.

22. Kaiser, G.E., Kaplan, S.M., and Micallef, J., Multiuser, Distributed Language-Based Environments. *IEEE Software* (November 1987), 58-67.

23. Kiper, J.D. A framework for characterisation of the degree of integration of software tools. *Journal of Systems Integration 4* (1994), 5-32.

24. Krasner, G.E. and Pope, S.T. A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *Journal of Object-Oriented Programming 1*, 3 (1988), 8-22.

25. Linton, M., Vlissides, J.M., and Calder, P.R. Composing User Interfaces with InterViews. *COMPUTER 22*, 2 (1989), 8-22.

26. Lyons, P., Simmons, C., and Apperley, M. HyperPascal: Using visual programming to model the idea space. In *Proceedings of the13th New Zealand Computer Society Conference*, Auckland, August 1993, pp. 492-508.

27. Magnusson, B., Bengtsson, M., Dahlin, L. An Overview of the Mjølner/ORM Environment: Incremental Language and Software Development. In *Proceedings of TOOLS '90*, Prentice-Hall, 1990, pp. 635-646.

28. Magnusson, B., Asklund, U., and Minör, S. Fine-grained Revision Control for Collaborative Software Development . In *Proceedings of the1993 ACM SIGSOFT Conference on Foundations of Software Engineering*, Los Angeles CA, December 1993, pp. 7-10.

29. Meyers, S. Difficulties in Integrating Multiview Editing Environments. *IEEE Software 8*, 1 (January 1991), 49-57.

30. Narayanaswamy, K. and Goldman, N. Lazy consistency: a basis for cooperative software development. In *1992 ACM Conference on Computer-Supported Cooperative Work*, ACM Press, 1992, pp. 257-264.

31. Nascimento, C. and Dollimore, J. A model for co-operative object-oriented programming. *IEE Software Engineering Journa 8*, 1 (1993), 41-48.

32. Ratcliffe, M., Wang, C., Gautier, R.J., and Whittle, B.R. Dora - a structure oriented environment generator. *IEE Software Engineering Journal 7*, 3 (1992), 184-190.

33. Reiss, S.P. PECAN: Program Development Systems that Support Multiple Views. *IEEE Transactions on Software Engineering 11*, 3 (1985), 276-285.

34. Reiss, S.P. GARDEN Tools: Support for Graphical Programming. In *Proceedings of Advanced Programming Environments*, LNCS 244, Springer-Verlag, Trondheim, Norway, June 1986, pp. 59-72.

35. Reiss, S.P. Connecting Tools Using Message Passing in the Field Environment. *IEEE Software 7*, 7 (July 1990), 57-66.

36. Reiss, S.P. Interacting with the Field environment. *Software practice and Experience 20*, S1 (June 1990), S1/89-S1/115.

37. Reps, T. and Teitelbaum, T. Language Processing in Program Editors. *COMPUTER 20*, 11 (November 1987), 29-40.

38. Roseman, M. and Greenberg, S. Groupkit: A groupware toolkit for building real-time conferencing applications. In *Proceedings of CSCW'92*, ACM Press, 1992, pp. 43-50.

39. *TurboCASE Reference Manual*, StructSoftInc, 5416 156th Ave. S.E. Bellevue, WA, 1992.

40. Venable, J.R. and Grundy, J.C. Integrating and Supporting Entity Relationship and Object Role Models. In *Proceedings of the 14th Object-Oriented and Entity Relationship Modelling Conferece*, LNCS 1021, Springer-Verlag, 1995.

41. Vlissides, J.M. and Linton, M. Unidraw: A framework for building domain-specific graphical editors. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, ACM Press, 1989, pp. 158-167.

42. Wasserman, A.I. and Pircher, P.A. A Graphical, Extensible, Integrated Environment for Software Development. *SIGPLAN Notices 22*, 1 (January 1987), 131-142.

43. Welsh, J., Broom, B., and Kiong, D. A Design Rationale for a Language-based Editor. *Software - Practice and Experience 21*, 9 (1991), 923-948.

Working Paper Series
ISSN 1170-487X

Coordinating Collaborative Work in an Integrated Information Systems Engineering Environment

by John C Grundy, John R. Venable, John G. Hosking, and Warwick B. Mugridge

Working Paper 96/5

March 1996

Department of Computer Science
The University of Waikato
Private Bag 3105
Hamilton, New Zealand

Coordinating Collaborative Work in an Integrated Information Systems Engineering Environment

John C. Grundy†, John R. Venable†, John G. Hosking†† and Warwick B. Mugridge††

† Department of Computer Science
University of Waikato
Private Bag 3105, Hamilton, New Zealand
{jgrundy,jvenable}@cs.waikato.ac.nz

††Department of Computer Science
University of Auckland
Private Bag, Auckland, New Zealand
{john,rick}@cs.auckland.ac.nz

Abstract. The development of complex Information Systems requires the use of many Information Systems engineering tools. These diverse tools need to be integrated in order to be effectively used by multiple cooperating developers. In addition, the users of these environments require features that facilitate effective cooperation, such as support for collaboratively planning cooperative work, notification of changes to parts of a system under development (but only when necessary or desired), support for keeping aware of other developers' work contexts, and the ability to flexibly engineer or adapt development processes and methods. We describe an integrated Information Systems engineering environment which includes a work coordination tool supporting these requirements.

1. Introduction

Development of complex Information Systems requires the use of many Information Systems engineering tools. These include CASE tools, databases and programming languages, documentation tools, version control systems and project management tools. Effective use of these tools together requires their integration. Ideally tools should be integrated in several ways, including: having a centralised data repository; supporting control and event flow between tools; having a consistent set of tool user interfaces; and allowing software processes to effectively use the tools together [3].

The use of such integrated tools by multiple developers introduces a further requirement for process integration - tool use needs to be coordinated [18]. Developers need to: collaborate to plan their work; know what other developers are doing or have done at various times and to various things; be informed of changes other developers are making to artefacts they are using or interested in; and need to collaboratively modify their work processes (and work plans) during development to suit the needs of the system under construction. Each of these tasks needs to be coordinated at a high level of abstraction in order to facilitate the development of complex systems by multiple developers.

We have developed a tool integration mechanism allowing a wide variety of IS development tools to be integrated [8, 34, 9]. This supports: data integration, by integrating repositories or linking them together and keeping their data consistent; control integration, by propagating events between tools; and user interface integration, by ensuring a consistent user interface is provided for all tools. We have now developed a powerful process integration mechanism which uses a visual work planning language to specify collaborative work plans and to capture modification histories for plan stages. A tool for constructing these plans is integrated with other integrated IS development tools. This supports the capture and presentation of work contexts, which are associated with descriptions of data changes in a tool, and these work contexts are presented with the changes to other developers, helping to coordinate tool use. Modification of work plans during development is supported, facilitating collaborative planning and method engineering.

The following section gives an overview of related research into IS Engineering Environment (ISEE) integration and tools which try to facilitate collaborative work and coordination of work. We then describe our integrated environment from a user's perspective and explain its need for a work coordination layer. This coordination layer is described, and its use for work planning, capture and presentation of work contexts, and specification of interest in changes is illustrated. We then show its usefulness for collaborative planning and method engineering. The architecture and implementation of our tool are briefly described and our research and current and future work summarised.

2. Related Research

Tools supporting different aspects of IS development can be used in isolation, but this results in much redundancy, an inability to effectively share data, and inconsistent user interfaces [23, 3]. Tools supporting different aspects of Information Systems development need to be integrated into a single ISEE which overcomes these problems [23, 26, 24]. Recent research into integrating such environments has focused on control, data, user interface and process integration [3, 23]. PECAN [25], MELD [15], and the Cornell Program Synthesiser [28] utilise a large centralised database to store shared information with restrictive structure editors allowing modification of views of this data. FIELD environments [26, 27] utilise selective broadcasting of events between Unix tools to achieve limited forms of control and user interface integration. Dora environments [24] integrate multiple textual and graphical views of software

development via a restrictive structure editing approach. CASE tools utilise code generation and reverse engineering but only partially keep design and code consistent [35, 32]. Federated approaches use a database, such as PCTE, spread over several locations and which may also utilise hetrogeneous data [3], but these often lack adequate user interface consistency and collaborative work and coordination facilities.

Many collaborative environments and CASE tools only support low-level editing mechanisms [1], including most Groupware systems [5], Mercury [16], Mjølner [21] and C-SPE [10]. As these systems do not facilitate coordination of work, nor the capture and presentation of work context information, effective collaborative work on large systems is not possible. Some groupware systems support limited group awareness capabilities, such as multiple cursors [29], but these usually only inform collaborators about the work artefacts collaborators are immediately interested in; they do not provide the context of others' work and are unable to delay informing collaborators of changes until a later time, when it is needed.

Process-centred environments utilise information about software processes to enforce or guide development. Examples include Marvel [2], CPCE [20], and ConversationBuilder [17]. These environments usually provide low-level text-based descriptions of work rationale, and often do not effectively handle restructuring of development processes while in use [33]. Computer-Aided Method Engineering (CAME) tools, such as Decamerone [14] and Method Base [30], provide support for configuring development processes and tools to a particular application, but often utilise complex textual specifications and don't facilitate work coordination itself.

Workflow-based systems, such as Active Workflow [22] and Domino [19], attempt to coordinate work by describing the flow of documents between collaborators. The workflow approach has proven to be inadequate for most real-world coordination activities. Exceptions to the workflows usually outnumber cases when they are useful, and the workflows often need to be modified while in use [33]. In addition, such systems usually do not model nor facilitate collaboration on the coordination (planning) activity itself [33]. Workflow systems and process-centred environments often have only a tenuous integration with the development tools, and thus the history and context of work artefact changes is difficult or impossible to obtain and to present to collaborating users.

3. An Integrated ISEE

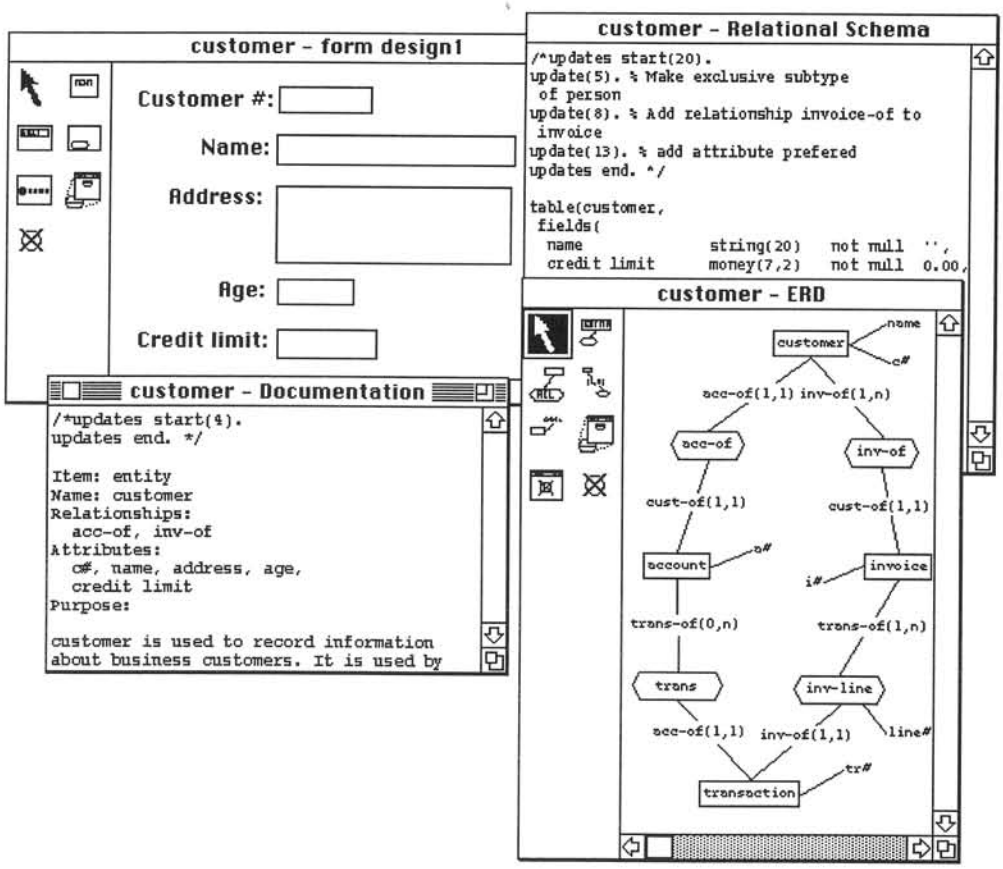


Figure 1. An integrated Information Systems Development environment

We describe an ISEE which incorporates several disparate IS development tools. These include tools for entity-relationship and relational schema modelling [8], form and report design [10], NIAM modelling [34], object-oriented analysis, design and implementation [7], and system documentation [7, 8]. Figure 1 is a screen dump from this environment showing some of the views of IS development it provides.

Window 'customer - ERD' is an entity-relationship diagram showing the major entities and relationships for an invoicing system, and window 'customer - Relational Schema' shows a relational schema for the customer entity. Both of these views are provided by the MViewsER ERD/schema modelling tool [8]. Window 'customer - form design1' shows a form specified for customer based on the customer schema, and these views are provided by the MViewsDP tool for form/report design [10]. Window 'customer - Documentation' shows a documentation view for the customer entity describing its relationships, attributes and purpose. This view is provided by the SPE tool for object-oriented software development, which also provides object-oriented analysis, design and implementation views [7]. Other views supported by our environment but not shown include NIAM views from the MViewsNIAM tool [34], and debugging views provided by the CernoII debugger [7]. We are currently implementing DFD views which will also be integrated into the environment.

It is not sufficient just to integrate tools in terms of their user interface (presentation) [23, 26]. Our integrated tools also support integrated repositories, and information in these repositories and in the tool views is kept consistent when the same or related information in other views is modified. For example, when an entity is modified in an ER view, this change is propagated to other affected views which include this entity information. We utilise the consistency management schemes used within the tools in our environment to also provide inter-tool consistency [7, 8, 10].

A consistency management example is shown in figure 2. The updates made to the customer entity in the ER view are propagated to the schema view and shown as *change descriptions* in the textual view header. These inform a developer of the changes made to the customer entity and that may require updates to the schema definition. Some are easy to make and can be automatically made by the environment (for example, adding, renaming or deleting attributes or changing attribute types). Others are more difficult to automatically implement, such as adding new relationships or changing relationship arity, and the environment leaves these modifications up to the developer. Similarly, changes need to be reflected in the form design view, and these are indicted in a dialog when the view is selected. Some can be automatically actioned, as is shown by the new attribute prefered having an edit box automatically added to the form design, but this still needs designer intervention to move it to an appropriate position (hence the greyed outline).

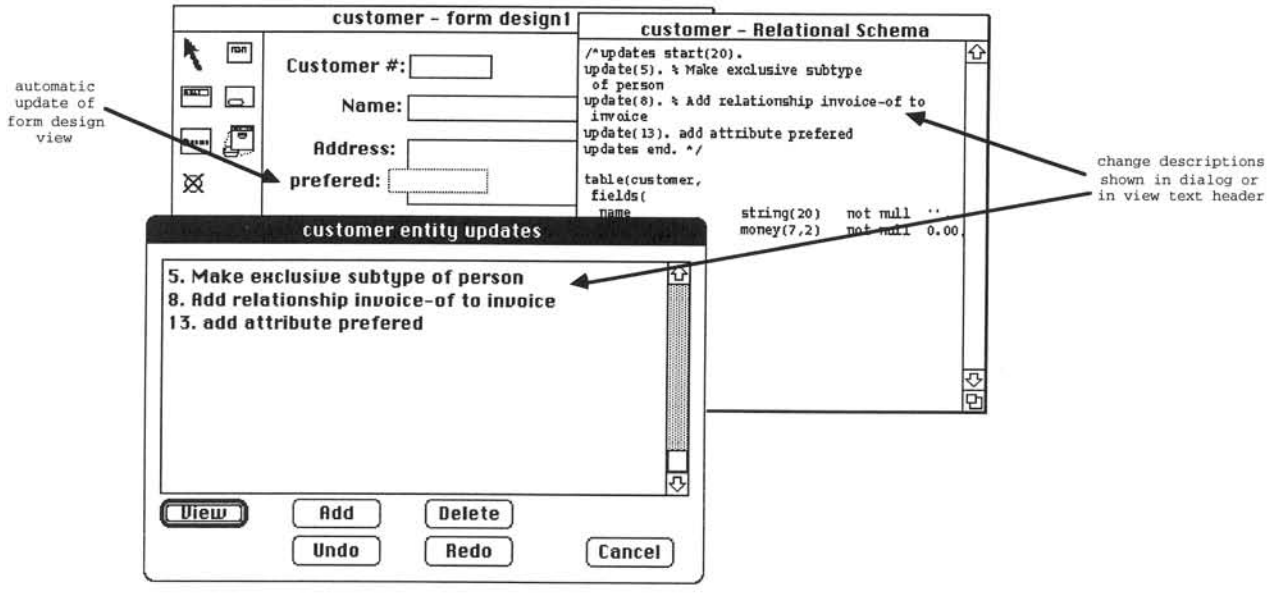


Figure 2. Keeping different views of development consistent.

In addition to the integrated tools and consistency management described above, our environment supports version control and collaborative facilities. Modification histories for all views and software components rendered in views are kept, recording every change made to these artefacts. These histories are partitioned into versions, with version revision, alternates and merging supported. As the storage of these software views and entities is in a shared repository, the versioning facilities support asynchronous collaborative development [11]. In addition, semi-synchronous and synchronous editors are provided for views. Semi-synchronous editing broadcasts change descriptions to all interested

collaborating developers and these are presented to them in dialogues or in view text. Synchronous editing allows developers to simultaneously edit the same version of a view [11].

4. A Work Coordination Layer

While these basic collaborative editing facilities are useful, they are not by themselves sufficient to support large groups of cooperating IS developers. A key problem of systems providing only this low-level coordination of work is the lack of information about the context that work artefact changes have been carried out in [12]. The collaborating user is not told why these changes have been carried out, only the sequence they were carried out in. No support is provided for planning work together nor for grouping changes into histories based on particular tasks and subtasks.

In large CSCW systems, such as collaborative ISEEs, coordination of cooperative work activities is needed [18]. Users must collaborate in the planning of work activities and be aware of the contexts in which other users' work is carried out. Support is needed for defining activities to be done (plans), coordinating the planning activity itself (meta-plans), and restructuring the history of work done to more effectively convey intent ("rewriting history"). The capture of work context and rationale information should be as unobtrusive as possible [4].

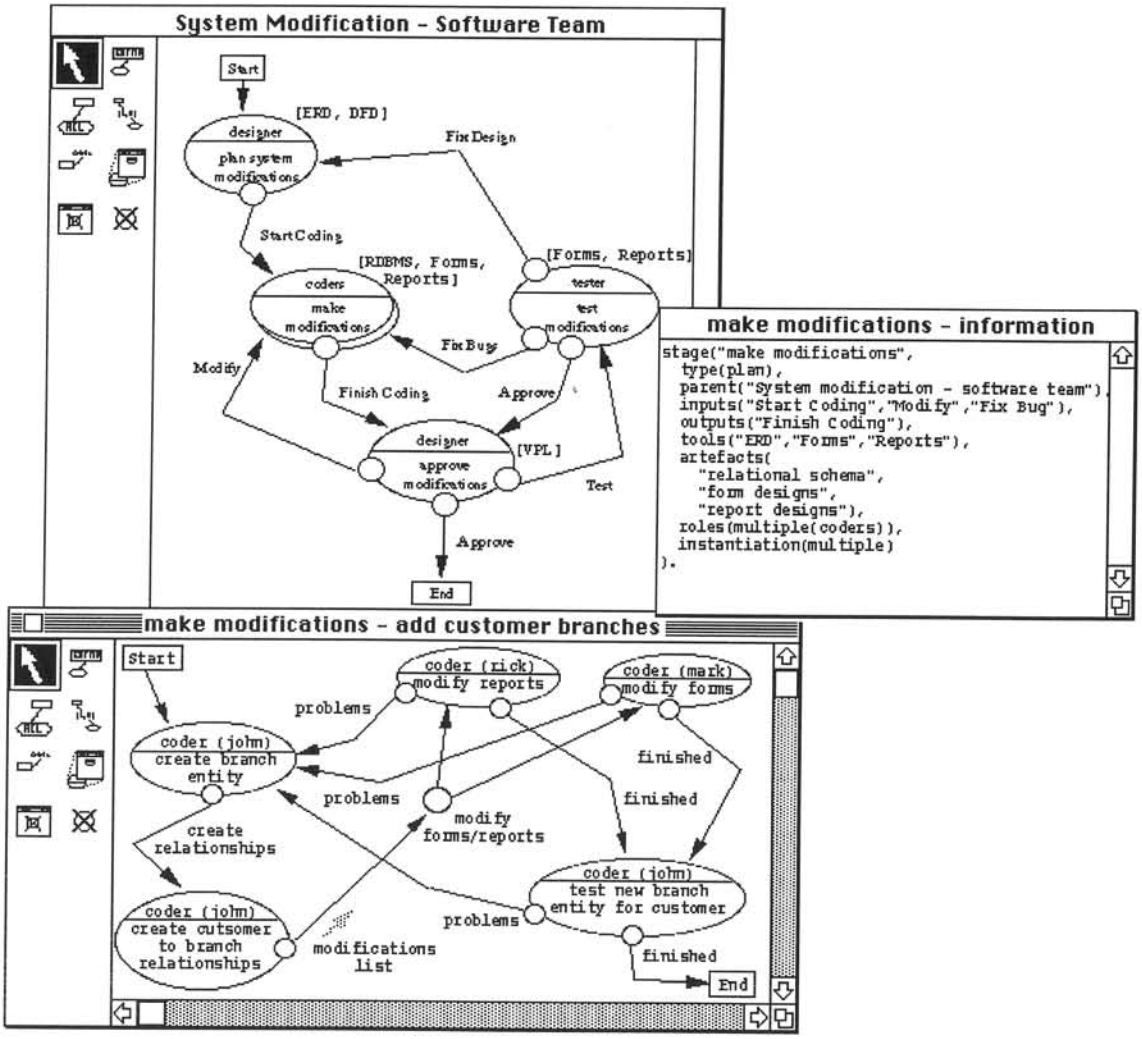


Figure 3. VPL+ views describing some IS development processes.

Rather than the inadequate workflow or process model approaches used by many systems, we have adapted the Visual Planning Language (VPL) [33] to define the context of work and planning activities. VPL allows the definition of plans and subplans for work tasks, and can be used to specify meta-plans for the planning process itself. Useful plans can be abstracted out into policies (basically software process models), which can then be instantiated into further plans. VPL supports flexible restructuring of plans while in use, and partially defined plans can be actioned and later completed or modified. The history of what actually happened when using a plan can be used to restructure the plan to better document

the process and to refine it into a policy. We have extended VPL to produce VPL+, in which a plan stage includes extra information about the work artefacts, CASE tools and collaboration mechanisms used.

Two VPL+ plan diagrams for a software process are shown in figure 3. VPL+ elements include: stages (steps in the process), denoted by ellipses and which include a role and stage description; split stages, which are duplicated for each person involved in the stage; and options, denoted by circles on the stage perimeter, which are used to specify the next plan stage(s). Extra annotations describe tools, work artefacts and communication mechanisms used during and between plan stages. These VPL+ elements are "plan artefacts", which can be modified in the same way work artefacts are modified. Plans are defined by users via synchronous, semi-synchronous or asynchronous editors. Plan modifications can be coordinated via meta-plans (themselves VPL+ plans).

In figure 3, the plan specifies a software process for modifying a software system. A subplan for the "make modifications" stage specifies steps for a particular system enhancement, in this case the extension of the customer entity to include multiple branches per customer. A plan history view is associated with each actioned plan, containing descriptions of changes made to both plan and software artefacts in accordance with this plan. Textual views describing extra information about the purpose of individual plan stages are provided. These views describe which tools are used to carry out the work and which artefacts are used by the plan stage. VPL+ views contain plan artefacts, as opposed to the work artefacts in our ISEE, forming a work articulation layer [31].

5. Capturing and Presenting Work Contexts

The notation for designing plans is used to display and manipulate plans in action. Active stages for a plan are highlighted, options interacted with via menus to advance plan stages and obligations specified between plans and plan stages to ensure users of related plans are informed of changes. The plan view thus specifies the current work context for each member in a collaborating team, and members may have more than one plan view open to see the status of other plans they are interested in or working with. When making work or plan artefact changes, information about the context of this work is captured and presented to interested collaborating users. This information includes who made a change, what was changed, when it was changed, why it was changed and the work or planning context the change was made in. We briefly describe this work context presentation/capture mechanism below. Further details can be found in [12].

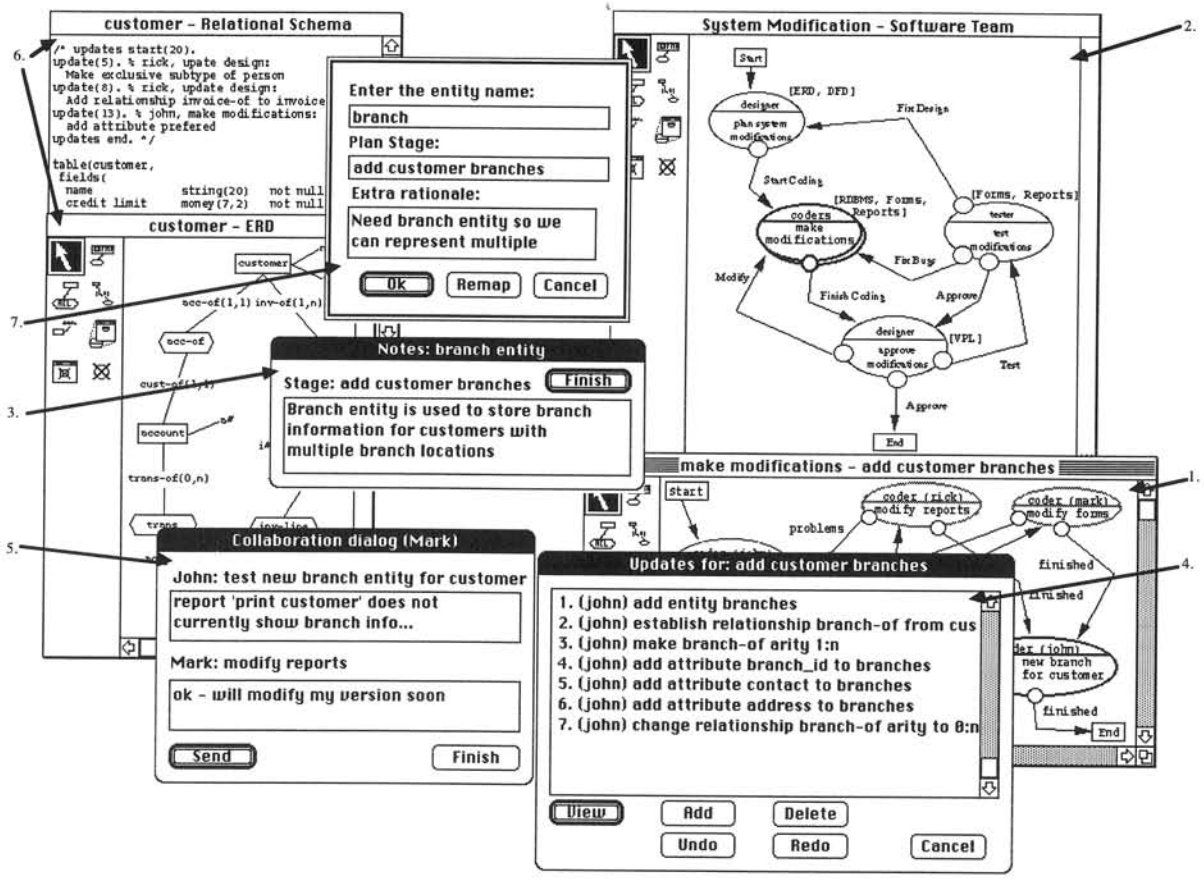


Figure 4. Capturing work context information.

Figure 4 shows how different views can be used to capture work contexts when using the ISEE views from Section 3 (note that a developer would not normally have all of these views open at once!). A "current plan" VPL+ view (1) shows information about a developer's current active stage status (bold border) and indicates the current plans of other collaborators (shaded borders). This captures the current work context for the artefact changes the developer is making. Other shared VPL+ plan views (2) show other plan stages the developer is interested in and also highlight active stages of other collaborators. Highlighting changes when active stages or the shared plan itself change. Artefact notes dialogs (3) specify extra documentation about individual work or plan artefacts, and are shared between all collaborators. Shared modification histories (4) for the current plan stage, or other plan stages the developer wants to review the history of, detail changes that have been made for a task. As plan or work artefact changes are generated, descriptions of these changes are forwarded to the appropriate current plan stage and stored to document the stage's subplan history. These historical changes may also be used to "rewrite history" if the plan is restructured. A collaboration dialog (5) is used for informal, context-dependent dialog between developers. Work artefact views (6) are graphical or textual ISEE editors which capture information about changes made to artefacts. Plan artefact views are manipulated in the same manner, co-ordinated by meta-plans. Augmented artefact dialogs (7) allow a developer to optionally specify extra rationale for low-level work or plan artefact changes. These reasons are stored with the plan stage history and communicated to collaborators.

The current context of work is modified by advancing VPL+ active stages. Incomplete VPL views can be extended as users become more familiar with their actual work processes, unlike many existing workflow and process-centred collaborative systems. Plans can also be restructured while in use, using meta-plans to coordinate this process. Plan history items can be split up if plan stages are split, or moved to other stage histories if plans are deleted.

Collaborating users must be informed of changes to work and plan artefacts they are interested in [12]. In most CSCW environments this amounts to presenting only artefact-level information to collaborators, either directly updating their work artefact views or using version control facilities to indicate changes made by other users. Our approach provides collaborating users not only with change descriptions describing actual work or plan artefact changes, but also with extra information about the work context the changes were carried out in. Presentation of work context information to collaborators is done in a similar way to the capture of work context information. Collaborators share many work and plan artefact views, and will have available the same kinds of views as shown in 4. Some changes a developer makes are directly relevant to their collaborators, such as renaming or deleting entities and attributes, and collaborators should be informed of these immediately. Other changes, such as the addition of new entities, relationships, attributes or forms and reports can be sent for later perusal, as they have more limited effects on collaborators' work. Low-level changes, such as the implementation of procedures, forms or reports not affecting a collaborator's work need not be presented. Collaborators can see from plan histories and various active stages the kinds of activities a developer is doing, and may choose to view these changes or modified artefacts on-demand.

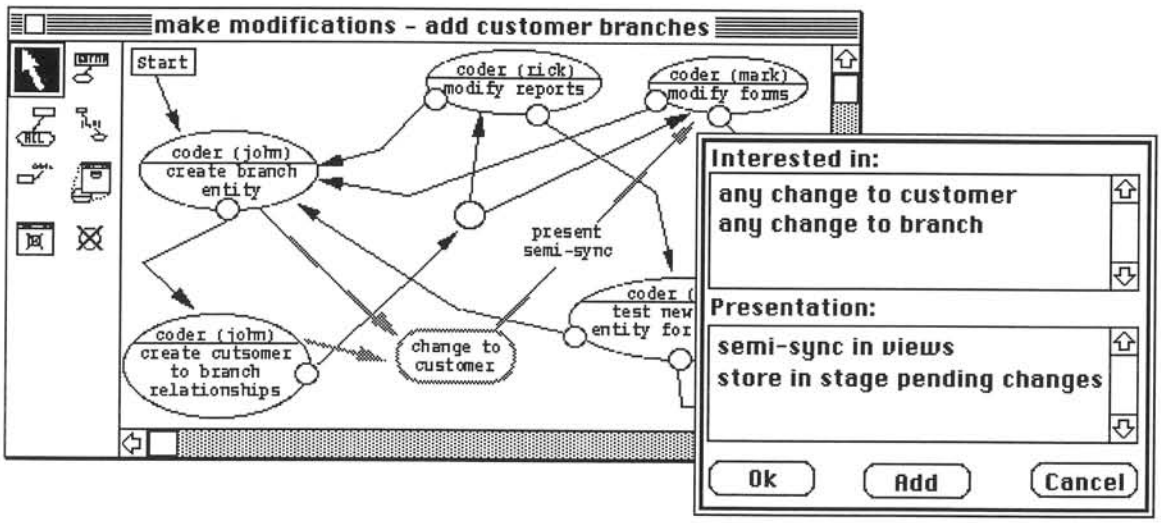


Figure 5. Describing interest in collaborators' work.

Collaborators are informed of changes in various ways. They have similar views to those in Figure 4, including: their own current plan view (1) and other shared VPL+ views (2), both which may indicate other developers' current work contexts, and any change to these contexts (new stage, changes made, change to shared plan(s)). Changes to shared work and plan artefact notes (3) are presented as they are made. Extra comments can be added by a collaborator, which are sent back to other collaborators. Changes to notes are documented in plan histories, and thus can also be viewed asynchronously on-demand. Collaborators can monitor work artefact changes semi-synchronously in a plan history

dialog (4). Informal messages are displayed in a collaboration dialog (5). and these can be responded to as they are received. Shared work artefact views (6) can be edited synchronously, semi-synchronously or asynchronously. Changed items in these views can also be highlighted, allowing collaborators to view the change history for the view or its work contexts' plan histories on-demand. Additional information is presented with change descriptions in work artefact views, including the work context (plan stage) the changes were made in. Hypertext links from these change descriptions allow quick access to plan histories and VPL+ views. Modification histories for individual work or plan artefacts also display the work context each artefact change was made in. Users can thus review change histories by grouping changes by artefact (artefact histories) or by grouping changes by work context (plan histories).

Collaborators need to be able to specify which changes they are interested in seeing and when and how they should be presented [12]. Our extensions to VPL include annotations for specifying obligations between plan stages, to allow users to register interest in particular kinds of plan or work artefact changes, and for defining extra change description processing. Figure 5 illustrates basic annotation indicating which changes to detect. Extra information about how to present these changes to a user can be specified in a dialog. This notation allows users to specify interest in either particular or general kinds of work or plan artefacts. Changes to these artefacts or instances of these artefact types are then communicated to the collaborating users as they occur. Note that many of these interest specifications can be inferred from collaborator plan stage information (for example, if collaborators are using the same artefact and tool but different artefact versions, semi-synchronous editing can be defaulted).

6. Collaborative Planning and Method Engineering

Current Information Systems development methodologies are generally situation-independent. However, researchers have found that due to the increasing complexity of Information Systems, development teams often require methods tailored to particular system development [14]. Our work coordination views assist in Situational Method Engineering [13] by allowing developers to incrementally refine their development methodology and plans. As plan stages record information about the tools to use, artefacts to modify/produce, subsequent plan stages, and also may be exploded into more detailed plans, they facilitate the engineering of software processes in a manner similar to Method Engineering tools.

Our approach has some advantages over comparable notations, such as MEL [14], in that its visual nature is more accessible to developers for visualising and modifying plans than the textual notations of other approaches. As VPL was designed for general work process modelling [33], its high-level nature allows developers to more readily understand and modify process descriptions than text-based process-centred environments or method-engineering tools. It also allows users to modify their plans while the plan is in use, and our VPL+ editors allow users to restructure plans and plan histories after completion so new, improved policies (i.e. software process models) can be developed for reuse.

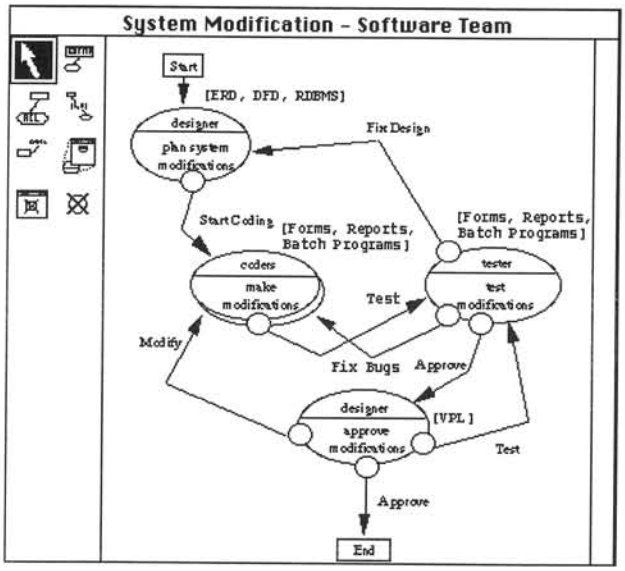


Figure 6. Method engineering techniques.

By providing collaborative editors for VPL+ views, and allowing other VPL+ views to act as meta-plan views (i.e. to plan and record the planning process itself), collaborative planning is supported. Collaborating developers share software process plans and can collaborate on modifying these plans. The use of these shared plans for work context capture and presentation, and specifying interest in changes, allows our VPL+ tools to be used for work coordination, collaborative planning, recording development histories, and method engineering. Figure 6 shows an example of method engineering

with VPL+ views. The System Modification plan (software process model) has now been collaboratively updated to use slightly different plan stages and tools. Database modifications are now done during the "plan modifications" stage (the "RDBMS" tool is now used in this stage and not during the "make modifications" stage), batch programs are now modified and tested, and the subsequent step to "make modifications" is now "test modifications", not "approve modifications" as previously. Any subsequent development using this policy will now use this modified software process.

Our integrated ISEE allows different tools to be used on the same problem domain, with tool data being kept consistent under change and the tools sharing a consistent user interface. Our VPL+ tool allows software processes to be reconfigured during development to better suit a particular development project. Software process models thus evolved can be reused in subsequent development by abstracting them into policies. The specification of artefacts, roles, CASE tools and interest obligations for plan stages gives our integrated environment similar method engineering capabilities to method engineering tools, in addition to its support for work coordination.

7. Design and Implementation

The individual ISEE tools are implemented as a collection of classes, specialised from the MViews framework [6, 10]. MViews supports the construction of Integrated Software Development Environments (ISDEs) by providing a general model for defining software system data structures and tool views, with a flexible mechanism for propagating changes between software components, views and distinct software development tools. MViews describes ISDE data as *components* with *attributes*, linked by a variety of *relationships*. Multiple views are supported by representing each view as a graph linked to the base software system graph structure. Each view is rendered and edited in either a graphical or textual form. Distinct environment tools can be interfaced at the view level (as editors), via external view translators, or multiple base layers may be connected via inter-view relationships.

When a software or view component is updated, a *change description* is generated. This is of the form UpdateKind(UpdatedComponent, ...UpdateKind-specific Values...). For example, an attribute update on Comp1 of attribute Name is represented as: update(`Comp1,Name,OldValue,NewValue`). All basic graph editing operations generate change descriptions and pass them to the propagation system. Change descriptions are propagated to all related components that are dependent upon the updated component's state. Dependents interpret these change descriptions and possibly modify their own state, producing further change descriptions. This change description mechanism supports a diverse range of software development environment facilities, including semantic attribute recalculation, multiple views of a component, flexible, bi-directional textual and graphical view consistency management, a generic undo/redo mechanism, and component "modification history" information.

New software components and editing tools are constructed by reusing abstractions provided by an object-oriented framework. ISDE developers specialise MViews classes to define software components, views and editing tools to produce the new environment. A persistent object store is used to store component and view data.

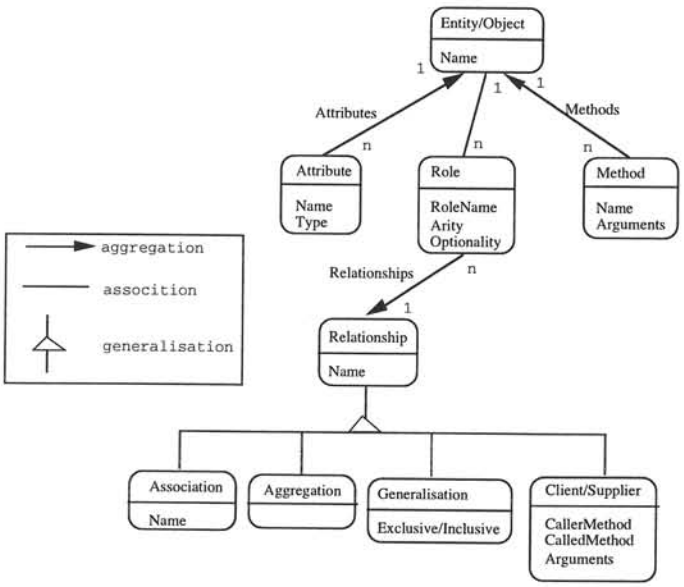


Figure 7. A tool meta-model.

We have developed a technique of integrating tools for multiple design notations using hierarchical, integrated repositories [8]. Figure 7 shows an example of a meta-model of an integrated tool repository for OOA and EERD notations (from [8]). We have recently extended this approach so that information in one tool repository (such as relational schema items) can be linked to similar items in another tool's repository (such as form components).

As these repository items generate change descriptions when modified, inter-repository relationships detect these change descriptions and use them to keep information between the repositories consistent. Figure 8 shows the architecture of our ISEE. Some tools are integrated at the repository level by having an integrated repository [8]. Others have links between repository items which keep related items consistent. Figure 8 shows an example of how data from quite disparate tools is kept consistent: 1) an object attribute is renamed in the OOA/D/P tool, generating a change description. 2) The OO repository is modified and a change description sent to the integrated OOEER tool repository (see [8] for details). 3) the ER tool repository is updated so the OO object's corresponding entity is modified. 4) a change description is sent to the form/report designer repository, updating corresponding form/report fields based on the entity attribute. 5) The form designer views are updated.

We have integrated the SPE OOA/D/P, MViewsER, MViewsNIAM, and MViewsDP tools to produce our integrated ISEE. We are currently implementing a DFD modelling tool and extending SPE to support functional and dynamic models. These behavioural modelling notations will then be integrated and different notations kept consistent using an integrated repository.

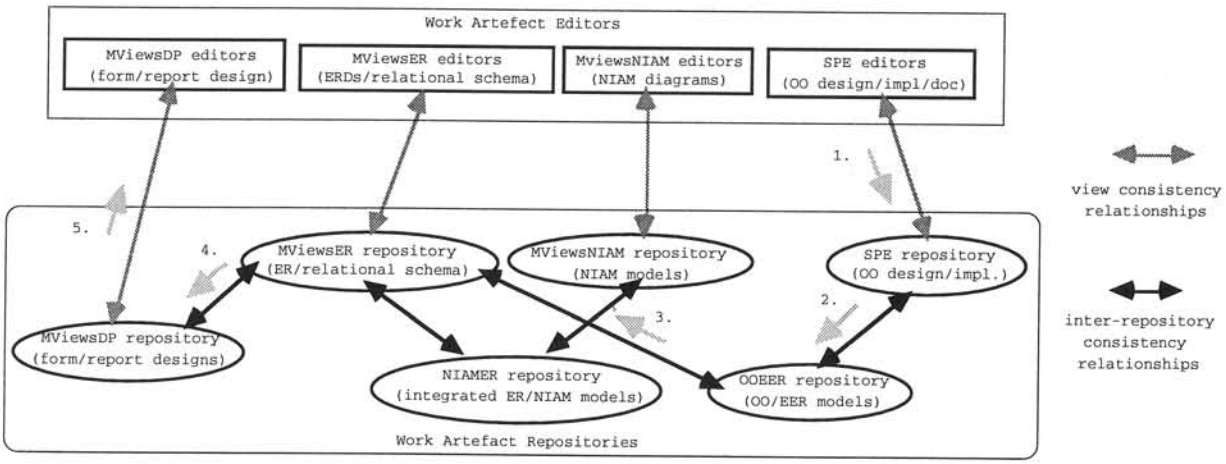


Figure 8. The architecture of the integrated tools.

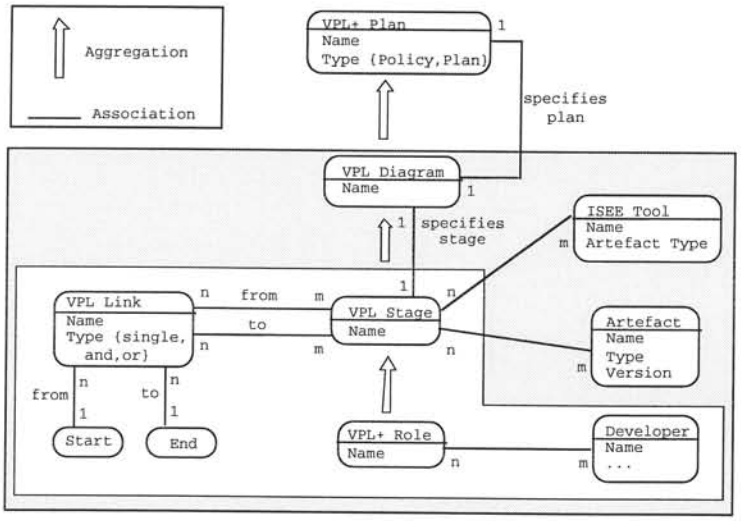


Figure 9. A VPL+ meta-model.

VPL+ views require a repository which defines VPL+ components (plan stages) and detailed information about plan stages (tools used, artefacts, etc.). Figure 9 shows a meta-model of VPL+ describing this information. A VPL+ plan is made up of VPL diagrams, themselves composed of plan stages and links. A plan stage has roles, ISEE tools and work

artefacts. Links can be single (1 stage to 1 stage), or multiple and- or or-dependency links between plan stages. We are currently implementing a VPL+ tool using MViews, which will form the coordination and method engineering layer for our integrated ISEE.

The approach we are taking to integrating our VPL+ tool and ISEE is to have coordination (VPL+) artefacts receive change descriptions from updated work artefacts. Work context information is associated with these change descriptions, then they are propagated to interested collaborators' views for presentation (or they may be stored for later presentation) [12]. Figure 10 shows how ISEE and VPL+ artefacts will interrelate: 1) if a developer modifies a work artefact, a change description is sent to the current VPL+ plan stage. 2) The plan stage augments the change description with work context information, and then forwards this to the work context (current plan stage) of collaborators interested in the change. Additional processing of the change description may also occur here, for example automatically advancing stages, or causing updating of plan or work artefacts. 3) The augmented change description is presented to collaborators in an appropriate manner to inform them of both changes to work artefacts and collaborating users' work contexts, thus facilitating the coordination of work.

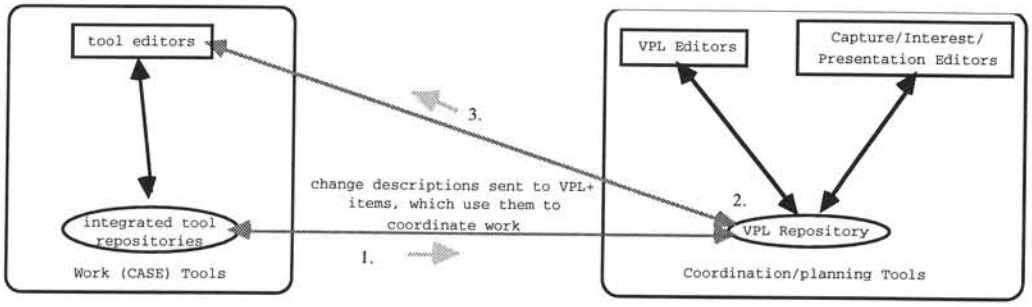


Figure 10. Integrating VPL+ views with integrated tool views.

8. Summary

We have described an integrated environment for Information Systems development which includes work coordination facilities to help manage large systems development. Integrated development tools allow developers to design and build complex Information Systems using a variety of integrated tools, with full data, control and presentation integration between the tools. A coordination layer using a modified version of the Visual Planning Language (VPL) is used to achieve process integration between tools. This includes the ability to coordinate work via work context capture and presentation, collaboratively develop, refine and reuse complex work plans (i.e. software process specifications), and use plans to document development via plan histories.

We are currently implementing VPL+ views to form a coordination layer for our prototype ISEE which incorporates integrated EER, OOA/D, NIAM, relational schema, form and report design, and documentation views. VPL+ views provide a work coordination layer and facilitate collaborative planning and method engineering. We are continuing to extend VPL+ to make it easier for collaborators to specify interest in changes, presentation mechanisms, and to visualise inter-plan stage communication mechanisms, artefact and tool usage, and developer roles. We are also developing a true MetaCASE environment which uses meta-models of notations as CASE tool repository specifications, and allows MViews environments to be more declaratively specified and generated.

References

1. Aean, I., Siltanen, A., Sørensen, C., and Tahvanainen, V.P. A Tale of Two Countries: CASE experiences and expectations. In *Proceedings of the IFIP WG8.2. Working Conference on The Impact of Computer Supported Technologies on Information Systems Development,* Kendall, K.E., DeGross, J.I., and Lyytinen, K. Eds, Minneapolis, June 14-17 1992, North-Holland.

2. Barghouti, N.S. Supporting Cooperation in the Marvel Process-Centred SDE. In *Proceedings of the 1992 ACM Symposium on Software Development Environments,* ACM Press, 1992, pp. 21-31.

3. Bounab, M. and Godart, C. A Federated Approach to Tool Integration. In *Proceedings of CAiSE'95,* Finland, June 13-16 1995, LNCS 932, Springer-Verlag, pp. 269-282.

4. Cockburn, A. and Jones, S. Four Principles for Groupware Design. In *Proceedings of OZCHI'94,* 1994, pp. 21-26.

5. Ellis, C.A., Gibbs, S.J., and Rein, G.L. Groupware: Some Issues and Experiences. *Communications of the ACM 34*, 1 (January 1991), 38-58.

6. Grundy, J.C. and Hosking, J.G. A framework for building visusal programming environments. In *Proceedings of the 1993 IEEE Symposium on Visual Languages,* Bergen, Norway, IEEE CS Press, 1993, pp. 220-224.

7. Grundy, J.C., Hosking, J.G., Fenwick, S., and Mugridge, W.B. Connecting the pieces, Chapter 11 in*Visual Object-Oriented Programming,* Burnett, M., Goldberg, A., Lewis, T., Manning/Prentice-Hall (1995).

8. Grundy, J.C. and Venable, J.R. Providing Integrated Support for Multiple Development Notations. In *Proceedings of CAiSE'95,* Finland, June 1995, LNCS 932, Springer-Verlag, pp. 255-268.

9. Grundy, J.C., and Venable, J.R. Developing CASE tools that support integrated design notations. In *Proceedings of the 6th European Workshop on Next Generation of CASE Tools,* Finland, June 1995, pp. 109-116.

10. Grundy, J.C., Mugridge, W.B., Hosking, J.G., and Amor, R. Support for Collaborative, Integrated Software Development. In *Proceeding of the 7th Conference on Software Engineering Environments,* Nordwijkerhout, Netherlands, April 5-7 1995, IEEE CS Press, pp. 84-94.

11. Grundy, J.C., Hosking, J.G., and Mugridge, W.B., "Supporting flexible consistency management via discrete change description propagation," Working Paper, no. 95/2, Department of Computer Science, University of Waikato, 1995.

12. Grundy, J.C., Mugridge, W.B., Hosking, J.G., and Apperley, M.D. Coordinating, capturing and presenting work contexts in CSCW systems. to appear in *Proceedings of OZCHI'95,* Wollongong, Australia, Nov 28-30 1995.

13. Harmsen, F., Brinkkemper, S., and Oei, H. Situational Method Engineering for Information System Projects. In *Proceedings of the IFIP WG8.1 Working Conference CRIS'94,* Olle, T.W. and Verrijn, A.A. Eds, Maastricht, 1994, North-Holland, Amsterdam, pp. 169-194.

14. Harmsen, F., and Brinkkemper, S. Design and Implementation of a Method Base Management System for a Situational CASE Environment. to appear in *Proceedings of the 2nd Asia-Pacific Software Engineering Conference (APSEC'95),* IEEE CS Press, Brisbane, December 1995.

15. Kaiser, G.E. and Garlan, D. Melding Software Systems from Reusable Blocks. *IEEE Software 4*, 4 (July 1987), 17-24.

16. Kaiser, G.E., Kaplan, S.M., and Micallef, J., Multiuser, Distributed Language-Based Environments. *IEEE Software* (November 1987), 58-67.

17. Kaplan, S.M., Tolone, W.J., Carroll, A.M., Bogia, D.P., and Bignoli, C. Supporting Collaborative Software Development with ConversationBuilder. In *Proceedings of the 1992 ACM Symposium on Software Development Environments,* ACM Press, 1992, pp. 11-20.

18. Krant, R.E. and Streeter, L.A. Coordination in Software Development. *CACM 38*, 3 (March 1995), 69-81.

19. Kreifelts, T., Hinrichs, E., and Klein, H.K. Experiences with the Domino Office Procedure System. In *Proceedings of the Second European Conference on Computer Supported Cooperative Work (ECSCW'91),* 1991, pp. 117-130.

20. Lonchamp, J. CPCE: A Kernel for Building Flexible Collaborative Process-Centred Environments. In *Proceedings of the 7th Conference on Software Engineering Environments,* Nordwijkerhout, Netherlands, April 5-7 1995, IEEE CS Press, pp. 95-105.

21. Magnusson, B., Asklund, U., and Minör, S. Fine-grained Revision Control for Collaborative Software Development. In *Proceedings of the1993 ACM SIGSOFT Conference on Foundations of Software Engineering,* Los Angeles CA, December 1993, pp. 7-10.

22. Medina-Mora, R., Winograd, T., Flores, R., and F., F. The Action Workflow Approach to Workflow Management Technology. In *Proceedings of CSCW'92,* ACM Press, 1992, pp. 281-288.

23. Meyers, S. Difficulties in Integrating Multiview Editing Environments. *IEEE Software 8*, 1 (January 1991), 49-57.

24. Ratcliffe, M., Wang, C., Gautier, R.J., and Whittle, B.R. Dora - a structure oriented environment generator. *IEE Software Engineering Journal 7*, 3 (1992), 184-190.

25. Reiss, S.P. PECAN: Program Development Systems that Support Multiple Views. *IEEE Transactions on Software Engineering 11*, 3 (1985), 276-285.

26. Reiss, S.P. Connecting Tools Using Message Passing in the Field Environment. *IEEE Software 7*, 7 (July 1990), 57-66.

27. Reiss, S.P. Interacting with the Field environment. *Software practice and Experience 20*, S1 (June 1990), S1/89-S1/115.

28. Reps, T. and Teitelbaum, T. Language Processing in Program Editors. *COMPUTER 20*, 11 (November 1987), 29-40.

29. Roseman, M. and Greenberg, S. Groupkit: A groupware toolkit for building real-time conferencing applications. In *Proceedings of CSCW'92,* ACM Press, 1992, pp. 43-50.

30. Saeki, M., Iguchi, K., Wen-yin, K. A Meta-model for representing software specification and design methods. In *Proceedings of the IFIP WG8.1 Conference on Information Systems Development,* Prakash, N., Rolland, C., and Pernici, B. Eds, Como, 1993.

31. Schmidt, K. and Bannon, L. Taking CSCW seriously: Supporting Articulation Work. *Computer Supported Cooperative Work (CSCW): An International Journal 1*, 1-2 (1992), 7-40.

32. *TurboCASE Reference Manual*, StructSoftInc, 5416 156th Ave. S.E. Bellevue, WA, 1992.

33. Swenson, K.D. A Visual Language to Describe Collaborative Work. In *Proceedings of the 1993 IEEE Symposium on Visual Languages,* Bergen, Norway, 1993, IEEE CS Press, pp. 298-303.

34. Venable, J.R. and Grundy, J.C. Integrating and Supporting Entity Relationship and Object Role Models. to appear in *Proceedings of the 14th Object-Oriented and Entity Relationship Modelling Conferece,* Gold Coast, Australia, Dec 13-15 1995, LNCS, Springer-Verlag.

35. Wasserman, A.I. and Pircher, P.A. A Graphical, Extensible, Integrated Environment for Software Development. *SIGPLAN Notices 22*, 1 (January 1987), 131-142.

Working Paper Series
ISSN 1170-487X

Conservative Parallel Simulation of ATM Networks

by John G Cleary and Jya-Jang Tsai

Working Paper 96/6

March 1996

Department of Computer Science
The University of Waikato
Private Bag 3105
Hamilton, New Zealand

Conservative Parallel Simulation of ATM Networks

John G. Cleary, Jya-Jang Tsai,
Department of Computer Science, University of Waikato, New Zealand.
email: {jcleary,jjtsai}@cs.waikato.ac.nz

Abstract

A new conservative algorithm for both parallel and sequential simulation of networks is described. The technique is motivated by the construction of a high performance simulator for ATM networks. It permits very fast execution of models of ATM systems, both sequentially and in parallel. A simple analysis of the performance of the system is made. Initial performance results from parallel and sequential implementations are presented and compared with comparable results from an optimistic TimeWarp based simulator. It is shown that the conservative simulator performs well when the "density" of messages in the simulated system is high, a condition which is likely to hold in many interesting ATM scenarios.

1 Introduction

ATM is a recent standard for broadband communications [3,7]. It is a rate independent system which uses fixed size (53 byte) *cells* for all transmission. The motivation of this work is the need to simulate large numbers of cells in ATM systems. Such simulations, which need to explicitly model the passage of cells through the fabric of switches and over the physical communications links, are particularly demanding for a number of reasons. There are indications that it is necessary to simulate in the order of 10^9 to 10^{12} cells in one run to measure some effects such as cell loss rates. Also the models of ATM at the individual cell level have very low compute grains. For example ATM-TN a detailed ATM model produced by the Telesim project [15] has average per simulation event granularities of 17 μseconds on a Sparc-2 55MHz CPU [11]. These granularities are close to the per event overheads that can be expected from sequential simulators. For example, the sequential simulator used to obtain the granularities has a measured per event overhead in the range of 11 to 18 microseconds (depending on the size of the event list).

The need for large numbers of high fidelity simulation events motivates the requirements for a high performance simulator. The low granularities imply that any simulator must be very efficient with per event overheads close to those of the best sequential simulators. Results reported later will show that expected simulation scenarios have significant available parallelism. Thus there is a strong motivation to construct parallel simulators to take advantage of this. The difficulty here is to keep the overheads low enough that the parallelism can be effectively used. To this end the conservative simulator is designed for a shared memory multiprocessor environment. This follows the lead of recent TimeWarp parallel simulators [8].

Any such simulation splits into two main parts: the models of the ATM switches, access points and links; and the models of the traffic generators. This paper is concerned mainly with the interior ATM objects and not so much with the traffic generators. There are two reasons for this: the ATM part is by far the most compute intensive; and it is more stereotyped and consistent than the generators.

ATM-TN is a generic cell-level ATM model [15]. The model deals with individual cells and passage through switches. As well it includes high-level models of TCP/IP, World-Wide-Web traffic, and self-similar traffic [1,2]. The current work is based around this model.

The proposed algorithm belongs to the class of conservative simulations. These, by definition, block until a process can ensure that it will not violate causality by processing the next event [10]. Nil messages, as proposed by Chandy and Misra [5,13], provide a method for communicating processes to exchange information regarding the lower bound on the time stamps of future messages. The effectiveness of nil messages depends greatly on the amount of lookahead available [9]. This is apparently application dependent [14,16]. In any case, the possible imposition of extra overhead by the use of nil

messages has caused much criticism of the usefulness of such approaches.

Several mechanisms based on nil messages have been developed since it was first introduced. Chandy and Sherman [6] present an alternative. Instead of nil messages, a conditional message is sent to the receiving process only when the sender will otherwise become blocked. As long as there are (real) messages waiting in the receiver's input buffer, conditional events are not used. Also, Jha and Bagrodia [12] suggested a scheme combining the above approach and conventional nil messages. Cai and Turner [4] proposed a "carrier null message" approach. In this approach, an additional field is included to the nil message so that a process can identify a nil message initiated by itself. When such a message is detected, the blocking is ended. The effectiveness of this approach depends on how much earlier such a carrier can be detected compared to regular nil messages. Although performance speedup can be observed in certain cases with these approaches, the applicability of these to large complicated communication networks (eg ATM) remains unknown.

In the next section the simulation algorithm itself is described, together with both sequential and parallel implementations. Section 3 focuses on a key data structure and its optimisation. Section 4 considers the theoretical performance of the conservative simulator and compares this with event based simulators including TimeWarp. As well actual performance results from a toy problem on a distributed version of the conservative simulator are reported together with comparable results from a TimeWarp based simulator. Initial results from a full port of ATM-TN are also provided. The paper concludes with a summary.

2 Simulator

In the simulator all *processes* receive (0 or more) *input links* and generate (0 or more) *output links*. *Messages* are sent down links between processes. Each message has a *send* and *receive* time. The critical part of the proposed algorithm is that each time a process is executed it merges all its input lists and executes one *event* for each incoming message, this continues until one of the input links is empty where-upon the process *suspends*. Because the merge is done in time order the process suspends on the earliest empty link, the earliest time that a message can be received down that link is the process's *current time* (CT).

The generic simulation algorithm is:

```
while there are messages in system
   select next process to execute;
   while lowest time stamped
    incoming link has a message
    select lowest message;
    process message;
   end while;
   set current time of process to
    minimum time on any in-link;
   update last time on all outgoing
    links to minimum possible
    receive time of next message (will
    be ≥ current time + lookahead of
    link);
   suspend process;
end while;
```

This algorithm is different from many event driven simulators, in that, when a process is selected for execution it consumes all of its incoming messages before suspending. That is, scheduling is not done on an event by event basis. In the worst case no event will be executed when a process is selected, although the current time of the process may be advanced which will cause the time on the output links to also advance. Such an "empty" execution corresponds roughly to a nil message in a Chandy-Misra conservative simulator [5] and to the conditional messages used in [12]. Many algorithms are possible for selecting the next process to be executed. The aim is to minimise the cost of scheduling both by minimising the number of suspension/scheduling steps and by minimising the cost of suspension and scheduling.

It is assumed that the links are monotonic - that is messages are received in the order of their receive times which is the same order that they are sent (physically this can be interpreted as "messages cannot pass each other in the links"). Of concern is the *lookahead* of each link which is the minimum difference between the receive and send time of a message. This must be positive (it may be 0 in some cases), and in many interesting cases will be non-zero.

Consider a simulation model as a directed graph where the links are arcs and the processes are nodes. The arcs are labelled with the lookahead of the link. For any loop (sequence of nodes and arcs that arrives back where it starts) the *loop time* (LT) is the sum of the lookaheads along the loop. The *Minimum Lookahead Time* (MLT) for a link or process is the minimum LT of any loop which passes through the link or process. I will assume that no process (and by implication link) has a zero MLT. In particular all ATM models will be seen to have non-zero MLTs.

The algorithm above can be executed by randomly selecting a process, executing it, and then selecting the next process and so on. So long as this process is fair - that is every process is eventually selected for execution - then it is easily shown that the simulator will make progress (the GVT or minimum time of any unprocessed message in the system will increase). The problem is to minimise the overheads of suspension, that is, the aim is to maximise the number of events executed and minimise the number of suspensions.

2.1 Sequential Execution

The first (sequential) algorithm always schedules (one of) the processes with the smallest CT. In such a simulator the best that any process can do is to advance its simulation time by its MLT on each execution/suspension cycle[1]. If the MLT of the i'th process is given by MLT_i then a lower bound for the average number of suspensions per simulation time unit, C, will be:

$$C \geq \sum_i \frac{1}{MLT_i} \qquad (2.1)$$

The process of scheduling on the basis of the CT is susceptible to many optimisations. Given that the processes can be scheduled at random, clearly scheduling on the basis of the CTs can be sloppy and still the algorithm will work. Note that selecting for execution on the basis of CT is different from using an event list (in particular, the times used for scheduling may correspond to no actual events in the system and the number of entries in the scheduling list is always equal to the number of processes in the system).

2.2 Static Schedules

Another possible algorithm is to have a loop in which each process is executed exactly once. This will clearly be sub optimal if the processes have differing MLTs - those with large MLTs will be executed too often. However, it seems that static schedules in which a process appears in inverse proportion to its MLT can approach close to the lower bound. The example systems in Figure 1 have close to optimum static schedules. In Fig 1a the MLTs for the three processes are: A=1, B=2, C=3. From formula (2.1) the lower bound for the number of suspensions per unit of simulation time is C=1.83.. Using either of the static schedules below achieves C=2..:

C B A A C B A A ..

or

C B A A B A C B A A B A ..

For Fig 1b the MLTs are: A=1, B=3, C=3 and the optimum of C=1.67.. is actually attained by

C B A A A C B A A A ...

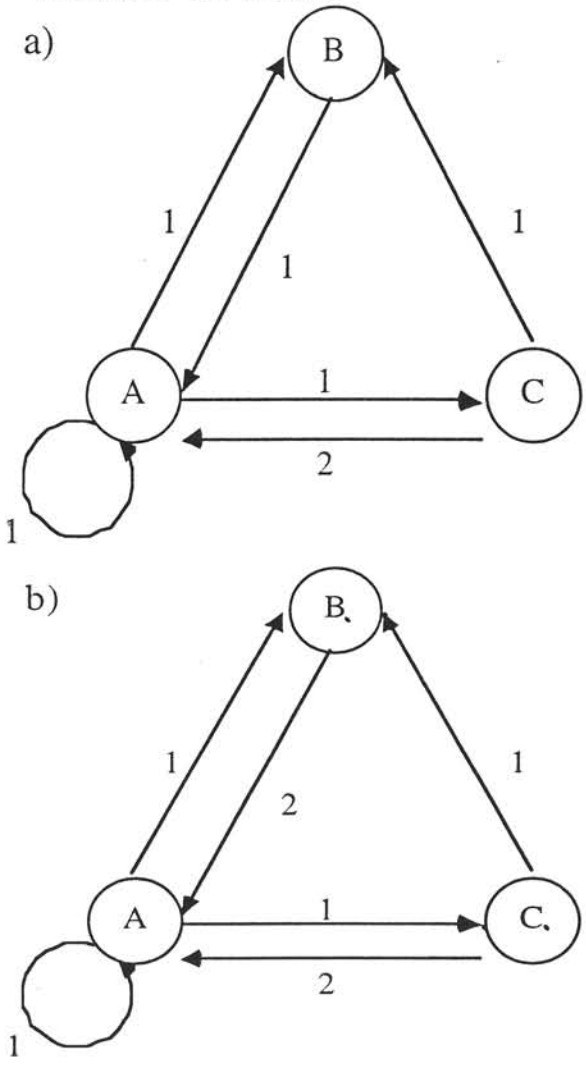


Figure 1. Example Process/Link Graphs.

Clearly the overheads of such static schedules can be made very small. It is not necessary to maintain an event list or any dynamic data structure. For small numbers of processes they could be directly compiled as an unrolled sequence of procedure calls.

[1]Consider the loop passing through the process that has the minimum LT. Assume that all the incoming links along the loop other than the one under consideration have minimum receive times that are infinite (or sufficiently far in the future that they do not contribute to the minimum). Then the minimum receive time on the incoming link will be equal to the sum of the lookaheads around the loop, that is, the MLT. It is this time that will limit the advance of the process when it is executed.

2.3 Parallel Execution

Consider a distributed system where the processes are each mapped to a processor. A static schedule can be followed locally on each processor. In the limit when each process is mapped to one processor the static schedule will be the repeated execution of the same process. In this case the process is effectively polling its input links.

For ATM systems this parallel case (with all processes mapped to their own processor) may be **more** efficient than the sequential one. The critical process, that is the one that on average runs the slowest may never suspend, as the other (lightly loaded) processes will run faster than the critical one and so the critical process will always have messages on all its input links. Thus the critical part of the computation will be essentially free from any suspension overheads, it will spend all its time executing in the user modelling code.

3 Link Data-structures

A key data-structure in any simulator based on these ideas is the one that holds the messages in the links. It should allow fast insertion and extraction of events and also be able to be shared between processes in a parallel system. The sharing is made easier by the fact that only two processes ever access the link - the sender and the receiver. The rest of this section consists of a series of refinements of an initial simple data structure. The refinements are designed to give greater efficiency and to provide effects such as bounding the time advance of the sending process. It is notable that the resulting data structure does not need any locking to work correctly, thus avoiding one important source of overheads.

3.1 Linked-list

Figure 2 gives a description and pseudo-code for a simple version of the link data structure. It uses single unidirectional pointers between the messages in the link together with two external pointers into the link: Top and Bottom.

This code works correctly even in the case where the sender and receiver are running in parallel. In this case `Top` is modified only by the sender and is read by the receiver. `Bottom` is used only by the receiver. To ensure correct parallel execution the order of assignments in `send` is important, in particular `Top` must be assigned after all the other fields have been set up. As well storing a pointer or a time-stamp must be an atomic operation.

3.2 Optimized

The first optimisation is to make the length of the list in the link constant. This achieves two things: it removes the overheads of allocating and deallocating links and it also allows the length of the list to be used as a flow control mechanism. For example, a process with no input links (a message generator) will have to suspend if its output links become full. Fixing the length allows the `next` fields to be initialised so that the list forms a circular loop and thus `next` need never be updated. There is one major change needed to the code for `send` to ensure that `Top` does not over-run `Bottom` (that is that too many messages are not inserted onto the link). This also means that the sender must be prepared to deal with the `send` function returning an indicator that the message cannot be sent - in this case the sender must suspend. The other code change is to `initialise` which now creates a fixed loop of links. This code works correctly in parallel.

The final optimisation is to put all the links adjacent to one another in a single block of memory. The main advantage of this is that it minimises cache misses when accessing the messages. The only code that needs to be changed is the initialise function (this is left as an exercise for the reader).

Two further refinements have been added to the implementation. First, links from an LP back to itself are treated specially - it is not necessary to delay on them when they are empty. Second, LPs forming a cycle of small LT can be clustered into a single LP. This optimisation has great impact on ATM-TN because a switch in ATM-TN is modeled by a number of LPs, which exchange control signals with small lookaheads forming low MLT loops.

4 Performance

4.1 Comparison With Event Driven Simulators

It is possible to make a simple comparison between the amount of computing expected in the type of conservative system outlined above and other event driven simulators such as TimeWarp. Let E be the maximum number of events possible in one (simulated) time unit, χ the cost of doing one suspension, ε the cost of executing one event and λ be a parameter that expresses the fraction of the maximum possible number of messages that are actually sent. Then the total cost of advancing one simulated time unit in the conservative scheme is given by the expression:

```
Top: pointer to next location to have a message stored;
Bottom: pointer to next message to be received;
structure link:
        time: receive time of message (or earliest time of next
                message if pointed at by Top);
        next: pointer to next message (undefined if
                pointed at by Top);
        data: user defined content of message (undefined if
                pointed at by Top);
end structure;
initialise: %Initialise link.
        t:= new link;
        t.time:=0;
        t.next := nil;
        Top:=t;
        Bottom:=t;
end initialise;
send(receive_time,next_time,msg): %Send a message.
next_time is the earliest possible receive time of the next message;
        Top.data:=msg;
        Top.time:= receive_time;
        t:= new link;
        t.time:= next_time;
        Top.next := t;
        Top:= t;
        send := true;
end send;
early_time(time): %Update earliest possible receive time:
        Top.time:= time;
%Receive a message - returns true if Bottom is left pointing at a valid message - in any
%case Bottom.time is left as the earliest time a message can arrive down the link;
receive:boolean;
        if Bottom = Top then
                receive:=false;
        else
                t:=Bottom;
                Bottom:=Bottom.next;
                deallocate t;
                receive:= boolean:(Bottom<>Top);
        end if;
```

Figure 2. Code for Link Datastructure.

$$t_c = \lambda \varepsilon E + \chi C$$

For an event driven system the cost is proportional only to the number of events:

$$t_o = \lambda \varepsilon E$$

Clearly as λ decreases the event driven system will eventually win out over the conservative as the costs of an event driven system are proportional to the number of events. The per event overheads of the conservative simulator will be inversely proportional to λ - an effect which is clearly seen in the empirical results below. This analysis is generous to real parallel event driven systems such as TimeWarp where the per event overheads will be higher than the conservative system because of the need to do state saving and rollback.

4.2 Empirical Results

The conservative simulator was implemented on a SPARC-1000 platform with 8 55MHZ Sparc processors. An initial series of experiments were conducted to collect performance measurements from the proposed mechanism in order to compare against a standard event list based

sequential simulator, as well as a parallel simulator based on Time Warp. The benchmark used was a 4-dimensional hypercube communication network. Upon receiving a message, a node forwards the message, with an exponential delay with mean 1.2, to a neighbouring node. The selection of a destination node is random. The transit time on each link is set constant to 1.2 time units. As in ATM systems it is assumed that a message prevents another message being sent for one time unit. The experiments use various message populations, ranging from 1 to 1000. Thus the number of pending events per process will vary from 1/16 to 64.

The event granularity for the basic sequential simulation varied from 11 µsecs to 18 µsecs (for a large event list). It is difficult to separate the user code and the simulator overheads but the user code takes about 4 µsecs (mainly for random number generation). A spin-loop is also employed in some cases to investigate the impact of larger event grains.

The average execution time per event versus the message population is shown for a single processor in Fig. 4a. Following the theoretical predictions above, the execution time per event for the conservative simulator increases as the message popu lation decreases. The execution time for the sequential and TimeWarp simulators show an opposite trend increasing slightly as the message population increases - this is probably caused mainly by an increase in the size of the event list (the sequential simulator uses a splay tree and the TimeWarp simulator a calendar queue).

The other three subgraphs of Fig 4 show the overhead on 2, 4 and 8 processors. The performance of the conservative simulator improves smoothly in the parallel execution. Interestingly the TimeWarp simulator has difficulties at low message populations on the parallel runs. There has not been a chance to investigate this more closely but it may indicate the onset of some form of dynamic instability.

The results from the proposed mechanism are encouraging. Even at very low granularities the conservative mechanism shows speedup over a significant range of message populations. Preliminary estimates indicate that average message populations in ATM simulations will be well above the cross-over where the conservative system performs better than TimeWarp.

Fig. 5a shows the absolute speedup of the conservative simulator and Time Warp compared to the sequential simulator without any added granularity. Four different message populations, namely, 20, 100, 200 and 1000, were used. For large message populations, the proposed mechanism shows a speedup of 3.96 when running on 8 processors. Even with moderate message population, eg 100-200, the speedup observed on 8 processors is 2.82. Interestingly, the single processor version of the distributed conservative algorithm shows a slow down of about 37% with a message population of 1000. In principle it should run at about the same speed as the sequential simulator (indeed a sequential implementation of the conservative simulator runs slightly faster than the standard sequential simulator for large message populations). The problem seems to lie in the implementation of the threads package on the Solaris operating system. This requires a system call to access memory which is local to a thread. The structure of the C++ system we are using forces such a call on every send. This seems to account for the slow down (investigations are proceeding on how to avoid this overhead).

In Fig. 5b-d, the same set of experiments are repeated, but with different event granularities. Each figure shows the speedup when a spin-loop of granularity around 10 µS, 100 µS, 1000 µS is added to each event (in addition to the original computation of the application). As seen from the figures, when the granularity increases, the speedup increases. Even with a small increase in event granularity (10µS), the speedup is elevated to about 4.91 on 8 processors when message population is high. Recent timing results on an implementation of the ATM-TN model indicates that the average event grains are about 17µS slightly higher than the grain used in Fig 5b. Time Warp also improves its performance drastically on larger event grains, leading to smaller differences between the two techniques. The results here are kind to TimeWarp in that the added compute grains include no provision for any additional state saving overhead.

4.3 ATM-TN Port

The full ATM-TN model has been ported to the a parallel version of the new simulator with encouraging preliminary results. A benchmark consisting of a perfect-port switch connecting three endnodes has been used to test the system. Two of the endnodes have an ethernet traffic source/sink modules running. The switch has three ports, at 45 Mbps each. The collected data in Table 1 shows that when the load on ethernet is low (λ=0.045), both the sequential simulator and the Time Warp simulator outperformed the conservative algorithm. However, as the ethernet load increased(λ=0.11 and 0.18), the execution time per event for the conservative simulator decreases greatly while the Time Warp based simulator and sequential simulators remain the same. The real strengh of the conservative simulator appears on executing on multiple processors. In one experiment, the speedup (vs. sequential simulator) on 4 processors is close to 2.5. Further experiments are planned for more realistic

ATM networks. The following table shows the execution time per event (in μS) for the different simulators.

λ	con	con (4)	TW	TW (4)	Seq
0.045	32.02	15.69	40.16	27.22	25.37
0.110	26.53	12.01	42.40	27.06	25.52
0.180	24.74	11.80	42.15	26.97	28.07

Table 1. Execution Times for ATM-TN

5 Summary

A new shared memory based conservative simulator has been proposed and implemented. It is capable of exhibiting high efficiencies with overheads on a parallel implementation less than twice that for an optimised sequential simulator. It is well suited to its proposed domain of application in ATM models with its high efficiency overcoming problems of very low compute grain sizes. The parallel simulator compares well with a TimeWarp simulator over a wide range of parameters.

The conservative algorithm relies on a number of features of ATM networks to achieve good performance:

- all nodes (switches) have a small number of incoming and outgoing links;
- the capacity of links is limited by the (fixed) length of cells (this ensures a minimum lookahead for links);
- most "interesting" simulations will have links loaded close to capacity.

Clearly the algorithm is not universal as there will be problems where other mechanisms including TimeWarp will outperform it, however, it is well suited to ATM models and similar communications systems which are demanding and economically important.

Acknowledgments

This work was supported by grant 95-UOW-S14-4321 from the New Zealand Public Good Science Fund and by Science Applications International Corp. We would like to thank Xiao Zhonge and the rest of Telesim team for providing us with both sequential and parallel simulators and the ATM-TN model.

References

1. Arlitt, M., and Williamson, C.(1995a) "*A Synthetic Workload Model for Internet Mosaic Traffic*," Proc. Summer Computer Simulation Conf., Ottawa, June.
2. Arlitt, M., Chen, Y., Gurski, R., and Williamson, C.(1995b) "*Traffic Modelling in the ATM-TN Telesim Project*," Proc. Summer Computer Simulation Conf., Ottawa, June.
3. Boudec, J.L.(1992) "*The Asynchronous Transfer Mode: a Tutorial*," Computer Networks and ISDN Systems, pp. 279-309.
4. Cai, W., and Turner, S.J.(1990) "*An Algorithm for Distributed Discrete Event Simulation*," Proc. Distributed Simulation Conference, San Diego California, pp. 3-8, JanuaryJanuary.
5. Chandy, K.M., and Misra, J.(1979) "*Distributed Simulation: a case study in design and verification of distributed programs*," IEEE Trans.Software Eng., **5**(5), pp. 440-452, September.
6. Chandy, K.M., and Sherman, R.(1989) "*The conditional event approach to distributed simulation*," Proc. Distributed Simulation Conference, San Diego, California, pp. 93-99, March.
7. Comm. ACM.(1995) "*Special Edition on Issues and Challenges in ATM Networks*," Comm. A.C.M., February.
8. Das, S., Fujimoto, R., Panesar, K., Allison, D., and Hybinette, M.(1994) "*A Time Warp System for Shared Memory Multiprocessors*," Winter Simulation Conference, December.
9. Fujimoto, R.M.(1988) "*Performance measurements of distributed simulation strategies*," Proc. Distributed Simulation Conference, San Diego, California, pp. 14-20, February.
10. Fujimoto, R.M.(1990) "*Parallel Discrete Event Simulation*," Comm. A.C.M., **33**(10), pp. 30-53, October.
11. Jade Simulations International Corp.(1995) "*Deliverable for ATM-TN Performance Project*," Science Applications Internationl Corp., August.
12. Jha, V., and Bagrodia, R.L.(1993) "*Transparent implementation of conservative algorithms in parallel simulation languages*," Winter Simulation Conference, Los Angeles, pp. 677-686, December.
13. Misra, J.(1986) "*Distributed Discrete-Event Simulation*," ACM Computing Surveys, **18**(1), pp. 39-65.
14. Nicol, D.M.(1988) "*Parallel discrete-event simulation of FCFS stochastic queuing networks*," ACM SIGPLAN Notices, **23**(9), pp. 124-137.
15. Unger, B.W., Gomes, F., Zhonge, X., Gburzynski, P., Ono-Tesfaye, T., Ramaswamy, S., Williamson, C., and Covington, A.(1995) "*A High Fidelity ATM Traffic and Network Simulator*," Winter Simulation Conference, Washington, D.C., December.
16. Wagner, D., and Lazowska, E.(1989) "*Parallel simulation of queuing networks: limitations and potentials*," Proc. International Conf. on Measurement and Control, pp. 146-155.

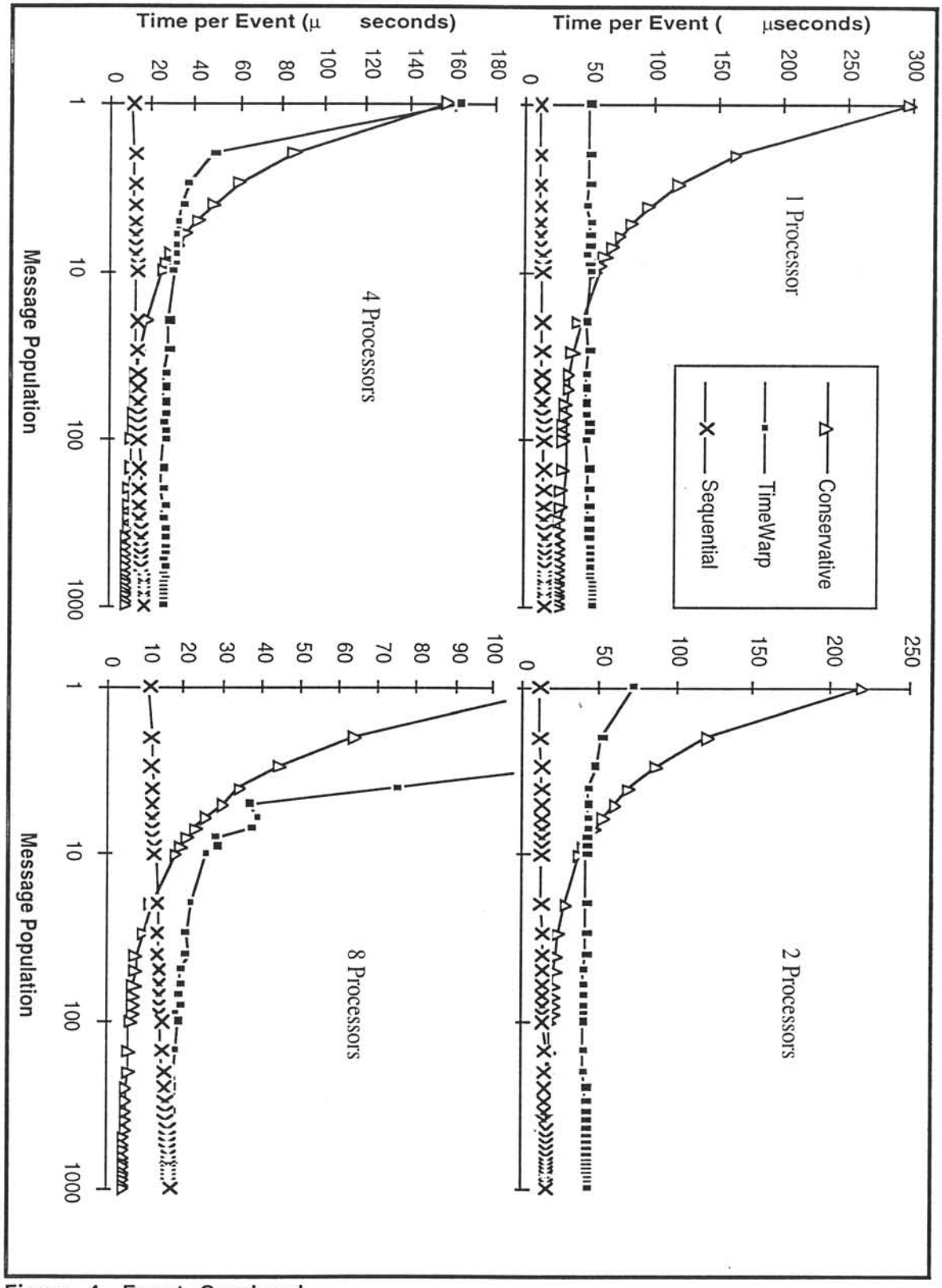


Figure 4. Event Overhead

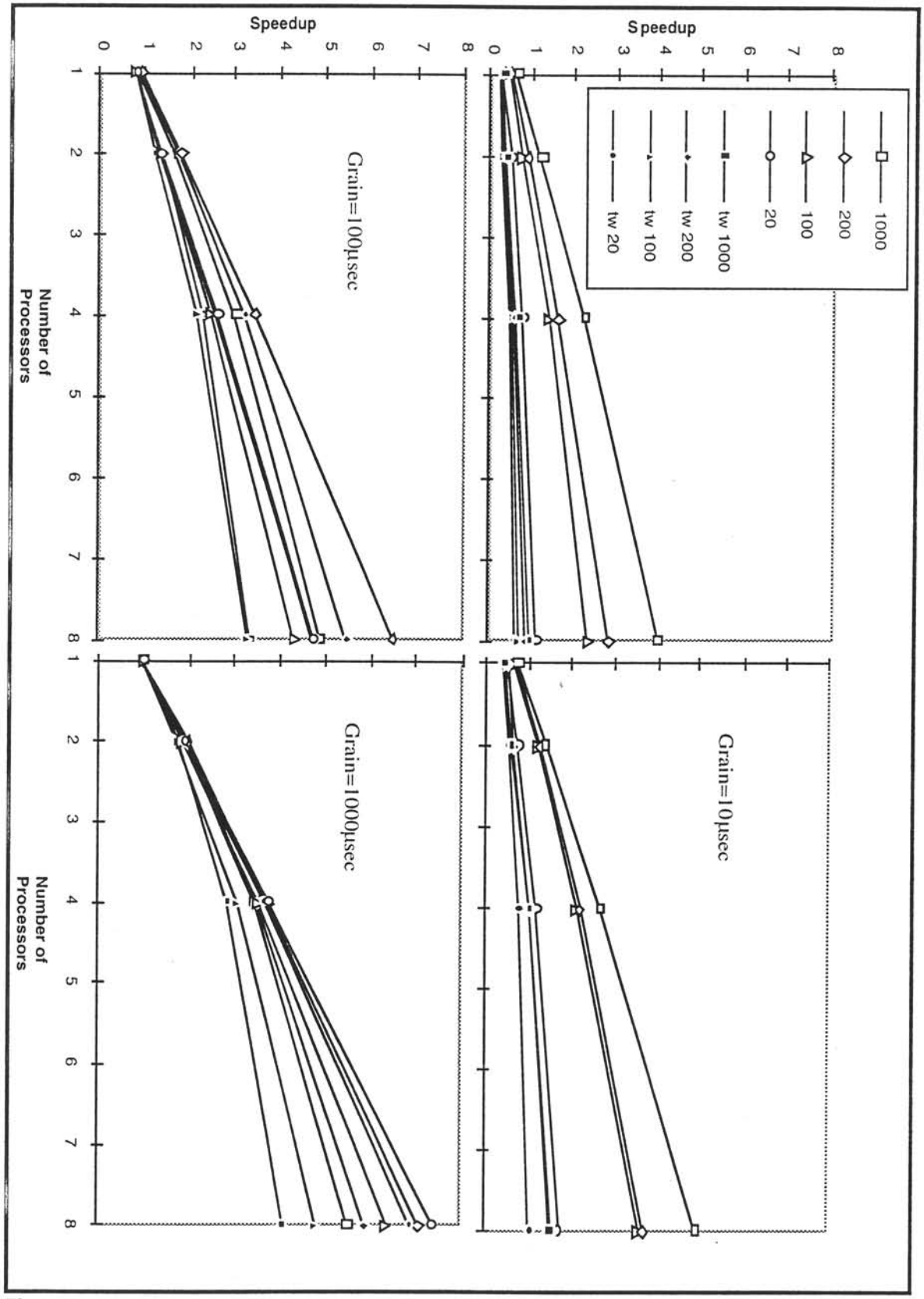


Figure 5. Speedup