A Generic Comparison Process for Documentation Files

A Dissertation Presented in Fulfilment of
the Requirement of the M.Sc. Degree

Michelle Timmons, B.Sc.

School of Computer Applications
Dublin City University

Academic Supervisor: Mr. Andrew Way

Declaration

I hereby certify that this material, which I now submit for assessment on the programme of study leading to the award of Masters of Science in Computer Applications, is entirely my own work and has not been taken from the work of others save to the extent that such work has been cited and acknowledged within the text of my work.

Signed: [illegible]

Michelle Timmons, B.Sc.

Date: 1-10-97

Acknowledgements

Thanks to Andy Way and Mark Roantree for their support and direction over the past two years. Thanks also to Forbairt, whose funding made this research possible.

I would like to acknowledge Leona, Ann, Anna and Mike in Lotus and to Tom, Robert and Sharon in ITP for their guidance in my research.

Thanks to all my friends, especially Séamus, for their help directly and indirectly in preparing this thesis, and more importantly, for being there. Thanks to Donal for his advice, and especially to Gary for his help at times when it was much needed.

Finally, my parents deserve many thanks and recognition for the support and encouragement they have given me throughout the years.

Abstract

One of the most important stages in the localisation process is the provision of high quality help and documentation in the target languages. Translation of computer manuals and Help files consists of :

(i) translating the text

(ii) maintaining the formatting of the source.

Although many tools are available to translators, the greatest need for standardised tools exists in the area of formatting and layout of documentation. At present a significant proportion of time allocated to this stage is spent checking that the formatting has not changed as a consequence of translation. For example, properties such as font type, font size and style (e.g. bold, italic) may accidentally be changed during translation. It is also possible that two paragraphs are combined into one, or even deleted altogether from the text

The aim of this research is to assess the viability of developing a generic comparison process for documentation files. This process should be able to take two text-based document files (e.g. TeX, MIF, RTF) and compare the underlying codes (called *markup*) that describe the format and structure of the documents, where format is its physical appearance (e.g. underlined text, margins) and the structure is its composition (e.g. paragraphs, chapters, headings). Although the localised documents will usually use the same markup scheme as the original, the possibility of incorporating the comparison of different file types into the process is investigated, in keeping with the concept of generality. However, each markup scheme has its own set of codes. In addition to this, the format is described by specific markup and the structure by generalised markup. The vast differences between these schemes means it is not always possible to make a direct comparison, complicating the process.

Table of Contents

1. Introduction

The localisation industry is one of the fastest growing areas in the world of software development. This world-wide drive to globalise software is partly due to the changing profile of computer users, but the main incentive is to sell more copies. Every company has its own approach to the adaptation of the software's functionality and the translation of text, but all strive to achieve a common aim - to make their product appear as if it was developed in their own country and not by a third party company. However, numerous problems exist in the industry. A major problem is that, given that no two products are the same, the localisation process can differ for each.

Our aim is to help generalise part of this process: the testing and verification of translated documentation files. At present, the task of ensuring that only the required changes (i.e. the text, screenshots, callouts, etc.) have been made during translation is often executed manually. Personal experience has shown that this results in a process that is slow, tedious and, because of the human factor, error-prone. Furthermore, it is very likely that translators will work with documents created in more than one application. For example, some of their clients may produce their documents in FrameMaker, whereas others may use Microsoft Word. The development of a system that would allow the automation of the verification of localised documents of any file type could provide a significant market advantage to software vendors interested in localising their products. In addition to being faster than manually checking the files, it would provide a more comprehensive and accurate analysis, reducing cost and time to market as well as increasing customer satisfaction. Furthermore, it eliminates the need for different tools to perform the same processing on specific documentation files.

The focus of this research is to generalise the comparison of the underlying codes specifying the *format* and *structure* of two documents where the format is the physical appearance of a document (e.g. bold text, underlines) and the structure is how the document is composed (e.g. chapters, paragraphs). This information is stored in the form of *markup* codes throughout the text. However, each markup language has its own set of codes. In addition to this, there are two types of markup: *specific markup* (e.g. RTF) describes the format of the document, whereas *generic markup* (e.g. an SGML document) describes the structure.

Although the localised documents will normally use the same markup scheme as the original, the comparison process will be extended to include the comparison of different markup schemes, in keeping with the concept of generality. This can be used to compare a document to a previous version in the case of the desktop publishing (DTP) system having been updated in between their creation. It could also be used to compare the equivalence of the same document published in different formats (e.g. printed documentation, on-line documentation, or pages on the internet). Despite the fact that such documents will obviously differ in appearance, the structure of the documents should remain similar. However, this can introduce different issues, e.g. links in HTML pages are not relevant for documents.

The vast differences between these markup schemes means that it is not always possible to make a direct comparison of tags from two documents. We have devised a generic process that will allow such a comparison by identifying the four different procedures:

1. Comparing documents with identical formats (e.g. two RTF documents)
2. Comparing two specifically marked-up documents (e.g. an RTF and a TeX document)
3. Comparing two documents with generalised markup (e.g. two SGML documents with different document type definitions (DTDs))
4. Comparing documents with different types of markup. (i.e. specific & generalised)

1.1 Overview of Chapters

Chapter two discusses the motivation behind this research. An introduction to the localisation industry in Ireland is provided as a background to the area. The localisation process for a product is outlined and some of the issues associated with this process are described, concentrating on those related to the localisation of documentation. Quality Assurance (QA) and the automation of QA testing are described, culminating in a discussion on the automation of testing documentation files.

Chapter three provides an overview of electronic documentation, focusing on the issues relevant to the localisation industry. It also introduces the concept of electronic markup, which is used to store extra information about a document. The two main types of markup are studied using *SGML* and *MIF* as examples. These two schemes are compared and contrasted, thereby providing a basis for creating a process for comparing the two schemes automatically.

Chapter four discusses our methods for designing a generic process for the comparison of two documents, based on the markup in those documents. Because of the differences in the features described by different markup, there are four different cases: comparing two documents with the same markup scheme, comparing two documents with specific markup, comparing two documents with generalised markup and the comparison of a specifically marked-up document with a document with generalised markup. The conversion of documents with differing schemes to a generic format, this being necessary in order to perform the comparison, is described.

Chapter five brings together the concepts of document markup discussed in chapter 3 and comparison algorithms discussed in chapter 4 with our research into the development of a generic process to compare two documents. This chapter discusses the implementation of a prototype that can take two documentation files and perform a comparison of the format and structure, based on the markup of the documents.

Chapter six analyses the results of the comparison made by the prototype in order to rate its overall performance. These results are analysed under the different areas of the process: the generic data preparation tool, the conversion of markup schemes to the internal generic tag set, the comparison of two documents with identical markup schemes and the comparison of documents having differing markup schemes but the same markup category. An explanation of any problems encountered is given and possible solutions are suggested.

Chapter seven summarises the ideas behind this research, and applies the conclusions derived from the implementation (as discussed in chapter six) to the localisation industry. Suggestions are offered for further work to extend and improve the work done here.

2. Localisation

2.1 Introduction to the Localisation Industry

Most software programs and documentation are first written in English, even by European companies such as Siemens, Ericsson, Nokia and Olivetti. They must then be translated and adapted into the local languages and culture for the non-English speaking market. Until recently, this involved little more than translating text in software, documentation and help from English into the major European languages. However, the end users have since become much more sophisticated and are quick to reject inferior quality. Consequently, "translation" has effectively been replaced by "localisation" [POLY96].

Every culture has its own national characteristics, legal requirements and accepted standards. The acceptance of a potentially successful product could be affected by not reflecting these in the software or its documentation. Therefore, to be successful in overseas markets, it is essential that one's company and its products should appear native to the target customers. The sign of successful localisation is when the product is perceived by the user as having been developed and produced in their own country rather than having been developed abroad, converted and then imported.

The profile of computer users is rapidly changing. No longer is it just highly educated people using sophisticated software; today's computer users extend across all layers of society and throughout a broad range of professions [HARS96]. Given the growing importance of the international market, companies are showing an increasing interest in adapting their products to multiple languages for use in foreign countries. The reason for this growth of the localisation industry is the desire to reach as broad a market as possible, the world.

There are various activities in the adaptation of products to local markets. These are described below:

Translation refers to the pure adaptation of words from the source language to the target language.

Localisation is the process of integrating the whole of the product cohesively into the language and culture of the target markets to meet their specific needs. It involves all components of a software product including the adaptation of the software's functionality, and the translation of manuals and on-screen text, as well as affecting technical specification and marketing literature. It also includes ensuring graphics, colours and sound effects are culturally appropriate.

Internationalisation is the behind-the-scenes work by software engineers to create a system or application software independent of natural language. It includes generic coding and design issues, such as keeping user interface (UI) text strings separate from the rest of the code so that translation will not introduce bugs into previously tested programs.

Globalisation is the term covering the entire process of creating a product with versions for users in multiple countries.

2.2 Software Localisation in Ireland

It is now widely accepted within the computer industry that Ireland is a world centre of excellence in software localisation with most major software firms having a significant presence in the field in this country. It is estimated that Ireland exports up to 60% of PC-based software sold in Europe, and is the world's second-largest exporter of software after the USA [LOCA97]. Those companies that have chosen Ireland for their product localisation centres include software publishers such as Microsoft World Product Group Ireland, Lotus Development Ireland, Corel Corporation, Symantec, Visio International, Novell, Oracle Corporation and Claris; hardware manufacturers such as Gateway 2000 and Sun Microsystems; Service Providers such as Berlitz International; and tools developers such as Trados.

There are also various localisation agencies operating in Ireland. Such agencies provide a range of services from software, documentation and help translation and localisation to technical services, project management and quality assurance. Many also offer advice on the best approach to internationalisation and on techniques which make information easier to translate.

As well as the Irish-owned companies, such as International Translation & Publishing Ltd, Translation Craft and DLG Software Services, many international agencies providing localisation services have established offices in various locations around the country, including Berlitz International, Bowne, Kudos and Rand McNally Media Services.

2.3 The Localisation Process

Although most of the larger companies have constructed their own framework manuals providing a model for internationalisation and testing, there is no formalised process because localisation will differ from product to product. For example, localising a multimedia product will require a different approach to traditional software because of different media, such as video, audio, 2D and 3D graphics and animation. World Wide Web applications may also need special treatment because of complex graphics, ShockWave and Java applets or CGI scripts. However, the localisation of different office applications is generally not that different.

However, we have outlined a general process below that can provide helpful guidelines to ease the task of the localisation team [TIMM96]. This will in turn reduce time and effort involved in the localisation of the product.

2.3.1 Original Development of the Product

The primary phase for any localisation project is the design of the product. The development teams strongly influence whether localisation will proceed with ease or difficulty. Within the teams, there needs to be a strong awareness of international issues to help internationalise, and subsequently localise, products more efficiently. "Software initially designed with features and code that support international conventions, foreign data and format processing will greatly facilitate the localisation process" [MILL94]. A number of guidelines for internationalisation can be considered during software development, for example:

- The use of non-US PCs, hardware and communications protocols should be supported.
- Keyboard layouts change according to locale, and not all characters exist in all keyboard layouts. Therefore, the use of non-US keyboards must be supported (e.g.

make sure shortcut key combinations can be reproduced using international keyboards).

- In many cases, localisation can be carried out on executable files. This property should be exploited by designing the software to eliminate the need to be recompiled to allow translation.
- All user-visible text should be separated from the product code and placed in resource files. These files can then be localised without affecting the software.
- When comparing strings with accented characters, the decision must be made as to whether the comparison should be ***accent-sensitive*** (where accented characters match only the same accented character, e.g. é = é, é ≠ è) or ***accent-insensitive*** (where accented characters match unaccented or other accented equivalents, e.g. ë = é = e). Comparing user-typed strings when searching help text, for example, should be accent-insensitive in order to be "user-friendly".
- Code should not byte-steps through strings as this assumes that all characters are single byte, but languages such as Japanese use double byte characters.

2.3.2 Technical Review

A technical review can be conducted to gather information about the product (both software and documentation) early in the development cycle. This can be used to influence the development team to provide a product that is more suitable to internationalisation by highlighting issues that may affect localisation, such as those mentioned above.

2.3.3 Localisation Analysis

Each file in the product is examined to see if it contains text needing translation or other information which must be changed due to localisation [LOTU95]. Files that do not need to be changed do not require rebuilding and can be imported straight into the localised product build. Rebuilding them will introduce the possibility of error. If the need arises for special tools to help translation, updating and verification, it will be identified here. The idea of using tools is to make localisation an exercise in language processing and preventing the localisation team being bothered by technicalities. Many companies will develop tools to overcome specific difficulties encountered in localising a product. A major disadvantage of this is that it will overcome only that problem for which it was designed. It is unlikely that it will be used again without considerable modification.

2.3.4 Translation and Verification

At this stage, all necessary information and software is provided to the translators, who may be either in-house or external vendors. As translation involves text which is external to the product, it is the activity most likely to be outsourced. This decision is usually based on the cost - if the resources are available in-house, it is less costly than outsourcing translation. However, if resources need to be specially hired for this purpose, it is often cheaper and more convenient to use an external vendor. It should already have been established whether the translators need to be provided with the tools, supporting documentation and materials necessary to implement the localisation process. The translators must also be made aware of any translation restrictions, e.g. line-length, deadlines.

In the ***evaluation and preparation stage of translation***, text often needs to be manipulated out of and back into desktop publishing (DTP) environments, while at the same time maintaining formatting information. Filters and layout formats that accommodate such pre-processing of text can be developed where needed. Software engineering tools can be used to verify that all of the original text in a User Interface (UI) is resourced and that it can be translated using relevant tools and editors [ITP96]. Unresourced text is therefore quickly identified. Similarly, text expansion and support for accented characters are tested at the early evaluation and preparation stage. Detecting potential difficulties such as these early on can help ensure that the critical path process of translating and building/testing software is not held up through the unexpected discovery of such issues.

Files are checked to see whether it is possible to re-use translations from previous versions of the product. This maximises the return on investment in translation and helps ensure consistency of translation. *Computer Aided Translation* (CAT) techniques are often employed to do this by building translation memories from previously translated versions of the product. These memories are used to batch translate (or pre-translate) the new source files, thus performing translation and ensuring quality and consistency in a fraction of the time it would take to carry out this task if the traditional "cut and paste" methods were used. [ITP96]

During the ***translation phase***, the same translation memories can be used by translators working in a CAT environment where further productivity gains can be achieved using repetition processing[1] and "fuzzy" matching[2] capabilities of the CAT tools.

In the ***verification phase***, a *functional test* is conducted to verify the quality of the translation, adaptation, layout and graphics design. Quality Assurance (QA) tools exist to help with this. QA tools can also be used in the software building and testing stage to automatically detect problems such as duplicate hot keys, truncated text, and a variety of other common UI localisation issues.

Often, only a section of the work will be completed and this is reviewed during a functional test. Generally, if files are being translated by an external vendor, this is carried out in-house if resources are available. Once approved, this section serves as the quality model for the remaining parts of the project.

2.3.5 Acceptance Test

Once the whole product has been translated, an acceptance test is conducted to help ensure a bug-free product. The whole product must be re-built with the translated files, using the localised build environment. A number of previously run checks on translation, layout and functionality are re-run as a precaution.

2.3.6 Evaluation

Once the localisation of the product is finished, the whole process should be reviewed to identify successes and failures of methods used, so that they can be evaluated and either maintained or corrected. It can also provide information to the

[1] Repetition Processing in the application of the same process many times.

[2] Fuzzy matching is used to find "almost matching" results, rather than exact matches.

development team on the product in terms of how easy the product was to localise, and suggestions should be provided on addressing any problems encountered.

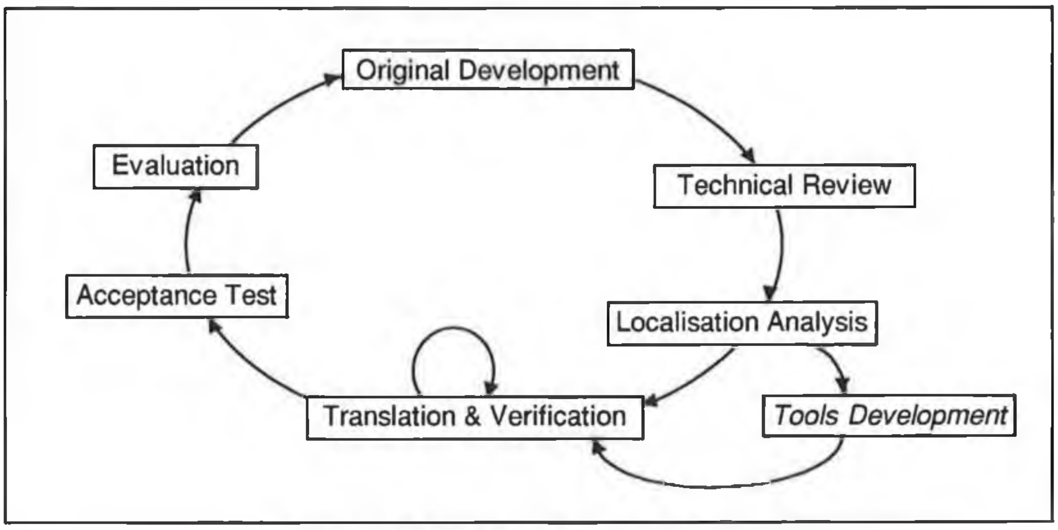


Figure 2.1 The Localisation Process

2.4 Issues in Localising Documentation

Although the localisation process concentrates on the adaptation of the software to other cultures, there are many issues that may occur in the localisation of documentation. This can mean that compromises are made to meet deadlines; for example documentation text may be simplified to cut down on the translation needed, or the documentation is published with errors. There are two main areas in which these issues are encountered: the content of the documentation and the process of translation itself. In this section, these issues are examined as many of them have a direct effect on this research.

2.4.1 Content-Related Problems

The content of documentation can cause confusion or complications in the translation. Some of these can be introduced in the original documentation by the thoughtlessness of the technical writer, and can be reduced by careful forethought. Other issues regarding the content are usually language and locale specific and commonly occur in the localisation of both documentation and applications.

2.4.1.1 Considerations for the Technical Writer of the Original Document

The cause of many other complications in translation is lack of thought in writing the text in the original documentation. “Making the information as clear and simple as possible promotes easier translation” [MILL94, p101]. There are a number of points the technical writer needs to consider:

- **Clarity and Simplicity:** The original text should be written concisely and using terminology that is easy to translate. English tends to be very metaphorical at times and this can cause the translator difficulty. Furthermore, the use of English words with multiple meanings can cause confusion to the translator. For example ‘because’ should be used instead of ‘since’, as it has a single meaning; ‘since’ can be confusing for a translator [MILL94]. Images should also be clear and uncomplicated as complex images are difficult to understand.
- **Use of Existing Terms:** Existing terminology should be used wherever possible. However, sometimes terms that are common in the original language have no equivalent in other languages (e.g. “*pop-up window*”). The translator is then faced with the decision between creating a new term in the language, or introducing the original “foreign” term. If a new term is created, there is the risk that no-one will understand it, but the reader will be unfamiliar with “foreign” terms and may not accept them.
- **Applicability of Content:** The applicability of content to other countries must be considered. Some terms or images may be irrelevant, or even offensive, in another country. For instance, examples can appear with zip codes, or refer to U.S. place names that are not as relevant to non-US users. Animals, religious and mythological symbols, colours, hand gestures and people (especially racial, cultural or gender stereotypes) may be misinterpreted or may offend users in another country [KANO95]. For example, some cultures associate the pointing-finger image (as used in a cursor) with thieves.

2.4.1.2 Issues for the Translator

- **Prior Knowledge of the Subject:** A common problem in the translation of software documentation can be that the translator does not know exactly what the product is about. Therefore, the translator should be thoroughly familiar with the software before attempting to understand the terminology used in the documentation.

- **Alphabetical Sorting:** Lists that were sorted alphabetically in the English version must be re-sorted after translation to re-order the list items alphabetically in the new language.
- **Local Data Conventions:** The conventions for displaying dates, times, currency, measurements and numeric formats can differ from place to place [DIGI91]. For example, the custom in the United Kingdom for numeric dates is to display the day first, followed by the month and then the year, e.g. 22/11/96. However, the U.S. standard is the month first, then the day and year. e.g. 11/22/96. When dealing with times, not only do different time zones and Daylight Saving Time affect times, but different locales also use different conventions [DIGI91]. For example, Ireland, the U.K. and the U.S. still use the 12-hour clock system, whereas many European countries such as France, Italy and Spain use the 24-hour clock system. Therefore during translation, issues such as these must be examined and localised (i.e. adapted for the target culture) rather than simply translated. The different conventions for currency, measurements and numeric formats require similar adaptation of the text.
- **Locale-Specific Information:** Text that is applicable only to a specific locale cannot be directly translated into the intended language. Although such text should be avoided in the original documentation where possible, it is sometimes necessary to include sections for specific locales. For example, the customer support section of a manual will differ depending on the intended destination. Not only will contact names, postal addresses, e-mail addresses and telephone numbers change, but in addition to this, certain information may be omitted. For example, some services such as a 24-hour help desk may not be available to customers outside the U.S. Furthermore, any references in the text to things changed by the translator, such as language, countries or formatting must be updated. For example, references to reading from left to right must be changed in Arabic documents (e.g. documentation for word processing packages may refer to the characters being typed from left to right across the screen).

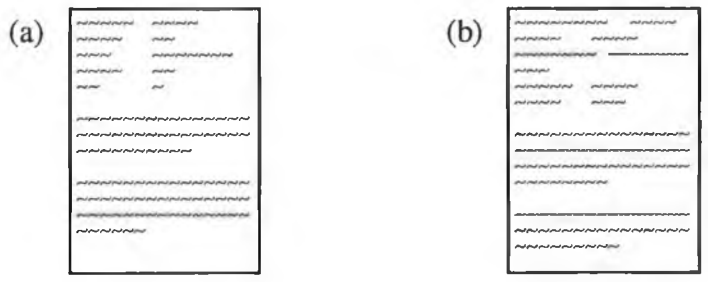
- **Proper Names:** The names of people, companies and products will not usually be translated in the localised documentation. Exceptions to this include localised versions of products which may be renamed, (e.g., the Japanese version of Microsoft Word is called *Word J*), in addition to companies and products that are known under a different name in other countries.

2.4.2 Issues Related to the Translation Process

A number of issues are encountered during the translation process that are not related to the content of the document, but to the planning and performing of the translation. Sufficient planning and forethought can reduce these problems. Some of these issues include:

- **Document Formats:** Documentation files may be in different formats. They will need to be converted to the same format for use with tools.
- **Modifications to Software:** The software many change a lot during development, which will have an effect on the documentation. If translation is started too soon, much of the text may change and have to be re-translated, and the product re-tested. Leaving it too late will delay the release date of the translated product.
- **Resources:** The lack of resources for translating into less popular languages may be an obstacle in the translation of documentation. For instance, a current EC project to design a CBT course on EC Structural Funds had great difficulty in finding skilled resources with lesser-used languages. To translate it from French to Irish, for example, it was necessary to translate from French into English first, and then from English to Irish by another set of translators. [MCD95]
- **Formatting Conventions:** Different cultures expect different layout and formatting conventions in documents. For example, not all cultures read text from left to right. The text of Arabic documents goes from right to left, whereas Korean text is in vertical lines from top to bottom, read from left to right. Mongolian documents are in vertical lines with characters from top to bottom, but are read from right to left [ZHEN92]. Therefore the document will require conversion to the appropriate layout.

- **Accidental Alterations to Document Markup:** The format and structure of the original document can be accidentally altered during translation, especially if the translator is operating in a WYSIWYG[3] environment, as it is easy to change the underlying markup without realising. For example, the translator may change text properties such as font type, font size and style (e.g. bold, italic). It is also possible that two paragraphs are combined into one, or even deleted altogether from the text.
- **Unreachable Text:** Text that is part of an image (such as a bitmap) cannot be translated and a new image must be included with the translated text. Therefore it is recommended that text is not included in the images.
- **Text Expansion:** Translation of text can result in an increase of up to 30% in its length. This can cause a number of problems. For example, text in diagrams may expand, requiring the components of the diagram to be rearranged. Text aligned using tabs may be displaced and need realignment (see Figure 2.2).

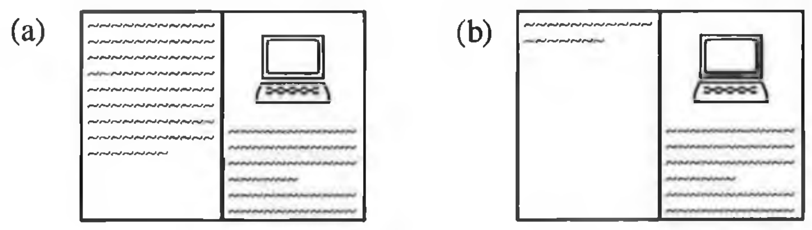


In (a), the text at the top of the page is neatly aligned into columns using tabs. However, after translation the text may expand to push the columns out of alignment as in (b).

Figure 2.2 Issues with Tabbed Text Due to Text Expansion

Another problem may arise if page breaks were enforced in the original document to move a diagram to a new page, for example (see Figure 2.3 (a)). The expansion (or reduction) of text can cause these page breaks to move, which may result in a page with very little text (see Figure 2.3 (b)).

[3] WYSIWYG (What You See Is What You Get) applications "display on screen a close representation of what will appear on the finished output" [NCC87, p19]



In (a), a page break was enforced after the text in the left-hand page to "push" the diagram onto the next page, as it did not fit. However, after translation, the text may have expanded onto the top of the next page. The diagram would now fit on the bottom of this page, as in (b). However, the enforced page break still forces the diagram onto the next page.

Figure 2.3 Issues Due to Enforced Page Breaks

2.5 Quality Assurance Testing in Localisation

The localisation of a product is a complex task. As we have observed, it does not simply consist of the translation of the software, documentation and help files of the product. It must be ensured that issues such as those highlighted above have been dealt with adequately. Therefore localisation also addresses questions such as whether the quality of the translation is acceptable, whether the product still retains the same functionality as the original product, and whether the text has maintained an attractive layout on the screen or in the printed documentation.

Section 2.3.4 introduced the issues encountered in the Translation and Verification phase. The quality assurance (QA) of the localised product is an important part of the localisation process to ensure acceptance in the world-wide market. There are three elements to quality in software:

- *The technical functionality.*

  The product should retain the same functionality after localisation. Many bugs can be introduced in re-building localised projects.
- *The linguistic quality of translation.*

  It must be ensured that the translated text still retains the same meaning as the original text.

- *The layout.*

  This entails checking to see whether the translations fit correctly on the screen, or in the case of documentation, ensuring that it maintains the same formatting and layout.

2.6 Automation in Localisation

Many of the processes described above use tools to automate them. However, many tasks in the localisation process are still carried out manually and are very labour intensive and costly as a result [LRC97]. There are many advantages in automating certain processes, including:

- *Reduction in time and cost of QA cycle.*

  Automated QA is much quicker, and therefore cheaper, than manually testing files.
- *Comprehensiveness and precision.*

  It can assure a certain degree of quality. For example, errors such as differences in fonts/font sizes may be very difficult for the human eye to spot, whereas an automated process can easily identify them, giving more accurate results. Automation also performs the exact same set of tests every time, ensuring consistency. There are no restrictions on the number of checks that can be conducted, so everything can be included, resulting in a more comprehensive test. Manually, this would be an unrealistic aim, mainly due to time.
- *Repetition for many languages.*

  The more languages into which the product will be translated, the greater the number of times the same tests need to be executed. Automation ensures that the same procedure is performed on each language version. It also executes these much more quickly than is humanly possible.

2.6.1 Automating QA Testing for Documentation Files

Some of the differences introduced during localisation are necessary, as described in section 2.4. However, this research concentrates on the problem of the accidental modification of the format and layout of a document during translation. For example, formatting information such as font, font size and style (e.g. bold, italic) may be accidentally altered by the translator. It is also possible that the structure of the document may be changed, e.g. two paragraphs could easily be combined into one, or sections deleted altogether from the text. At present a significant proportion of time allocated to this stage is spent manually checking that the formatting has not changed as a consequence of translation. Even if this process has been automated, current tools apply only to a particular markup scheme and a new one is required for different formats.

The aim of this research is to develop a generic process to facilitate the automation of the comparison of the original English files with the translated files. This process would work by comparing the markup in the document files that describes the format and structure of the documentation, and report all differences between the files. It is assumed that the two files to be compared are similar in format and structure. Although such a system still relies on the user to manually examine the differences reported to determine which are required and which are errors, its development would dramatically reduce the time spent in the QA phase and would give a more comprehensive and accurate analysis than a manual comparison. This system would benefit the translator, as the responsibility of verifying the effects of their translation currently lies with them. As they are likely to work on a wide range of documents from different applications, a generic tool is would facilitate this task.

2.6.1.1 Generic Processes

We also seek to contribute towards the standardisation of the localisation process throughout the industry, leading to reusability of software and translation materials. As mentioned in section 2.3, many companies develop tools in-house to support localisation. These are usually designed with the sole purpose of quickly solving a specific problem with a particular product. In order to re-use these tools later on for other products, they normally require modification and re-engineering.

The advantage of the external development of localisation systems (as opposed to development in-house) is that the developer is not restricted to having to overcome an immediate need. Therefore they have the time to investigate the problem area and develop a process applicable to a more general domain. This project will examine the possibility of creating a QA system of such generality that it could accept files of any format for analysis.

2.7 Summary

In this chapter, the localisation industry in Ireland was outlined. We described a general process that can be applied to the localisation of a software product, and discussed some of the issues that can be encountered during this process, concentrating on those regarding the localisation of documentation. The different areas in which quality assurance of the localised products is necessary were mentioned and the advantages of automating QA testing were described. This served as a background to the area of this research: the development of a generic comparison process to facilitate the automation of the verification of localised documentation files. We intend to do this by locating the markup codes in the documents and comparing these, as they encapsulate the layout and structure of the documents. The next chapter examines the markup in documents that are created by Desktop Publishing (DTP) applications, with a view to comparing the different markup schemes.

3. Overview of Electronic Documentation

3.1 Introduction

The term electronic documentation can be applied to any document created using software and stored in a computer file. In this chapter, we provide an introduction to Desktop Publishing (DTP) and Word Processing as ways of creating electronic documentation. The problems currently encountered when processing documents created in DTP packages are discussed, concentrating on the issues relevant to the localisation industry. We then examine the *markup* that all electronic documentation uses to store the appearance and composition of a document. The two main types of markup are studied using *SGML* and *MIF* as examples and the differences between these two schemes are highlighted, indicating difficulties in comparing them.

3.2 Desktop Publishing

Desktop Publishing (DTP) is the use of personal computers and page-layout software to perform all or most of the steps of publishing [GURG90]. A DTP package is an application program which allows the user to manipulate pieces of textual and graphical data to produce a publication, e.g. course brochures, newsletters, pamphlets. In general, DTP software is not used until the pieces of text have been created by a word processing package, and the graphical data has been created by scanning a picture or by using draw or paint packages. These pieces are then assembled into a final publication with the use of the DTP package.

DTP software offers great flexibility and fine control over text formatting. A comprehensive, but by no means exhaustive list of DTP features is given below. Most DTP applications contain a subset of these features.

Multiple Columns
Page insertion/removal
Automatic Page Numbering
Rulers
Style Sheets
Tabs & Tab Leaders
Reverse Type

Character and Paragraph Formatting
Multiple Fonts
Font Sizing
Automatic Text Flow and Wrap
Horizontal/Vertical Text
Hyphenation
Text Rotation

Import Raw & Formatted Text	Tracking
Import Graphics	Kerning
Search/Replace	Letterspacing
Headers & Footers	Condensing
Automatic Index Generation	Expanding
Automatic Table of Contents Generation	Baseline shifting
Automatic Foot Noting	Set text in special shapes
Automatic Figure Numbering	Super/Subscripts
Grouping and Ungrouping of Objects	Graphics Manipulation
Undo Capability	Rotation of Graphics
Master Pages	up to 0.0001" accuracy in placing objects
Thumbnail view of document	Drawing Capability

3.2.1 DTP Software

There are basically 4 types of publishing software, Command-Driven, SGML, WYSIWYG and Document Description Languages.

3.2.1.1 Command-Driven

This software requires the user to insert the appropriate formatting codes into the text of the document. It is suited to uncomplicated work where changes to the type style or size are likely to be minimal, such as a book [NCC87]. The main disadvantage of command-driven software was that the finished appearance could not be viewed until the document was printed. However, readers to view the document before printing are now available. Popular examples of command-driven software are TeX and LaTeX.

3.2.1.2 Standard Generalised Markup Language (SGML)

SGML is the International Standards Organisation's (ISO) standard for document description allowing users to create a set of markup tags with rules defining when they are applicable and how they relate to each other. As with command-driven software, the applicable tag is inserted around the text. In this case however, SGML allows the structure rather than the appearance of a document to be defined. By remaining neutral with respect to formatting, SGML allows the same information to be presented in many formats across many different hardware and software systems. The most popular application of SGML is Hyper Text Markup Language (HTML) the standard for documents on the World Wide Web.

3.2.1.3 What You See Is What You Get

WYSIWYG software is probably the best known type of DTP application [NCC87]. It overcomes many layout problems by displaying on the screen a close representation of what will appear on the finished output. The elements of a page can be changed as often as necessary until the desired effect is achieved. It is less probable that mistakes will occur because it is more likely that the mistake will be noticed on the screen before the page is printed [BOVE87].

However WYSIWYG is a time-consuming process, as each page must be designed individually. The text is usually prepared using standard word processing software (often provided as part of the DTP software) and stored. The page layout is then designed and the pre-prepared text and graphics 'poured' into the available space. If the text does not fit, the point size and line spacing can be adjusted until it fits in an acceptable manner. Similarly, graphics can be enlarged, reduced or cropped. Using this kind of system the elements of a page can be changed as often as necessary until the desired effect is achieved. WYSIWYG applications include Adobe FrameMaker, Adobe PageMaker and Corel Ventura.

3.2.1.4 Document Description Language

DDL is a piece of software used to smooth the transfer of material from the input and storage device to the output device [NCC87]. Using a DDL improves the speed and efficiency with which material is passed from the input stage to the output stage. PostScript is one of the best known examples.

3.2.2 Style sheets

Many systems provide the ability to use style sheets in documents. Style sheets are a collection of pre-defined styles. A style is a set of text formatting information applied to characters or paragraphs that cause the text to be reformatted according to the specifications of that style [ADOB90]. For example, the default font for a document in Microsoft Word is of the style Normal, defined to be font Times New Roman, size 10. The user can create styles or use those supplied with the software. Not only do style sheets make document formatting quicker and easier, but they also help maintain a consistent look throughout a document.

3.2.3 Desktop Publishing versus Word Processing

Until recently, Desktop Publishing was the only way to perform elaborate formatting on text (e.g. flowing text around graphics, using different fonts or rotating text) and then view on-screen exactly what that document would look like when printed. Word processors were simply a means of creating, editing and printing documents, without any complex formatting facilities.

Publishers of many word-processing packages suggested that people bought DTP software because they wanted WYSIWYG output that they could not get from word-processors [WHEE94]. Now that Windows has made WYSIWYG available to everyone using a PC, the marketplace is changing dramatically. Today, most Windows word processing (WP) packages can provide the day-to-day document production requirements of the occasional users, leaving the professional users needing the precise and delicate control offered by DTP applications.

DTP producers did not anticipate the convergence of the DTP package and word-processor because DTP mimics professional typesetting, where the user had to learn basic publishing skills (e.g. the layout of columns mixed with pictures) [WHEE94]. However, this view did not take into account the increasing range of features of today's word-processors that overcome the need for many traditional layout and publishing skills. For example, most Windows word processors facilitate the creation of multiple columns and automatic text wraparound graphics. Many WP packages also provide document templates to guide the user through working with standard layouts, giving adequate DTP results without the complications of using features like multiple column layout and frames.

Tanaka [TANA94] compared DTP and WP applications as follows:

- **Text Placement:** Although a DTP package is better at precisely placing text blocks so as text appears in different sizes and locations on a page, using frames in a word processor can accomplish a similar layout.
- **Graphics:** Word processors can also adequately handle graphics in documents. Using frames, drawings, charts, tables, equations or scanned images can be

inserted, resized and positioned anywhere on the page. Some even include freehand drawing tools.

- **Text Streams:** One area in which DTP has an advantage is when the document is comprised of multiple text streams. The master document function allows separate files to be associated with one document. Although WordPerfect 5.0 (and above) and AmiPro 3.1 provide this facility, most WP applications require all the documents to be collected into a single file for indexing, page numbering and creating the contents pages.
- **Publication Format:** Word Processors are designed for printing office documents (e.g. letter or legal-size paper, envelopes, labels), so the maximum paper size is usually much smaller than that allowed by a DTP package. However, most offices do not need such capabilities as they do not have the facilities to print large documents.
- **Colour:** One of the key features distinguishing high end DTP programs from word processors is the ability to deal with colour. The latest versions of DTP software, such as QuarkXpress or PageMaker, are far more capable of dealing with colour than a word processor.

To summarise, most Windows word processing packages are adequate for the document production requirements of the majority of users. However, since WP software can be used to create satisfactory documentation, we must also consider the output from these in our research. Nevertheless, as the work gets more complex, word-processors become less and less suitable. Documents that need precise, delicate control with many small frames of text, lots of graphics and complex layouts are better handled in packages such as PageMaker and Quark Xpress.

3.3 Automating Electronic Document Processing

The formatting produced by most word processing and DTP software is proprietary, thus making it restrictive. Also, each of these packages has its own storage structure for this information. As a result, tools used to analyse documents from these applications generally concentrate on a limited number of these formats. Although most handle formats for leading word processing and DTP products, the fall-back position will always be ASCII text [OVUM95]. For example, the Logos Intelligent Machine Translation System claims to provide support for Microsoft Word for Windows, Word Perfect, Lotus AmiPro, Windows Help Source, FrameMaker and InterLeaf [SOFT96]. However, on examining the requirements of the system, documents from Word, AmiPro and Windows Help must be saved as RTF[4] files, FrameMaker files must be in MIF[5] format and Interleaf files must be saved as plain ASCII. Even more limited in its performance is the S-Tagger, which works solely on one document markup scheme. ITP released two separate versions, one for FrameMaker and another for InterLeaf documents [ITP96].

Documents often need to be converted to the format used by the particular tool [OVUM95]. Conversion utilities use filters to conserve the formatting characteristics of the source text. For example, AmiPro has a filter to save it as an RTF file. Once finished, the text must then be converted back to the original format, restoring all tagging and formatting information. Without filters, the task of re-formatting the documentation after processing can be very time-consuming and prone to corruption. Even with filters, a lot of work is involved, and the possibility of introducing errors still exists.

Much effort has been put into devising a solution to eliminate the complications and cost of translating text between different editing platforms. Most of this has been focused on devising a standard format to which all documentation would conform. The OVUM Report [OVUM95] states that this concept has been promoted (by ISO and others) for roughly ten years. There are currently two standards in existence: the Open Document Architecture standard and the ISO standard (SGML). These are in competition with certain proprietary standards that have gained wide acceptance (such as Microsoft's rich text format, RTF), but it is unlikely that a purely proprietary

[4] The Rich Text Format (RTF) Specification is Microsoft's text-based format for "encoding formatted text and graphics for easy transfer between applications" [MICR95, p3].

[5] MIF (Maker Interchange Format) is a group of ASCII statements that can represent all the text, graphics, formatting, and layout constructs in a Frame document [ADOB95].

standard could ever serve as a truly open international formatting standard [OVUM95] due to the number of vastly different formats in existence.

As a single formatting standard has not yet emerged, the solution we propose is to develop generic systems that can take documents of any format as input and perform identical processing steps on them. This will involve trying to devise a generic format onto which different forms of markup can be mapped. This will be the basis for a tool that can be used during localisation for comparing the format of translated documents to that of the originals.

3.3.1 Automating the Comparison of Two Localised Documents

The localisation process involves as a sub-process the translation of all text to another language. The post-translation process checks the quality and accuracy of the translation. This can be performed using applications developed for this purpose, or by manual checking.

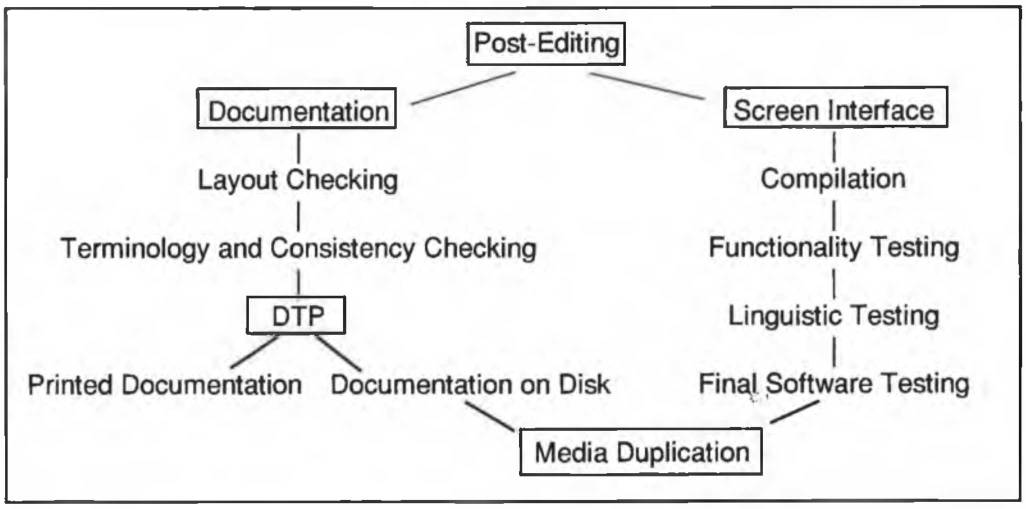


Figure 3.1 The Post-Translation Process [OVUM95]

One of the problems encountered after translation is that the formatting of the document has often been accidentally changed during editing. Much time and effort is invested in verifying that the layout of the translated document does not wrongly differ from the original. Our aim is to develop a process that will reduce the quality assurance (QA) process by automating several structure and consistency checks on the translated files. It should not only be faster than manually checking the files, but will give a more comprehensive and accurate analysis.

At present, the format checking when translation tools are used is usually executed after translation while the document is still in the intermediate format. This is the "layout checking" stage in the post-translation process. A generic tool for verifying the structure of the translated document against the original could be used on whatever format is output from the software in the DTP stage. Using the final result is more beneficial because the possibility of errors occurring during the conversion back to the original format is eliminated. The post-translation process for documentation would then take the form of Figure 3.2.

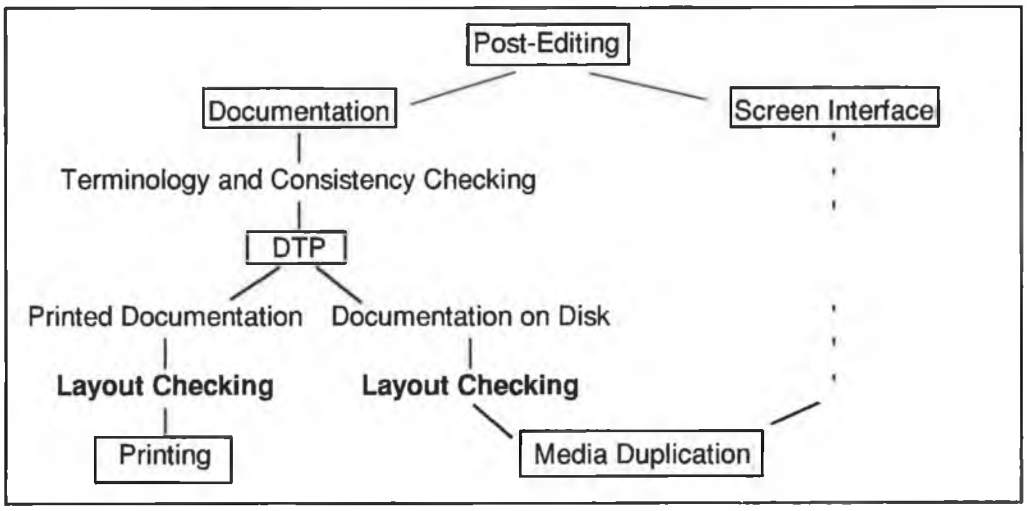


Figure 3.2 Revised Post-Translation Process

Because the formatting information is stored as *markup* in the document file, the markup must be extracted for comparison. However, the output files from the majority of DTP packages (e.g. Adobe PageMaker, Quark Xpress, Microsoft Word, Microsoft Publisher) are stored in a proprietary binary format that can only be parsed if the format of these files is known in advance. We have found that most vendors do not wish to publish their format so we are limited to text-based formats (such as MIF, RTF and SGML), as designing binary parsers for each vendor format is beyond the scope of this research.

Therefore the tool can only be used after the DTP stage if the application used outputs ASCII files, or if the user wished to convert the output file to a textual format. Otherwise it can still be used at the layout checking stage before DTP, as is currently the case.

3.4 Markup

Electronic markup is the additional information interspersed among the natural text of the document, which is not part of the text or content, but describes it. A markup language is a set of markup conventions used together for encoding texts. It must specify the markup allowed, the markup required, how it is to be distinguished from text and what its role in the structure if the document. Markup serves two purposes [GOLD90]:

1. to separate the logical elements of the document; and
2. to specify the processing functions to be performed on those elements.

Markup originally referred to the annotation added to a text instructing the typesetter how the manuscript should be laid out. With the introduction of automation in publishing, the term was extended to cover markup codes that indicated processing (such as formatting) used in electronic texts, and consequently text formatting languages were written. A typesetter would convert the annotated markup into the equivalent markup for the text formatting language being used and insert this into the electronic text [WATS92].

As computers became more widely available, authors began using word processing software to write and edit their documents. Systems that store text for output generally use some form of markup, even though it is not always apparent to the user. For example, HTML and TeX are text-based markup schemes. Markup usually takes the form of start and end tags delimiting the text. These tags may be visible, hidden, entered by the user, or automatically generated. They can be stored as binary data or alphanumeric text characters. Although these systems are powerful and effective in formatting documents, the fact that each usually has its own method of markup can cause compatibility issues. When exchanging documents or changing hardware or software, it may be necessary to convert data to the new format. This can often mean re-entering at least the formatting information, if not the whole document.

3.4.1 Types of Markup

The following figure is used by Coombs et al. to illustrate text that has no markup at all:

miltonexpressesthis ideamostclearlylaterin the tracticannotpraise
afugitiveandcloisteredvirtueunexercisedandunbreathedthatneversa
lliesoutandseesheradversarybutslinksoutoftheracewherethatimmortalgarlandistober
un fornotwithoutdustand heatsimilarlywordsworth . . . .

(This example may look artificial, but "ancient writing was often in such scriptio continua, with virtually no interword spaces and little punctuation" [CRD87].)

Figure 3.3 Text Without Any Markup [CRD87]

Authors instinctively mark up a document as it is written, for instance, by putting spaces between words, and using fullstops to indicate sentence boundaries [CRD87]. Although spaces and punctuation are not tags, they are still valid markup as they identify the "logical elements" of the text, e.g. humans as well as computers require spaces to identify each word, and punctuation is required to denote sentences, clauses, paragraphs, etc. The use of punctuation is called **punctuational** markup [CRD87]. Because such punctuation is common, it is naturally assumed that authors will punctuate their document files as they type them. Therefore, some form of markup will always occur in documents because our writing systems require it.

The introduction of text-processing systems brought with it new types of markup and processing. Documents stored in electronic files often have special electronic types of markup designed for processing by computers. There are two main categories of electronic document markup (see Figure 3.4):

- **specific markup**, encompassing *presentational markup* and *procedural markup* (describing the procedures that a particular application should follow); and
- **generalised markup** which identifies the entity type of the current string.

Coombs et al. [CRD87] illustrate the differences in the principal types of markup using the same text as in Figure 3.3 as follows:

Presentational Markup

Milton expresses this idea most clearly later in the tract:

I cannot praise a fugitive and cloistered virtue, unexercised and unbreathed, that never sallies out and sees her adversary, but slinks out of the race where that immortal garland is to be run for, not without dust and heat.

Similarly, Wordsworth . . . .

Procedural Markup

.sk 3 a;.in -10 +10;.cp 2;.ls 1 Milton expresses this idea most clearly later in the tract: **.sk 3 a;.in +10 -10;.ls 0;.cp 2** I cannot praise a fugitive and cloistered virtue, unexercised and unbreathed, that never sallies out and sees her adversary, but slinks out of the race where that immortal garland is to be run for, not without dust and heat. **.sk 3 a;.in -10 +10;.cp 2;.ls 1** Similarly, Wordsworth . . . .

Generalised Markup

<p>Milton expresses this idea most clearly later in the tract: **<lq>**I cannot praise a fugitive and cloistered virtue, unexercised and unbreathed, that never sallies out and sees her adversary, but slinks out of the race where that immortal garland is to be run for, not without dust and heat.**</lq>** **<p>**Similarly, Wordsworth . . . .

Figure 3.4 Different Forms of Markup

There are other forms of markup that can be used in conjunction with these, i.e. *punctuational*, *referential* (referring to entities external to the document) and *metamarkup* (which defines or controls the processing of other forms of markup).

3.4.1.1 Specific Markup

Most WP and DTP software uses specific markup, each with its own set of markup codes that only it can understand [ARBO95]. This markup is usually in the form of formatting codes that are mixed in with the text of the document. These codes represent a single way of presenting the information, such as a printed page, and do not allow the user to define the appearance of the text for any other media, such as hypertext.

It should be noted that specific markup can use style sheets to emulate generalised markup, but this is at the discretion of the user. Therefore documents with such schemes can be regarded as generalised if the style sheets are used consistently throughout the whole document. However, because this is not enforced, the schemes are generally considered as specific markup.

Presentational Markup: This requires the user to specify the proper layout or appearance of a text. The components in a document can be marked up in many ways to clarify the presentation, including horizontal and vertical spacing, page breaks and enumeration of lists [CRD87]. For example, an author generally marks the beginning of a paragraph by leaving some vertical space and often horizontal space as well.

Procedural Markup: In many text-processing systems, presentational markup is replaced by procedural markup, which defines what processing is to be carried out at particular points in a document [CRD87]. The user inserts commands into the text stream, which the output device interprets as formatting instructions, rather than text. This markup is obviously specific to a particular text formatter and style sheet. It is also device-dependent. For example, the instruction to skip three lines could be changed to a value such as eighteen points for a high-resolution printer [CRD87]. Procedural markup is typically associated with batch text formatters.

3.4.1.2 Generalised Markup

Generic coding involves identifying each element in a document and marking it with tags that specify the document's structure instead of its appearance. Generalised markup extends generic coding. The assumption behind generalised markup is that documents have a structure consisting of logical components which should remain separate from the style of the document.

Generalised markup is based on two concepts [GOLD90]:

- markup should describe a document's structure and other attributes, rather than specify processing to be performed on it, as generalised markup needs to be done only once and will suffice for all future processing.

- markup should be formally defined, so that techniques available for processing formally defined objects, such as programs and databases, can be used for processing documents as well.

By separating presentational information from the structure, elements within that structure (such as chapters or paragraphs) can be identified, which tell the computer what the fundamentals of the text are. They can then be programmed to make intelligent choices about formatting and organisation. For example, generalised markup can create multiple presentations of the same information [ARBO95]. A single set of source files can be processed by different pieces of software, with each applying different processing instructions to the relevant parts. This is because the software first reads a set of rules that establish the procedure for each occurrence of each element type [CRD87]. By updating this set of rules, different processing instructions can be associated with any one part of the file. For instance, one program might extract names from a document to create an index or database, while another operating on the same text might print names in a distinctive typeface.

Generalised markup languages often allow the user to define tags that describe a format (e.g. bold). However, this is against the principles of generalised markup and is therefore discouraged.

3.4.1.3 Other Forms of Markup

Referential Markup refers to entities external to the document and is replaced by those entities during processing [CRD87]. For example, it can refer to entities stored in a separate file (such as graphics), as well as being used for device-dependent punctuation or abbreviations, (e.g., &dcu can be replaced with "Dublin City University" during processing).

Metamarkup provides a facility for controlling the interpretation of markup and for extending the vocabulary of descriptive markup languages [CRD87]. For example, procedural and descriptive systems allow markup delimiter characters to be defined. Procedural systems also enable the user to define macros, which can be used to create descriptive markup representing a series of processing instructions.

3.4.2 Markup Handling

Goldfarb [GOLD90] identifies three distinct stages in marking up a document:

1. Element recognition
2. Markup selection
3. Markup performance

1. **Element Recognition**: The author analyses the document, identifying each separate element and characterising it appropriately (e.g. as a paragraph, heading, ordered list, footnote). This step is the same for all forms of markup. [GOLD90]

2. **Markup Selection**: A processing function is associated with the element recognised in the first step and the corresponding markup is applied to all occurrences of it. [GOLD90]

3. **Markup Performance**: Markup can be performed, including typing the markup almost as if it were text, using function keys or selecting items from menus. Any of these methods can be applied to each form of markup. [GOLD90]

Once the text has been manually marked up, there are three more steps taken by the software:

4. Representing markup
5. Storing markup
6. Processing markup

4. **Representing Markup**: Once the markup has been performed it is depicted in the text editor interface. Coombs et al. [CRD87] define 4 categories:
   - **Exposed**: Formatting codes are shown as they occur in the source file, without performing any special formatting. This is typical in systems with separate editors and formatters, such as TeX.
   - **Concealed**: A formatted representation of the markup is displayed, but the underlying formatting codes are concealed entirely. This is typical of WYSIWYG software, such as Microsoft Word.

- **Disguised**: Markup is processed and then disguised behind a special character that is shown to the user. An example of this is the *show/hide non-printing characters* option in Word which allows the user see special representations of the scribal markup such as tab characters, spaces and paragraph marks. e.g. "¶" represents a paragraph mark. (However this option does not display electronic markup such as fonts, bold, etc.).
- **Displayed**: Codes in the source file are displayed on-screen along with the formatted text. For example, WordPerfect 5.x formats text for editing but displays markup along the bottom of the editing window.

5. **Storing Markup**: Markup can be stored in many ways. Moreover, systems can elicit one type of markup but store another. For instance, a system can elicit presentational markup but store procedural markup [CRD87], e.g. if text is marked as centred, the line is centred in the editing interface but the markup recorded around the text could be procedural markup. In other words, text displayed as:

```
                              CENTRED TEXT
```

many be stored as:

```
.cm center
.bd .ce  "CENTRED TEXT"
```

(where the commands used are from the text formatting language Waterloo Script [HERW, p5]).

The text also could simply be surrounded with blank spaces which are not differentiated from the text, either on screen or in the file, as follows (where the "_" represents a space):

```
______________________CENTRED TEXT______________________
```

6. **Processing Markup**: There are currently three main types of markup processing [CRD87]:

   1. Reading (by humans).
   2. Formatting.
   3. Open-ended (including formatting).

   Presentational markup is designed for reading as it focuses on the final appearance of a document. ***Procedural markup*** is designed for formatting, but usually only by a single program. ***Descriptive markup*** can be read but is primarily designed to support an open class of applications (for example, information retrieval).

3.4.3 Comparing Forms of Markup

Of the six[6] types of markup, only the specific markup (procedural and presentational) and generalised markup offer a choice. The rest are used along with another form of markup.

The superiority of descriptive markup can be shown by the following comparisons between specific and generalised markup:

- **Information Stored**: In marking up a document, information about appearance (e.g. page layout, font sizes) is kept separate from its structure (e.g. the number of chapters, the order of paragraphs), despite the fact that formatting is generally determined by the structure. A major difference between the two types of markup is that specific markup records the format or appearance, whereas generalised markup stores structural details of the document. However, formatting can be applied to this structural information, whereas the structure cannot be inferred from format recorded by specific markup.

- **Maintenance**: Markup may have to be modified during the development of a document. Specific markup requires the author to repeat the markup process throughout the whole document to reflect the changes. Generalised markup just needs a single change to the text formatter's rule base, reducing the time, costs and possibility of error normally incurred by editing the document itself.

- **Portability**: Due to their widespread distribution, document portability is of major concern. Exchanging documents electronically between different systems can cause huge difficulties. Documentation with specific markup will mean an agreement must be reached on the format, or else the recipient will need to translate the data into the new format. Generalised markup is not tied to any particular system as the

[6] Procedural, presentational, generalised, punctuational, referential and metamarkup.

structure of the document will not change. It also protects the text from misinterpretation by identifying each element's purpose.

- **Machine Readability** : Documents must have a defined structure, for computer analysis. Without structure, text is simply a "character string that has no form other than that which can be deduced from the analysis of the document's meaning" [GOLD90, p17] and computers cannot do this themselves. Using generalised markup transforms text into a collection of highly structured text elements, enabling selective and systematic processing (e.g. the ability to generate table of contents and indexes) and full text retrieval.

Recognising its superiority, many publishers and organisations have joined in an effort to establish an industry-wide standard based on generalised markup [CRD87]. In its Electronic Manuscript Project, the Association of American Publishers (AAP) found that generalised markup to be the most effective means of establishing a consistent method for preparing electronic manuscripts which can feed the publishing process [CRD87]. The AAP has endorsed the ANSI-ISO SGML (a language for defining generalised descriptive markup) and developed its first application.

In the following sections, we look at MIF as an example of a specific markup scheme, and SGML as generalised markup. We then compare the two schemes, using these as the basis for our discussion. This comparison is used to emphasise the difference between the schemes that must be overcome to create a generic process to handle both.

3.5 MIF (Maker Interchange Format)

Maker Interchange Format is a format that can represent all the text, graphics, formatting, and layout constructs in a Frame[7] document as a group of ASCII statements. Because MIF is a textual representation of a document, it can be read by

[7] Frame Technology Corporation (recently taken over by Adobe Systems Inc.) produced a number of Frame Products, including FrameMaker, FrameBuilder and DL Composer. The Frame documents referred to here are those created from these products.

most systems and is easily parsed [ADOB95]. Therefore it can be used to allow Frame products and other applications to exchange information while preserving graphics, document content, and format. It is usually generated by a Frame product but can be created using a text editor.

3.5.1 Objects in MIF

Frame products treat each document as an *object* and store document preferences as properties of the document, e.g. a document's page size, pagination style, view options and current user preferences [ADOB95]. A Frame product also represents document components as objects. Different types of objects represent different components in a Frame document. For example, a paragraph is considered an object, as is a paragraph format, called a *formatting object*. Each object has *properties* that represent its characteristics. For example, a paragraph has properties that represent its left indent, the space above it, and its default font. A rectangle has properties that represent its width, height, and position on the page.

When a Frame product creates a MIF file, it writes an ASCII statement for each object in the document. The statement includes substatements for the object's properties. For example, consider a document (with no text frame) containing a rectangle that is 2 inches wide and 1 inch high. The rectangle is located 3 inches from the left side of the page and 1.5 inches from the top. MIF represents this rectangle with the following statement [ADOB95]:

```
<Rectangle                    # Type of graphic object
   <ShapeRect 3.0" 1.5" 2.0" 1.0"> # Position and size: left offset,
                                   # top offset, width, and height
>
```

Therefore, even though MIF is essentially a specific markup system (i.e. its properties describe the appearance of the final document), it also has some object-oriented features which means it has a greater capability to describe a document's structure.

3.5.2 Statements in MIF

When a Frame product creates a MIF file, it writes an ASCII statement for each object in the document. MIF also enables macros to be designed and used with the define statement.

The following conventions are used in MIF files to describe syntax [ADOB95]:

<token data>

where *token* is an indivisible group of characters that identify one of the MIF statement names (such as Pgf, representing paragraph) and *data* represents one or more numbers, a string, a token, or nested statements.

Some MIF statements can contain other statements, called substatements. A MIF main statement appears at the top level of a file. A main statement cannot be nested within other statements. Some substatements can only appear within certain main statements.

3.5.3 MIF Files

The only statement that is compulsory is the <MIFFile> statement, which must appear on the first line of the file. Without it, a Frame product simply reads the file as a text file. Frame products provide all of the other objects, even if the object is empty. Because of this, MIF files generated by a Frame product can be very lengthy. This is true of most specifically marked-up documents that are generated by a package (e.g. MIF from Frame products, RTF from Microsoft). However, files generated manually usually only have the minimum number of statements necessary, although such files are rare.

Below is an example of a MIF file that uses only four statements to describe a document containing one line of text [ADOB95].

```
<MIFFile 5.00>              # The only required statement
<Para                       # Begin a paragraph
   <ParaLine                # Begin a line within the paragraph
     <String `Hello World'> # The actual text of this document
   >                 # End of <ParaLine> statement
>                  # End of <Para> statement
```

Using this 6-line file, the MIF interpreter will generate over 1,000 lines of MIF statements that describe all the default objects and their properties [ADOB95]. Although this may be overkill, it makes parsing the file easier as the interpreter knows exactly what to expect and where to expect it.

3.5.3.1 Parsing MIF files

Most[8] Frame products have a MIF interpreter that reads and parses MIF files [ADOB95]. When a MIF file is opened or imported, the interpreter reads the MIF statements and creates a Frame document that contains the objects described in the MIF file. The algorithm used by the interpreter as outlined in the MIF On-Line Reference [ADOB95] is as follows:

- Markup statements are always delimited by angle brackets ("<" and ">"); macro statements are not, but when using a macro in a MIF file, macro names must be enclosed in such brackets to comply with the MIF syntax.
- The MIF interpreter scans the file for a left angle bracket marking the beginning of a MIF statement. When the MIF interpreter finds white space characters that are not part of the text of the document (e.g. in < Units Uin >), it interprets the white space as token delimiters. When parsing the example statement, the MIF interpreter ignores any white space characters between the left bracket (<) and the first character of the token, Units.
- After reading the token, the MIF interpreter checks its validity. If the token is valid, the interpreter reads and parses the data portion of the statement. If the token is not valid, the interpreter ignores all text up to the corresponding right angle bracket (>), including any nested substatements. The interpreter then parses the file for the next left angle bracket starting the next MIF statement.

3.5.4 Why choose MIF?

The decision to use MIF to represent specific markup in our project was taken for a number of reasons:

[8] All Frame products with the exception of FrameReader understand MIF.

- It is a textual representation of output from a DTP package rather than a WP package. Despite the growing popularity of the use of WP software to create documentation, DTP is still the most commonly used.
- FrameMaker is one of the most widely used DTP packages for product documentation. Many translation companies, such as ITP, design their tools to work with FrameMaker output.
- The specification for MIF is readily available.

3.6 SGML

SGML has become the leading international standard for data and document interchange in open systems environments. In fact, it is the ISO's most widely accepted standard [INTE94]. SGML has the support of many well known members of the SGML Open Consortium (including Adobe Systems, Corel Corporation, Oracle Corporation) who have used it in a wide variety of applications such as books, articles, technical reports and hypermedia, published both on paper or electronically. SGML is not limited to textual applications; it is perfectly suitable for use in Electronic Data Interchange (EDI) and can also be used successfully as an intermediate language for data conversion. The use of generalised markup languages is becoming increasingly popular. Many organisations use SGML in text processing, including the US Department of Defence, the Association of American Publishers (AAP), Hewlett-Packard and Kodak [USER95]. The most popular application of SGML is HTML, the formatting standard at the heart of World Wide Web documents.

3.6.1 Introduction to SGML

Standard Generalised Markup Language (SGML) is the International Standards Organisation's standard for document description (ISO 8879). SGML is a metalanguage for formally defining markup languages. In other words, SGML does not impose its own tag set but defines a language for authors to describe the structure of their documents and mark them accordingly. It is therefore flexible and open to new applications.

SGML provides a vendor-neutral, formal international standard for information interchange which frees that information from the constraints of particular formats, applications, and computing platforms so that it can be used by any system. All of the information about the text is coded in ASCII characters allowing the interchange of text across platforms.

3.6.1.1 Why is SGML so Different from Specific Markup?

Burnard and Sperberg-McQueen [BSM] identify three characteristics of SGML which distinguish it from other markup languages: it is *generalised descriptive markup*, all documents are of a *document type* and it exploits the notion of *data independence*:

- **Generalised Descriptive Markup**: SGML has the benefits of a generalised markup system (as described in the previous section) and does not restrict documents to a single application, formatting style, or processing system.
- **Document Type**: Every document can be categorised as being of a particular document type, and must conform to the corresponding document type definition (DTD). By specifying what parts documents will and will not contain, it is possible to create documents that computers can work with predictably [ENL96]. Humans intuitively know that different documents have different components, and can determine if they are of a certain type by checking to see whether they have certain components. With a little help, computers can do the same using well-known parsing techniques.
- **Data Independence**: Most formatting information (e.g. typesetting codes, specific font names, page breaks) is proprietary, which makes it restrictive [INTE94]. SGML ignores these formats, and focuses on the content and structure (the relationships among the data) of the information, allowing it to be used and reused by a wide range of applications. Because it is independent of any one system, it enables the interchange of text across platforms.

3.6.2 Components

SGML represents documents by modelling them as tree structures (with additional connections between the nodes). This technique works well in practice because most conventional documents are in fact tree structures, and because tree structures can easily be flattened out for representation as character sequences [GOLD90, p 127].

The document as a whole is called the "*document element*". In the tree structure, it is represented by the top node. Each node represents an "*element*", an identifiable part or object within the document. Each element is classified as being of a particular "*element type*", which is a class of elements with similar characteristics, e.g. paragraph, chapter, footnote. The descendants of a node are considered the "*content*" of that element. An element can contain simple text elements, elements of other types or nothing at all. The terminal nodes comprise of the actual characters or other data (e.g. images).

SGML documents have three required elements:

- ***The Document Type Definition***: A DTD defines the structure of a document by telling the computer what to expect in that document.
- ***The SGML declaration***: This defines "which characters are used in the DTD and the document text" [HERW90, p13]. It defines any special SGML features used in the document, such as the base character set used, the maximum length of tag names, symbols used for tag descriptions. It can be stored independently of the document that uses it.
- ***The Document Instance***: This is the actual marked-up text that has been encoded by SGML. It contains the text, a reference to the DTD, and is marked-up based on the rules of the DTD.

3.6.2.1 The Document Type Definition

The tree structure for any particular document is represented by its *Document Type Definition* (DTD). The DTD is a set of declarations which define the elements that can occur in a document, what they can contain, their relationships and the tag set to mark the document. These rules help ensure that documents have a consistent, logical structure.

The three most important kinds of declaration that can occur in a DTD are [GOLD90, p26]:

- "An ***element declaration***, which defines the general identifiers (GIs) that can occur in each element and in what order". An element is a component of the hierarchical structure defined by a document type definition. Elements are classified as being of a particular *element type*, a class of elements with similar characteristics, e.g. paragraph, chapter, footnote.

- "An ***attribute definition list declaration***, which defines the attributes that can be specified for an element, and their possible values". The attribute of an element in SGML is "a characteristic quality, other than type or content" [GOLD90, p252]. An important use of attributes is for creating cross-references in a document. The attribute definition list declaration establishes the attributes for elements in the DTD.
- "An ***entity declaration***, which defines the entities that can be referred to in documents of this type". An entity is a "symbolic name for any type of data" [HERW90, p36] where the parser substitutes the symbolic name with the data each time it occurs in a document, for example they can be used as a short-hand notation for text strings that are lengthy or cannot be entered conveniently with the available keyboard or to imbed documents stored in separate files into the main document.

3.6.2.2 Example DTD and Conforming Document

The following DTD describes a simple memo. The document type is "Memo". The elements allowed in a Memo are To, From, Body, Para and Close. The description of these elements and the relationships between them are described in the ELEMENT declarations. The To, From, Para and Close elements contain only text. A Body element contains any number of Para elements. The order in which these can occur in the document is defined in the Memo element declaration: The To and From must both occur, in any order, but before the other elements. They are followed by a Body element, and a Close element can follow this, but is not necessary.

A Memo can also be considered public or confidential, the default setting being public. The ParaRef element is used for creating cross-references to paragraphs in the document.

```
<!ENTITY % doctype "Memo" -- document type generic identifier --                 >
<!--          ELEMENTS   MIN    CONTENT (EXCEPTIONS) --                           >
<!ELEMENT  %doctype;     - -    ((To & From), Body, Close?)                       >
<!ELEMENT  To            - 0    (#PCDATA)                                         >
<!ELEMENT  From          - 0    (#PCDATA)                                         >
<!ELEMENT  Body          - 0    (Para*)                                           >
<!ELEMENT  Para          - 0    (#PCDATA)                                         >
<!ELEMENT  ParaRef       - 0    EMPTY                                             >
<!ELEMENT  Close         - 0    (#PCDATA)                                         >
<!--         ELEMENTS   NAME     VALUE                DEFAULT                     >
<!ATTLIST  Memo         status   (confiden|public)    public                      >
<!ATTLIST  Para         id       ID                   #IMPLIED                    >
<!ATTLIST  ParaRef      refid    IDREF                #REQUIRED                   >
```

A Memo document conforming to this DTD could look like the following example, where the first line references the DTD:

```
<!DOCTYPE Memo SYSTEM "C:\DTDS\MEMO.DTD">
<Memo>
<To>John</To>
<From>Joe</From>
<Body>
<Para>I cannot make our meeting scheduled for tomorrow afternoon. Can we re-
schedule it for Friday?</Para>
</Body>
<Close>Regards, Joe</Close>
</Memo>
```

3.6.3 Summary of the Advantages of an SGML-based Approach

The decision to include SGML in the markup schemes dealt with in this project was based on the following reasons:

- its many advantages over specific markup, as already outlined.
- its growth in popularity in industry. Numerous influential companies named above have given their support to it, and many of the popular DTP packages, such as FrameMaker, have versions that work with SGML.
- its differences from specific markup, especially its descriptive qualities. By incorporating the ability to handle SGML into the process increases its scope and therefore improves its generality.

3.7 LaTeX

Although LaTeX includes tags describing formats (e.g. italic), it can also be considered as generalised markup because of its macro commands with logical names, such as “title”, “section” and “quotation”. “LaTeX can thus be said to be a generic markup language, though it can be used in an old, physical way or in the newer logical way” [DILL97].

The following example (with LaTeX commands highlighted in bold text) shows how LaTeX can be used descriptively as generalised markup:

```
\documentclass [12pt]{article}
\begin{document}
\title{LaTeX Overview}
\maketitle
\section{Overview}
LaTeX is considered to be generic markup because of its macro commands with
logical names, such as:
\begin{itemize}
\item “title”
\item “section” or
\item “quotation”
\end{itemize}
\section{Logical Or Physical}
“These logical tags coexist with the physical ones, so the user can define the
physical appearance if they wish, but otherwise this can be done using style sheets
for the type of document they declare their work to be. LaTeX can thus be said to
be a generic markup language, though it can be used in an old, physical way or in
the newer logical way” [DILL].
\end{document}
```

Figure 3.5 Example LaTeX file

The decision to include LaTeX in our project was taken because it is one of the most popular text-based markup languages, and its specification is readily available. It also is an example of a language that can be used both as a specific or generalised scheme.

3.8 Comparison of Specific and Descriptive Markup

Documents comprise three types of information: content, structure, and formatting. Whereas specific markup only records the format of a document, descriptive markup recognises that these are separable elements. It preserves the content and structure, but does not specify the format of the document, maintaining that format should be optimised to user requirements at the time of delivery [OPEN96].

We now give a comparison of specific markup (using MIF as an example) and generalised markup (represented by a sample scheme defined using SGML) under these three factors. This highlights the differences between the markup schemes that must be overcome to allow a process for their comparison to be formulated.

3.8.1 Content

The content in a document is the information itself. Even though this is usually in the form of text, images, graphics, charts and even multimedia objects such as video and sound can be included in electronic documents.

When a document is tagged with specific markup there is no extra information recorded about the content. It is given no regard, except to deduce its place in the structure of the document in order to apply a formatting style during the document's creation.

In documents that conform to SGML, each element in the content (as signified by a DTD) is recognised and its purpose identified by tags marking the beginning and end of the element. A content model defines which sub-elements and character strings are allowed in the content, and where they can occur.

The following example shows a single topic containing a paragraph element. The paragraph contains another element, a note. The text in (a) is tagged using specific markup which does not record the meaning of the content it marks:

```
a)  <Para>This is the content of this document. <Font <FAngle 'Italic'>> Note:
    content is the information itself. <Font <FAngle 'Regular'> This is the next
    sentence in the paragraph.
```

In (b), the use of generalised markup clearly indicates the start and end of each element, and what it is:

```
b)  <Topic><Par>This is the content of this document. <Note> Note: content is
    the information itself. </Note> This is the next sentence in the paragraph.
    </Par></Topic>
```

3.8.2 Structure

The structure of a document is informally defined as the set of elements in that document and the relationship between those elements [MARC96]. The appearance of a document is deduced from its structure - as Marchal [MARC96] states, "ideally a text is formatted to expose its structure to the reader because good formatting when constantly applied is a real help to a reader". People rely on typographic conventions (such as titles in bold) to help build a mental image of the document structure.

When authors use specific markup, they must first determine the role of the text in the document (e.g. a paragraph, a footnote) before choosing an appropriate format. Because the markup only specifies the appearance of the text, information about the document structure is lost. For example, if italics are used to mark both quotations and emphasised words, then no difference in the meaning is recorded by the tags. Many specific markup schemes do identify certain elements in the text, such as paragraphs, but these are very limited.

SGML uses generic coding to determine the structure of a document. Each element in the document is identified and marked with tags that specify its purpose instead of its appearance. The structure of the elements within the document is enforced by the particular DTD being used.

The following example shows a single section containing two elements: a title and a paragraph. The paragraph contains a note element. The text tagged with specific markup in (a) only denotes how it is to appear. The reader must determine what the structure of the text is, and it is very difficult for humans, let alone computers, to do this.

```
a)  <Para> <Font <FWeight 'Bold'>Content <Font <FWeight 'Regular'>
    <Para> This is the content of this document. <Font <FAngle 'Italic'> Note:
    content is the information itself. <Font <FAngle 'Regular'> This is the next
    sentence in the paragraph.
```

The structure and hierarchy of a document is exposed to both humans and computers by the nesting of descriptive tags in an SGML markup scheme (b):

```
b)  <Section><Title>Content </Title><Par>This is the content of this document.
    <Note> Note: content is the information itself. </Note> This is the next
    sentence in the paragraph. </Par></Section>
```

SGML recognises that documents are processed according to their structure and formalises this practice to replace the implicit manual treatment with an explicit automatic one. Recognising the structure enables many processes to be automated [MARC96], including:

- **formatting**: Mapping the structure to formatting attributes is a simple task. For example, an element marked as a "title" will be in a larger bolder font, "paragraph" elements will be in normal font, etc.
- **indexing**: This is simply a matter of extracting relevant elements.
- **conversion**: Structure provides semantically-rich information and therefore conversion into any other format is almost always possible.

3.8.3 Format

Even with the introduction of graphical tools like FrameMaker and Microsoft Word, the underlying text is still marked with commands specifying the format, which imply only the document's structure [APPL94]. SGML is a neutral encoding language, where the underlying markup commands store the structures in the document, with the appearance determined from the structure by the specific software application. Formatting can be updated simply by changing the program that composes the source file.

There are many different ways to convey the meaning of text depending on the medium and audience in question, e.g. the same text can be used to create help, printed documentation, on-line documentation and WWW pages, but each of these formats has its own conventions and requirements. With specific markup, one must try to specify how the text should appear on every conceivable output (an almost impossible task), as the tags instruct a formatter as to how the document should look. Using generalised markup, the meaning is tagged and the formatting software is able to map that meaning to the desired target output.

For example, the text in (a) below is marked up using MIF to make the sentence "This is an important note." appear in bold text.

```
a)  <Font <FWeight 'Bold'>> This is an important note. <Font <FWeight
    'Regular'>>
```

In an SGML document (b), the meaning of the text is marked up, as opposed to specifying the appearance.

```
b)  <NOTE> This is an important note. </NOTE>
```

The formatting software determines how the NOTEs appear. In HTML, they might be bolded. In the Netscape extensions to HTML they might be blinking. In a colour printout they might blue. In black and white printout they could be underlined. If any particular formatting language had been used, it would not have been possible to output so many different formats.

Since SGML is neutral, formatting is determined by the software application itself [INTE94]. However, some formatting information is useful in SGML - that which transcends any particular display system, like specifying the number of columns in a table. SGML permits tags to have specific formatting significance, but does not encourage this as it detracts from its generality.

3.9 Summary

In this chapter we discussed Word Processing and DTP software, and identified why our research must deal with both. The current methods of processing documents were described, and our idea of a generic process introduced. We then identified how the formatting and structure of documents are represented by markup codes within the document file. Using MIF and SGML as examples, the two main types of markup (specific and generalised) were examined, showing the differences in the schemes which indicates difficulties in trying to create a process for comparing them.

This discussion provides a basis for our research into the formulation of a process that can manipulate documents with either type of markup scheme, and allow the markup in any document to be compared to the markup in any other document, regardless of its type. In the next chapter, we outline our design for such a process.

4. Generalising Markup Comparison in Documents

4.1 Introduction

The main aim of this research is to devise a generic process to compare the format and structure of two documents. Because the format and structure are recorded by markup codes, it is the set of tags from each document that must be compared.

However, due to the vast differences in the markup schemes outlined in the previous chapter, this is not always a direct comparison of tags. A tag-for-tag comparison can only be applied when the documents to be compared both use identical markup schemes, for example two MIF documents or two SGML documents with the same DTD. This process is discussed in section 4.2. We have identified three other cases in which some conversion process is necessary to allow this type of comparison to be implemented:

1. Comparing two specifically marked-up documents.
2. Comparing two documents with generalised markup.
3. Comparing documents with different types of markup (i.e. specific & generalised).

To compare two specifically marked-up documents, they must first be converted to the same tag set to allow a direct comparison of tags, as discussed in section 4.3.1. Two documents with generalised markup must both describe the same elements before a comparison can be made, as described in section 4.3.2. For documents of different markup categories, they must first be converted to an intermediary format, as discussed in section 4.4. All of these processes are then brought together to give an overview of the conversion process. Finally, we outline the tag-for-tag comparison that can be applied to the documents after any necessary conversion.

4.2 Comparing Documents With Identical Formats

Because the same tag set is used for both documents, this is a tag-for-tag comparison. However there are a number of possible complications to this process, concerning style sheets and matching tags.

4.2.1 Style sheets

Style sheets are a collection of pre-defined styles, with each style having a name and a set of formats that can be applied to text. When style sheets are used in documents, this raises the issue of whether style names or formatting information should be compared. For example, <heading1> may be defined as being bold, with size 24 font Arial in one document, but underlined with font size 20 in the other. By examining how style sheets are used in documents it can be seen that this will not cause the difficulties first supposed. It is often possible to compare both separately.

4.2.1.1 Comparing Style Properties

Many file formats, such as RTF, store the properties of the style sheet in the header of the document. The example below is extracted from the start of an RTF document. (Note: the layout has been changed marginally for clearer presentation).

```
{\stylesheet
      {\widctlpar \f4\fs20\lang2057 \snext0 Normal;}
      {\s16\widctlpar \b\f4\ul\lang2057 \sbasedon0\snext0 Heading;}
}
```

where [MICR95]:

- **\widctlpar** indicates that widow/orphan control[9] is used.
- **\fN** is the font number, where N refers to an entry in the font table.
- **\fsN** is the font size in half points.
- **\langN** applies a language to a character[10], where N is the number of the corresponding language from the language table in the RTF header.
- **\snextN** defines the style for the paragraph that follows the current style e.g. \snext0 Heading: the paragraph after a Heading is 0, which is Normal.
- **\sN** identifies the paragraph style in the style sheet.
- **\b** is bold.
- **\ul** is underline.

[9] Widow/Orphan Control "prevents the last line of a paragraph by itself being printed at the top of a page (widow), or the first line of a paragraph being printed by itself at the bottom of a page (orphan)" [MSWord help]

[10] The spell checker and other proofing tools use the dictionaries of the specified language.

- **\sbasedonN** defines the id number of the style on which the current one is based, e.g. \sbasedon0\snext0 Heading: The Heading style is based on 0, which is Normal.

In cases such as this, each style is defined in the document, so when the documents are compared any differences in the style sheets will be noticed. Other file formats refer to an external style sheet. Lotus AmiPro records the name of the style sheet used in the header information in the document, rather than the details of each style, as in:

```
[sty]
     ut2suite.sty
```

Because each style is defined in the external style sheet, comparing the properties would require a comparison of the style sheets independently of the documents. However, we can recognise when different style sheets are being used.

4.2.1.2 Comparing Usage of Styles

When a style is applied to text, some applications write the style name to the output file. The following example is an extract from an AmiPro SAM file, where the text "The Document Title" is in the style of "Title":

```
@Title@The Document Title.
```

When the style name is used in the body of the document, we can detect if the two styles differ and report an error.

However, formats such as RTF store the properties of the style with the text, along with the style identifier. In the following example of text, the "Style Sheets" paragraph was formatted to "Heading" style and the paragraph of text was "Normal":

```
\par \pard\plain \s16\widctlpar \b\f4\ul\lang2057 Style Sheets
\par \pard\plain \widctlpar \f4\fs20\lang2057 Style sheets are a collection of pre-
     defined styles.
```

where [MICR95]:

- **\par** represents a new paragraph,
- **\pard** resets to default paragraph properties
- **\plain** resets the language property to the default.

Although **\sN** identifies the paragraph style in the style sheet, in such cases we must also compare the formatting information as well.

To summarise, the choice of comparing the style name or its properties cannot be decided by our process, but by the way in which the markup scheme stores styles and style sheets.

4.2.2 Matching Tags

An algorithm must be formulated for those instances in the tag-for-tag comparison where two tags do not match. It must determine if this is a case of a wrong tag used or whether tags are missing. If the latter is true, it must identify from which document the tags are missing and at what point they start to match up again. It is also possible that part of the document text was moved rather than deleted. The process devised to deal with this is described in section 4.6.

4.3 Comparing Documents of the Same Markup Category

To compare two documents of the same markup category (i.e. specific or generalised), one or both must first be changed to allow a direct comparison. For example, an RTF and a MIF document cannot be directly compared because of their differing markup. To formulate an algorithm for such a conversion, it was necessary to perform a review of existing work in this area.

Many applications such as word processors employ document conversion. For example, Microsoft Word can open documents created in Word Perfect or Lotus 1-2-3, and can save Word documents in Word Perfect format. However, this requires a filter for each pairing of formats used, defining the equivalence between them. There are also numerous tools to convert documents from one format to another, such as *LaTeX2HTML*, a LaTeX to HTML converter [DRAK94], *fm2html*, a FrameMaker to HTML converter [STEP94] and *Tex2RTF*, a LaTeX to RTF and HTML converter [SMAR95]. However all of these utilities were designed and written specifically to work with the specific pairs of file types. If one wished to convert to any other format, a new tool would have to be created. Some utilities even rely on the user to prepare the document first, making conversion little more than replacement. For example, the plug-in for converting Microsoft Word documents to HTML (before the widespread availability of WYSIWYG HTML tools) simply mapped certain formats to HTML tags. For example, text in the style of "Heading 1" in the Word document was surrounded by <H1> and </H1>, representing a heading in HTML.

Rather than providing a filter for each combination of markup schemes required, which is impractical for a large number of file types, or insisting on certain formatting in the document, which is cumbersome for the user, we propose to convert both documents to a generic internal format. This will require a single mapping for each tag in a markup scheme to its equivalent in the internal generic tag set. We have created our own file format, even though many existing ones are considered standard. This way, we can ensure that all tags from existing formats can be mapped to an equivalent, as we cannot guarantee this with any existing format. If one does not already exist in our tag set, it can easily be added, as described in the next chapter.

Due to the different concepts behind specific and generic markup, our algorithm will deal with each category separately. For specific markup, the internal tag set will need to represent formatting information. For generalised markup, the structural elements in documents must be able to be recorded by the internal format. The process for converting each markup type is explained below. Once the conversion is performed, the next step is the same as for identically marked-up documents.

4.3.1 Comparing Two Specifically Marked-up Documents

Comparing different specific markup schemes involves creating a mapping between them. We intend to accomplish this by mapping each markup scheme to an intermediary generic tag set. Such a tag set would need to encompass all possible formatting information to be able to represent any document's markup. However, this does not necessarily mean that an equivalent for every tag in all markup schemes must exist, but rather that the format described by a tag or combination of tags in each scheme has a corresponding tag. For example, MIF represents paragraphs with a Para tag containing as a number of ParaLine tags representing lines in the paragraph, whereas most other markup schemes consider the paragraph as a single element containing only formatting information (e.g. bold, italic). The following example shows the markup for a tag in MIF.

```
<Para
   <ParaLine
      <String 'This is '>
   >
   <ParaLine
      <Font
         <FWeight 'Bold'>
      >
      <String 'bold '>
   >
   <ParaLine
      <Font
         <FWeight 'Regular'>
      >
      <String 'text in MIF.'>
   >
>
```

Figure 4.1 Extract from MIF document

The following figure is an example of a paragraph in RTF.

```
/par This is {\b bold} text in RTF.
```

Figure 4.2 Extract from RTF document

Because there is no equivalent format or structure in the majority of markup schemes for a ParaLine tag in MIF, we do not wish to include it in our generic tag set. This allows for a better comparison of documents, as the format and structure of the documents are being compared instead of the actual tags. For example, it is inappropriate in trying to find a tag corresponding to a ParaLine tag in the RTF document, as no such tag exists.

For the purpose of this research we propose using a smaller subset, composed of the widely-used formats (e.g. font, font size, bold, underline), but allowing the easy addition of new ones as necessary.

4.3.1.1 Generic Tag Set

The formatting information we wish to represent in our tag set can be grouped into four different categories: character formatting, paragraph formatting, page formatting and objects.

Character Formatting can be applied to any single character or groups of characters in a document. For example, one word of a piece of text can be underlined. The following list contains examples of formats that can be applied to characters:

- Font Type
- Font Size
- Bold
- Italic
- Underline: Single, Double, Dotted, Words-Only
- Subscript
- Superscript
- Strike-through
- Colour

Paragraph Formatting is applied to the text of a whole paragraph, as opposed to groups of characters within a document. For example, if a piece of text is centred, the entire paragraph in which that that piece of text is contained will be centred. Paragraph formatting includes:

- Justification - left, right, centre, full
- Bullets & Numbering
- Indentation - left indent for first line, left indent for body of paragraph, right indent
- Line Spacing
- Paragraph Spacing - before paragraph, after paragraph
- Character Spacing

Page Formatting refers to the formats that can be applied to a single page of a document. This includes:

- Paper Size
- Orientation - portrait, landscape
- Margin - left, right, top, bottom

Objects: A document can contain many objects, even when specifically marked-up. However, these objects are usually related to the appearance of the document, rather than its structure. For example, borders and shading are physical, rather than logical, attributes. Specific markup schemes can contain objects such as:

- Drawing Objects - line, text box, shape, etc.
- Graphics (i.e. bitmaps, etc.)
- Tables / Cells
- Page Break
- Section Break
- Carriage Return / Paragraph
- Header/ Footer
- Borders - left, right, top, bottom
- Shading
- Frames
- Links
- Cross Reference
- Index
- Table of Contents

This list of formats can be used as a basis to generate an internal tag set to which most tags found in documents can correspond. To allow the conversion of the markup in documents to its equivalent tag from our generic scheme, we must store the tag for each markup scheme being used (e.g. MIF, RTF) and its relationship with our design. An example is as follows:

GENERIC TAG	MIF	RTF
BOLD	FWeight 'Bold'	\b
ITALIC	FAngle 'Italic'	\i
PARAGRAPH	Para	\par
etc.	etc.	etc.

In this table, GENERIC TAG is our internal tag set and the MIF and RTF columns contain the equivalent tag for that scheme.

4.3.1.1.1 Other Considerations

Any parameters or attributes given in a tag must also be recorded. For instance, marking the left indent for a paragraph in MIF uses the following statement, where the 1.0" represents the size of the indent and therefore is significant:

```
<PgfLIndent 1.0">
```

We intend to do this by associating a parameter field with each internal tag that can take such values. For example, When the PgfLIndent is encountered, the 1.0" will be stored with it, in a separate field .

There are other rules in these schemes to which the document must adhere to be valid within that scheme. Consider the following example from a MIF document:

```
<Para                                                   # Begin a paragraph
    <Pgf                                                # Begin paragraph format
        <PgfAlignment Left>                             # Specify text alignment
    >                                                   # End of paragraph format
    <ParaLine                                           # Begin a line in the paragraph
        <String 'This paragraph is left justified.' >   # The actual text
    >                                                   # End of <ParaLine> statement
>                                                       # End of <Para> statement
```

A Pgf statement can only occur in a Para, a PgfCatalog or a TblFormat statement but never on its own. The PgfAlignment statement can only appear within other statements such as Pgf, or FmtChangeList, and can only have one of a defined list of parameters. As we are only concerned with comparing the tags to those of another document, verifying such rules is beyond the scope of this research.

4.3.2 Comparing Two Documents with Generalised Markup

As with specific markup, generic markup requires that both tag sets must first be converted to an internal format to allow a direct comparison. This appears a relatively easy task, similar to that for specific markup, and in some instances this is the case. A generic markup scheme such as LaTeX has a pre-defined, and therefore limited, tag set, it can be treated in the same way as described for specific markup i.e. we can specify an internal tag set to which these can be mapped. However, because generic markup describes the document's structure rather than the format, the internal tag set for specific markup cannot be used. Therefore we must extend our internal tag set to include structural elements such as:

- Document / Main Body
- Title
- Chapter
- Section
- Subsection
- List
- List Item
- Heading
- Table
- Table Cell
- Text
- Comment
- Note
- Highlighted Text (Emphasis)

Generalised markup languages such as SGML allow the user to identify their own elements in documents instead of conforming to a pre-defined set, and allow the definition of tags to mark up these elements. This can also be applied to style sheets, which can be used as generalised markup when the style name describes the elements in the text instead of the appearance required. Therefore, the technique used for specific markup would not work if we tried to apply it to generalised markup schemes. Although the majority of tags would be used for similar purposes, the name for each may differ. For example, one user many identify a paragraph as <para>, another with <p>, or <paragraph>. In fact, usually the only restrictions on naming tags are related to the length and characters permitted, not with the actual name given. Therefore <xyz> is a nonsensical, yet entirely valid, identifier for a paragraph tag, assuming the combination of the characters xyz is allowed by the scheme.

Another problem when applying the same procedure used for specific markup is the elements the user may need to categorise. A report may be structured as a title followed by a number of sections, each starting with a heading and containing paragraphs. Another document may contain poetry, with elements such as poem, verse, line, etc. Therefore, because a document can contain elements of many types of which we have no advance knowledge, our solution is to define a simple generic document structure which can be applied to any document. Because we know the reason behind the development of this process (i.e. software documentation), the elements allowed could be confined to those normally found in typical documentation, such as chapters, titles, headings and sections. However this would reduce the generality of the process, which conflicts with the basis of this research, the development of generic tools.

4.3.2.1 Generic Document Structure

Our generic document structure considers all documents to have a main body of text, which can be composed of one or more sections (e.g. chapters, or sections in a book). Each section can have sub-sections, paragraphs, or a combination of both in any order. These sub-sections have the same composition as a section, with the root sub-sections always containing a paragraph. Paragraphs can represent different elements, e.g. a heading or title, a list item, or simply text. However, each of these will still be composed of the same components - any combination of text, external entities (e.g. graphics) or links (if an on-line document). These are always considered the terminating elements in this structure. From this description we can create a standard template for the main body of a document, represented by the following graph:

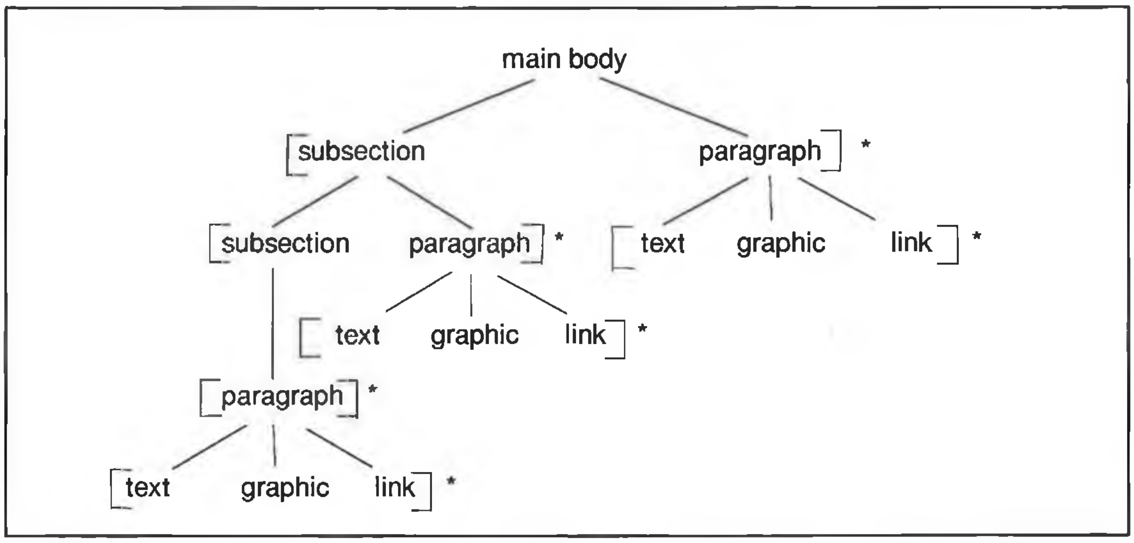


Figure 4.3 Generic Structure of a Document

This structure considers elements such as the title, introduction, abstract, appendices, etc. to be sections also, because in our representation a section can contain just a single paragraph. Hence the title could be regarded as a paragraph in a section of its own.

4.3.2.2 Converting Documents with Generalised Markup to a Generic Structure

This structure contains components which can be used to describe any type of element. For example, the poem element in a poetry document could correspond to a subsection, with the verse element matching a subsection of that section, and the line considered a paragraph under the assumption that each line of the poem will be ended with a carriage return. (This constitutes a new paragraph in word processing). Although it is relatively easy for a human to make these comparisons, defining a process to accomplish the same task is more difficult. The method we chose is to consider the document as a tree structure. For example, a document containing poetry could be represented as follows:

```
                 poetry document
                   /           \
           [ poem                 notes ] *
              |                      |
           [ verse ] *            [ words ] *
              |
           [ line ] *
              |
           [ words ] *
```

Figure 4.4 Example structure of a Poetry Document

By comparing this to our generic structure of a document, we can try to deduce a mapping for each element. The document element poetry document obviously corresponds to the main body of the generic structure. The terminal nodes, words, correspond to one the terminal elements of the generic structure: text, graphic or link. By examining the content of the element words allowed by the markup system, it can be determined to which of these three options it is equivalent. For example, if this were represented in an SGML DTD, the content list of the element words explains its allowed content. If this were PCDATA[11], this means anything delimited by the words tags can contain only pure text.

The intermediary elements can then be either subsections or paragraphs. This can be deduced by examining the permissible contents of these elements. Any element containing only terminal nodes (e.g. "line" contains only "words") can be considered the equivalent of a paragraph. All other elements can contain at least one other element that is not terminal (e.g. "verse" contains "line") and are therefore regarded as subsections.

Applying this structure to a document results in a great loss of detail and information. For example, lists are reduced to a series of paragraphs, as are headings and titles.

[11] PCDATA (Parsed Character Data) is "zero or more characters that occur in a context in which text is parsed and markup is recognised. They are classified as data characters because they were not recognised as markup during parsing" [GOLD90, p140].

The addition of optional elements common to the majority of documents, e.g. chapters, headings, etc. (as listed in 3.3.2 above for the generic tag set) would recover some of this lost information. However in an automated process, it would be very difficult to recognise such elements in the document. Although we can identify structural elements, we cannot determine the purpose of these. For example, we can identify that a title is stored as a paragraph, but we cannot tell that the text in this paragraph is a title. This can be overcome by presenting the user with our mapping and allowing them to choose a more suitable alternative for each tag from the additional list, if one exists.

4.4 Comparing Documents with Different Types of Markup

To compare documents with different markup schemes, some conversion process must first be performed to ensure both use the same tag set to allow a direct comparison, as with documents of the same markup type. The options available are to convert both documents to an internal format or to convert one document to the markup scheme of the other.

Converting the markup of both documents to a single internal format involves formulating a tag set that is capable of encompassing the characteristics of both. However, because there is no direct correspondence between the markup schemes, we cannot create a tag to which both the specific tag and the equivalent tag in the document with generalised markup can be mapped. For example, in the specifically markup up document a heading may be centred:

```
<CENTER>Document Comparison<\CENTER>
```

However, in the document with generalised markup, the title will be identified with a generic tag indicating what it is:

```
<HEADING> Document Comparison <\HEADING>
```

There is no tag to which both of these tags can be mapped as we cannot assume centred text will always be used for a heading and vice versa. Therefore this option is unrealistic.

Converting specific markup to a generalised markup scheme involves deducing the structure of the specifically marked-up document from its appearance. Although all documents have an inherent structure [QUIN90], this deduction is complicated by two facts:

1. there is no standard format for documents so, for example, one author may use large bold text for titles whereas another may have underlined centred text.
2. there is no guarantee that an author will consistently use the same formatting conventions throughout the whole document.

Using the generic structure as a basis for the deduction will simplify this process by limiting the number of elements to be recognised, and will also standardise the documents we are working with. Once a structure has been recognised for the specifically marked-up document, the document with generalised markup must be examined to ensure it also conforms to the generic structure. It may have elements specific to that document: for example, a document containing poetry may have elements such as poem, verse, line, etc. that cannot be incorporated into a generic structure.

To transfer a generalised markup to a specific markup scheme, we can easily apply formatting information to the generalised markup. However, there are a number of difficulties. We still need to impose a generic structure on the document with generalised markup for the same reason as above. Also, there is no guarantee that the formatting we apply will be the same as that used in the specifically marked-up document. To ensure that it is, the specific markup must be examined to determine its usage, identifying how the format is applied to the structure. In other words, the same process as for converting specific markup to generalised markup must be performed before we can even do this. Therefore we are converting from specific to generalised markup before we can convert from generalised to specific, doubling the processing.

Obviously much information will be lost in such a conversion process because of the huge difference in the information recorded by the two schemes. Transferring text from specific to generalised markup will result in the loss of all formatting information, as generalised markup does not store such information. Similarly, converting generalised markup to specific loses all structural information, because specific markup does not have the facility to store this. However, after processing, formatting can easily be re-applied to a document with generalised markup if required, whereas it is much more difficult to convert specific markup back to generalised markup. Using generalised markup also has many other advantages over specific markup as already discussed in Chapter 2. Because of this, and the fact that replacing specific with generalised markup involves less processing and is no more difficult, we would convert all documents to generic markup describing a generic structure.

4.4.1 The Process for Converting Specific Markup To Generalised Markup

To convert the documents from specific markup to generalised would involve using a set of rules to apply the generic structure to the document. Each piece of text must be examined to determine its purpose, as defined by the generic structure. Existing work in the area of deducing document structure from layout includes a system developed by Porter and Rainero [PORT92]. Their system is capable of "deriving a high level document structure from the layout and content of the document" [PORT92, p43]. It can accept documents in paper (e.g. scanned raster) or Postscript form. The document is then passed through three processes:

- The *Low Level Structure Reconstruction* (LLSR) process
- The *High Level Structure Reconstruction* (HLSR) process
- The *Output Conversion* process

The *Low Level Structure Reconstruction* process converts either raster or Postscript input into a layout for high level reconstruction [PORT92]. Therefore existing documents must be converted to Postscript for processing. For Postscript documents, this process executes the code to determine the primitive elements on the page: chars, lines, etc. High Level Segmentation constructs a tree containing textual elements such as word fragments (WF), spaces, graphical elements such as rule and line art, and images. The result of this process is the layout view structure containing the structural elements and the associated layout (Figure 4.5). This is passed to the High Level Structure Reconstruction process.

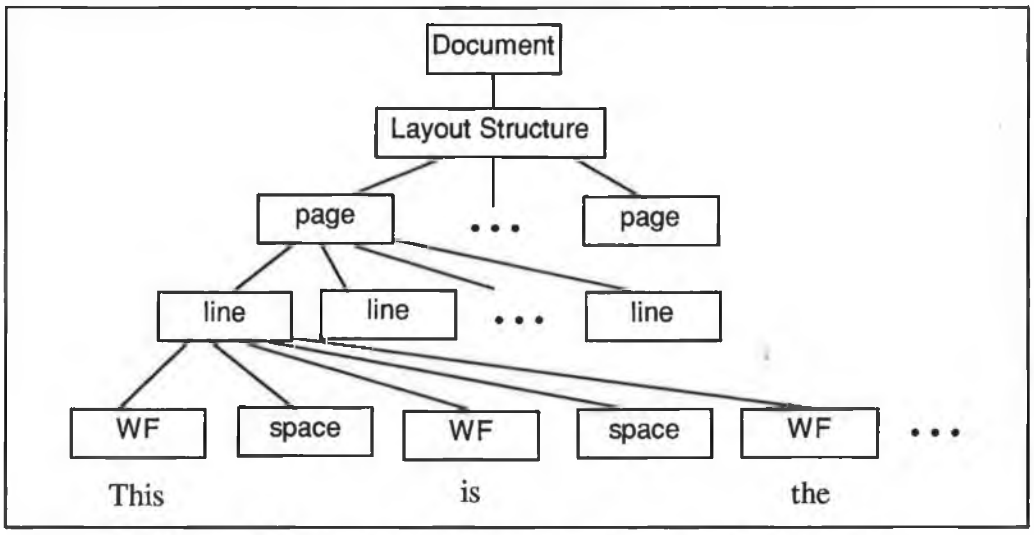


Figure 4.5 The Layout View of a Document [PORT92]

The *High Level Structure Reconstruction* process maps the layout view into a set of additional views [PORT92]. This process builds high level structural views of the document using various knowledge sources to improve classification, e.g. hyphenated words are looked up by a lexicon to decide whether to remove the hyphen. For example, if the document is a technical article, this involves classifying each line as part of a known structure type such as a PARAGRAPH, TITLE-PART, AUTHOR-PART, SECTION-HEADING, INDENTED LINE, HEADER, FOOTER, etc. (Figure 4.6) Currently reconstruction code has only been written for article style documents, but “additional document styles are being implemented” [PORT92, p51].

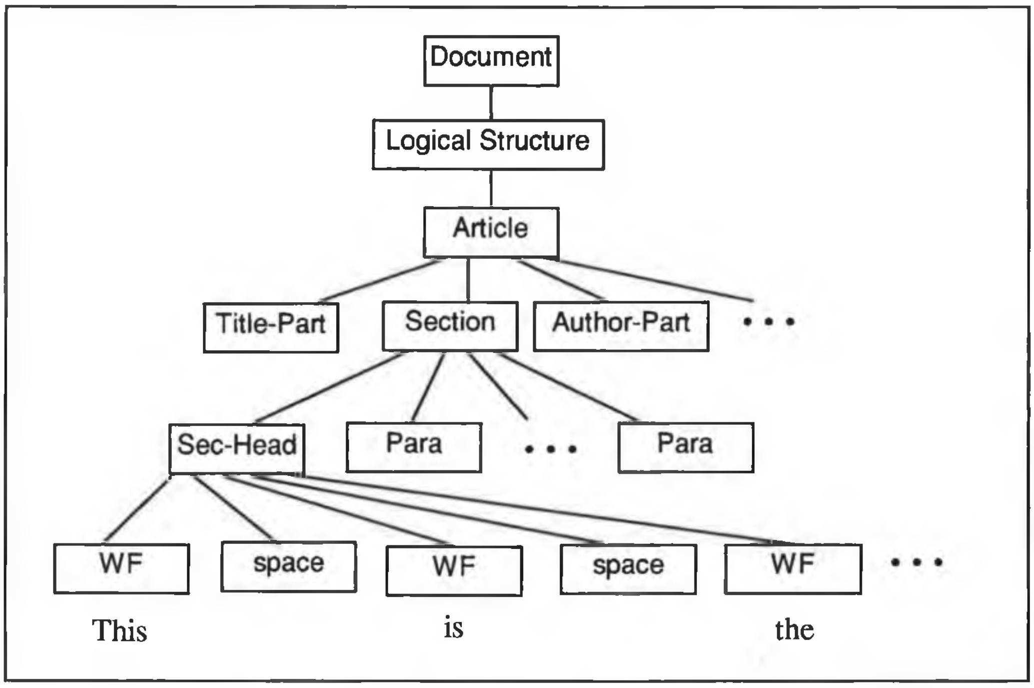


Figure 4.6 Logical Structure for a Technical Article [PORT92]

The result of the HLSR process is a mapping of the layout and logical views of the document (Figure 4.7) that is passed to the output conversion process for conversion to a specific form such as SGML.

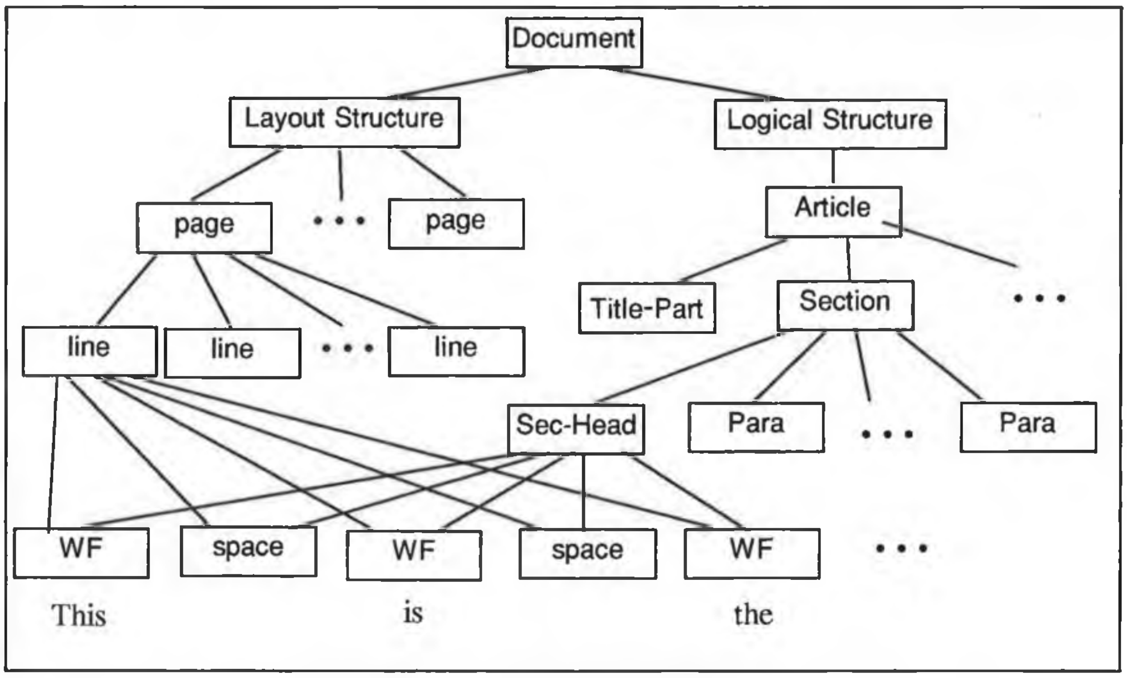


Figure 4.7 The Mapping Between Layout and Logical Structure [PORT92]

The *Output Conversion* process converts the multi-view representation into various external representation languages such as SGML [PORT92]. “Conversions are being written to map the layout/logical structure onto additional document description languages, such as LaTeX and Frame Maker Interchange Format (MIF)” [PORT92, p51].

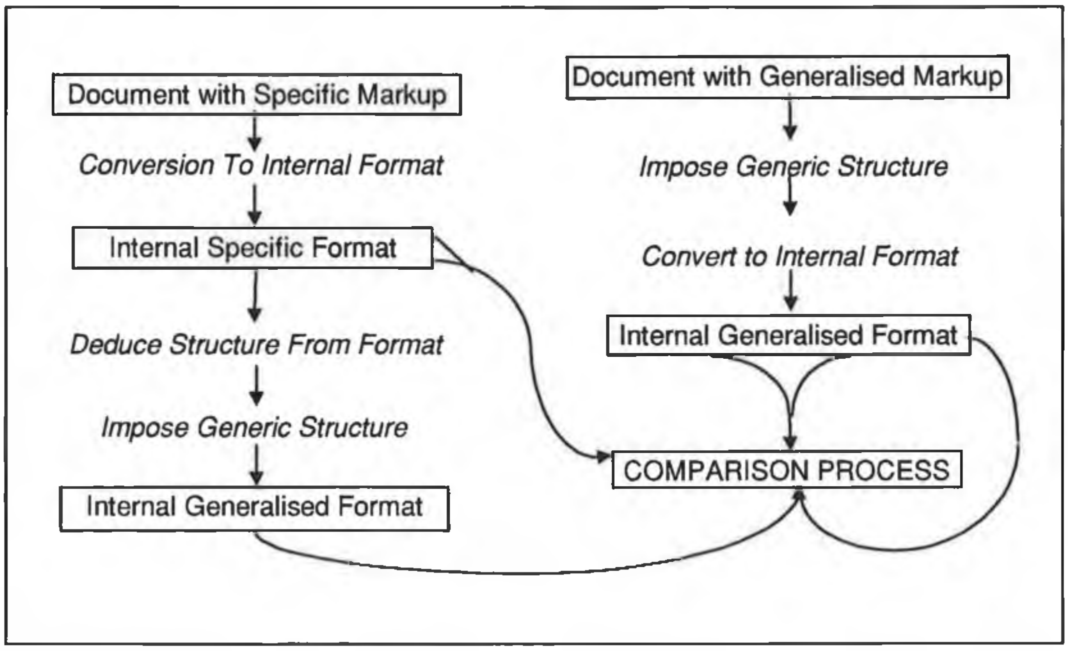
To summarise, this system takes the following steps:

```
  Document → Postscript
→ Postscript Interpreter → Rendered Page Description Language
→ High Level Segmenter → Layout View
→ High Level Structure Reconstruction → Logical View
→ Structured Document Conversion → SGML
```

This system uses a template for certain types of documents, reducing its generality. However, it takes a specifically formatted document and converts it to generalised markup which is what we want to achieve. Therefore it is proposed that the process on which this system is based could be incorporated in the conversion of specific markup to generalised markup to compare these markup types to each other.

4.5 Conversion Process for Marked-up Documents

Bringing together all the components described in this chapter results in a process that works as outlined in the following diagram:



Document with Specific Markup
Conversion To Internal Format
Internal Specific Format
Deduce Structure From Format
Impose Generic Structure
Internal Generalised Format
Document with Generalised Markup
Impose Generic Structure
Convert to Internal Format
Internal Generalised Format
COMPARISON PROCESS

Figure 4.8 System Overview of Conversion Process

If the two documents have the same markup schemes, no conversion is necessary, as they already use identical tag sets. Otherwise all specific documents are converted to the internal tag set describing the formatting information. All documents with generalised markup are converted to the internal generalised tag set describing the elements, having first been made to conform to a generic structure.

Two documents of the same markup category can be directly compared in the corresponding internal tag set. For example, two specifically marked-up documents would be converted to the internal specific tag set.

For documents with different markup types, the specific markup has an extra processing step before comparison. Its structure must be deduced from its formatting information, and then have a generic structure applied to it. It is then converted to the internal generic tag set for comparison with the internal representation of the document with generalised markup.

4.6 Tag-for-Tag Comparison of Two Documents

To perform a direct tag-for-tag comparison, both documents must first use the same tag set, as already described above. Once any necessary conversion is completed, the next step is to compare the second document with the first. Because of the nature of localised documents, a direct line-by-line comparison will not work, as each line is *expected* to be different after translation. Therefore the tags must be separated from the text.

The comparison process reads in the tags from each document, starting with the first, and comparing them. If they are the same, the next tag from each file is read in and compared. However, if the tags are different, this indicates either that there are tags missing from the second document, there are extra tags in the second document, or tags have been moved in one of the documents. The main difficulty in the direct comparison of tags is, determining the reason for any difference encountered and re-aligning the two documents appropriately. There are a number of existing difference algorithms for file comparison. However, they are designed to deal with files of text, rather than tags, and so are generally unsuitable.

For example, Lindsay's text file difference utility, *diff*, [LIND89] is based on the algorithm described in "A Technique for Isolating Differences Between Files" [HECK78]. The utility scans through each file and finds any lines that occur only once in the file. These lines can then be matched up in both files. It then checks lines which are next to matched lines, taking adjacency as enough reason to match such lines, even though other matches exist. This approach is totally unsuitable for comparing lists of tags, as is our requirement. The majority of lines in a text file will be different, giving an excellent start for the adjacency rule to work on non-unique lines. However, in our files, the probability of a tag being used only once is very small. Even looking at groups of tags will not greatly improve this approach, as in a consistently marked-up document the same combination of tags will be used for similar elements, e.g. all level two headings may be font size 12, and underlined. Therefore this gives a very poor starting point for the adjacency rule to work from.

Other work in the area of comparing two files includes Hearne's [HEAR97] "QA Tool for Help Files". This takes two help files and performs a basic comparison on each topic. However, because each topic in a help file is assigned a unique id number, this task is greatly simplified and does not require the re-alignment necessary with documentation files, which have no identification on chapters or sections. It also does not implement a detailed analysis of the format of the topics, which is our aim for documents.

GNU diff [SUNS97] works by identifying 'hunks', which are groups of differing lines in documents. It tries to "minimise the total hunk size by finding large sequences of common lines interspersed with small hunks of differing lines" [MACK93]. By conducting tests using the *diff* command on files containing lists of tags, we have discovered that it performs as well with these as with text files. Therefore we have decided to implement the concepts of this command in the comparison process. This algorithm is described briefly below.

Assuming that the tags in both documents are in the same format, we can compare the two documents tag for tag until a difference is found. When a difference between two tags is encountered, we need to determine if it is because of:

- *a changed tag*: The next X tags in both files are compared. Ideally if all the following tags are the same, then one of the two has been changed. However, we must allow for the possibility of one or more of the following tags being wrong also, so instead, we assume that if the majority are the same, one of the two tags that we are comparing has been changed during translation. Regarding the first document as the original, the tag in the second document is reported as having been changed.
- *An extra tag in document two*: if the tag has not been changed, the next matching tag in the second document is found. If none is found, then the tag is obviously missing from the second document. If a match is found, we must check the next X tags in both documents to ensure this is the matching tag and not a coincidence. If the majority of the tags are the same, we assume we have found our match. If not, this process is repeated until either a match is found, or we deem the tag to be

missing from file 2 because we cannot find a suitable match. Using the adjacency concept, we assume the subsequent unmatched tags in the first document to be missing from document two also. To re-align the two documents we follow the same procedure as if tags were missing from document 2, with the two original unmatched tags.

- *A missing tag from document two*: This is the same as checking for extra tags, except we try to find a match for the tag in document two in document one.
- *A moved tag*: Once all the matching up has been completed, the still unmatched tags are checked for groups that match up. If any are found, these are considered as having been moved during translation.

This algorithm is explained in more detail in the chapter on the implementation of a prototype.

4.6.1 Issues in the Tag-for-Tag Comparison

This process assumes that both documents should be identical (except for translated text) and reports any differences found as errors. However, for two reasons, certain differences are not actually errors. Firstly, the content of the documents may differ because of country-specific information, as described in Chapter 2. These differences may be due to extra or removed text, examples specific to the locale, enforced page breaks or the inclusion of different bitmaps for localised images. The second reason is differences in the markup schemes. For example, some schemes allow optional end tags. If one document omits the optional tags and the other uses them, this will cause a difference between the files that is entirely valid. Certain schemes also use different tags to end the same format. The order of tags marking the same text may differ in the second document. For example, the first document may specify the text to be bold and italic, whereas the second may be italic and bold. Another allowable difference in the documents is due to the different methods of representing paragraphs, as described in section 4.3.1. However, these have not been taken into consideration in this process as the following assumptions have been made:

- Each application will output tags in the same order every time, e.g. Word always outputs bold before italic in RTF.

- Each application will always use the same end tag to end formats, e.g. Word always ends tags with a "}" in RTF documents, even though there are other ways of doing this.
- Applications will be consistent in the use of end tags, e.g. FrontPage always inserts the end-paragraph tags into a HTML document, even though they are optional.

Therefore, in cases where these assumptions fail, the comparison will report errors for these differences, as discussed in the results chapter.

4.7 Summary

In a professional environment, source documents come in a wide variety of formats [OVUM95]. In chapter 2, we categorised and described the markup of these formats, highlighting the vast difference between them. To allow for the interoperability of these schemes, it is necessary to create a way of bridging the gap between generalised and specific markup. This chapter discussed our design for a generic process to compare these markup schemes in documentation files.

We have listed four different situations that the comparison may encounter: two documents of identical file types, two documents with specific markup, but different schemes (e.g. MIF and RTF), two documents with generalised markup and two documents of different markup types. The process for handling each of these cases was outlined: the generic tag set for specific markup and the generic document structure for generalised markup. We discussed the issues involved in converting specific markup to generalised markup, and the process which we would use to implement it. The conversion of each markup scheme and internal format was described, giving an overview of the system. The algorithm for the comparison was briefly discussed. This provides the basis for the implementation of the comparison process, discussed in the next chapter.

5. A Prototype Implementation

5.1 Introduction

The design behind each component in our generic process for comparing the markup of two text-based documents was outlined in Chapter 4. This chapter discusses how this design was implemented. Before the main comparison can be applied to the documentation files, they must first be altered to the format it expects. This preparation is described in section 5.2. The main implementation is discussed in section 5.3. This section first provides an overview of the data structures used to store the tags during the process, and the method used to store the mapping of the markup languages to the generic tag set. The operation of the overall system is summarised, before introducing a description of each procedure and the algorithms it uses. The conversion of the markup languages to the generic tag set is discussed in section 5.3.4; the comparison process is outlined in section 5.3.5; and finally section 5.3.6 describes the analysis of the results of the comparison.

5.2 Data Preparation

To perform the necessary conversion and comparison, the main program needs to be able to recognise each tag in a document, any parameter associated with it (e.g. in the tag <PgfLIndent 1.0">, the "<" and ">" are the delimiters and the 1.0" is the parameter), and the text of the document. The document files can be prepared by a pre-processing tool to output each in a format that can be recognised by the main program. The format with which we have chosen to work has the following features:

- Each tag is at the start of a new line, with its delimiters removed.
- Any parameter for the tag is placed after the tag, separated by a tab character.
- Any comments are ignored.
- If the markup does not identify text with a tag, the text of the document is output with the word "TEXT" as the tag, with a tab inserted between it and the text itself.

5.2.1 Generic Parser

The preparation tool must parse each document file to identify each of the elements described above, and output them in the required format. Research by Hearne

[HEAR97] has suggested that a fully generic parser is an unrealistic aim due to the vast differences in markup schemes. For example, text in HTML documents are delimited by a start tag and end tag, with each tag surrounded by the TAGOPEN "<" and TAGCLOSE ">" symbols. Only the tag and any related parameters are allowed between these. For example:

```
<p> This is a paragraph.</p>
```

Tags in MIF are also surrounded by the same TAGOPEN and TAGCLOSE delimiters, "<" and ">". However, MIF allows certain tags to be "nested" inside other tags. In the following example, the String tag is nested within a ParaLine tag, which is itself nested within a Para tag.

```
<Para
<ParaLine
  <String `This is a paragraph.'>
 > # end of ParaLine
> # end of Para
```

In addition to this, markup schemes use different conventions to identify document text. The above example illustrates how the text of a MIF document is included as a parameter within the String tag, rather than outside all tags as in HTML, thereby complicating the task of a generic parser.

The purpose of the required parser is to identify tags, their parameters and the document text. However, despite the difficulties mentioned above, we propose that a generic parser could be used to perform this task on differeht markup schemes if it could recognise all possible components in a markup scheme (e.g. tags, parameters, etc.). To compile a list of these, the components in existing text-based markup schemes and the delimiters used to distinguish them were first examined to identify what the parser should expect. Obviously, each markup has a tag, but the delimiters differed for each scheme. For example, MIF and HTML surround tags with "<" and ">" (as in the above examples), whereas RTF and TeX precede the tag with a "\".

All the schemes examined used parameters, but again the method in which they were stored also differed. For example, MIF includes the parameter within the tag delimiters, separated by a space, but RTF appends the parameter onto the end of the tag with no separator. As well as parameters, LaTeX uses arguments. However, in this parser these stored in the param field, as few tags actually use parameters. In the case of a tag having both a parameter and an argument, both will be stored in the parameter field in the order in which they occur in the document. Nevertheless, the arguments must be identified in the document, so delimiters are required. Other components that exist and therefore need inclusion are groups (RTF groups tags together with the text) and comments.

Although none of the schemes examined used separate delimiters to identify a style, the text-based files from Lotus AmiPro surround a style name in the text with @'s, as in the following example:

```
@Title@ The Document Title
```

However, the markup from AmiPro has not been considered in this research for two reasons. Firstly, it has since been replaced by Lotus WordPro with a different output file format, and secondly, the specification for the markup is not published so to determine the delimiters used in each case would require reverse engineering[12], which is outside the scope of this research. However, we must allow for delimiters identifying styles, as this demonstrates that they exist in certain formats.

The most obvious component of a document is the text. However, this is identified as being outside all tags in most schemes, except in MIF where text is recorded as a parameter within a specific tag. In both of these cases, no delimiters exist to identify text.

From this analysis, the following list of element delimiters was recognised:

TAGOPEN	tag-open delimiter
TAGCLOSE	tag-close delimiter
GROUPOPEN	group-open delimiter
GROUPCLOSE	group-close delimiter
PARAMSTART	parameter start

[12] Reverse engineering is "the process of analysing an existing system to identify its components and their interrelationships and create representations of the system in another form or at a higher level of abstraction" [HOWE97].

PARAMCLOSE	parameter end
COMMENTSTART	comment start
COMMENTEND	comment end
STYLESTART	style start
STYLEEND	style end

We believe that by identifying the symbols representing these delimiters for each markup scheme, this process could recognise all components and text of a document. However, some schemes, such as TeX and LaTeX, do not use delimiters to identify paragraphs, but rather use a blank new line to denote a new paragraph. Although this will not cause problems when comparing two TeX documents, it complicates the comparison of a TeX document to a document in a markup scheme that uses tags to identify paragraphs, such as RTF. This can be overcome by keeping count of the number of new-line characters that occur in a row. If there is more than one, the generic tag PARAGRAPH is written to the file to signify a new paragraph, and the next characters are read in until another new line character is encountered. The generic tag PARAEND can be written to the output file after the text has been written to it. Therefore PARASTART and PARAEND must be added to the above list to identify schemes in which this is the only method of recognising a paragraph.

Given the above, we propose that the implementation of such a process would create a generic parser for the purpose of recognising tags, text and delimiters. The main difficulty in this process is identifying the occurrence of tag delimiters in the text, as described in the Results chapter. Nevertheless, given the already extensive coverage of the parser, it was deemed worthwhile to implement it, with a view to solving this problem at a later stage.

5.2.2 The Data Preparation Tool

The data preparation tool based on the process described in section 5.2.1 reads in a file of any markup scheme and defines the delimiters listed above depending on the scheme.

It keeps track of what type of element is being read in: either a tag, a parameter or text. Each character is read and examined to determine its purpose. If a new line character is not used as a delimiter, it is ignored. If it is a TAGOPEN, we are about to read in a tag. Any tag and its parameter currently open are written to the output file (e.g. MIF embeds tags so the previous tag must be recorded before reading the new one). If the text of the document outside any tags was being read in, this must be written to the output file. A GROUPOPEN symbol is treated in the same way as a TAGOPEN. When TAGCLOSE is encountered the tag is finished and is written to the file. Any GROUPCLOSE symbols are treated in the same way. If the current character is a PARAMSTART, and if a tag is currently being read, this tag is ended and we prepare to accept a parameter. If a PARAMEND is encountered, the parameter is ended but is not considered finished yet, as there may be an argument to be read into the param field. However, most tags do not have a PARAMEND delimiter and the TAGCLOSE will end both the parameter and the tag. If a STYLESTART is encountered, the tag or text being read is finished and is written to the file. The style is read into the tag parameter until a STYLEEND is reached, and then written to the file with no parameter. On reading a COMMENTSTART symbol, all characters between it and the COMMENTEND are read in but ignored.

For example, Figure 5.1 shows an extract from a MIF file.

```
<Para
 <Font
  <FWeight 'Bold'>
 > # end of Font
 <ParaLine
  <String ` Pre-processing Input Files'>
 > # end of ParaLine
 <Font
  <FWeight 'Regular'>
 > # end of Font
 <ParaLine
  <String `To convert the markup in a document to our tag set, '>
 > # end of ParaLine
 <ParaLine
  <String `the document tags must be read in one at a time and '>
 > # end of ParaLine
 <ParaLine
  <String `compared to our mapping of each markup type to the internal tags. '>
 > # end of ParaLine
> # end of Para
```

Figure 5.1 Extract from MIF file before pre-processing

The symbols for MIF are defined in the process as follows:

TAGOPEN	<
TAGCLOSE	>
GROUPOPEN	*none*
GROUPCLOSE	*none*
PARAMSTART	*space*
PARAMEND	*none*
COMMENTSTART	#
COMMENTEND	*new line*
ARGSTART	*none*
ARGEND	*none*
STYLESTART	*none*
STYLEEND	*none*
PARASTART	*none*
PARAEND	*none*

The above process would read the file one character at a time, and process each character, depending on what it is. Using the MIF document extract in Figure 5.1 as an example, a description of the actions taken by the process is as follows:

- The first character processed is the TAGOPEN delimiter "<". This is considered the start of a new tag. The current tag string is empty (because this is the start of the file) so there is no previous tag to be written to the file.
- The character read in is "P", and is stored in the tag string.
- The next characters "a", "r" and "a" are read in and processed in the same way because they are not recognised as delimiters.
- The *new line* character is read in and because it is not a delimiter, it is ignored and the next character is retrieved.
- The *space* is assumed (wrongly) to be a parameter separator, as we are reading a tag. The tag string is ended, and it is assumed the next character is part of the parameter.
- The "<" indicates the start of a new tag. The current tag is written to the file, and the tag and parameter strings are emptied.
- The next characters ("F", "o", "n" and "t") are read into the tag string until a delimiter is encountered.
- The *new line* is ignored and the next character read.
- The *space* indicates a parameter separator so the tag string is ended and the parameter is expected next.

- The “<” causes the current tag to be written to the file, the tag and parameter strings are emptied.
- “F”, “W”, “e”, “i”, “g”, “h” and “t” are stored in the tag string.
- The *space* causes the tag string to be ended, and a parameter is expected next.
- “‘”, “B”, “o”, “l”, “d” and “’” are stored in the param field.
- The > indicates the end of the tag, so the tag and parameter are written to the output file. The tag and parameter strings are emptied. The process expects the next character to be document text unless it is a recognised delimiter.
- etc.

Figure 5.2 shows the output of this process. As can be seen, all tags are considered to be of the same level, even though the tags were originally nested. The nesting of the tags is of no relevance to the comparison process, as each tag is considered individually and it is outside the scope of this research to determine the syntactic validity of the document.

```
Para
Font
FWeight ‘Bold’
ParaLine
String ` Pre-processing Input Files'
Font
FWeight ‘Regular’
ParaLine
String ‘To convert the markup in a document to our tag set, '
ParaLine
String `the document tags must be read in one at a time and '
ParaLine
String `compared to our mapping of each markup type to the internal tags. '
```

Figure 5.2 Extract from MIF file after pre-processing

The output file is a representation of the original file in a form that can be recognised by the main program. The output from all markup schemes will be in the same format so each can be processed identically.

5.2.2.1 Data Preparation Tool for Generalised Markup

Both SGML and LaTeX documents can be prepared using the same tool as for specific markup schemes to identify tags, parameters and document text, as each element (TAGOPEN, TAGCLOSE, etc.) can be specified. However, problems can arise in mapping generalised markup to the internal generic tag set. While LaTeX has a predefined tag set which allows a mapping of each tag to be recorded, SGML is a language which allows the user to define their own tag sets, so tags cannot be known in advance. The mapping must therefore be made either by the user or a separate tool specific to SGML. Although we have not implemented such a tool, the issues for an implementation were discussed in section 4.3.2.2. The system currently expects the user to map each element in the SGML DTD to a tag in the generic tag set, and save this in a file called *DTDname*.map where *DTDname* is the name of the DTD to which the mapping belongs.

5.3 Main Implementation

5.3.1 Storing the Mappings between Tags

The mappings between the tags in the markup languages and the generic tags are stored in a simple text file (called tagmap.ini) that is read by the program if the documents need conversion. Each file type has its own "section", which is headed by the extension of documents of that file type surrounded by square brackets, all on a separate line. The section itself contains each tag recognised by our process and the generic tag for it, separated by a tab character. Each such mapping appears on a new line. The section is ended with the heading for the next file type. Here is an example tagmap.ini file. (The entire file as used by this system is included in Appendix A):

```
[MIF]
FWeight 'Bold'                  BOLD
FWeight 'Regular'               BOLDOFF
FUnderlining FSingle            UNDERLINE
FUnderlining FNoUnderlining     UNDERLINEOFF
Para                            PARAGRAPH
[RTF]
b                               BOLD
b0                              BOLDOFF
ul                              UNDERLINE
ul0                             UNDERLINEOFF
ulnone                          UNDERLINEOFF
par                             PARAGRAPH
```

Figure 5.3 Example Content of Tag Mapping File

As can be seen from this example, more than one tag from the markup language can be mapped to the same generic tag. RTF has two tags to turn off underlining, so both are equivalent to UNDERLINEOFF in our generic tag set. This will not cause a problem because when a tag is encountered in the input file, we search for it in the tag mappings and replace it with our generic tag, irrespective of what it is. However, only one instance of each tag from the markup languages is allowed as more than one would cause ambiguity when searching for the generic replacement.

5.3.2 Internal Representation of the Documents as Lists

To manipulate the tags internally in the program, they are stored as linked lists[13]. For this implementation, there are two types of list defined, one to store the tags read in from the input file (file_tag), and one to store the mapping between tags (tag_map).

5.3.2.1 File_tag: List of Tags from Input File

File_tag stores all information about each tag from an input file in a single node. The following fields are stored in the nodes of the file_tag list:

- **tag** is a character string that stores the tag read in.
- **param** is a character string to store any parameters for the tag (described below).
- **id** is a unique identification number assigned to the tag.
- **match** is a number that stores the id number of the matching tag. This is initially 0 for all nodes because no tags have been identified as matching.
- **error_status** is a number indicating the type of error associated with this tag, or 0 if none.

The tag itself is separated into the keyword of the tag and the parameter, if one exists. For example, in the tag FWeight 'Bold' from the file in Figure 5.2, FWeight is the keyword and the 'Bold' is the parameter. There are two reasons for storing the parameter separately from the tag. Firstly, in a direct comparison of tags, if only the parameter differs, this should not be considered an error. For instance, in MIF the left indent of a paragraph is specified using the tag PgfLIndent *n* where n is the size of the

[13] A linked list is "a data structure in which each element contains a pointer to the next element, thus forming a linear list." [HOWE97].

indent. If one file had the tag PgfLIndent 1.0" and the corresponding tag in the second file was PgfLIndent 1.5", the comparison including parameters would consider these different tags, rather than the same tags with differing parameters. Storing the parameter separately overcomes this problem.

The second reason for isolating the parameter is for conversion. This conversion process searches a list of tags in the appropriate markup scheme until it finds the tag required. For example, if we wish to convert the tag PgfLIndent 1.0" to its generic equivalent, the list of MIF tags is searched for this tag. Only the tag PgfLIndent is stored, so the tag will not be found. By separating the tag and parameter we can search solely for tags, and replace only the tag with the generic tag, retaining the parameter.

Other information stored for each tag includes an identification number, the id of its matching tag (or 0 if no match is identified), and an error number, denoting the reason for any error with the tag, or 0 if no error exists. The error reporting is described in detail in section 5.3.6.

Each tag from the input file has its own node with the fields describing it, and each input file has its own linked list of these nodes. For example, Figure 5.4 represents the first four tags in the extract in Figure 5.2.

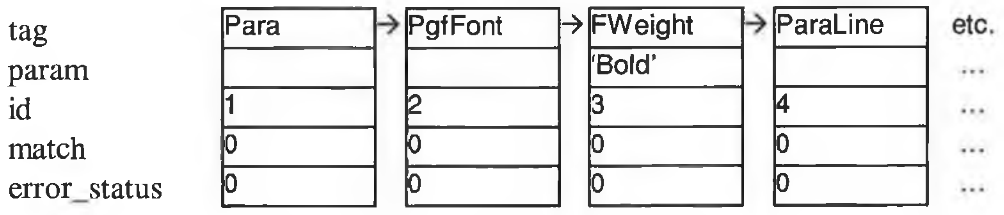


Figure 5.4 List of Tags from Input File

5.3.2.2 Tag_map: List of Mappings from Markup Tags to Generic Tags

Tag_map records the mapping between the tags in the markup languages and the generic tags. It is a simpler structure than file_tag, with the following fields:

- **spectag** is a character string that stores a tag specific to the markup language.
- **gentag** is a character string that stores the generic identifier for the specific tag.

A list of these nodes for each markup scheme will be used in the program, but only if conversion is required. Each list is filled with the tags for the file type of the input file from tagmap.ini, the file recording the mappings between specific and generic tags. Each node has one tag from the markup language, plus its equivalent generic tag. For example, if the linked list contained MIF tags, it could be represented as follows:

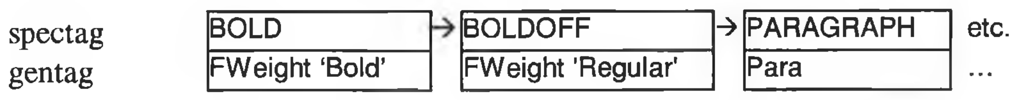


Figure 5.5 List of Tags from Tag Mapping File

5.3.3 System Overview

After pre-processing the documentation files, the main program performs any necessary conversion and the comparison. To manipulate the tags in the system, they are read into linked lists from the output files of the preparation tool. Because both files are in the format described in section 5.3.1, we know that the tag is composed of all characters from the start of the line to the tab character, and that everything after the tab until the end of the line is the parameter. An identifier is then assigned to the tag. The list of tags from the first input file will be referred to as list 1 and the tags from the second input files as list 2.

In order to determine whether conversion is necessary, the file extension of both input files is examined. If both files are of the same type, no conversion is necessary. Otherwise the tag_map lists are filled with the relevant mappings and the conversion process is executed for each list. If both files are SGML files, the user is asked to specify the *DTDname*.map for the conversion, otherwise tagmap.ini (the file of pre-defined mappings) is used. After any conversion, the comparison function is called. This compares the two lists tag by tag and finds any differences. The reason for each difference is identified as one of the following: a tag could be changed in the second list, tags may be missing from list 2, there could be extra tags in list 2, or the parameter of the tag in list 2 may have changed. It can also perform a basic check for untranslated text if the user chooses this option. The results of the comparison are analysed to report the errors to a file.

5.3.4 Conversion

Before the conversion of a file can be performed, the tag_map list must be filled with the mapping from the markup scheme of that file to the generic tag list.

5.3.4.1 Filling the Tag Mapping List

The FillTagList function scans the tag mapping file to find a section for the relevant file type. This is done by searching for an opening square bracket, "[". When one is found, the file extension between it and the closing bracket, "]", is compared to the file type passed in. If they are not the same, then this is not the correct section, so the heading of the next section is sought in the same way. If the correct section is not found, the program reports to the user that there are no tag mappings for files of that markup scheme. Otherwise, when the correct section is found, a list (henceforth referred to as the generic list) is filled from the tag mapping file.

Because each tag is on a new line, the characters from the start of the line are saved in the spectag string in the node, until a tab is reached. The tab character separates the generic tag from the specific tag. All characters after the tab are saved in the gentag string, until the end of the line is reached. If the first character of a line is a "[", this is the heading of a new section indicating that all tags for the current section have been retrieved. The resulting list is a list similar in structure to Figure 5.5.

5.3.4.2 Conversion of Specific Markup to the Internal Tag Set

The conversion function accepts the list to be converted and the tag_map list containing the appropriate tag mapping. The list of tags from the input list are read in one at a time and compared to the specific tags in the generic list, one by one, until a match is found. If the tag is found, it is replaced by the generic equivalent. If it is not found, it is replaced with NOMATCH. The NOMATCH tags are ignored in the comparison. Because we do not recognise them, we cannot attempt to match them to any tag in another scheme. This means that only the elements and formats recognised by the system are compared. The implications of this are discussed in the Results Chapter.

For example, the first tag in list 1 for the extract in Figure 5.2 would be Para. If the generic list is the tag_map list in Figure 5.5, Para is compared to each specific tag in the generic list (FWeight 'Bold', FWeight 'Regular') until it finds a match. The current tag is then replaced by the generic tag, in this case PARAGRAPH, the next tag in list 1 is examined and the process repeated, starting with the first tag in the generic list.

Finding a matching tag in the specific tags of the generic list is complicated by parameters in two ways. Firstly, part of a tag may wrongly be considered a parameter. All tags in this system are separated into the keyword and the parameter. However, sometimes the parameter is an essential part of the tag, not just extra information. For example, FWeight 'Regular' is the tag in MIF for BOLDOFF. FWeight is stored as the tag and 'Regular' is saved in the param field. Therefore when we try to find a match for this tag in the specific tags, it will not be found as no entry exists for the tag FWeight. By concatenating the parameter onto the tag and comparing this, we can locate the matching tag if one exists. If a match is found in this way, then we know that the parameter field is part of the tag, so it is deleted, and the tag is replaced with the generic equivalent. For example, FWeight (the tag to be converted) is not the same as FWeight 'Bold' (the first tag in the generic list), so we compare the concatenation of the tag and parameter, FWeight 'Regular', to the tag. This still does not match, so the next tag, FWeight 'Regular', from the list is read in. FWeight does not match this either, so we compare the concatenation of the tag and parameter, FWeight 'Regular', to it. This does match, so the tag FWeight is replaced with BOLDOFF, and the parameter 'Regular' is deleted, as it was part of the tag.

The second issue with the conversion of tags with parameters is that the parameter may not have been identified at all. Markup schemes such as RTF do not use any character to separate the parameter from the tag. For example, to specify the left indent of a paragraph, RTF uses \li*N*, where the *N* is the indent size. If the indent is 2, the tag is /li2. A comparison of this to the specific tags of the generic list will fail, as the tag in the list obviously does not have a parameter. To overcome this, the paragraph is represented in the generic tag set as a "?"., e.g. \li?. When the comparison fails, the process compares the tags character by character until a difference is encountered, e.g. l = l, i = i, 2 ≠ ?. If the character in the specific tag in the generic list (i.e. \li?) at this point is a ?, as here, the rest of the other tag (in this case 2) is assumed to be the parameter, and is stored in the param field. The tag is then replaced with the generic equivalent, LEFTINDENT. If the differing character is not a ? the tags do not correspond, so the next tag in the generic list is read in and the process continues.

5.3.5 Comparison

The files on which the comparison is to be performed are the output files from the generic parser, possibly after conversion. However, before the comparison is performed, these files require modification for two reasons. Firstly the text must be removed, as translated text obviously differs from the English and will cause numerous errors in the comparison. Secondly, if the files have been converted, they will contain numerous NOMATCH tags which must be ignored, as we do not wish to try and find a match for them.

5.3.5.1 UNIX *diff* Utility

The UNIX *diff* utility was used to perform the comparison. It compares the contents of two files and outputs a list of changes necessary to convert the first file into the second. No output will be produced if the files are identical. The output from *diff* consists of one or more "hunks" of differences, with each hunk indicating one place where the files differ [MACK93]. Each hunk is output in the following format:

```
change-command
< from-file-line
< from-file-line...
---
> to-file-line
> to-file-line...
```

There are three types of change commands. Each consists of a line number or range of lines in the first file, a single character indicating the kind of change to make and a line number or range of lines in the second file. All line numbers are the original line numbers in each file. The change commands are displayed in one of the following forms (which are explained below):

```
n1 a n3,n4
n1,n2 d n3
n1,n2 c n3,n4
```

where n1 and n2 represent lines in file 1 and n3 and n4 represent lines in file 2. The "a" stands for "add", "d" represents "delete" and "c" means "change".

- **n1 a n3,n4**

This indicates that the lines from n3 to n4 (inclusive) in the second file must be added after line n1 of the first file to make both files the same. For example, "8a12,14" means append lines 12, 13 and 14 of file 2 after line 8 of file 1. When applied to our files of tags, this indicates that the tags on lines n3 to n4 are extra in file 2, with no match in file 1.

- **n1,n2 d n3**

To make the second file resemble the first, the lines from n1 to n2 in the first file must be deleted; line n3 is where they would have appeared in the second file had they not been deleted. For example, "5,7d3" means delete lines 5 to 7 of file 1. When applied to our files, this indicates that the tags on lines n1 to n2 are missing from file 2.

- **n1,n2 c n3,n4**

This means that the lines from n1 to n2 in the first file should be replaced with the range n3 to n4 of the second file. This is a more compact method of a combined add and delete. For example, "5,7c8,9" means change lines 5, 6 and 7 of file 1 to read as lines 8 and 9 of file 2.

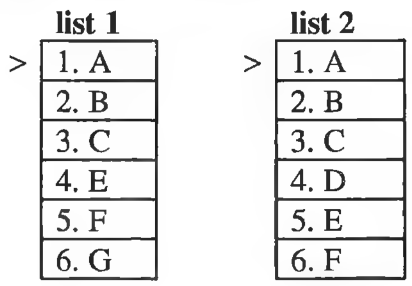
The lines that are affected by the difference follow each of these change commands. The range of lines from the first file are displayed, preceded by "<", followed by the range from the second file, which are preceded by ">". However, this information is not required by our system as the tags associated by the errors can be determined from the line numbers.

5.3.5.2 Analysis of *diff* Output

After calling the *diff* command, the output must be analysed by our system to be applied to the list representations of the files. By checking each "hunk" at a time, the reasons for the differences and where they occurred in our lists can be determined.

Locating the erroneous tags in documents is simplified by the fact that the id number of each tag is the same as the line number in the file. Even if the documents were converted and all NOMATCH tags ignored, the NOMATCH tags are deleted from the lists, so the line numbers still correspond to the id numbers. The analysis function uses the following process to set the appropriate error_status to the erroneous tags in both lists.

The change command for the first difference identified by *diff* is examined to extract the line numbers and reason for the change (i.e. the character representing the change required). It sets node 1 to the first node in list 1 and node 2 to the first node of list 2. The function then steps through both lists from the beginning, setting the tags to match each other by letting match in node 1 equal the id of node 2 and match in node 2 equal the id of node 1 (see Figure 5.6).



In this example, the current position in either list is indicated by ">", the number in each list is the id of the tag, the letter is the tag, and any following characters are the parameter for the tag. The result of the diff command for these files would indicate that line 4 of list 1 is missing. Therefore the function would let the tag A in list 1 equal A in list 2, B = B and C = C, until the line with the error (i.e. tag 4) is reached in list 1.

Figure 5.6 Setting Tags to Match

5.3.5.3 Checking for Changed Text

As the function steps through the lists, all text must be checked for translation. However, because understanding the text of the document is beyond the scope of this research, this process cannot verify translation. Instead, we assume that the text of the localised document should differ from the original, and simply check for changed text.

To do this we employed the concept of "pseudo-translation" – altering the English text to simulate translation.

To determine if the parameter contains text, its tag is examined. However, some markup schemes use a particular tag for text, whereas others have none and will use the generic tag TEXT. Therefore, the tagmap.ini file must first be searched to find the tag that identifies text in the markup scheme of either file. Then, when the analysis function processes the tags, it can identify text tags regardless of the markup scheme. A string comparison is used to determine if the text has changed. If both strings are the same, this means that the text was not altered. In this case, the error_status of both tags is set to 6 (see Figure 5.20 in section 5.3.6.1 for a full table of errors).

5.3.5.4 Locating Errors

This process continues until the tag with the first error is reached. This is detected by checking the id of each node against the number of the first line in the erroneous range. When a tag with an error is reached, the character representing the change required in the change command is examined to determine the reason for the difference.

5.3.5.5 Checking for Extra Tags

If the character is "a", this indicates that the tags with ids matching the range between n3 and n4 are extra tags in file 2 with no match in file 1. The error_status for each of these tags is set to 3, indicating they are "extra" (see Figure 5.7). To continue processing, the tag after the extra tags in list 2 corresponds to the next tag in list 1.

list 1	list 2
1. A	1. A
2. B	2. B
3. C	3. C
4. D	**4. X**
5. E	**5. B**
6. F	**6. D**
7. C	**7. A**
8. A	**8. G**
9. B	9. D
etc.	10. E
	11. F
	etc.

The tags in list 2 marked in bold are extra tags that have no match in list 1. The result of the diff command for these files would be **3a4,8**, *in other words lines 4 to 8 are extra in list 2. Therefore the function would match the first three tags in the lists, and then set the error-status of tags 4-8 (inclusive) in list 2 to indicate that they are extra. It would then continue the process with the tag after the errors in list 2, i.e. tag 9, and the next tag in list 1, i.e. tag 4.*

Figure 5.7 Identifying Extra Tags in File 2

5.3.5.6 Checking for Missing Tags

If the character representing the required change in the change command is "d", this indicates that the lines indicated by n1,n2 are missing from file 2. In other words, there are tags missing from list 1 in our representation. These tags can be identified by matching the line numbers in the change command with the id numbers of the tags, and the error_status of each of these erroneous tags is set to 3, indicating they are "missing" (see Figure 5.8) To continue processing, the tag after the extra tags in list 1 corresponds to the next tag in list 2.

	list 1		list 2
	1. A		1. A
	2. B		2. B
	3. C		3. C
>	**4. X**	>	4. D
	5. B		5. E
	6. D		6. F
	7. A		7. C
	8. G		8. A
	9. D		9. B
	10. E		etc.
	11. F		
	etc.		

The tags in list 1 marked in bold are missing from list 2. The result of the diff command for these files would be **4,8d3**. *In other words lines 4 to 8 in list 1 are missing from list 2 after line 3. Therefore the function would match the first three tags in the lists, and then set the error-status of tags 4-8 (inclusive) in list 1 to indicate that they are missing from list 2. It would then continue to process the next tags, i.e. the tag after the errors in list 2 (tag 9) and the next tag in list 2(tag 4).*

Figure 5.8 Identifying Missing Tags in List 1

5.3.5.7 Checking for Changed Parameters

If the character representing the required change is "c", this indicates the tags corresponding to n1,n2 in file 1 have changed to those represented by n3,n4 in file 2. The reason for the "change" must be determined to report an accurate error message. To check if the parameters have changed, the tags corresponding to n1 and n3 must first be compared. If these are the same, then the parameters are checked and if they are not the same, it is assumed that the error is due to a changed parameter (assuming the tags are not TEXT tags, in which case they must be checked for translation). Therefore the error_status of both tags is set to 5 to indicate that the tags match, but the parameters differ (see Figure 5.9). If the tags are not the same, the reason for the difference is determined in the following sections.

	list 1		list 2
	1. A a		1. A a
>	2. B **b**	>	2. B **z**
	3. C		3. C
	4. D		4. D

In this example, the parameter for tag 2 in list 2 was changed, as highlighted in bold text. The result from diff would be **2c2**, *indicating that line 2 in file 1 has changed to tag 2 in file 2. To determine if the reason for this is because of the parameter change, the tags on line 2 in list 1 and line 2 in list 2 are compared. They are the same (B=B) so they are set to match each other. The parameters of these tags are then compared. They are not the same (b ≠ x) so the error_status for both tags is set to 5. The process continues with the next two nodes, i.e. C and C.*

Figure 5.9 Identifying Tags With Differing Parameters

5.3.5.8 Checking for Changed Tags

If the character representing the required change is "c" and the change is not due to the parameters, it is possible that the tag was changed in document 2. Whereas this is the most likely explanation if there is only one tag in either range (e.g., a bold tag was changed to italic), it is not always accurate to assume this, as it may possibly be a case of a deletion in one file and an coincidental addition in the same position in the other. For example, consider the following extracts from two HTML documents:

Document 1	Document 2
P	P
TEXT	TEXT
IMG SRC...	IMG SRC...
P	**B**
CENTER	TEXT
TEXT	A HREF...
A HREF...	TEXT
TEXT	/A
/A	**/B**
/CENTER	/CENTER
etc.	etc.

Figure 5.10 Example of an Incorrect Judgment by diff

The text was removed for comparison (as translated text will obviously differ and therefore must be removed). The tags highlighted in bold in document 1 are missing from document 2, and those in document 2 are extra, with no match in document 1. However, *diff* does not recognise this, and instead assumes that these changes are related and therefore reports that P and CENTER have been changed to B, which is incorrect. The /B tag is correctly assumed to be missing from document 2.

Therefore, in cases like this, the tags following the two tags are compared by the CompStraight function (described in section 5.3.5.10) to determine if there is a change, or a misjudgement by *diff*. If most of the subsequent tags are the same, then it is assumed that the tag in the second list was changed. Ideally all tags should be the same (see Figure 5.11).

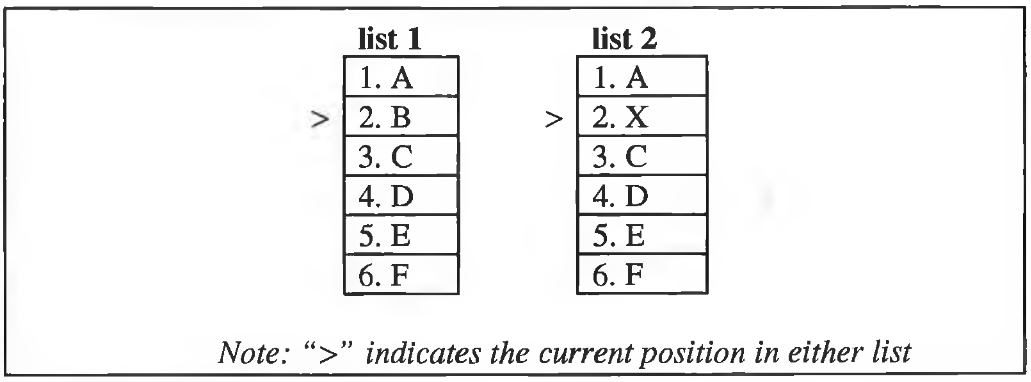


Figure 5.11 Example of all Following Tags Matching

However, we must allow for other changes or omissions in the tags, so if the vast majority of tags are the same, we assume that one of the tags must have changed (see Figure 5.12). The error_status of the node 1 and node 2, in which the tags are contained, are set to 1, indicating this. Although the tags are not the same, they are in matching positions in the documents, so they are set to match each other.

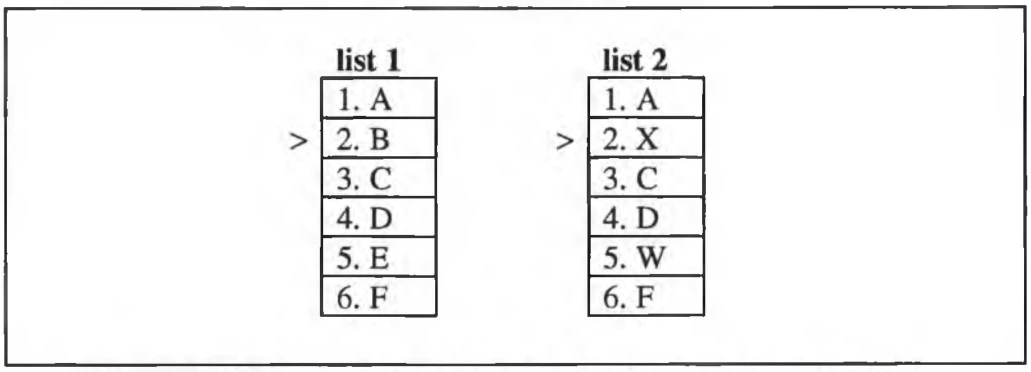
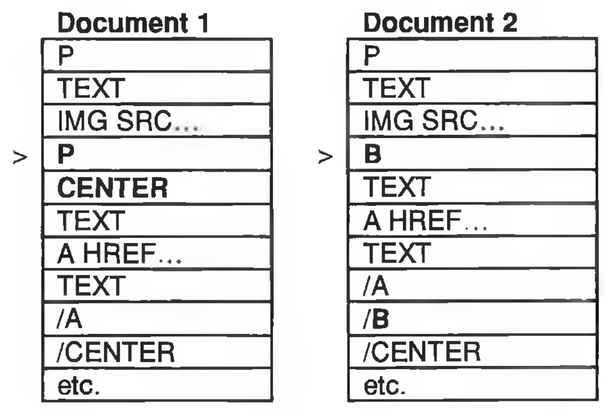


Figure 5.12 Example of Most of the Following Tags Matching

However, if most of the subsequent tags differ, it is assumed that the tags indicated by n1,n2 in list 1 are missing from list 2 and the tags represented by n3,n4 are extra tags in list 2.

Using CompStraight, the incorrect "change" identified by *diff* in Figure 5.10 above is solved and reported as separate sets of missing and extra tags, as demonstrated in the figure below:



In this example, the missing tags are in bold. CompStraight would compare the tags after P *and* B *i.e.* CENTER *and* TEXT, TEXT *and* A HREF, A HREF *and* TEXT *etc. Because most of the tags differ, the tags are considered not have changed.* P *and* CENTER *are deemed to be missing from list 1, and* B *is deemed to be extra in list 2.*

Figure 5.13 Correcting the Inaccurate Judgement from diff using CompNext

If the tags are not deemed to have changed, the reason for the difference must be determined. For example, the tag in file 2 may match a tag in list 1 that was marked as missing, or vice versa. To determine this, the ReAlign function is called to attempt to find a match for either tag in the following tags.

	Document 1		Document 2
	P		P
	TEXT		TEXT
	/P		/P
	P		P
>	**A HREF x**	>	A HREF y
	TEXT		TEXT
	/A		/A
	/P		/P
	P		P
	A HREF y		A HREF z
	TEXT		TEXT
	/A		/A
	/P		/P
	etc.		etc.

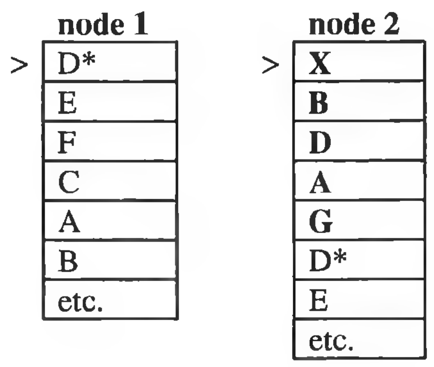
In this example, the missing tags are in bold. Diff identified a change in the tags marked with a ">". However, A HREF y matches the same tag further down in list 1. ReAlign would identify this, and then the A HREF x is marked as missing, instead of changed.

Figure 5.14 Using ReAlign to Identify a Missing Tag Incorrectly Considered as Changed by diff

If a match is found for one, then the other tag is deemed to be missing. If this is not the case, then the tag in file 1 is missing from file 2 and the tag in file 2 is missing from file one (as was the case in Figure 5.13).

5.3.5.9 ReAlign

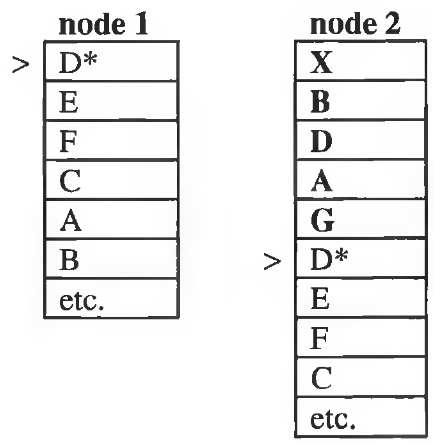
The ReAlign function accepts two nodes, (node1 and node2) of the type file_tag. Its purpose is to find a match for the tag in node1 in the nodes following node2, as illustrated in Figure 5.15.



ReAlign compares the tag in node1 (D) to the tag in node2 (X). Because these differ, node2 is let equal the node following node2, (i.e. node 2 now is the node containing the tag B) and node1 and node2 are compared. This continues until a match for the tag in node1, D, is found.

Figure 5.15 Searching for a Match in ReAlign

If a match is found, we must ensure that it is not a coincidence. The next node after both node 1 and node 2 are passed to the function CompNext (described in section 5.3.5.10), which compares the next *n* tags of both nodes. If a significant majority are the same, it is assumed we have found the match. Otherwise we find the next tag matching the tag in node 1 and repeat this process.



When the next match is found (indicated above by ">"), we can tell it is the correct match by examining the subsequent nodes of both.

Figure 5.16 Finding a Correct Match

5.3.5.10 CompNext

Whether two tags are a correct match can be determined by comparing the subsequent tags. CompNext is given the next node of both nodes that are to be checked. DEPTH_LIMIT is defined in the program to be 15. This means that the 15 tags following both nodes are compared. If DEPTH_LIMIT is too low, we may not consider corresponding tags to match because if too few of the subsequent tags were checked, the erroneous tags will have too much weight and the two tags will not be deemed to match. However, if DEPTH_LIMIT is too high, we may consider tags to match even if they do not. This is because most of the tags in the documents probably correspond, and the erroneous tags will lose their significance if too many surrounding tags are deemed to match.

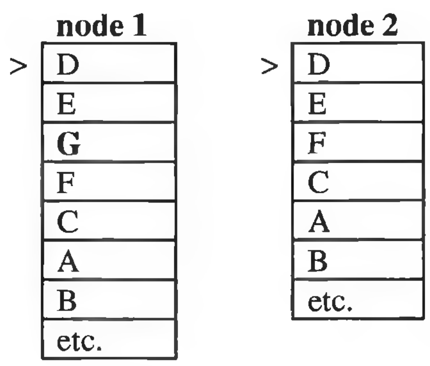
CompNext is comprised of three similar recursive comparison functions: CompStraight, CompNext1 and CompNext2, each of which accepts the two nodes passed into CompNext. These perform different comparisons on the subsequent nodes of those passed in, and set the variables sametags to the number of tags found to match and difftags to the number of tags that differed.

To determine if the tags in node 1 and node 2 are a correct match, sametags and difftags are analysed. By dividing difftags by sametags we get the ratio of the number matching to the number differing, and this is compared to the acceptable cut-off level. To determine the cut-off point, it must first be decided how many differing tags we will accept and still consider the tags a correct match. For example, if we want at least 80% of the tags to be the same, this ratio is 3 differing to 12 matching for 15 tags. The cut-off level is then calculated as follows:

$$\frac{difftags}{sametags} = \frac{3}{12} = 0.25 = \textit{cut-off level}$$

Therefore, if the ratio of difftags to sametags is greater than the cut-off level of 0.25, we will not accept the tags as matching. When two tags are compared and found to be the same, the parameters must then be checked. If these also match, one is added to sametags. If the parameters differ, it is assumed the tags correspond. However, because there is a possibility that they do not, 0.75 is added to sametags.

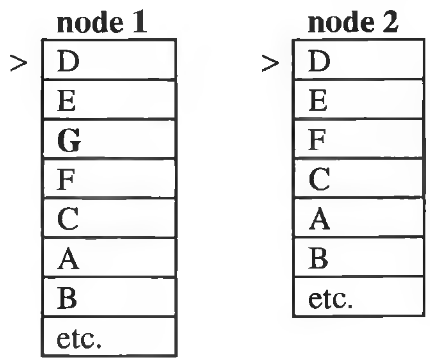
When CompNext is called, it calls CompStraight first. This compares each of the corresponding subsequent tags to each other in sequence, as illustrated in Figure 5.17.



CompStraight would compare E with E, G with F, F with C, C with A, etc. Although the current tags are a correct match, this comparison does not identify this because the tag G (bold in list 1) is missing from list 2.

Figure 5.17 Determining the Validity of a Match using CompStraight

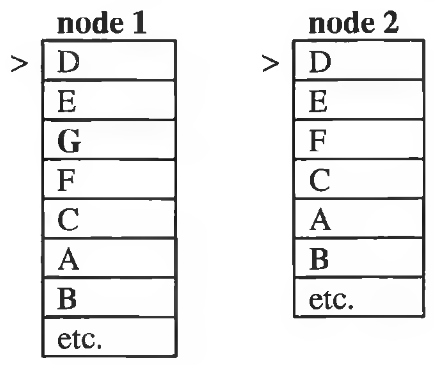
The values of sametags and difftags are compared to determine if CompStraight found the tags to be an accurate match. If it did not, then sametags and difftags are reset to 0, and CompNext1 is called. This will establish a match if tags were added to the list after node 2, or tags were deleted after node 1. The example in Figure 5.18 demonstrates how this function works.



CompNext1 would compare E with E, G with F. When these do not match, it tries the node after F, comparing G with C, G with A, G with B, etc. The ratio of sametags to difftags will not be acceptable, so this function will also fail to identify the match.

Figure 5.18 Determining the Validity of a Match using CompNext1

The values of sametags and difftags are checked, and if this does not prove the tags to have matched, CompNext2 is called to find a match if tags were added in list 1, or missing from list 2 (see Figure 5.19). If this also fails, the two tags are not considered to be equivalent; otherwise the tags are said to be a correct match.



CompNext2 would compare E with E, G with F. When these do not match, it tries the node after G, comparing F with F, C with C, A with A, etc. With only one differing tag, the ratio of sametags to difftags is acceptable, and the tags are deemed to match.

Figure 5.19 Determining the Validity of a Match using CompNext2

5.3.6 Error Analysis

When the comparison is finished, the lists must be examined to print an error log for the user. This function starts at the first node in each list, and checks their error_status and match fields to determine if an error message is required, and moves on to the next nodes. The reason for moving through the lists in parallel is to report error messages that correspond to both files, e.g. changed tags.

The first node in list 1 is read into node 1, and the first node in list 2 is read into node 2. The match field in node 1 is examined. If it is 0, this means that it is unmatched, indicating that it is missing from list 2. The node is sent to the PrintError function to display the position in the document and a relevant error message. The match field for node 2 is then checked, irrespective of the value for match in node 1. If it is 0, it is unmatched in list 1, and therefore is an extra tag in list 2. The node is sent to the PrintError function to display the position in the document and a relevant error message.

If both nodes have the value of 0 for match, this means that both tags have been reported, so we read the next node in list 1 into node 1 and the next node in list 2 into node 2, and repeat the analysis. If node 1 has a nonzero value for match and node 2 has a value of 0, this means that node 1 requires no more processing and the next value in list 1 can be read into it. However, node 2 has a matching tag in list 1 with an id corresponding to the value in match. Because we are moving through list 1, its match will be reached eventually, so we do not change it. This is because matching tags are processed together, thereby keeping the two lists in alignment.

If node 2 has a nonzero value for match and node 1 has a value of 0, then the next value for node 2 is read in. Otherwise, if both node 1 and node 2 have matching tags, it must first be determined if they match each other to process them. If they do, this does not guarantee that there are no errors. For example, even if tags are said to match, they may still have parameters that differ. Therefore the error_status of the current tags of both lists must be checked.

If both have a value of 0, then the tags match and have no errors so there is nothing to report. The next two nodes are read in and the process repeated. If the error_status of node 1 is greater than 0 and the error_status of node 2 is 0, it is sent to the PrintError function, and then read the next node into node 1. We do not process node 2 because it must be processed with its matching tag. If the error_status of node 1 is 0 and the error_status of node 2 is greater than 0, it is sent to the PrintError function, and then read the next node into node 2. If both nodes have a non-zero error_status, then both are sent to the PrintError function together, as errors such as a parameter change or a tag change need both matching tags for an accurate error message.

The only time both nodes would have a nonzero match field and not match each other is when moved tags had been identified. However in our system this is considered to be an error, as we do not check for tags that have moved. Therefore this should never happen, but if it does the system will detect it, print an error and read in the next two tags.

If the documents have been converted, the user is given a warning that there may be other errors in the documents than those printed due to the possibility of tags not being recognised. For example, the NOMATCH tags may have represented different elements in both documents, but the parser could not detect this, as it did not recognise what they were. Careful maintenance of a full set of all possible formats and elements would eliminate this risk because the parser would recognise all elements and therefore ignore only those tags with no equivalent in other schemes, such as ParaLine. However, this requires a user with an in-depth knowledge of the markup schemes to recognise the correspondence between tags, and identify tags with no equivalent.

5.3.6.1 PrintError

This function accepts one or two nodes to print the relevant error message. If only one node has an error, an empty node is passed in as the other node. The error_status of the nodes are checked and the error message printed. All error messages have a similar format:

"ERRORTYPE: the *X* tag between *previous text* and *following text error text*

where ERRORTYPE is one of the error types in Figure 5.20; the tag *X* is obtained from the combination of the tag and parameter fields in the node; the *previous text* is obtained from the function GetPrevText, described in section 5.3.6.2, the *following text* is retrieved by GetNextText (section 5.3.6.2), and *error text* depends on the type of error. The error type is determined from the error_status of the node(s). These are outlined in Figure 5.20, along with the relevant error text.

error_ status	Description	ERRORTYPE	Error Text
0	The tag has no match	MISSING if node 1 EXTRA if node 2	"in file 1 is missing from file 2" "has been added to file 2"
1	The tag was changed in file 2	TAG CHANGE	"in file 1 was changed to *tag from node* 2 between *previous text for node* 2 and *next text*
2	The tag is missing from file 2	MISSING	"in file 1 is missing from file 2"
3	The tag was added to file 2	EXTRA	"has been added to file 2"
4	The tag was moved in the text	MOVED	This has not been implemented
5	The parameter of the tag was changed in file 2	PARAMETER CHANGE	"has had its parameter changed from *parameter from node 1* to *parameter from node 2*"
6	The text was not changed in file 2 (to mimic translation)	UNCHANGED	"The text between *previous text* and *following text* has not been changed"

Figure 5.20 Description of Possible Errors and Their Related Messages

The first node passed into the function, errornode1, is checked to see if it is NULL. If it is not, its error_status is checked. Errornode1 comes from list 1, so it cannot have an error_status of 3 or 6 as these only apply to tags from list 2. Therefore there is no need to check for these. If the status is 1, 2 or 4, the message for these also requires the information in errornode2, so it is checked to ensure it is not NULL, before printing the error to the error file. If the first node is NULL, this means that only the second node, errornode2, has an error. The only error codes applicable only to list 2 are 0, 3 and 6 (as the rest either apply only to list 1or to both lists), so only these are checked for. Depending on the error_status, the relevant message is printed and the function returns to the error analysis.

5.3.6.2 Locating Document Text Surrounding an Error

GetPrevText: This function is passed the node with the error. Its purpose is to find the text before this tag in the document. CurrPos records the position of the current position in the list, i.e. this node. The previous text is found by moving from the current node, back through the relevant list, checking the tag in each node until the tag 'TEXT' is found.

GetNextText: The text following the error tag is found in a similar manner to GetPrevText, except that instead of moving backwards through the list, the function moves forward until the text is found.

5.4 Summary

This chapter outlined how the design of our system was implemented. A description was given of the tools necessary for preparing the files before they can be used in our system. The files for storing the mapping were described, as were the lists for storing the data internally. We discussed the method for converting the documents to the generic tag set. Each function in the comparison was outlined, describing how this process works. Finally, we discussed the algorithm used for identifying and reporting errors in the tags. Our system was tested with a series of test cases to determine the success of its design. The following chapter discusses the results of these experiments.

6. Results

6.1 Introduction

In Chapters 4 and 5, the design and implementation of a generic process for the comparison of two documents were described. To assess the performance of this process, a number of sample documents were used to test the system. Some of these documents have artificially created errors in the markup in order to test each function of the system. The next section discusses the data preparation tool. We then describe the results of the comparison of two documents of the same markup scheme. These documents were tested for changed tags, changed parameters, missing tags, extra tags and untranslated text. We then deal with the comparison of two documents with different markup schemes, with the conversion process necessary to allow the comparison discussed first, followed by the issues encountered in the comparison of these converted documents. Finally, we discuss the accuracy of the errors reported.

6.2 Data Preparation Tool

To test the data preparation tool, documents with several markup schemes were used. The overall performance is summarised in Figure 6.1, and details of each markup scheme are then described in the following sections.

	MIF	**RTF**	**HTML**	**LaTeX**
No. of documents tested	10	10	10	6
% of actual tags identified	100%	100%	100%	100%
% of erroneous tags	1.3%	0.5%	1.1%	1%
% of actual parameters identified	100%	n/a[14]	100%	100%
% of erroneous parameters	3.1%	n/a	8%	0.8%
% of actual text identified correctly	n/a[15]	96.2%	97.5%	73.5%
% of erroneous text	n/a	45.8%	1.2%	26.5%

Figure 6.1 Results of Tests on Data Preparation Tool

where:

- **% of actual tags identified** is the percentage of tags in the document that were correctly identified, calculated as follows:

[14] Parameters in RTF are not separated from the tag, and are therefore identified as part of the tag.

[15] MIF identifies text within a tag, so this is not applicable as it deals with text outside tags.

$$\frac{\text{(tags found - erroneous tags)}}{\text{actual tags in document}}$$

- **% of erroneous tags** is the percentage of tags identified by the system that were not tags in the document, calculated as follows:

$$\frac{\text{erroneous tags}}{\text{total tags found}}$$

- **% of actual parameters identified** is the percentage of parameters in the document that were correctly identified.

$$\frac{\text{(parameters found - erroneous parameters)}}{\text{actual parameters in document}}$$

- **% of erroneous parameters** is the percentage of tags identified by the system that were not tags in the document.

$$\frac{\text{erroneous parameters}}{\text{total parameters found}}$$

- **% of actual text identified** is the percentage of pieces of text[16] that were correctly identified.

$$\frac{\text{(text found - erroneous text)}}{\text{actual text in document}}$$

- **% of erroneous text** is the percentage of pieces of text identified by the system that were not actually document text.

$$\frac{\text{erroneous text}}{\text{total text found}}$$

As can be seen from this, the parser identified all of the tags and parameters in the documents. However, it also recognised some of the text as tags or parameters. The reasons for this are explained for each markup type in the following sections. The reasons for the high percentage of erroneous text in RTF and LaTeX is also explained.

6.2.1 MIF

The following symbols are defined in the system as the delimiters for tags in MIF files:

TAGOPEN	<
TAGCLOSE	>
GROUPOPEN	*none*
GROUPCLOSE	*none*

[16] Where the pieces are words surrounded by tags, e.g. in the following example, there are three pieces of text: <P>This is <B>bold<\b> text.</P>

PARAMSTART	*space*
PARAMEND	*none*
COMMENTSTART	#
COMMENTEND	*new line*
ARGSTART	*none*
ARGEND	*none*
STYLESTART	*none*
STYLEEND	*none*
PARASTART	*none*
PARAEND	*none*

Using these delimiters, a number of MIF files were passed through the system. The parser correctly recognised all tags and parameters in the MIF documents on which it was tested. Because text is stored as a parameter in a tag, this was found in the tag identification.

6.2.1.1 Issues Encountered & Possible Solutions

If a TAGOPEN delimiter was found in the text string of a document, the generic parser assumed this to be a new tag and wrote the text following it to the file as a tag. This caused problems in the comparison, as the text considered to be the tag was translated in the second document, meaning that the two erroneous tags could not be matched and an irrelevant error indicating that these non-existent tags were missing was reported. In the conversion, no generic equivalent could be found for the tag, so it was ignored, causing no problems. For example, consider the following String statement:

```
<String 'The < character is used as a TAGOPEN in MIF'>
```

The generic parser reads String as the tag and "'The " as the parameter. On encountering the "<" in the text, the parser assumes it is a new tag and the next characters in the text are read until a parameter separator (a space) is reached. This text, the "character", is written to the file as a tag and "is used as a TAGOPEN in MIF'" is considered its parameter. In the localised document, this text will be translated, but will encounter the same problem. However, the erroneous "tag" in that document will be a translated word from the string. As neither tag will have a match in the other document, the user will be given the messages "The tag 'character' is missing from file

2" and similarly for the extra tag, both of which are irrelevant and confusing to the user.

Similarly, a TAGCLOSE symbol in the text will end the tag, and the following characters are considered text as they occur outside the tags. To overcome this, the parser needs to know when a delimiter occurs in the text. For MIF, it would have to know that all characters between the quotes in a String tag are document text. This could be recognised in a rule-based system where each markup scheme has a set of specific rules by which it can be processed [HEAR97]. However, due to the complexity of implementing such a system, it was not used by this parser and remains an issue for further work.

6.2.2 RTF

The symbols for RTF delimiters are defined in the system as follows:

TAGOPEN	\
TAGCLOSE	*space*
GROUPOPEN	{
GROUPCLOSE	}
PARAMSTART	*none*
PARAMEND	*none*
COMMENTSTART	*none*
COMMENTEND	*none*
ARGSTART	*none*
ARGEND	*none*
STYLESTART	*none*
STYLEEND	*none*
PARASTART	*none*
PARAEND	*none*

All tags were found in the RTF documents used to test the system. However, because the parameter is appended to the tag with no intervening space in RTF, the entire expression is assumed to be the tag. The combination of tag and parameter is allowed for in the conversion, so this does not cause a problem for the main program.

6.2.2.1 Issues Encountered & Possible Solutions

Text is identified by either a space after a tag, or the text after a GROUPCLOSE bracket (i.e. "}") if the next character is not a recognised delimiter, such as the

TAGOPEN symbol (i.e. "\"). However, in the header information, expressions such as style names in the style sheet definition and font names in the font table are also deemed to be text, as they are also separated from the tags with a space. For example, the following example defining a style sheet is taken from the header of an RTF document:

```
{\stylesheet
      {\widctlpar \f4\fs20\lang2057 \snext0 Normal;}
      {\s16\widctlpar \b\f4\ul\lang2057 \sbasedon0\snext0 Heading;}
}
```

The style Normal is composed of all the tags in the group. It is not a markup tag, so it cannot be stored in the tag field. It cannot be considered a parameter to a tag as it is associated with all tags in the group, and most of the tags already have parameters. Because of the space separating the \snext0 from the style name, Normal is judged to be text by our system. This is not an accurate description of such expressions, but in the system there is no other field in which to store it. In order to recognise it as a style name requires a knowledge of RTF, which a generic parser cannot have. However, it sufficed for our comparison as all RTF documents are processed in the same way and thus all will contain these components as text. The only problem caused by considering such components as text is when comparing documents of the same type is in error reporting. Because the previous text is displayed in the message, these expressions may be used, but this will only happen for errors in the header information of the document, not the document text itself. However, if we are comparing an RTF document to another document, this TEXT field will be considered part of the document text and will be reported as missing from the other document. To overcome this problem requires the parser to recognise textual components in RTF, such as style names and font names. A distinction can be made between these components and the document text, as the text that appears in the header information is part of the markup, and the rest can then be assumed to be document text.

As with MIF, the use of a TAGOPEN delimiter in the text will be regarded as the start of a new tag, causing the same problems as described in section 6.2.1.1. A GROUPOPEN or GROUPCLOSE delimiter in text will signify the start or end of a group in this system. Even though RTF precedes delimiters in text with a "\" [MICR95], this can only be detected if the pre-processor has a previous knowledge of RTF, which a generic tool cannot have.

6.2.3 HTML

The following symbols are defined in the system as the delimiters for tags in HTML:

TAGOPEN	<
TAGCLOSE	>
GROUPOPEN	*none*
GROUPCLOSE	*none*
PARAMSTART	*space*
PARAMEND	*none*
COMMENTSTART	*none*
COMMENTEND	*none*
ARGSTART	*none*
ARGEND	*none*
STYLESTART	*none*
STYLEEND	*none*
PARASTART	*none*
PARAEND	*none*

One of the HTML documents on which the tool was run is given in Appendix B. The output file is also given. All of the tags were correctly identified, as was all text.

6.2.3.1 Issues Encountered & Possible Solutions

Tags in HTML have attributes instead of parameters. Attributes typically consist of an attribute name (which is a defined keyword), an equal sign and a value. For example, the IMG tag is used to insert images into a HTML page. This tag has a number of attributes [GRAH96], including:

- SRC, specifying the image to insert (this is compulsory).
- ALT provides a text description of the image.
- ALIGN, which specifies how the image is positioned relative to the text line in which it occurs.
- HEIGHT and WIDTH, specifying the intended height and width of the image in pixels.

e.g. <IMG SRC="picture.gif" ALIGN=middle>

Our system will consider everything after the IMG tag as part of the parameter. Therefore if any one attribute has changed, it is not distinguished from the others and a general message reports that the parameter has changed. For example, if the tag above was compared to <IMG SRC="picture.gif" ALIGN=right>, the images would be found to differ because the parameters differ, rather than being considered the same image with different alignments. Ideally, these attributes should be extracted and treated as tags also, e.g. the tag "IMG SRC" with the parameter "="picture.gif"", the tag "IMG ALIGN" and parameter "=middle". However, the comparison process matches the tags, not the parameters, so the correct match will still be made.

It may be possible to overcome this problem generically by storing any attributes for a tag together with how the end of the attribute can be recognised in a text file, as in:

TAG	**ATTRIBUTE**	**ENDATTRIBUTE**
IMG	SRC	*space*
IMG	ALIGN	*space*
etc.		

On encountering a tag, we can check the file to see if the parameter contains any of the attributes. If it does, each can be written to the file as a tag and a parameter, namely:

Tag	**Parameter**
IMG SRC	="picture.gif"
IMG ALIGN	=middle

Although it is usually omitted, white space is allowed around the equal sign, and therefore cannot be used accurately as a delimiter. Therefore, a processor could either remove the white space when it knows that it is dealing with an attribute, or by recognising the second quotes in the attribute as the end of the attribute. However, this requires a process specific to HTML which is in conflict with the generic nature of our research.

The inclusion of a TAGOPEN delimiter in the text can cause problems as already discussed. For instance, a TAGCLOSE delimiter used in the attributes of a tag can indicate the end of the tag, e.g. in the following tag, the tag will be ended when the ">" is read in from the ALT value:

```
<IMG SRC="equation.ps" ALT="a > b">
```

However, many HTML documents use an entity or numeric character reference to the symbol to allow compatibility with applications that consider any occurrence of ">" as to signify the end of a tag [RAGG95]. For example, the number corresponding to a TAGCLOSE symbol is 62, so the following tag could be replaced with:

```
<IMG SRC="equation.ps" ALT="a &#62; b">
```

Otherwise, a tool specific to HTML would have to be used, which is not the object of the exercise here.

6.2.4 LaTeX

The following table describes the symbols in the system used for the delimiters:

TAGOPEN	\
TAGCLOSE	*space*
GROUPOPEN	*none*
GROUPCLOSE	*none*
PARAMSTART	[
PARAMEND	]
COMMENTSTART	*none*
COMMENTEND	*none*
ARGSTART	{
ARGEND	}
STYLESTART	*none*
STYLEEND	*none*
PARASTART	*BLANKLINE*[17]
PARAEND	*new line*

The following document was used to test the parser for LaTeX:

[17] In the Implementation Chapter, we discussed how a paragraph identified only with a preceding blank line can be recognised in the parser by keeping count of the number of new line characters in a row. If more than one new line character is read in before some text, and the PARASTART delimiter is BLANKLINE, the text is assumed to be in a new paragraph.

```
\documentclass[12pt]{article}
\begin{document}
\title{LaTeX Overview}
\maketitle
\section{Overview}
LaTeX is considered to be generic markup because of its macro commands with
logical names, such as:
\begin{itemize}
\item "title"
\item "section" or
\item "quotation"
\end{itemize}
\section{Logical Or Physical}
"These logical tags coexist with the physical ones, so the user can define the
physical appearance if they wish, but otherwise this can be done using style sheets
for the type of document they declare their work to be. LaTeX can thus be said to
be a generic markup language, though it can be used in an old, physical way or in
the newer logical way" [DILL].
\end{document}
```

Figure 6.2 Sample Input LaTeX File

The output file is as follows:

```
documentclass 12pt article
begin document
title LaTeX Overview
maketitle
section Overview
TEXT LaTeX is considered to be generic markup because of its macro commands
     wit
begin itemize
item
TEXT "title"
item
TEXT "section" or
item
TEXT "quotation"
end itemize
section Logical Or Physical
TEXT "These logical tags coexist with the physical ones, so the user can define the
     ph
end document
```

Figure 6.3 Parsed Output File for LaTeX Document

The tags and text were all correctly identified in this document.

6.2.4.1 Issues Encountered & Possible Solutions

LaTeX has both arguments and parameters associated with tags. For example, in the markup \documentclass[12pt]{article}, the 12pt surrounded by "[" and "]" is a parameter, and article with the "{" and "}" is an argument. Because we only have one text field, param, associated with the tag, both must be stored in this one field. This will cause similar problems to those described for storing more than one attribute in the parameter of a HTML tag. However, the comparison will still find matching tags as the tag itself remains intact.

Another problem related to the arguments in LaTeX documents is the fact that they sometimes contain keywords (e.g. article, document, itemize) and sometimes document text, e.g. \title and \section tags take the document and section titles as arguments. Therefore the first type of argument should be stored as a parameter to the tag, but the latter as text to be checked for translation. This can only be achieved by processing specific to LaTeX to distinguish keywords from text. If all keywords were stored in a text file, the generic parser could check each argument against this file. If the argument is found it is recognised as a keyword, otherwise it is considered as text. However, if this process is applied to most other markup schemes (such as MIF, RTF), all parameters will be considered as text as these schemes do not use the concept of keywords. It is therefore not an acceptable solution and a separate tool is required to overcome this problem, which again contradicts the concept of generic tools.

As in other markup schemes, the generic parser cannot cope successfully with the occurrence of a delimiter in the text, so a separate processor would need to be used to overcome any related problems. The impact of this on our research is discussed in the summary of this chapter (section 6.6).

6.2.5 Overall Evaluation of Generic Data Preparation Tool

The tool identified all tags and the majority of parameters. The inaccuracies of the parser are summarised as follows:

- Attributes that are a key part of the tag, such the SRC and ALIGN keywords in <IMG SRC="picture.gif" ALIGN=middle> in HTML, are considered part of the parameter. However, the current method still allows an accurate comparison. If the order of the attributes have changed or any one attribute has changed, it is not distinguished from the others and a general message reports that the parameter has changed.

- Certain tags in LaTeX have both parameters and arguments. This requires both to be stored in a single field as the system only uses one parameter field. Again, the current method still allows an accurate comparison of tags. If any either the parameter or argument has changed, the error message only reports that the tag's parameter has changed, not which specific part, as discussed in section 6.2.4.1.
- Text in document header information can be wrongly identified as document text in certain markup schemes such as RTF, because it is neither a tag nor a parameter. They are also associated with more than one tag. Therefore there is no appropriate place to store them in our system and they are incorrectly considered as text, as discussed in section 6.2.3.1.
- Any delimiter with more than one character cannot be detected in our system as it works on a character by character basis. For example, the following example is a comment from a HTML document where <-- and --> are the delimiters:

  ```
  <-- This is a comment. -->
  ```

  The generic parser ignores all comments, as they are irrelevant in the comparison of the tags in two documents. However, in HTML comments, the parser recognises the "<" as a TAGOPEN delimiter and considers the "--" as a tag, with "This is a comment. --" as the parameter. This can result in numerous messages being reported for missing "--" tags, as the comments in the files will not necessarily match up.
- The occurrence of delimiters in the document text causes problems in all markup schemes. Because an inclusion is handled differently by each scheme, the only way to identify a delimiter in the text is to use a tool for each format.

This generic parser is successful in locating document tags and parameters which are the key to our comparison, but there are some problems with text in some markup schemes. Therefore, although it works well for our system as the tags are identified, it is an impractical process for an all-purpose generic parser that would require an accurate and detailed representation of the parameters and text.

Because each file is treated in the same way, each output file will be in the same format. Therefore, even with some of these errors, the comparison still works adequately once all tags are identified. Problems will arise, however, with the incorrect identification of text as tags. If text considered as a tag is a direct comparison, no match will be found for it in the other document and inappropriate messages will report that a non-existent tag is missing from the document. If it is to be converted, no match will be found for it in the generic tag set and will be ignored.

The data preparation tool was designed to work only on the body of the document because of the different conventions in document header information. However, most schemes include extra information with the document, and each has its own way of distinguishing it from the body of the document, so the main body cannot not be identified and extracted by a generic tool. Therefore the header information had to be processed along with the body of the document, causing a number of difficulties such as the style names or font names in RTF, described in section 6.2.2.

6.3 The Comparison of Documents

To test the comparison tool, documents with several markup schemes were used. The overall performance is summarised in Figure 6.4, with details of the errors encountered described in the following sections. The figures were not broken down by file type as the results for each were similar.

% changed tags correctly identified	92.3%
% tags incorrectly identified as changed	1.1%
% changed parameters correctly identified	93.5%
% parameters incorrectly identified as changed	3%
% missing tags correctly identified	91.4%
% tag incorrectly identified as missing	8.1%
% extra tags correctly identified	90.1%
% tags incorrectly identified as extra	7.9%
% untranslated text identified	100%
% translated text incorrectly identified as untranslated.	16.3%

Figure 6.4 Results of Tests on Comparison

where:

- **% changed tags correctly identified** =

$$\frac{\text{(changed tags found - erroneous changed tags)}}{\text{actual changed tags}}$$

- **% tags incorrectly identified as changed** (where this is the percentage of tags that were identified to have been changed, even though they had not) =

$$\frac{\text{erroneous changed tags}}{\text{changed tags found}}$$

- **% changed parameters correctly identified** = $$\frac{\text{(changed parameters found - erroneous changed parameters)}}{\text{actual changed parameters}}$$
- **% parameters incorrectly identified as changed** = $$\frac{\text{erroneous changed parameters}}{\text{changed parameters found}}$$
- **% missing tags correctly identified** = $$\frac{\text{(missing tags found - erroneous missing tags)}}{\text{actual missing tags}}$$
- **% tag incorrectly identified as missing** = $$\frac{\text{erroneous missing tags}}{\text{missing tags found}}$$
- **% extra tags correctly identified** = $$\frac{\text{(extra tags found - erroneous extra tags)}}{\text{actual extra tags}}$$
- **% tags incorrectly identified as extra** = $$\frac{\text{erroneous extra tags}}{\text{extra tags found}}$$
- **% untranslated text identified** = $$\frac{\text{(untranslated text found - erroneous untranslated text)}}{\text{actual untranslated text}}$$
- **% translated text identified as untranslated** = $$\frac{\text{erroneous untranslated text}}{\text{untranslated text found}}$$

The errors in the comparison were due to many factors, including the problems of constructing a generic parser. The results are discussed for each possible difference below. A list of errors that must be taken into consideration were compiled and are discussed for the comparison of documents with identical schemes in section 6.3.6, and for the comparison of different schemes in section 6.4.3.

6.3.1 Recognising Changed Tags

The comparison process was successful in recognising most of the changed tags in a document. However, it will not recognise changed tags if there are a number of changes reported in the same "hunk", and it there are errors in more than 20%[18] of the 15 tags immediately after the changed tag. In the following example many of the following tags have changed. The changes are highlighted in bold:

	Document 1
	P
*	**CENTER**
*	**B**
	TEXT Phone:
	/B
	TEXT 01-7045618
	/P
	P
	B
	TEXT Project Title:
	/B
	TEXT Generic Comparison...
	/P
	etc.

	Document 2
	P
*	**I**
	TEXT *Phone:
	/I
	TEXT 01-7045618
	/P
	HR
	P
	I
	TEXT *Project Title: ***Generic ...**
	/I
	/P
	etc.

Figure 6.5 Example of a Tag Change That is Not Detected in the System

The tag B in the first hunk of changes (i.e. those marked with a "*") has been changed to I. However, because the CENTER tag is missing from file 2 at the same position, the ReAlign function was called to verify which of these tags has been changed to I. Many of the subsequent tags had changed, so the process did not consider either CENTER or B to have changed to I. Therefore, CENTER and B are considered as missing from list 2, and I is considered to be missing from list 1. Despite this error, the inclusion of the CompNext and ReAlign functions have improved the identification of changed tags, and tags that have not changed are less likely to be wrongly identified.

6.3.2 Recognising Changed Parameters

Once the correct tags had been matched, all changed parameters were recognised by the system. However, in some instances, a tag may have been incorrectly considered to match another, and if the parameters differed, an error reported that the parameter had changed, when in reality, the tags were in fact an inaccurate match.

[18] In Chapter 5, we described how up to three differing tags in the next fifteen will be accepted and the tags will still be considered to match. The percentage is then 3/15 = 20%

6.3.2.1 Issues in Recognising Changed Parameters

Some parameters were reported as having changed when the attributes were the same but the order was different, or where only some of the attributes were missing, as described for HTML in section 6.2. Ideally, a comparison system would recognise this and report an appropriate message detailing the reason for the difference instead of simply stating that the parameter has changed. However, in order to identify missing sections, the comparison would need to know which markup schemes use attributes, which tags have attributes and how the attributes are composed. As indicated in section 6.2, this requires separate processing for each scheme. This could be done either by a separate preparation tool outputting each section as a separate tag, or in the comparison itself, if it were adapted for particular markup schemes. However, the use of specific tools is nt an option for us here, as it is outside the scope of this research.

6.3.3 Recognising Missing Tags

The comparison process was successful in finding most of the missing tags in the document. However, certain issues such as the order in which tags occur, different end tags for the same format, tags that have been moved and different paragraph representations will cause tags to be wrongly considered as missing. These issues are explained later in this chapter. Tags that had changed but were not identified correctly were also deemed as missing. Furthermore, some missing tags were matched to tags, causing others to be deemed missing, e.g.

Document 1
P
TEXT Phone: 01-7045618
/P
P
B
TEXT Project Title:
/B
TEXT Generic Comparison...
/P
P
B
TEXT Description:
/B
etc.

Document 2
P
TEXT *Phone: 01-7045618
/P
P
B
TEXT *Description:
/B
TEXT *This research is involved
/P
HR
CENTER
P
etc.

Figure 6.6 Example of an Incorrect Identification of a Missing Tag

In this example, the tags in bold are missing from document 2. However, because of the similarities between the tags in the same position in document 2, the text "Project Title:" is considered to match "*Description" in document 2, and "Description" in document 1 is deemed to be missing. This is explained in more detail in section 6.3.6.1.

6.3.4 Recognising Extra Tags

The comparison process was successful in finding most of the extra tags in the second document. However, certain issues such as the order in which tags occur, different end tags for the same format, tags that have been moved, different paragraph representations and the inclusion of accented characters will cause tags to be wrongly considered as missing. These issues are explained later in this chapter. Some tags that had changed but were not identified correctly were also considered as additional tags. Similarly, groups of similar tags as described above will again cause tags to be incorrectly considered as extra.

6.3.5 Recognising Changed Text

To test for the translation of text, any text in the document that would have been translated was instead prefixed with a * to indicate such a change. The process compares the strings and when they are found to differ, no error is reported. All text that had been changed was identified. However, numerous pieces of text that had not changed were reported as being errors, as not all text in a document is translated. For example, names, numbers, companies or products will not usually change during translation, and should not cause an error. If unchanged words such as these are part of a "translated" sentence, the sentence as a whole had changed and no error is reported. However, if the text was broken up by tags, each piece of text between the tags is considered separately, and the unchanged text may be stored on its own. For example, Author is the only word to be translated in the following text, but because the text is considered as a whole, no error is reported to say that Joe Soap had not changed.

```
<P>Author: Joe Soap </P>
```

However, in the following text, Joe Soap is separated from Author by the bold tag and stored in a TEXT tag of its own.

```
<P><B>Author:</B> Joe Soap</P>
```

Therefore, the comparison considers the unchanged Joe Soap as an error. This results in messages for unchanged text being reported, even though it is not an error. However, to identify parts of the text that should not be translated requires the program to understand the text, which is well beyond the scope of this research.

6.3.6 Issues Encountered During the Comparison of Documents of the Same Markup Scheme

6.3.6.1 Groups of Similar Tags

If a document is formatted consistently, certain groups of tags can occur repeatedly throughout the document. For example, each section may start with a similar heading. Therefore, if the documents need to be re-aligned, the wrong group of tags may be chosen to match the current tags. In the example in Figure 6.7, a paragraph is missing from document 2. The missing tags are highlighted in bold in document 1.

Because the tags in the missing group are similar to the tags following it (i.e. they both consist of the tags P, B, TEXT, /B, /P), the group of tags numbered as 2 in document 1 will be matched with group 2 in document 2. However, group 2 in document 2 should match group 3 in document 1. Therefore, the comparison will not identify the missing tags. When the comparison tries to match group 3 in document 1 with group 3 in document 2, it will find a difference, and report the tags in group 3 in document 1 are missing. The only way to solve this is to examine and match the document text. However, the text is expected to differ after translation, so a direct comparison cannot be applied and the deduction of the meaning of the text is beyond the scope of this research.

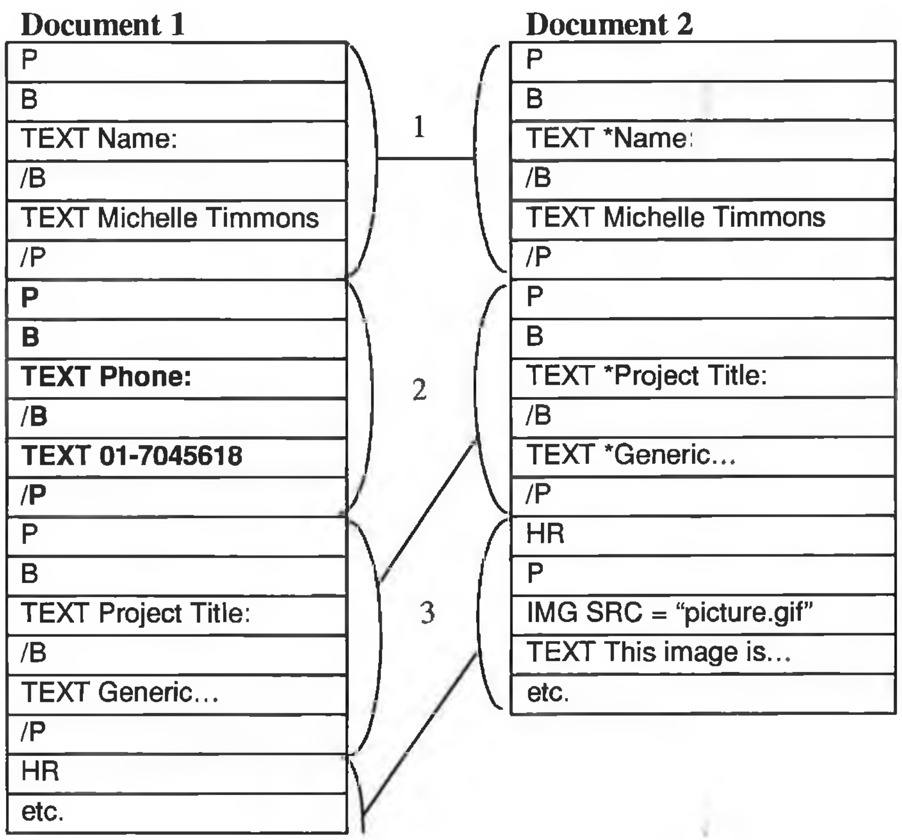


Figure 6.7 Example of Two Documents with Similar Groups of Tags

6.3.6.2 Optional Tags

In some markup schemes, such as tag sets created using SGML, certain tags can have end tags that are optional. The end tag may be omitted for any element that cannot contain another element of the same type, e.g. a paragraph cannot contain another paragraph. The occurrence of another paragraph indicates the end of the previous one, whether an end tag is specified or not. Tags other than end tags can be optional also. For example, "the HTML, HEAD and BODY start and end tags can be omitted from the markup as these can be inferred in all cases by parsers conforming to the HTML 3.2 DTD" [RAGG97]. Therefore, one document in the comparison may use the optional tags and the other may not. For cases such as this, our system reported numerous errors for missing tags in the document in which the end tags were omitted, even though this is not an error for certain schemes.

6.3.6.2.1 Possible Solution

For the system to ignore missing tags that are optional requires a knowledge of the markup to know which are optional and which are not. A text file could store a list of optional tags for each scheme, e.g.

```
[HTM]
HTML
/HTML
HEAD
/HEAD
/P
etc.
```

This information could be stored in the tag mapping file, but this would result in the information only being available for those tags that have generic equivalents in the system. Therefore, the use of a separate text file is recommended.

The preparation tool could then check each tag as it is encountered against this file and if the tag is optional, it would not write it to the output file. This ensures that the optional tags will not occur in any document, so a difference will not be identified in the comparison. However, if the document is being compared to a document from a different markup scheme in which end tags are compulsory, numerous errors will be reported for missing end tags, as discussed in section 6.4.3.2.

6.3.6.3 Different End Tags for the Same Format

Even though two documents of the same markup scheme are being compared, a markup scheme can have different end tags for the same format or object. In the following example from RTF, the GROUPCLOSE tag ends both the \i and \b formats:

```
\par this paragraph contains text that is both {\i\b bold and italic.}
```

However, in the following example, the \i0 ends the \i tag and \b0 ends \b.

```
\par this paragraph contains text that is both \i\b bold and italic.\i0\b0
```

Therefore, although these two examples are equivalent, the tags are not identical, so an error will be reported.

6.3.6.3.1 Possible Solution

It may be possible to overcome this problem in the data processing tool by keeping track of all tags opened since the GROUPOPEN symbol (e.g. by implementing a stack). When a GROUPCLOSE tag is reached an end tag for each opened tag in that group can be output. To determine the end tag for each tag, a list of all start tags and their corresponding end tags must be known by the system. An external text file can store this information for each scheme and the process can examine this to find the appropriate end tag. The following example shows a sample file containing only RTF tags, with the start tag on the left and the corresponding end tag on the right:

```
[RTF]
b           b0
i           i0
ul          ul0
uldb        ul0
etc.
```

Using this method, the first example would be processed and output as:

```
par
TEXT this paragraph contains text that is both
i
b
TEXT bold and italic.
i0
b0
```

This is identical to the output that would result from processing the second example above, allowing a correct comparison. Also, this format can be compared without error against a similar file in a scheme such as HTML that uses end tags. The original format would cause errors for a missing end-bold tag and end-italic tag.

6.3.6.4 The Order in Which Tags Occur

The order in which certain tags can occur in a document may differ. For example, if a piece of text is both bold and italic, there is no standard for the order in which the bold and italic tags should appear. Although each application has its own conventions, not all applications will adhere to this order. For example, Microsoft Word will output the bold tag and then the italic tag in an RTF document, but other applications creating RTF documents may not necessarily output the same order. Also, many documents such as HTML and LaTeX are often created manually, so the order is determined by the user.

In our system, if the tags in the two documents are not in the same order, tags in the second document will be skipped over to find the match for the tag in the first document. For example, consider the following extracts from two RTF documents, where the \b tag represents bold, \i is italic, \ul is underline and TEXT is the generic tag used by the system for document text:

Document 1	Document 2
...	...
\par	\par
ul	i
i	b
b	ul
TEXT	TEXT
\b0	\ul0
\i0	\b0
\ul0	\i0
etc.	etc.

Figure 6.8 Examples of Groups of Similar Tags in Different Orders

The \ul tag in the second document will be found by skipping over the \i and \b tags, reporting them as missing. When the comparison process attempts to find a match for the \i tag from the first document, it will not backtrack to the tags it skipped over, and therefore will not find the correct matching tag. The \i and \b tags will then be reported as missing from the second document because they will not be found.

6.3.6.4.1 Possible Solution

This problem can be overcome in a number of ways. Firstly, the process could check previous unmatched tags. This is not implemented in our system as the algorithm on which it was based does not incorporate this. It may be possible to update the algorithm to include this at a later stage.

The second way to overcome this problem is for the pre-processor to keep track of certain formats which are active (i.e. those that have been opened but not yet closed), and output them in the order specified by our system. To do this, the pre-processor must know which tags can be rearranged, e.g., a paragraph tag cannot be moved. The only tags that can be moved are those specifying the format of the text, such as underline, bold, end bold, etc. This information can be stored in an external file to let the process know the tags in each scheme for these formats. The process also needs to know what groups of tags can be rearranged. For example, consider the following extract from an RTF document:

\par \ul\b This is some text **\b0\ul0\i\b** and this is more text \b0\i0

The tags \b0, \ul0 \i and \b (in bold) are all formatting tags that can be re-ordered. However, if these are re-ordered alphabetically, they become:

\par \ul\b Text **\b\b0\i\ul0** more text \b0\i0

The start tags and end tags have been mixed together, which is not a correct representation of the document. For example, the \b tag is immediately followed by a \b0 tag which turns off bold. Therefore, start tags can only be rearranged among themselves, and similarly with end tags. Furthermore, character formatting tags should not be mixed with paragraph formatting tags such as line spacing, indentation, etc. The information to be stored in the external file is the tag, whether it is a start tag or end tag, and whether it is a character or paragraph format, e.g.:

```
[RTF]
b           STARTTAG     CHAR
i           STARTTAG     CHAR
b0          ENDTAG       CHAR
i0          ENDTAG       CHAR
\sl         STARTTAG     PARA
\li         STARTTAG     PARA
\ri         STARTTAG     PARA
etc.
```

where \sl is line spacing, \li is left indent of a paragraph and \ri is the right indent of the paragraph.

Using this method, the data preparation tool would read in each tag and check the external file to determine if it can be moved. If it can, it would be stored in the program using a stack, for example. The next tags are read in and checked until a tag is encountered that either is not in the file, or it is in the file but is of a different type (i.e. if the tag is a start tag, we stop when an end tag is reached and vice versa. The same steps are taken when checking for character and paragraph formatting). The group of tags are sorted and output in this new order, and the process continues in the same manner. In the example in Figure 6.8, the tool could output \b \i \ul, if the order chosen for the system was alphabetical. The second document would be processed by the same tool, and so the tags would be written in the same order allowing an accurate comparison.

6.3.6.5 Accented Characters

If one of the documents being compared has been localised, it is possible that it will contain accented characters. These are usually included in the document using another tag. The following example shows how an "↔" is included in an RTF document:

```
\par An accented a: {{\field{\*\fldinst SYMBOL 171 \\f "Times New Roman Special
G1" \\s 10}{\fldrslt\f45\fs20}}}
```

The text "An accented a:" precedes the accented character, and the {{\field{*\fldinst SYMBOL 171 \\f "Times New Roman Special G1" \\s 10}{\fldrslt\f45\fs20}}} represents the character. It is referencing a symbol on the "Times New Roman Special G1" font. The comparison of this to the original document will obviously report numerous errors for the extra tags.

6.3.6.5.1 Possible Solution

A list of tags representing all accented characters in a text file could be checked when an error is encountered in the comparison. However, because of the number of tags that constitute each accented character, it may be better to wait until after the comparison and then check all groups of unmatched tags with the tags comprising each accented character. If a matching group is found, then these tags are not reported as errors.

6.3.6.6 Moved Tags

If a group of tags had been moved in a document, the intermediary tags were reported as missing. In the following example, the tags marked in bold were moved in the documents.

	Document 1		Document 2
	P		P
	B		B
	TEXT *Name:		TEXT Name:
	/B		/B
	TEXT Michelle Timmons		TEXT Michelle Timmons
	/P		/P
>	P	>	**HR**
	B		**P**
	TEXT *Project Title:		**IMG SRC = "picture.gif"**
	/B		**TEXT This image is...**
	TEXT *Generic...		**/P**
	/P		P
	HR		B
	P		TEXT Project Title:
	IMG SRC = "picture.gif"		/B
	TEXT This image is...		TEXT Generic...
	/P		/P
	etc.		etc.

Figure 6.9 Example of Two Documents with Groups of Moved Tags

When the comparison reaches the tags marked with the ">", it recognises the difference in the two documents, as P does not match HR. It skips over the bold tags in document 2 until it finds the tags matching those in document 1. These tags in document 2 are marked as extra in that document. The comparison continues until the moved tags (marked in bold) are found in document 1. No match can be found for them as the matching tags have already been processed and the function does not backtrack. Because of this, the moved tags in document 1 are considered to be missing.

6.3.6.6.1 Possible Solution

It may be possible to solve this problem by examining the unmatched tags in both documents after the comparison and trying to find groups of tags that match. These can then be deemed as having been moved. However, it is possible that this would match groups of similar tags even though they do not correspond, as described in section 6.3.6.1.

6.4 Comparing Two Documents with Different Markup

In order to allow the comparison of two documents in different formats, they were first converted to the generic tag set. The performance of the conversion process is described below. We then discuss the comparison of two converted documents, followed by a discussion of the issues encountered therein. Because the results of the comparison of converted documents were so similar to those of documents with the same markup, they are incorporated into the results of the overall comparison in Figure 6.4.

6.4.1 Conversion

The conversion process is limited by the number of tags of each file type associated with generic tags in the tag mapping file. All unrecognised tags are changed to NOMATCH as we cannot make a guess as to what the unrecognised tags mean and are subsequently ignored in the comparison. Because of this limitation, not all tags were identified for the comparison process in the tests, resulting in a loss of accuracy. In fact some of the tags that do not have generic equivalents are commonly used within the markup scheme. For example, the MIF tags Pgf (defining a paragraph format), PgfFont (defining character formats for a paragraph), and ParaLine (defining a line within a paragraph) were not converted. However, on examining the tags with no mapping to the generic tag set, it was discovered that many of them do not have an equivalent in other markup languages, and therefore a matching tag would not be found in a document of another markup scheme.

6.4.2 Comparing the Equivalence of Two Documents

Although there is a loss of detail during the conversion for tags that are not contained in the generic tag set, this results in the equivalence of the documents, rather than each tag, being compared. For example, consider the following paragraphs. Figure 4.2 illustrates a paragraph in RTF:

```
/par This is {\b bold} text in RTF.
```

Figure 6.10 Extract from RTF document

The figure below is the same paragraph marked up using MIF:

```
<Para
   <ParaLine
      <String 'This is '>
   >
   <ParaLine
      <Font
         <FWeight 'Bold'>
      >
      <String 'bold '>
   >
   <ParaLine
      <Font
         <FWeight 'Regular'>
      >
      <String 'text in MIF.'>
   >
>
```

Figure 6.11 Extract from MIF document

The conversion of the RTF and MIF extracts would result in the figure below:

(a)
```
PARAGRAPH
TEXT This is
BOLD
TEXT bold
BOLDOFF
TEXT text in RTF
```

(b)
```
PARAGRAPH
NOMATCH
TEXT 'This is '
NOMATCH
NOMATCH
BOLD
TEXT 'bold '
NOMATCH
NOMATCH
BOLDOFF
TEXT 'text in MIF.'
```

Figure 6.12 Extract from Documents After Conversion: (a) RTF (b) MIF

If ParaLine and Font had equivalent generic tags, then comparing these two paragraphs would give three errors for extra ParaLine tags and two for the extra Font tags in the MIF document. However, because there is no generic equivalent and NOMATCH tags are ignored in the comparison, no errors are reported. Therefore, this loss of detail results in a comparison with an emphasis on the equivalence of the documents.

This is more beneficial to the user because reporting that a ParaLine tag is missing from an RTF document, for example, is irrelevant because RTF does not support the concept of lines in paragraphs so nothing can be done to the RTF document to "fix" this error. However, reporting that a tag such as PARAGRAPH or BOLD from the generic tag set is missing, informs the user of an error that can possibly be corrected in another markup scheme, as the generic tag set consists of elements and formats common to most schemes. Therefore it is recommended that if the user wishes to extend the generic tag set, they only do so for widely used formats.

Instead of relying on the user to maintain the generic tag set correctly, there is another way to eliminate the problem of differing amounts of tags describing the same format. This is to create a tool to identify the combination of tags in a MIF document that constitute bold text, for example, and output a single tag for them. However, this requires a tool for each separate markup scheme in which this problem can occur, and this conflicts with the concept of generic tools.

Other tags that are common in many schemes such as line spacing, borders and those relating to tables, were also ignored by the system, but this was because the existing tag mappings do not cover all formats and elements as only a subset was included to test the design.

6.4.3 Issues in the Comparison of Two Converted Documents

The issues encountered when comparing documents with identical formats (e.g. recognising changed tags, missing tags) are also applicable to the comparison of two documents of different markup schemes, given that the comparison process is the same after conversion. In addition to these, a number of other issues were discovered when comparing two converted documents.

6.4.3.1 Paragraph Representations

The way in which different markup schemes represent paragraphs can cause problems in the comparison. HTML and RTF record only the paragraph and consider all the text between the paragraph start tag and paragraph end tag as one block of text. However a paragraph in MIF is broken down into separate strings of text in ParaLine elements, where the ParaLine marks a single line of the paragraph's text. Therefore the ParaLine tags have no equivalent in other schemes and would simply be reported as missing from the other document. For instance, consider the following paragraph in RTF:

```
/par Another problem that can arise in comparing different markup schemes is the
way in which each scheme represents paragraphs.
```

Figure 6.13 Example Paragraph in an RTF Document

The paragraph in the figure below is the equivalent paragraph in MIF:

```
<Para
   <ParaLine
      <String 'Another problem that can arise in '>
   >
   <ParaLine
      <String 'comparing different markup schemes schemes is the way '>
   >
   <ParaLine
      <String 'in which each scheme represents paragraphs.'>
   >
>
```

Figure 6.14 Example Paragraph in a MIF Document

The conversion of the RTF paragraph would result in the following:

```
PARAGRAPH
TEXT Another problem that can arise in comparing different markup schemes is
the way in which each scheme represents paragraphs.
```

Figure 6.15 Converted Paragraph in RTF

The MIF paragraph would resemble the following figure after conversion:

```
PARAGRAPH
NOMATCH
TEXT 'Another problem that can arise in '
NOMATCH
TEXT 'comparing different markup schemes schemes is the way '
NOMATCH
TEXT 'in which each scheme represents paragraphs.'
```

Figure 6.16 Converted Paragraph in MIF

Comparing these documents reports that there are numerous extra TEXT tags in the MIF document.

6.4.3.1.1 Possible Solution

Any text separated with just NOMATCH tags could be combined into a single TEXT tag because NOMATCH tags are ignored by the comparison anyway. However, this may not be an accurate representation as the NOMATCH tag may have been an element such as a table that is not currently recognised by this system. Elements such as this would split the text in the document. Nevertheless, both documents would be processed in the same manner and therefore the text would be combined in both documents.

For a more accurate solution, a separate parser for MIF and other such schemes could join all the ParaLine's within the paragraph tags into a single paragraph for comparison to other file formats while ensuring that it is valid union, but this would not be a generic tool as it would rely on processing specific to the markup scheme.

6.4.3.2 Differing Methods of Ending Tags

Many markup schemes, including HTML, use a start tag to identify the start of an element or format, and an end tag identifying the end of it, e.g.

```
<P>This is a HTML paragraph and <B><I>this text is italic and bold.</B></I></P>
```

In this example, the paragraph is started with the <P> tag, and </P> signifies the end of the paragraph element. The text is set to bold with the <B> tag, </B> turns it off, and likewise for the italic.

Other schemes such as MIF nest tags within each other, rather than using end tags. Therefore the delimiter symbol used to close the tag also ends the element itself, as in:

```
<Para
    <ParaLine
        <String 'This is a MIF paragraph and '>
    > # end of ParaLine
    <Font
        <FWeight 'Bold'>
        <FAngle 'Italic'>
    > # end of Font
    <ParaLine
        <String 'this text is italic and bold.'>
    > # end of ParaLine
    <Font
        <FWeight 'Regular'>
        <FAngle 'Regular'>
    > # end of Font
> # end of Para
```

In this example, the String tag is nested inside the ParaLine tag, where it itself is nested inside the Para tag. The ">" marking the end of the Para tag ends the paragraph, and similarly the ">" marking the end of the ParaLine tag ends the ParaLine element, i.e. there is no separate tag used to end the elements.

Schemes such as RTF use yet another method of ending formats and elements. Although many RTF tags have end tags, (e.g. \b0 ends a \b tag), usually tags are grouped together using group-open and group-close delimiters, { and }, and a group-close delimiter will end all tags within that group, e.g.

```
\par This is an RTF paragraph and {\i\b this text is italic and bold.}
```

Here, the \i and \b tags are grouped together, and on reaching the } for that group, both tags are ended.

These differences in the methods of ending tags can cause problems when trying to compare them. An actual end tag in HTML is equivalent to a ">" in MIF that ends the tag, or an end group in RTF to which many end tags can correspond. Also, our data preparation tool removes all delimiters, leaving no indications of where tags are closed in MIF and RTF. However, for the system to have used this information, it would have to understand each format, requiring separate processing for each scheme. If no match is found for an end tag, we cannot assume that this is because the markup in the other document does not record them, as it could be the case of a missing end tag.

6.4.3.2.1 Possible Solution

To overcome this problem, a separate tool could process each markup scheme that does not use an end tag, outputting a relevant tag on meeting the TAGCLOSE or GROUPCLOSE symbol. For example, when parsing a MIF document, on finding a > that closes a Para tag, for example, the tool could output ENDPARA for the tag. Detecting which tag it is ending will also differ depending on the scheme. For MIF, the tool could keep track of the tags that have been opened and write an end tag for the most recently opened. For example, in the following paragraph, the first TAGCLOSE delimiter encountered is for the most recently opened tag, String:

```
<Para
    <ParaLine
        <String 'This is a paragraph.'>
    > # end of ParaLine
> # end of Para
```

Alternatively, the tools could use the comments output by Frame products for each tag-close, in the same way as fm2HTML [STEP], although manually generated files may not include comments. A tool for RTF could use flags to record what tags are open, and output appropriate end tags on encountering the group close delimiter, as described in section 6.3.6.3 above.

However, each of these solutions requires a tool for each specific markup language, which is in conflict with the concept of generic tools. The impact of this on our research is discussed in the summary of this chapter (section 6.6).

6.4.3.3 The Storage of Measurements

Another problem encountered during testing is the way in which numbers and measurements are stored in different markup schemes. For example, numbers can be stored as integers or decimals by the markup scheme. Because they are treated as characters by this system, a string comparison of 1 and 1.0 will fail to find them the same.

6.4.3.3.1 Possible Solution

We could try to convert the parameters to numbers, and compare these (i.e. compare the values of 1 and 1.0 instead of the characters they are comprised of). However, some of these numbers may be used to represent a measurement, and different markup schemes use different units of measurement. For example, RTF measures in twips[19], whereas MIF uses inches. To specify a left indent of 1 inch for a paragraph, RTF will use the tag \li1440, with the parameter being 1440, and MIF will use <PgfLIndent 1.0"> so the parameter is 1.0". Even though these are equivalent, they are not the same. To convert these to numbers will not work in this case, as firstly, the MIF indent contains

[19] A twip is "1/1440th of an inch or 1/20 of a printer's point. There are thus 1440 twips to an inch or about 567 twips to a centimeter" [HOWE97].

a non-numeric character ("), and even if they both could be converted to numbers, a comparison will fail because they are not identical.

Our system reports an error stating that the parameters of these tags are different in cases such as this. Ideally, it would recognise the equivalence between the parameters and not give an error. As most markup schemes use only one unit of measurement, this could be used to convert the measurements to a single unit used in our system. For example, if we decided to use centimetres in the system, all measurements in MIF are in inches, so any parameters that are numeric and are ended with a " (identifying the unit of measurement) could easily be changed to centimetres in the conversion process. However, in schemes such as RTF that do not use a symbol to indicate the unit, this is not possible. For example, RTF uses numbers for many purposes, including the indication of fonts in the font table, font size, styles in the style sheet, dates and colours. Because there is no indication of the purpose of any number, it is wrong to assume all numbers are twips and then convert them to centimetres. We would need to know the purpose of each number, which can only be determined by understanding the markup scheme. It may be possible to do this by listing the tags that use measurements for each scheme, along with the default units used, in an external file that can be examined by the preparation tool. If the tag is in this list, it could then convert it to the unit used in our system. If this is not possible, a separate tool for each scheme would be required, which is conflicting with the concept of our research.

6.5 Error Reporting

An example of the errors reported in a comparison of two documents as follows:

```
RESULT OF COMPARISON OF c:\bcw\test\htmltest1\myoutput.htm AND
c:\bcw\test\htmltest1\myoutput2.htm
******************************************************************
UNCHANGED: The text "E-mail:" between
      Michelle Timmons
and
      mtimmons@compapp.dcu.ie
has not been changed in file 2
******************************************************************
MISSING TAG: The 'B ' tag between
      mtimmons@compapp.dcu.ie
and
      Phone:
in file 1 is missing from file 2.
******************************************************************
MISSING TAG: The '/B ' tag between
      Phone:
and
      01-7045618
in file 1 is missing from file 2.
******************************************************************
```

Figure 6.17 Extract from an Error Report File

6.5.1 Issues In the Error Reporting of the System

6.5.1.1 Displaying the Erroneous Tag

Displaying the actual document tag is not a desirable way of presenting the user with the problem because they will not necessarily understand the tag, especially if the document was created in a WYSIWYG package. However, because the process is designed to be generic and accept document of any type, we cannot give an explanation of the tag without a list of all tags of all types.

6.5.1.1.1 Possible Solution

It may be possible to add this description to the tag mapping file, as in:

```
[RTF]
par             PARAGRAPH          Paragraph
b               BOLD               Bold
b0              BOLDOFF            End bold
qj              JUSTIFY            Full justification of paragraph text
etc.
```

Alternatively, we could check the existing tag mapping file for its generic equivalent if one exists, and display that in the message, e.g. using BOLDOFF is more informative than displaying the \b0 from RTF.

6.5.1.2 Displaying the Associated Text

For each tag with an error, the system displays the tag with the text preceding and following it in the document, even though the tag will apply to only one of the pieces of text displayed. This may confuse the user as to which piece of text is in. Displaying the associated piece of text would be more beneficial. However, to do this requires an understanding of each tag to determine whether it is a start tag or end tag. A start tag applies to the text following it, and an end tag is associated with the previous text. Consider the following example:

```
<P>Text 1 <B> text 2 <\B><I> text 3 <\I> text 4.<\P><P><B><I> text 5 ....
```

The <P> is a start tag and is therefore associated with the text following it, i.e. "Text 1". <B> is setting the text following it to bold. As a start tag, it has no bearing on the previous text, "Text 1", so it is associated with text 2. <\B> turns off the bold formatting of the text preceding it, so is associated with "text 2". <I>, even though it is directly beside <\B>, has nothing to do with "text 2", but rather is applied to "text 4" to make it italicised, and so on. This demonstrates that determining the text to which the tag applies is not related to the proximity of the tag to the text (e.g. the second <P> tag in the above example is closer to "text 4" but actually is associated with "text 5") or which tags are adjacent (e.g. the <\B> and <I> are beside each other, but yet they apply to different text). Rather, determining the associated text depends on whether the tag is a start tag or end tag.

Deciding if a tag is an end tag is complicated by the use of different end tags for the same format as described in 6.3.6.3 and the different methods of ending tags, as discussed in section 6.4.3.2. For example, in MIF the text is embodied in a String tag and the end of the tag is indicated by the TAGCLOSE delimiter. There is no separate end tag so there is no uncertainty as to which piece of text the tag applies. However, HTML uses end tags like those in the example above. RTF can use different ways of ending the same tag, so this can cause ambiguity. Therefore, to decide whether a tag is a start tag or end tag depends on the markup scheme. Information indicating the role of a tag could be stored in an external file which could be examined during the error reporting process. This file could be similar to the layout of the tag mapping file, as in:

By careful maintenance of the tag mappings in the system, the conversion will result in a comparison of the equivalence of documents of differing schemes, which is more beneficial to the user.

The comparison worked well in most cases. However, problems were encountered that were again due to the peculiarities of each markup scheme. These can be categorised in two ways: those that require information about each markup scheme and those that require special processing. Our method of allowing the process to "understand" each scheme is to store information on each scheme in an external text file to be examined during processing, as used in the conversion. Many of these, such as optional tags, the order of tags, and the specification of numbers could be overcome by extending the text file(s) to cover other characteristics of the schemes. However, the issues requiring special processing (such as the number of tags used to specify a single format, the representation of paragraphs, different ways of ending tags) can only be overcome a specific tool to process each scheme to solve them.

The system as a whole would obviously benefit greatly from a knowledge of each scheme to handle the issues of each individually, as is currently the case for tools used in the industry. However, each of these tools is designed to work specifically with a single scheme, and a new tool must be developed for each new markup scheme. Our aim was to discourage this practise by developing a generic process that can be used on all schemes. To keep the comparison as generic as possible required omitting all details of any markup scheme. Despite this, however, the comparison successfully dealt with most of the test cases.

7. Conclusion

The purpose of this research was to examine the viability of a generic process to compare two documentation files. The motivation behind this research came from the localisation industry. An important part of the localisation process is the quality assurance of all localised products, including help and documentation. Translation of text inevitably introduces accidental alterations into the formatting and layout of the document, requiring the verification of the localised document against the original.

We also aimed to contribute to the standardisation of localisation software by devising a solution that is generic, allowing reusability. Therefore the process was developed to work on the markup of any documentation. In keeping with the concept of generality, the process also incorporates the comparison of documents with different markup schemes. Although localised documents usually employ the same scheme, this process could be applied to the comparison of the equivalence of documents published in different formats (e.g. printed documentation, on-line documentation, WWW pages).

We proposed the development of a generic process that can compare any two documentation files and explain the differences found. This process involves the identification and comparison of the markup codes that specify the *format* and *structure* in both documents, where the format is the physical appearance of a document (e.g. bold text, underlines) and the structure is how the document is composed (e.g. chapters, paragraphs). This markup is extracted from the document by a generic parser that identifies the tags from the specified delimiters.

Because of the differences between the two categories of markup (specific and generalised), four different cases were identified in which different treatment is required:

- the comparison of two documents of the same markup scheme.
- the comparison of two documents with specific markup.
- the comparison of two documents of with generalised markup.

- the comparison of two documents of the different markup categories (i.e. one specific and one generalised).

The comparison of two documents of the same markup scheme required an algorithm to compare the documents tag for tag and to identify and report the reasons for any differences found. The process checks for tags that have changed, parameters to tags that have been altered, tags missing from either document and untranslated text. Our process could be implemented with any comparison algorithm deemed suitable for the task in hand. The comparison we considered most suitable was the algorithm on which the UNIX *diff* command is based. However the comparison algorithm used is extraneous to our process, as it is only used to allow us to implement our generic process. This research involved the modification of a comparison algorithm to be applied to document markup, rather than its development as such.

The comparison of two documents with specific markup uses the same comparison as for documents with identical markup schemes. However, because each markup scheme uses its own set of tags, the markup of both documents must first be converted to the same tag set. Our generic tag set to which the documents are mapped incorporates formats and elements common to most markup schemes.

The comparison of two documents with generalised markup required the markup of both documents to conform to a generic structure to allow the elements to be compared. For schemes with a defined tag set, each tag is mapped to a tag in our generic tag set describing the structure. However, for generalised markup languages, the elements in each document need to be assessed to determine their role in the structure of the document, e.g. a paragraph, section, etc. In our prototype, the user must create a separate text file with a mapping for each tag identifying an element in the document to a tag in our generic tag set describing the structure. However, this may be automated by developing a process to examine the contents of each element to determine the generic element to which it corresponds.

In order to compare two documents of the different markup categories, we proposed that both documents should be converted to the generic elements in the internal tag set to allow the comparison process to be applied. This was not implemented in our prototype, but is a topic for future work. We described how this could be done by identifying textual and graphical elements (e.g. words, lines, images) and determining their role in the structural elements of the document.

A prototype was developed to implement these ideas and assess the viability of our design. We found that the concept of a generic parser was successful for its intended purpose. It identified all tags in the documents, which was the main aim, as well as all of the parameters. However, some of the text was incorrectly identified as tags or parameters. From an analysis of the results, it can be seen that the results for the identification of text vary greatly over the different schemes, which leads us to believe that this is due to the different characteristics of the schemes, rather than the process itself.

The generic comparison process worked well in most cases, correctly identifying over 90% of the differences between the files. However, not all of these differences were errors. Some are necessary as part of the localisation, for example, the deletion of U.S.-specific information. However, many of the discrepancies are due to issues with the markup such as optional tags, the order of tags, different end tags for the same format, different paragraph representations and differing methods of ending tags. For example, if optional tags are used in one document, but omitted from the second, all the optional tags are reported as missing from the second document. Although this causes differences between the files, they are not errors. If the results were adjusted to take account of all such issues, the accuracy of the comparison would be considerably reduced. However, many of these issues were anticipated, but we chose to ignore them partly due to time constraints, but more importantly because of the infrequency of their occurrence. For example, applications are consistent in outputting documents, so tags from the same application will always be in the same order, with the same end tags and the with same treatment for optional tags (e.g. most applications always use optional tags). Nevertheless, these issues should be overcome to develop a more beneficial process, as they are of no concern to the user.

The conversion process was limited by the number of mappings from specific to generic tags stored in the system. The design of the process requires that all of the common formats and elements are covered, but for the purpose of this research only a subset was used and therefore needs to be extended. However, the conversion was successful for all mappings defined in the system. This approach resulted in a comparison of the equivalence of the two documents, as tags with no counterpart in other schemes are ignored. The success of this process is dependent on the careful maintenance of the mappings. The inclusion of tags particular to a scheme introduces unnecessary detail in the comparison, yet if too few tags are included, formats and elements are overlooked.

To summarise, there were many issues that affected the overall performance of the process, most of which were caused by the different features of each markup scheme. To keep the process as generic as possible required omitting all details of any markup scheme. As discussed in Chapter 6, for a process to be successful it needs to understand each scheme it deals with to handle each of its characteristics and process them correctly. Some information about each scheme can be incorporated through the inclusion of text files during processing. This can help to overcome issues caused by optional tags, the order in which tags occur and different methods of ending tags in the same format. However any scheme that requires special processing rather than a knowledge of its characteristics needs a specific tool to perform this. These include problems caused by different representations of paragraph and differing methods of ending tags in different schemes. Therefore there is a trade off between accuracy and generality.

The majority of issues encountered in this research were due to the differences in each markup scheme, and the whole process would obviously benefit from a knowledge of each scheme. However, our aim was to overcome the development of separate tools for each markup scheme by creating a generic comparison process. Therefore this knowledge could not be built into the system. Despite this, it still achieved satisfactory results, illustrating that there are benefits to using generic processes.

However, if a specific parser was introduced to handle the peculiarities of each application area, many of the obstacles previously encountered by the generic process would be eliminated. If this research were to be extended in this way, each file could be output in a standard format (e.g. all elements are marked with separate start and end tags), thereby maintaining the comparison process as generic. Although this is not a truly generic solution, it is still in keeping with the intention of this research, to develop a generic comparison process rather than a wholly generic system for localisation verification. A suite of generic tools could be developed to take the standardised files as input and perform identical processing steps on each. This requires one parser for each markup scheme and a single tool for each process that can work on the output from all of the specific parsers. Therefore, we propose that generic tools are a viable and beneficial option when used in conjunction with tools specific to the application area.

7.1 Future Work

Expansion of the Generic Tag Set: The expansion of the generic tag set is necessary to include other text-based markup schemes and to incorporate elements and formats such as:

- tables
- frames
- cross references
- page breaks
- section breaks
- columns
- document headers
- document footers
- footnotes
- endnotes
- colours
- borders and shading
- drawing objects (e.g. lines, text boxes)
- automatic heading numbering

Comparison of Specific and Generalised Markup: A process was described in Chapter 4 in which the textual elements of the document are identified and examined to determine their role in the structure of the document. It is proposed that this is implement to allow the comparison of documents of differing markup categories.

Recognition of Elements in Generalised Markup: An automated process could be used to examine the content allowed in each element of the document to determine its role. For example, if an element can contain only text, it would be considered part of a paragraph. The process is discussed in chapter 4.

A number of anticipated problems were not addressed in the prototype for our generic comparison process. Solutions for these, as well as the other issues encountered in the assessment of the prototype, were suggested in the results chapter. These are summarised as follows:
Optional Tags: All optional tags for each scheme should be listed in an external file to be included during processing. If the tag considered missing is in this list, the difference is not reported as an error.

Different End Tags for the Same Format: All start tags should be matched with their end tags in an external file to be included when processing schemes that allow different end tags. When a difference is encountered, the file is checked and if the erroneous tag is found, the other equivalent end tags are examined to see if replacing the erroneous tag with one of these will solve the problem. If the problem is due to the use of end-group delimiters to end all tags in the group, the end-group delimiter is replaced with the end tags found in this file for any tags opened in the group.

Accented Characters: An external file should also store a list of all accented characters in each scheme. When extra tags are encountered in the translated document, this file can be examined to see if the difference is caused by the inclusion of tags representing an accented character.

The Order in Which Tags Occur: A list distinguishing start tags from end tags, and character formatting from paragraph formatting can be used to identify groups of tags in which rearrangement is allowed. By re-ordering the tags in the preparation tool, no differences caused by the order of the tags can occur in the comparison.

Moved Tags: It was suggested that the process is extended to examine the documents after comparison to find groups of unmatched tags. If a group in one document could be matched to another group in the second document, it can be assumed that these tags were moved.

Different Paragraph Representations: If a paragraph is represented as separate lines in a scheme, the lines can be merged into a single paragraph by ignoring all NOMATCH tags between the TEXT tags, and considering a group of TEXT tags as a single entity. A more accurate solution requires separate processing.

Differing Methods of Ending Tags: Separate tools are required to identify the end tag in each scheme and output a generic equivalent. This problem cannot be solved simply by providing extra information through a file, but requires a different process for each scheme.

The Storage of Measurements: A list of all tags that have numeric parameters representing measurement, and the units in which these measurements are stored, would allow all measurements to be converted to the unit of measurement used by the system. For example, if a parameter represents a measurement in inches, it can be converted to centimetres, if that was the unit chosen for the system.

It is recommended that the solutions for issues caused by specific markup schemes are implemented in the data preparation tool as we proposed that all characteristics of the schemes are removed in the pre-processing stage to allow the comparison to remain generic.

Bibliography

[ADOB95] ***On-line MIF Reference manual*** (shipped with FrameMaker 5.1), Adobe Systems Inc., 1995

[APPL94] ***Introducing SGML, the Word Processing Standard for the 90's***, AppleSeeds (Newsletter of the Society of Technical Communication, New York Metro Chapter), 1994

[BOVE87] ***The Art of Desktop Publishing: Using Personal Computers to Publish it Yourself***, T. Bove, C. Rhodes, W. Thomas, Bantam Books, New York, 1987.

[BSM96] ***Living with the Guidelines : An Introduction to TEI Tagging***, Lou Burnard & C. M. Sperberg-McQueen (http://www.lib.virginia.edu/)

[CRD87] ***Markup Systems and the Future of Scholarly Text Processing***, James H. Coombs, Allen H. Renear, Steven J. DeRose. Communications of the ACM, vol. 30, no. 11, 1987. p933 - 47 (http://www.sil.org/)

[DIGI91] ***Digital Guide to Developing International Software***, Digital Equipment Corporation, Digital Press, Massachusetts, 1991

[DILL97] ***Markup Languages: LaTeX, SGML, HTML***, George Dillon, http://weber.u.washington.edu/~dillon/netbook/markup/node2.html

[DRAK94] ***From Text to Hypertext: a Post-Hoc Rationalisation of LaTeX2HTML***, Nikos Drakos, Computer Networks and ISDN Systems (Special Issues) Vol. 27, no. 2, in: *Selected Papers of the First World-Wide Web Conference, Geneva, Switzerland, 25-27 May 1994*, page 215-224, 1994

[ENL96] ***SGML Introduction*** *(ENL210E Introduction to SGML)*, Technical Writing course "English 210E", English Department of the University of Waterloo, 1995. (http://watarts.uwaterloo.ca/ENGL/courses/engl210e/)

[HOWE97] ***Free On-line Dictionary of Computing***, D. Howe, 1997, http://wombat.doc.ic.ac.uk/foldoc/index.html

[GOLD90] ***The SGML Handbook***, Charles F. Goldfarb, Oxford University Press, New York, 1990

[GRAH96] ***The HTML Sourcebook***, Ian S. Graham, 2nd Ed., Wiley Computer Publishing, New York, 1996

[GURG90] ***Mastering PageMaker***, G. Keith Gurganus, Blue Ridge Summit PA, Windcrest, 1990

[HAND90] ***ADAPT - Automated Document Analysis Processing and Tagging***, John Handley, Stuart Weibel, in: *EP90 - Proceedings of the International Conference on Electronic Publishing, Document Manipulation & Typography, Maryland*, The Cambridge Series on Electronic Publishing, Cambridge University Press, New York, 1990, p183 - 194

[HARS96] ***Organizing Babylon***, Adele Hars, BYTE Magazine, March 1996, Vol. 21, no. 3.

[HEAR97] ***A Generic Quality Assurance Tool for Windows Help Systems***, Gary Hearne, School of Computer Applications, Dublin City University, 1997

[HECK78] ***A Technique for Isolating Differences Between Files***, Paul Heckel, Communications of the ACM, Apr 1978, Vol. 21, No. 4, p 264

[HERW90] ***Practical SGML***, Eric van Herwijnen, Kluwer Academic Publishers, Dordrecht, 1990

[INTE94] ***The SGML Guide***, Interleaf Inc., 1994 (http://www.ileaf.com/)

[KANO95] ***Developing International Software for Microsoft® Windows® 95 and Windows NT®***, Second Edition, Nadine Kano, 1995

[KAY94] ***Software goes Global***, Russell Kay, BYTE Magazine, Vol. 19, No 6, June 1994.

[LIND89], ***Literate programming: A file difference program***, Donald Lindsay, Communications of the ACM, June 89, Vol 32, no. 6, p740-755

[LOCA97] ***Localisation Ireland*** Newsletter, Localisation Resources Centre, U.C.D., Volume 1, Issue 2, June 1997.

[LOTUS95] *Lotus Notes* © information and help databases in Lotus Development Ireland Ltd.

[MACK93] ***Comparing and Merging Files***, D. MacKenzie, P. Eggert, R. Stallman, http://math.unice.fr/laboratoire/help/info/diff/diff_toc.html, September 1993

[MARC96] ***An Introduction to SGML***, Benoit Marchal, 1996, http://www.brainlink.com/~ben/

[MCD95] ***Multimedia Development,*** Marion McDonald, in the Proceedings of SLIG '95 Conference, LRC, UCD, Oct 1995.

[MICR95] ***Rich Text Format (RTF) Specification and Sample RTF Reader Program***, Microsoft Product Support Services Application Note for RTF version 1.4, http://www.microsoft.com, 1995

[MILL94] ***Transborder Tips and Traps***, L. Chris Miller, BYTE Magazine, Vol. 19, No 6, June 1994, p93-102.

[NCC87] ***The NCC Interconnecting Applications Handbook - Desktop Publishing***, National Centre for Information Technology, Manchester, 1987

[OPEN96] ***Standard Generalized Markup Language***, SGML Open Consortium Home Page, http://www.sgmlopen.org/

[OVUM95] ***Globalisation - Creating New Markets With Translation Technology,*** Rose Lockwood, Jean Leston, Laurent Lachal, Ovum Reports, Ovum Ltd., 1995

[PORT92] ***Document Reconstruction: A System for Recovering Document Structure from Layout***, Gilbert B. Porter, Emil V. Rainero, in: *EP92 - Proceedings of Electronic Publishing Document Manipulation & Typography, Maryland*, The Cambridge Series on Electronic Publishing, Cambridge University Press, 1990, p17 - 30

[QUIN90] ***Towards Document Engineering***, V. Quint, M. Nanard, J. Andre, in: *EP90 - Proceedings of the International Conference on Electronic Publishing, Document Manipulation & Typography, Maryland*, The

Cambridge Series on Electronic Publishing, Cambridge University Press, 1990, p17 - 30

[RAGG95] ***HyperText Markup Language Specification Version 3.0 INTERNET DRAFT,*** Dave Raggett, http://sparc2.lib.cuhk.edu.hk/~ernest/info/html3/, 1995

[SMAR95] ***Manual for Tex2RTF 1.52***, Julian Smart, Artificial Intelligence Applications Institute, University of Edinburgh, October 1995, http://www.calband.berkeley.edu/calchart/docs/tex2rtf_contents.html

[SOFT96] ***Software Localisation***, Quarterly Newsletter of the Localisation Resources Centre, University College Dublin, Volume 1, No. 2, October 96

[STEP94] ***Converting Formatted Documents to HTML***, Jon Stephenson von Tetzchner, presented at *WWW94: First World-Wide Web Conference, Geneva, Switzerland, 25-27 May 1994*, http://pigeon.elsevier.nl/cgi-bin/WWW94link/28/overview, 1994

[SUNS97] ***Diffutils 2.7***, GNU Diff freeware (written by M. Haertel, D. Hayes, R. Stallman, L. Tower, P. Eggert), Solaris Freeware, SunSITE Ireland, http://sunsite.compapp.dcu.ie/solaris-freeware/solaris_2.5.html

[TANA94] ***DTP Software: The Right Tool But for What?***, David Tanaka, (http://www.tcp.ca/Nov94)

[TEIP96] ***A Gentle Introduction to SGML,*** Edited by C. M. Sperberg-McQueen & Lou Burnard (http://info.ox.ac.uk/~archive/teip3sg/)

[TIMM96] ***The Localisation Industry in Ireland and the Issues it Encounters***, M. Timmons, G. Hearne, A. Way, M. Roantree, School of Computer Applications Working Paper Series: CA-0896, Dublin City University, 1996

[USER95] ***SGML Users' Group History***, International SGML Users' Group, 1995, http://www.sil.org/

[WATS92] ***Brief History of Document Markup***, Dennis G. Watson, University of Florida, Circular 1086, November 1992 (http://hammock.ifas.ufl.edu/txt/fairs/ae/)

[ZHEN92] ***"A Document Composition Environment for Multi-Language Processing"***, Min Zheng, in: *EP92 - Proceedings of Electronic Publishing 1992*, Cambridge University Press, Cambridge, 1992, p43-52

Company Home Page Web References

[CLOC97] Clockworks Multimedia, *http://www.clockworksmultimedia.com/*

[DLG96] DLG Software Services, *http://ireland.iol.ie/~dlg/*

[GECAP96] GECAP, *http://www.gecap.de/homepage.htm*

[ITP96] International Translation and Publishing Ltd, *http://www.itp.ie/*

[LRC97] Localisation Resources Centre, *http://lrc.ucd.ie/*

[POLY96] Polylang Limited, *http://www.polylang.com/*

[SYM96] Symantec Corporation, *http://www.symantec.com/*

Appendix A

This appendix contains the tag mappings file, tagmap.ini that was used in the system.

MIF:

```
[MIF]
STARTTAG                <
ENDTAG                  >
PARAGRAPH               Para
PARAEND
BOLD                    FWeight 'Bold'
BOLDOFF                 FWeight 'Regular'
ITALIC                  FAngle 'Italic'
ITALICOFF               FAngle 'Regular'
UNDERLINE               FUnderlining FSingle
UNDERLINEOFF            FUnderlining FNoUnderlining
DOUBLEULINE             FUnderlining FDouble
DOUBLEULINEOFF          FUnderlining FNoUnderlining
STRIKETHRU              FStrike Yes
STRIKEOFF               FStrike No
EMPHASIS                FTag 'Emphasis'
FIRSTINDENT             PgfFIndent
LEFTINDENT              PgfLIndent
RIGHTINDENT             PgfRIndent
LEFTALIGN               PgfAlignment Left
RIGHTALIGN              PgfAlignment Right
JUSTIFY                 PgfAlignment LeftRight
CENTRE                  PgfAlignment Center
FONTTYPE                FFamily
FONTSIZE                FSize
FONTCOLOUR              FColor
SUPERSCRIPT             FPosition FSuperscript
SUPEROFF                FPosition FNormal
SUBSCRIPT               FPosition FSubscript
SUBOFF                  FPosition FNormal
TEXT                    String
LINE
LISTITEM
ENDLISTITEM
```

RTF:

```
[RTF]
STARTTAG                \
ENDTAG
PARAGRAPH               par
PARAEND
BOLD                    b
BOLDOFF                 b0
ITALIC                  i
ITALICOFF               i0
UNDERLINE               ul
DOUBLEULINE             uldb
UNDERLINEOFF            ul0
UNDERLINEOFF            ulnone
STRIKETHRU              strike
STRIKEOFF
EMPHASIS
FIRSTINDENT             fi?
LEFTINDENT              li?
RIGHTINDENT             ri?
LEFTALIGN               ql
RIGHTALIGN              qr
```

	JUSTIFY	qj
	CENTRE	qc
	FONTTYPE	
	FONTSIZE	
	FONTCOLOUR	col?
	SUPERSCRIPT	super
	SUPEROFF	nosupersub
	SUBSCRIPT	sub
	SUBOFF	nosupersub
	LINE	
	STARTLIST	
	ENDLIST	
	LISTITEM	pntext
	ENDLISTITEM	
	TEXT	TEXT
HTML:	[HTM]	
	STARTTAG	<
	ENDTAG	>
	PARAGRAPH	P
	PARAEND	/P
	BOLD	B
	BOLDOFF	/B
	ITALIC	I
	ITALICOFF	/I
	UNDERLINE	U
	UNDERLINEOFF	/U
	DOUBLEULINE	
	DOUBLEULINEOFF	
	STRIKETHRU	STRIKE
	STRIKEOFF	/STRIKE
	EMPHASIS	EM
	EMPHASISOFF	/EM
	FIRSTINDENT	
	LEFTINDENT	
	RIGHTINDENT	
	LEFTALIGN	
	RIGHTALIGN	
	JUSTIFY	
	CENTRE	CENTER
	CENTREOFF	\CENTER
	FONTTYPE	
	FONTSIZE	FONT size
	FONTCOLOUR	FONT COLOR
	SUPERSCRIPT	
	SUPEROFF	
	SUBSCRIPT	SUB
	SUBOFF	/SUB
	LINE	HR
	STARTLIST	UL
	ENDLIST	/UL
	LISTITEM	LI
	ENDLISTITEM	/LI
	TEXT	TEXT

Appendix B

This appendix contains a full set of files for one of the tests on the system. CSE.HTM is the original HTML file.

```
<HTML> <! Creation 05/01/96>
<HEAD>
<TITLE>Centre for Software Engineering Home Page</TITLE>
</HEAD>
<BODY  BGCOLOR="FFFFCC" TEXT="000000" LINK="0000FF"
VLINK="C40026">
<CENTER>
<TABLE WIDTH=95% CELLSPACING=5>
<TR>
<TD WIDTH=25%><CENTER>
<IMG SRC="cse/gifs/cselogo.gif" HEIGHT="67" WIDTH="126">
<H6>The Irish government designated <BR>
 IT support organisation</H6></CENTER>
</TD>

<TD ROWSPAN=3  VALIGN=MIDDLE> <HR>

<CENTER> <P><H4><I> The Centre for Software Engineering is
committed to raising the standards of quality and productivity
within the software development community, in Ireland and
internationally.</P>

<P>Our goal is to make the most flexible and comprehensive range of partnership
programmes available to software developers.</I></H4></P>
<HR></CENTER>
<CENTER>
<TABLE WIDTH=80%>
<TR>
<TD>
<P> <IMG SRC="cse/gifs/blue.gif" HEIGHT="14" WIDTH="14">
<A HREF="cse/c_whatis.html">About the CSE </A></P>
<P> <IMG SRC="cse/gifs/blue.gif" HEIGHT="14" WIDTH="14">
<A HREF="cse/c_sparea.html">Specialist Areas</A></P>
<P> <IMG SRC="cse/gifs/blue.gif" HEIGHT="14" WIDTH="14">
<A HREF="cse/c_prog.html">Membership Programmes</A></P>
<P> <IMG SRC="cse/gifs/blue.gif" HEIGHT="14" WIDTH="14">
<A HREF="cse/c_service.html">Services</A></P>
</TD>
<TD>
<P> <IMG SRC="cse/gifs/blue.gif" HEIGHT="14" WIDTH="14">
<A HREF="cse/c_proj.html">Projects</A></P>
<P> <IMG SRC="cse/gifs/blue.gif" HEIGHT="14" WIDTH="14">
<A HREF="cse/c_pubs.html">Publications</A></P>
<P> <IMG SRC="cse/gifs/blue.gif" HEIGHT="14" WIDTH="14">
<A HREF="cse/c_staff.html">Staff</A></P>
<P> <IMG SRC="cse/gifs/blue.gif" HEIGHT="14" WIDTH="14">
<A HREF="cse/c_misc.html">Miscellaneous</A></P>
</TD>
</TABLE> <HR>
<I>Last updated 13/08/97</I>
</CENTER>
</TR>

<TR >
```

```
<TD  WIDTH=25%>
<CENTER>
<H6>Centre for Software Engineering Ltd., Dublin City University Campus,  <BR>
Dublin 9, Ireland.
<P>Tel: +353 1 7045750    <BR>
 Fax: +353 1 7045605    </P> </H6>
<P> <A HREF="mailto:admin@cse.dcu.ie"><IMG SRC="cse/gifs/mail.gif"
HEIGHT="40" WIDTH="40" ALIGN=MIDDLE HSPACE=4 VSPACE=4>
</A> <H6> Email admin@cse.dcu.ie</H6></P>
</TD>
</TR>
<TR >
<TD  WIDTH=25%>
<CENTER><H6>To fully view other pages at this site you will need a browser that
supports FRAMES<H6></CENTER>
</TD>
</TR>
</TABLE>
</CENTER>
</BODY>

</HTML>
```

CSE2.HTM is the “translated” file with artificial errors to simulate translation.

```
<HTML> <! Creation 05/01/96>
<! Translated 22/03/96>
<HEAD>
<TITLE>*Centre for Software Engineering Home Page</TITLE>
</HEAD>
<BODY  BGCOLOR="FFFFCC" TEXT="000000" LINK="0000FF"
VLINK="C40026">
<CENTER>
<TABLE WIDTH=95% CELLSPACING=5>
<TR>
<TD WIDTH=25%><CENTER>
<IMG SRC="cse/gifs/cselogo.gif" HEIGHT="90" WIDTH="150">
</CENTER>
<H5>*The Irish government designated <BR>
 *IT support organisation</H5>
</TD>
<TD ROWSPAN=3  VALIGN=MIDDLE> <HR>
<CENTER> <P><H4><I> *The Centre for Software Engineering is
committed to raising the standards of quality and productivity
within the software development community, in Ireland and
internationally.</I></P>
<P>*Our goal is to make the most flexible and comprehensive range of partnership
programmes available to software developers.</H4></P>
<HR></CENTER>
<CENTER>
<TABLE WIDTH=80%>
<TR>
<TD>
<P> <IMG SRC="/gifs/orange.gif" HEIGHT="14" WIDTH="14">
<A HREF="cse/c_about.html">*About the CSE </A></P>
<P> <IMG SRC="/gifs/orange.gif" HEIGHT="14" WIDTH="14">
<A HREF="cse/c_sparea.html">*Specialist Areas</A></P>
<P> <IMG SRC="/gifs/orange.gif" HEIGHT="14" WIDTH="14">
<A HREF="cse/c_prog.html">*Membership Programmes/A></P>
<P> <IMG SRC="/gifs/orange.gif" HEIGHT="14" WIDTH="14">
<A HREF="cse/c_service.html">*Services</A></P>
</TD>
<TD>
<P> <IMG SRC="/gifs/orange.gif" HEIGHT="14" WIDTH="14">
<A HREF="cse/c_proj.html">*Projects</A></P>
<P> <IMG SRC="/gifs/orange.gif" HEIGHT="14" WIDTH="14">
<A HREF"cse/c_staff.html">*Staff</A></P>
<P> <IMG SRC="/gifs/orange.gif" HEIGHT="14" WIDTH="14">
<A HREF="cse/c_misc.html">*Miscellaneous</A></P>
</TD>
</TABLE> <HR>
<B>*Last updated 12/05/97</B>
</CENTER>
</TR>
<TR>
<TD  WIDTH=25%>
<CENTER>
<H6>*Centre for Software Engineering Ltd., Dublin City University Campus,  <BR>
*Dublin 9, Ireland.
<P>*Tel: +353 1 7045750    <BR>
 Fax: +353 1 7045605    </P> </H6>
```

```
<P> <A HREF="mailto:admin@cse.dcu.ie"><IMG SRC="cse/gifs/mail2.gif"
HEIGHT="40" WIDTH="40" ALIGN=MIDDLE HSPACE=4 VSPACE=4></A> <H5>
Email admin@cse.dcu.ie</H5></P>
</TD>
</TR>
<TR >
<TD  WIDTH=25%>
</TD>
</TR>
</TABLE>
</CENTER>
</BODY>
</HTML>
```

CSE.HTMX is the output files from the generic parser for CSE.HTM.

```
HTML
! Creation 05/01/96
HEAD
TITLE
TEXT Centre for Software Engineering Home Page
/TITLE
/HEAD
BODY BGCOLOR="FFFFCC" TEXT="000000" LINK="0000FF" VLINK="C40026"
CENTER
TABLE WIDTH=95% CELLSPACING=5
TR
TD WIDTH=25%
CENTER
IMG SRC="cse/gifs/cselogo.gif" HEIGHT="67" WIDTH="126"
H6
TEXT The Irish government designated
BR
TEXT IT support organisation
/H6
/CENTER
/TD
TD ROWSPAN=3  VALIGN=MIDDLE
HR
CENTER
P
H4
I
TEXT The Centre for Software Engineering is  committed to raising the standards of
/P
P
TEXT Our goal is to make the most flexible and comprehensive range of productivit
/I
/H4
/P
HR
/CENTER
CENTER
TABLE WIDTH=80%
TR
TD
P
IMG SRC="cse/gifs/blue.gif" HEIGHT="14" WIDTH="14"
A HREF="cse/c_whatis.html"
TEXT About the CSE
/A
/P
P
IMG SRC="cse/gifs/blue.gif" HEIGHT="14" WIDTH="14"
A HREF="cse/c_sparea.html"
TEXT Specialist Areas
/A
/P
P
IMG SRC="cse/gifs/blue.gif" HEIGHT="14" WIDTH="14"
A HREF="cse/c_prog.html"
TEXT Membership Programmes
/A
/P
```

```
P
IMG SRC="cse/gifs/blue.gif" HEIGHT="14" WIDTH="14"
A HREF="cse/c_service.html"
TEXT Services
/A
/P
/TD
TD
P
IMG SRC="cse/gifs/blue.gif" HEIGHT="14" WIDTH="14"
A HREF="cse/c_proj.html"
TEXT Projects
/A
/P
P
IMG SRC="cse/gifs/blue.gif" HEIGHT="14" WIDTH="14"
A HREF="cse/c_pubs.html"
TEXT Publications
/A
/P
P
IMG SRC="cse/gifs/blue.gif" HEIGHT="14" WIDTH="14"
A HREF="cse/c_staff.html"
TEXT Staff
/A
/P
P
IMG SRC="cse/gifs/blue.gif" HEIGHT="14" WIDTH="14"
A HREF="cse/c_misc.html"
TEXT Miscellaneous
/A
/P
/TD
/TABLE
HR
I
TEXT Last updated 13/08/97
/I
/CENTER
/TR
TR
TD WIDTH=25%
CENTER
H6
TEXT Centre for Software Engineering Ltd., Dublin City University Campus
BR
TEXT Dublin 9, Ireland.
P
TEXT Tel: +353 1 7045750
BR
TEXT Fax: +353 1 7045605
/P
/H6
P
A HREF="mailto:admin@cse.dcu.ie"
IMG    SRC="cse/gifs/mail.gif"    HEIGHT="40"    WIDTH="40"    ALIGN=MIDDLE
HSPACE=4 VSPACE=4
/A
H6
TEXT Email admin@cse.dcu.ie
```

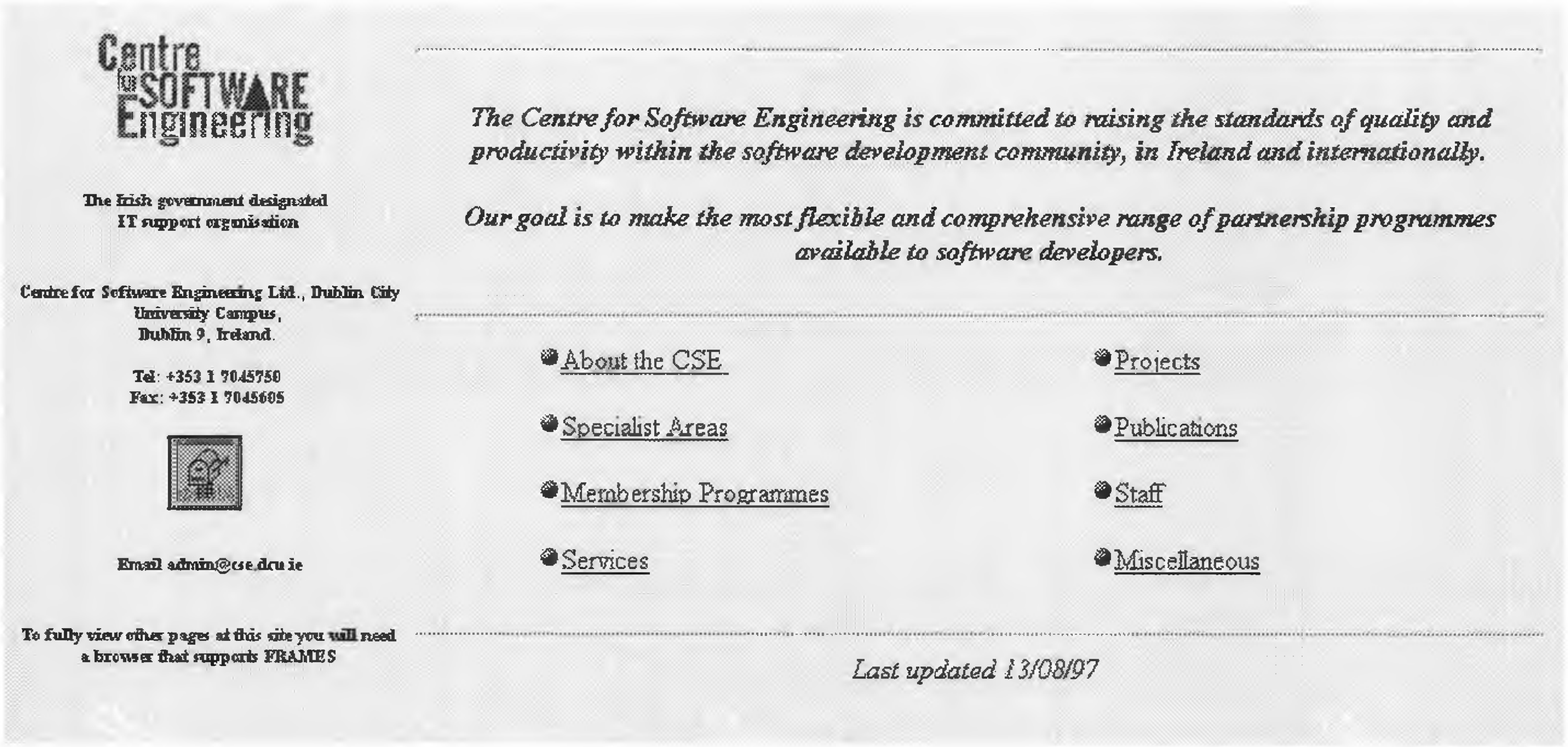
```
/H6
/P
/TD
/TR
TR
TD WIDTH=25%
CENTER
H6
TEXT To fully view other pages at this site you will need a browser that supports FR
H6
/CENTER
/TD
/TR
/TABLE
/CENTER
/BODY
/HTML
```

CSE2.HTMX is the output files from the generic parser for CSE2.HTM.

```
HTML
! Creation 05/01/96
! Translated 22/03/96
HEAD
TITLE
TEXT *Centre for Software Engineering Home Page
/TITLE
/HEAD
BODY BGCOLOR="FFFFCC" TEXT="000000" LINK="0000FF" VLINK="C40026"
CENTER
TABLE WIDTH=95% CELLSPACING=5
TR
TD WIDTH=25%
CENTER
IMG SRC="cse/gifs/cselogo.gif" HEIGHT="90" WIDTH="150"
/CENTER
H5
TEXT *The Irish government designated
BR
TEXT *IT support organisation
/H5
/TD
TD ROWSPAN=3  VALIGN=MIDDLE
HR
CENTER
P
H4
I
TEXT *The Centre for Software Engineering is  committed to raising the standards o
/I
/P
P
TEXT *Our goal is to make the most flexible and comprehensive range of partnersh
/H4
/P
HR
/CENTER
CENTER
TABLE WIDTH=80%
TR
TD
P
IMG SRC="/gifs/orange.gif" HEIGHT="14" WIDTH="14"
A HREF="cse/c_about.html"
TEXT *About the CSE
/A
/P
P
IMG SRC="/gifs/orange.gif" HEIGHT="14" WIDTH="14"
A HREF="cse/c_sparea.html"
TEXT *Specialist Areas
/A
/P
P
IMG SRC="/gifs/orange.gif" HEIGHT="14" WIDTH="14"
A HREF="cse/c_prog.html"
```

```
/P
P
IMG SRC="/gifs/orange.gif" HEIGHT="14" WIDTH="14"
A HREF="cse/c_service.html"
TEXT *Services
/A
/P
/TD
TD
P
IMG SRC="/gifs/orange.gif" HEIGHT="14" WIDTH="14"
A HREF="cse/c_proj.html"
TEXT *Projects
/A
/P
P
IMG SRC="/gifs/orange.gif" HEIGHT="14" WIDTH="14"
A HREF"cse/c_staff.html"
TEXT *Staff
/A
/P
P
IMG SRC="/gifs/orange.gif" HEIGHT="14" WIDTH="14"
A HREF="cse/c_misc.html"
TEXT *Miscellaneous
/A
/P
/TD
/TABLE
HR
B
TEXT *Last updated 12/05/97
/B
/CENTER
/TR
TR
TD WIDTH=25%
CENTER
H6
TEXT *Centre for Software Engineering Ltd., Dublin City University Campus,
BR
TEXT *Dublin 9, Ireland.
P
TEXT *Tel: +353 1 7045750
BR
TEXT Fax: +353 1 7045605
/P
/H6
P
A HREF="mailto:admin@cse.dcu.ie"
IMG   SRC="cse/gifs/mail2.gif"   HEIGHT="40"   WIDTH="40"   ALIGN=MIDDLE
    HSPACE=4 VSPACE=4
/A
H5
TEXT Email admin@cse.dcu.ie
/H5
/P
/TD
/TR
TR
```

```
TD WIDTH=25%
/TD
/TR
/TABLE
/CENTER
/BODY
/HTML
```


Centre for SOFTWARE Engineering

The Irish government designated IT support organisation

Centre for Software Engineering Ltd., Dublin City University Campus, Dublin 9, Ireland.

Tel: +353 1 7045750
Fax: +353 1 7045605

Email admin@cse.dcu.ie

To fully view other pages at this site you will need a browser that supports FRAMES

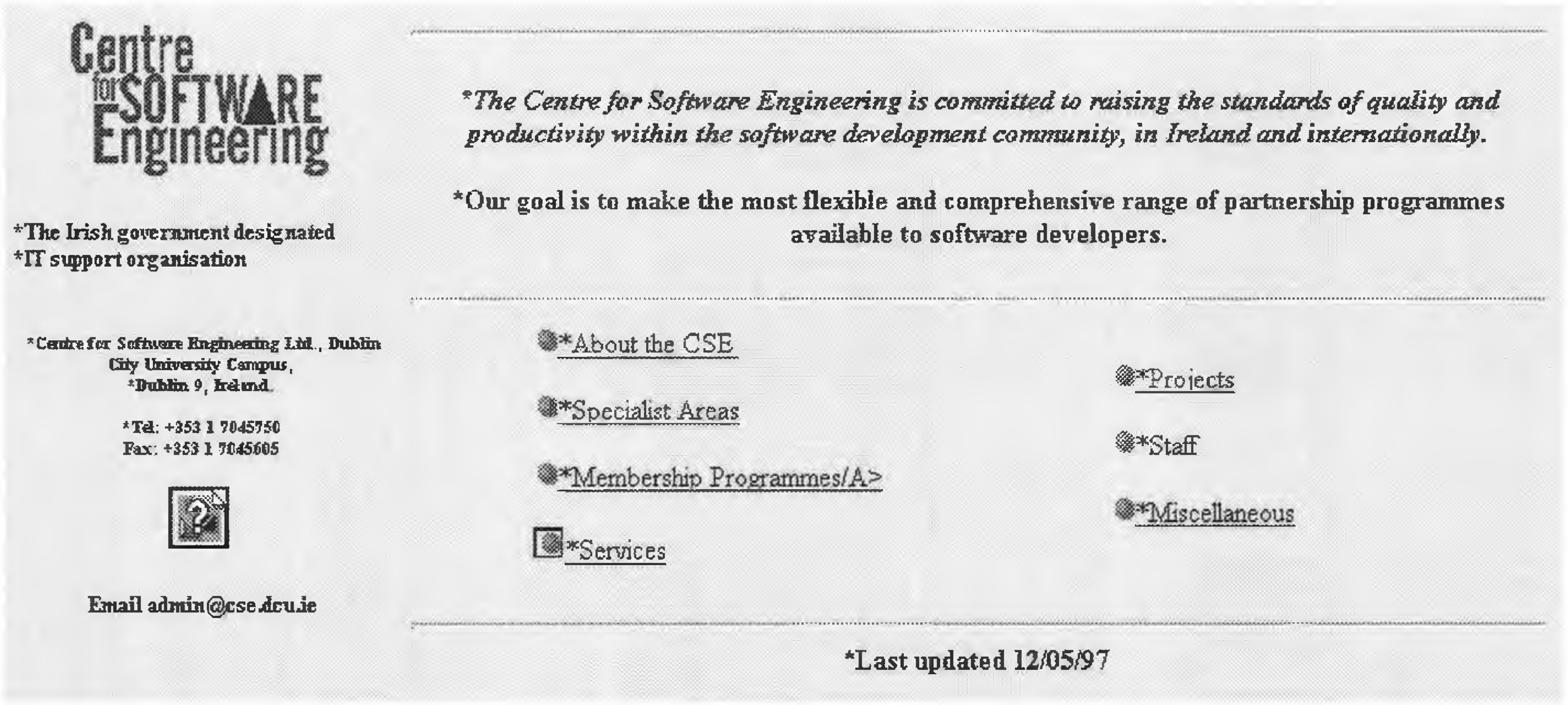
The Centre for Software Engineering is committed to raising the standards of quality and productivity within the software development community, in Ireland and internationally.

Our goal is to make the most flexible and comprehensive range of partnership programmes available to software developers.

- About the CSE
- Specialist Areas
- Membership Programmes
- Services
- Projects
- Publications
- Staff
- Miscellaneous

Last updated 13/08/97

Figure 1 This is an Image of the CSE.HTM File Opened in Netscape



Centre for SOFTWARE Engineering

*The Irish government designated
*IT support organisation

*Centre for Software Engineering Ltd., Dublin City University Campus,
*Dublin 9, Ireland.

*Tel: +353 1 7045750
Fax: +353 1 7045605

Email admin@cse.dcu.ie

*The Centre for Software Engineering is committed to raising the standards of quality and productivity within the software development community, in Ireland and internationally.

*Our goal is to make the most flexible and comprehensive range of partnership programmes available to software developers.

*About the CSE

*Specialist Areas

*Membership Programmes/A>

*Services

*Projects

*Staff

*Miscellaneous

*Last updated 12/05/97

Figure 2 This is an Image of the "Translated" CSE2.HTM File Opened in Netscape (with a "*" used to simulate translation of a string, as described in Chapter 6).