It's time to bring word processing out of the dark ages: A position paper

Tamir Hassan

Abstract

This paper presents a view that word processing (and *document authoring* in general) today is very inefficient, and that this problem is mainly due to inadequately designed software whose approach reflects too much on its historical development, showing little consideration for HCI principles and how users actually interact with it. A number of profound improvements are proposed, which aim to offer the reader a fresh look at how document authoring systems of the future could function. In Part 1, three main types of document authoring packages are analysed; word processors, DTP packages and LaTeX; and suggestions are proposed to reunite the benefits of all three. Part 2 deals with document authoring in a wider context, the GUI desktop, and suggestions are proposed to improve interaction between applications and themselves, and the operating system. Part 3 is concerned with a number of remaining issues pertaining to application interaction, which don't directly concern HCI design.

Part 1: Document authoring software

Nowadays, people take word processors for granted. They spend many hours working with them—often struggling with them—but generally accept the situation as it is, because they do not believe that it could get any better. After all, word processing software hasn't changed very much over the last decade, and *Microsoft Word* is still the market leader with millions of users all over the globe, so the current formula must be right, mustn't it? Yet anyone who has ever had to prepare a long or complex document will know what a struggle this can be.

This paper shows that there are indeed many improvements that can be made to the present "state of the art" in document preparation software. These improvements are proposed by way of open-ended questions, together with examples of how these questions may be addressed. This is indeed a complex situation, whose approach requires a thorough investigation of HCI (human computer interaction) issues and empathy on the user's part. History has shown us that any "quick fix"

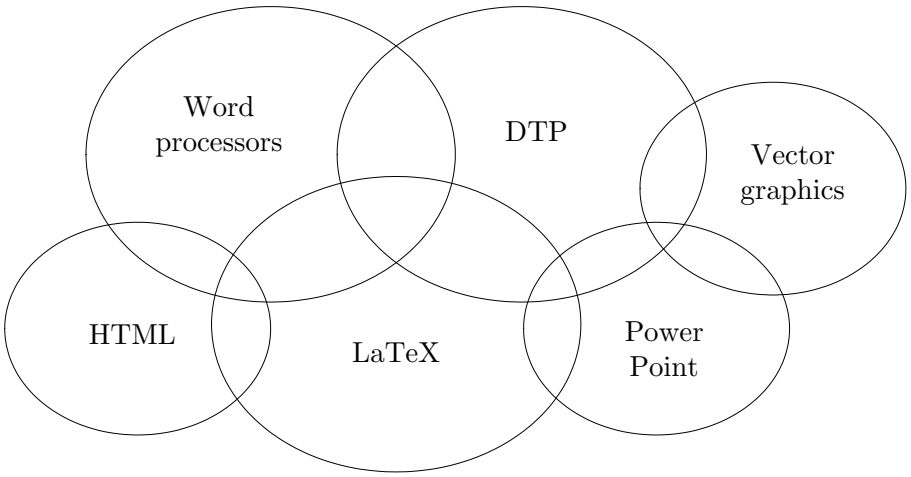

Figure 1: A diagram showing the common types of document authoring software, and how their functionalities overlap

solutions, such as those often brought out by commercial vendors, only serve to cause a hindrance in the long term. Rather, we need to make a number of profound changes that have been fully researched and thought out. To begin with, we need to take our heads out of the sand and change the way we *think* about document authoring, instead of continuing to accept the established norms without question.

Before we start, just a quick note about the term *document authoring*. In most word-processing tasks today, the same user not only writes the words, but actually takes control over the whole process including formatting and page layout, right up to when the finished product comes out of the printer. So the term document authoring refers to the whole process, considering each of these tasks on an equal footing. Secondly, many programs, which are not markted explicitly as word processors, can also be used to create documents from start to finish, or by interaction with other programs. This also falls under document authoring. The diagram in Fig. 1 shows several common *document authoring application paradigms* and how their functionalities overlap. This will be described in more detail in the next section.

Word processor or desktop publisher (DTP)?

Most current off-the-shelf desktop publishing software falls into one of two groups: *word processors*, which focus more on the textual content, and *desktop publishers*, where the focus is geared towards the layout and appearance of the page. Before the advent of personal computing, these were indeed separate tasks, each undertaken by specialists in their field. But, as soon as GUIs moved into the

mainstream, word processors began to extend their range of formatting features, many of which came from the DTP arena.

So now we have two distinct types of product, word processors and desktop publishers, which provide overlapping functionality but are inherently aimed at different parts of the document creation process. Some might disagree with me, but I believe this not to be a good situation at all. A user often has to ask himself the question: "Should I move all this stuff into a DTP package, or does this word processor let me do all that I need to?" And the requisite functions will most likely be duplicated in both DTP and word processor but be controlled by totally different user interfaces, each geared to the focus of the particular program. The user may well find that a word processing package enables him to draw ruling lines on the page (such as the example in Figure 2), but may not give him the fine control that he requires, or may be very tedious to use.

The obvious way to remove this redundancy would be to **design a document preparation system which is equally good at both drafting and layout**. This is the first of the suggestions for improvement in this paper. A starting point would perhaps be to combine elements of an existing word processing product with those of an existing DTP product, and rework the user interfaces so that a smooth transition is possible between them. This is indeed a challenging task in HCI design, particularly if we also want the result to present a shallow learning curve to users of existing DTP and word processing packages.

WYSIWYG (What-You-See-Is-What-You-Get) vs. markup (or code)

A small number of users have jumped off the word-processing bandwagon and have sought solace in markup-based systems such as LaTeX. LaTeX, in particular, has found a niche in the typesetting of texts in mathematics and computer science; the former being due to its superior typesetting of mathematical formulae and the latter being due to its markup language, which presents a relatively shallow learning curve to computer scientists. Theoretically, when one is not distracted by the coding aspects, a markup language enables the writer to focus on the content without being distracted by formatting issues. This approach makes LaTeX particularly suitable for longer documents; its intelligent layout engine positions text and figures automatically throughout the document. But there are a number of significant obstacles to the uptake of LaTeX: First, although the layout engine is intelligent, trying to deviate from one of the predefined layouts is very difficult. And secondly, most people—myself included—find it difficult to concentrate on writing when looking at source code.

As someone who grew up with the likes of Microsoft Word, I actually need a somewhat WYSIWYG display while I am drafting. I like to see my headings and tables more or less like they will look in the final version. This enables me to get a

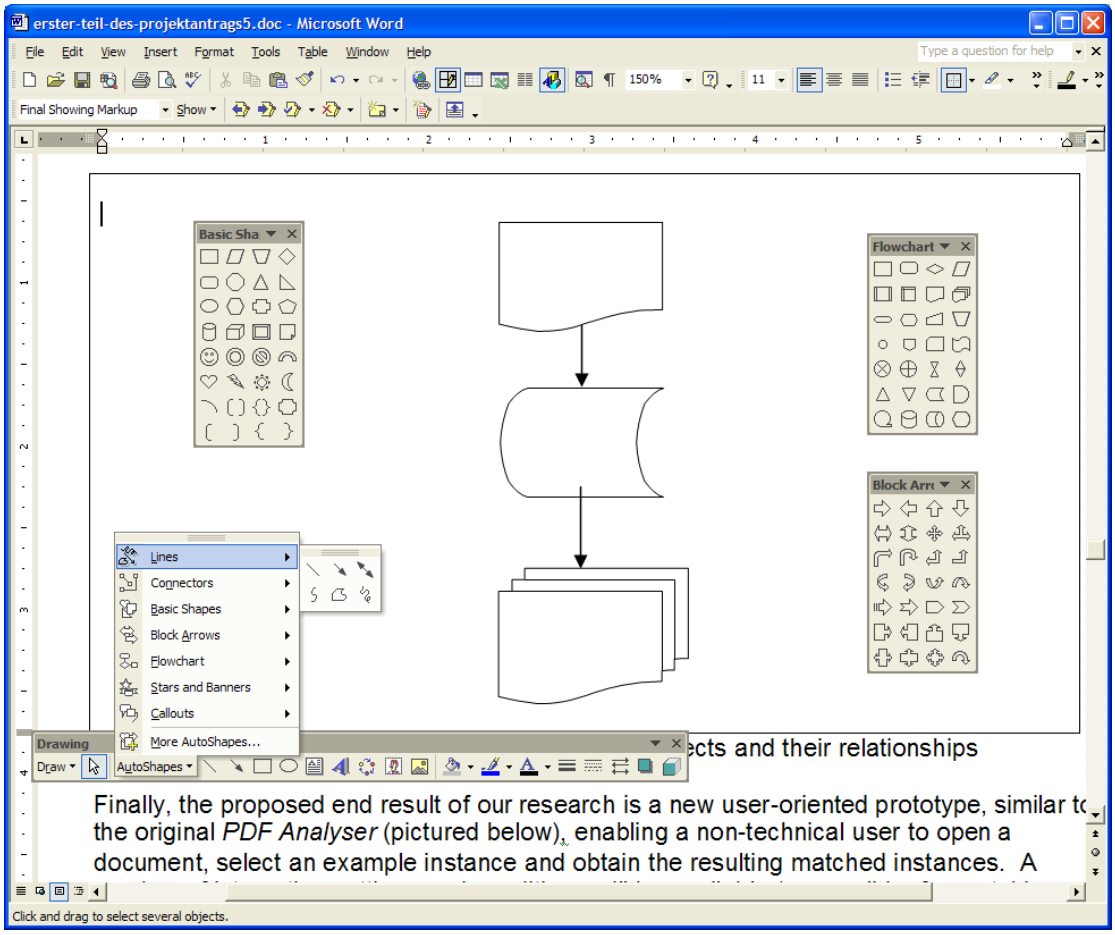

Figure 2: Example of Word's graphics tools—comprehensive, but neither suitable for DTP nor for diagramming purposes

good overview of the document that I am working on; of its outline structure and of its current length. When working directly on LaTeX source code, I find that I often get "lost" in the document, and need to keep a constantly updated preview side-by-side. There are a number of GUIs for LaTeX which aim to provide a semi-WYSIWYG interface, such as LyX[1] and TeXmacs[2]. Another option is the use of drafting tools, such as *Mellel*[3] and *Scrivener*[4], which include export functionality to LaTeX. But neither method offers true WYSIWYG, and therefore not practical for more complex layouts or for documents incorporating graphics.

The second suggestion of this paper is therefore to **integrate LaTeX into a fully WYSIWYG graphical user interface**. The next section goes into more detail about LaTeX, and the issues that must be faced when designing such a GUI.

The "flexible layout" concept

Let's go back to the layout engine mentioned in the last section. It really is intelligent. What sets LaTeX apart from other software, including today's DTP, word processing and graphics packages, is that the page layout is *flexible*; many of the layout attributes—character and line spacings, hyphenations, positions of figures, etc.—are dynamic and are algorithmically determined to produce the best result for the given content.

To take a simple example, in most word processing and DTP packages, when images are placed within the flow of text and then become wrapped to the next page because of lack of space, an ugly gap is left on the bottom of the page, which must then manually be adjusted by the user. In LaTeX, this adjustment takes place automatically. Depending on the content, the line spacing of the text in the preceding pages may be slightly adjusted or the positioning of the figures altered so that the gap does not occur. Of all the possible adjustments, the layout engine finds the one that is the least perceptible to the user. In mathematical terms, whilst current DTP and word processing packages find the local maximum by obeying simple rules that wrap individual words, lines or figures, the LaTeX layout engine looks for the global maximum[5], changing the layout significantly if necessary. In a nutshell, this is what will be referred to as the *"flexible layout" concept* in the rest of this paper.

Unfortunately, LaTeX does not always make the best decisions with regard to the author's intentions. This can either occur because the template parameters

[1] LyX, `http://www.lyx.org`

[2] TeXmacs, `http://www.texmacs.org`

[3] Mellel, `http://www.redlers.com/mellel.html`

[4] Scrivener, `http://www.literatureandlatte.com/scrivener.html`

[5] This is actually a simplification; in certain cases, even LaTeX does not find the best possible layout due to limitations in the search algorithm that it uses.

have not been specified correctly (a common example being the default behaviour of overfilling columns rather than underfilling them[6]), or because the user would like to customize the layout and deviate from the standard template design. These cases stem from poor user interface design: if the default values are not displayed to the user, how can he know that they even exist, let alone change them?

The LaTeX (and TeX) system, originally developed in the '70s, was way ahead of its time regarding the typesetting quality of its documents. Now, in 2007, it still is! The trouble is, the quality of competing systems, in particular off-the-shelf DTP and word processing packages, is nowadays deemed by most to be "good enough". The biggest obstacle in the uptake of LaTeX is therefore the lack of a fully WYSIWYG GUI. Many proponents of LaTeX even cite this as a "feature, rather than a bug". But a suitable GUI could not only reduce the learning curve for new users, it could also make many common tasks easier and far more intuitive. Such a GUI could offer the user the ability to switch seamlessly between outlining, editing and page layout views, in each of which live editing of the document is possible. During the later stages of document production, the average user is concerned as much about presentation as he is about content. There is nothing wrong with that!

Another benefit of such a GUI is that it could help the user better understand the "flexible layout" concept, as the document will need to be redrawn whenever edits are made. The fact that small edits can sometimes cause significant changes in layout poses a further challenge. Perhaps animation or scrolling could make it clear to the user how the layout has changed.

But what if the user does not like the decisions made by the computer's auto-formatting algorithm and would rather keep certain elements on a certain page, for example? The UI should also enable the user to make changes to the layout and specify additional constraints if necessary. After all, he knows what he wants! If the user does not feel empowered by the software, he will choose something else next time.

Following on from this point, my third suggestion for this paper is to **further develop the "flexible layout" concept to enable interactive manipulation of page layout**. Although LaTeX is focussed on the creation of mainly textual documents of arbitrary length, there are also downloadable templates that provide suitable layouts for letters, presentation slides, research papers, etc. However, such a template has to be manually written, and any adjustments have to be made manually by coding. It is my vision that an intelligent formatting system such as LaTeX could also be employed in the creation of newsletters, brochures and other complex layouts which are currently the realm of DTP software. In order for such

[6] See the section on *line breaks* in
`http://www.economics.utoronto.ca/osborne/latex/PMAKEUP.HTM`

documents to be produced efficiently, this needs to be done visually. For example, while editing a newsletter, the user could simply click on an article and drag it to a different part of the page. The rest of the page would then be automatically reformatted and redrawn so that the existing content fits perfectly again. Thus the tedious work would again be carried out automatically by the computer, and the user would be able to experiment with different page layouts with just a few clicks.

Some years ago, when I worked at a computer retailer, we had to produce monthly price lists from the product database. This was a long, arduous task, as we had to copy each item manually and enter it into a DTP package, ensuring that everything fits on the page at the end. Such a system could also have been very practical here. The layout need only be designed once; the data could then be updated automatically from a database, with little or no user intervention being required. If too much space is remaining, the user could simply add some illustrations with a few clicks.

This brings us to another common problem that I do not believe to be successfully handled by LaTeX or by any other current publishing software at the moment: the need to fit a certain amount of content within a given number of pages or page area. Such situations always result in the user having to manually change the size of graphics, figures, etc. in order to coerce the program to fit all the content into the requisite space. Clearly, a computer program cannot be expected to change the textual content itself, but it should offer a user-friendly and non-tedious approach to balance the various trade-offs involved and perform any textual alterations that may be necessary. The mock-up in Fig. 3 shows how this could be achieved.

The "flexible layout" concept lies at the very heart of my vision of what document preparation software could and should become. I believe that the current crop of open source word processing packages, such as *OpenOffice*[7] *Writer* and *AbiWord*[8], are a missed opportunity. Instead of continuing to produce an inferior imitation of *Microsoft Office*, the development team should make their own innovations and perhaps try addressing some of the issues raised here. Certainly the design of such a system poses many challenges, and requires a fully researched and thought-out solution. But I do believe that such work will ultimately not be in vain; it is time that we break with decades of legacy and finally improve the current "piecemeal" approach that has blighted document preparation with unnecessary redundancy and inefficiency to this day.

[7] OpenOffice, `http://www.openoffice.org`
[8] AbiWord, `http://www.abisource.com`

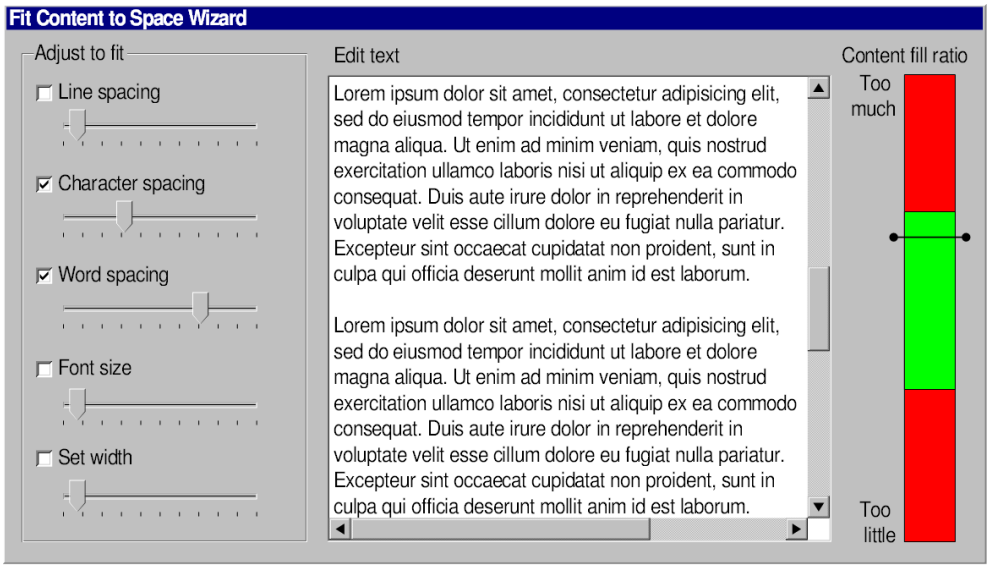

Figure 3: Mock-up of a user interface dialogue to enable efficient adjustment of content to fit a given page area

Part 2: GUI operating systems and "the desktop"

OLE (Object Linking & Embedding)

In Part 1 of this paper we considered the three main types of document authoring software; word processors, DTP packages and LaTeX; and suggested how these could be combined into one complete software package, providing the advantages of all within a consistent user interface, reducing redundancy. In Part 2 we take this idea further, considering the interaction between applications on a more general level, and begin with OLE, a mechanism for increasing this interaction in Microsoft products.

Figure 1 gave an example of the main software products used to create documents nowadays, together with their respective overlapping functionalities. But documents often contain graphics and other objects from a much wider variety of applications, for example bar charts, diagrams and mathematical equations. Some documents—those intended primarily for on-screen viewing—even include sound and video clips. In Windows, OLE enables such *embedded* objects to be edited from within the document itself. When the object is double-clicked, its native application loads in the background and displays a toolbar within the host program's interface, enabling the object to be edited, as shown in Figure 4. When the user clicks outside of the object's bounds, the changes are effected in the hosting document and control is passed back to the hosting application.

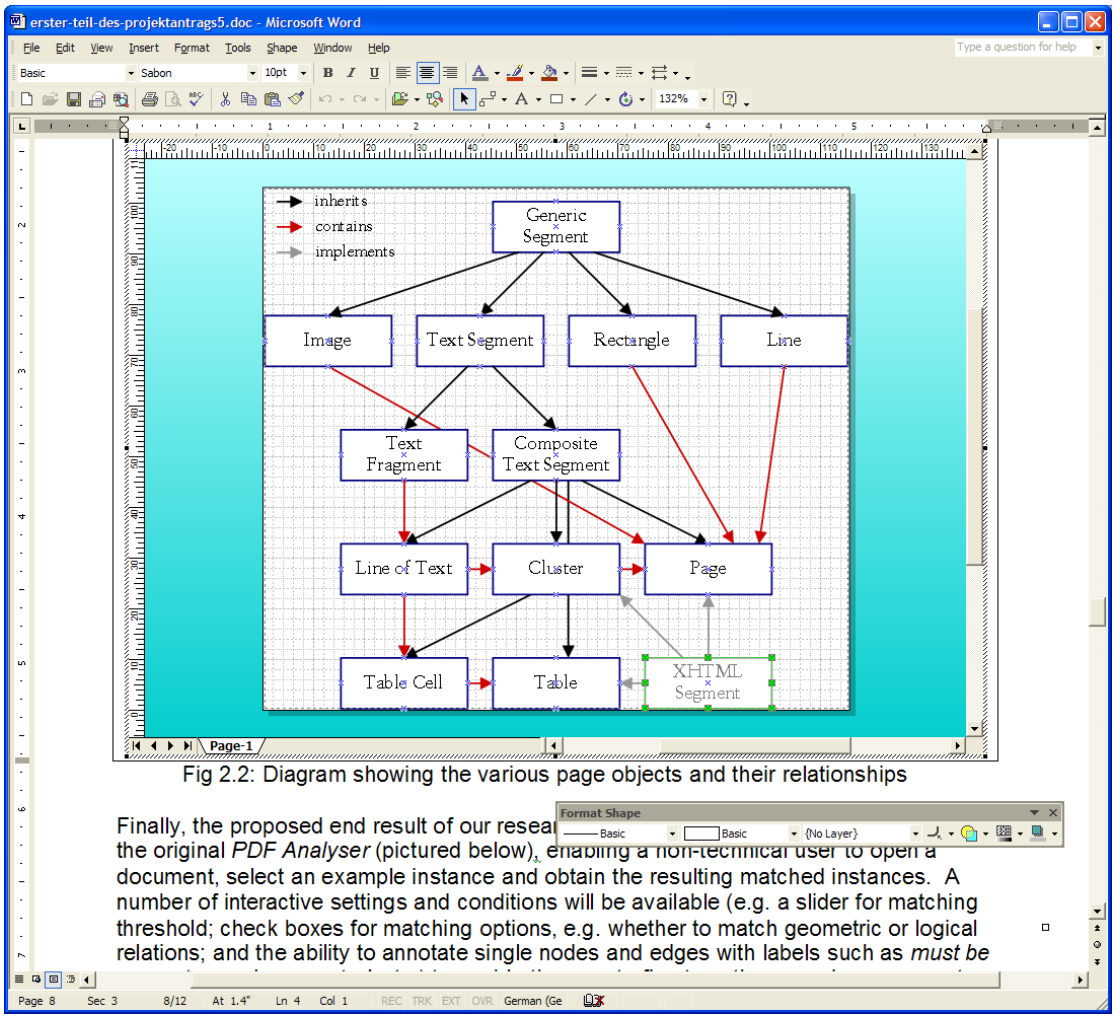

Figure 4: Example of a *Visio* diagram being edited from within a Word document using OLE. The floating toolbar belongs to the Visio application

The trouble with OLE is that its implementation is rather poor. Indeed, it masquerades as a feature providing "user friendliness" yet brings absolutely no benefit to the user at all. Moreover, it actually hinders the situation by providing confusion, unreliability and even putting data at risk (see the last bullet point below). The following list describes some common problems with the implementation of OLE:

- The separation between host application and embedding application is not made very clear to the user. If the object can only be edited using controls belonging to its native application, which is usually the case, it is better to display the object in a totally different window.

- Editing the object is not always non-destructive. Sometimes, the mere act of entering edit mode can cause strange formatting changes.

- OLE objects are non-portable; if the document moves between computers that have slightly different versions of software installed, OLE may not work at all, or have unexpected consequences.

- When editing the embedded object, the user is restricted to the controls on the OLE toolbar, which usually represent a small subset of the controls available in the originating application. If a user needs to perform other functions, he must go back to the originating application itself.

- It is often not possible to change the bounding box of an embedded object (e.g. when an extra box is added outside of a diagram's current dimensions).

- Making detailed alterations is very awkward as zooming usually takes place in the host application.

- It is not always possible to copy the embedded object and paste it back into its originating application. Therefore, one always needs to keep a copy of the original file from which the embedded object was created. However, most users, having seen that they can edit the embedded object directly, will not know this, and may well discard the original before they find out they need it.

The net result is that I find it much easier and more reliable to make edits to embedded objects within the originating program and copy them anew. But I only know this after many hours lost through experience. And Microsoft seems to have acknowledged the poor state of OLE and responded by reducing the amount OLE functionality in newer versions of *Office*. I find this a pity, as I believe

that application interaction is becoming increasingly important in today's GUI applications, particularly in document authoring.

If OLE had been better designed and implemented, it would have been enabling, not limiting. It would have exploited redundancy, allowing similar attributes to be changed in the same way in an embedded object as in the host application itself. It would have provided extra functionality, for example enabling text styles in the host document to be used in the embedded object. And it would have been reversible, making it possible to copy any embedded objects back into their respective originating applications.

The fourth suggestion of this paper is indeed to take this notion a step further and to **improve interaction between applications on the desktop**, therefore eliminating redundancy at the user interface level, in such a way that **boundaries between applications disappear completely**. Thus the notion of an *application* or *program* could change quite considerably. Although such a program may still run as a discrete process on the operating system level, it would simply offer additional functionality integrated into other *application paradigms* (see the following section) or on the desktop itself. Of course, a prerequisite for such behaviour is fully documented and, ideally, open interfaces and file formats. And ultimately, the "Word processor of DTP?" dilemma mentioned in Part 1 of this paper could be solved this way. Essentially, such a development could be thought of as providing the same type of interoperability that is available on UNIX-like operating systems using Pipe, but at the GUI level.

Freedom and restrictions—Less can sometimes be more

By using the phrase *application paradigm*, I refer to the use of a tangible metaphor for a particular task. For example, in the creation of documents, the "page" metaphor is commonly used to show a WYSIWYG display of the page and enable the contents to be manipulated interactively. For longer stretches of text, a more typewriter-like metaphor with a continuous page, style list and outline view (Figure 5) is often considered better, as the user has less *freedom*, and is therefore better able to concentrate on the task at hand. Formatting and pagination, which serve only to distract the user, are carried out automatically. This is even more true for charting and diagramming software, which imposes further *restrictions* on how objects can be manipulated, therefore allowing the user to finish the task more efficiently. Examples as shown in Figures 6, 7 and 8. Multimedia applications, on the other hand, use a completely different metaphor, which is again suitable for the nature of the content that is being manipulated.

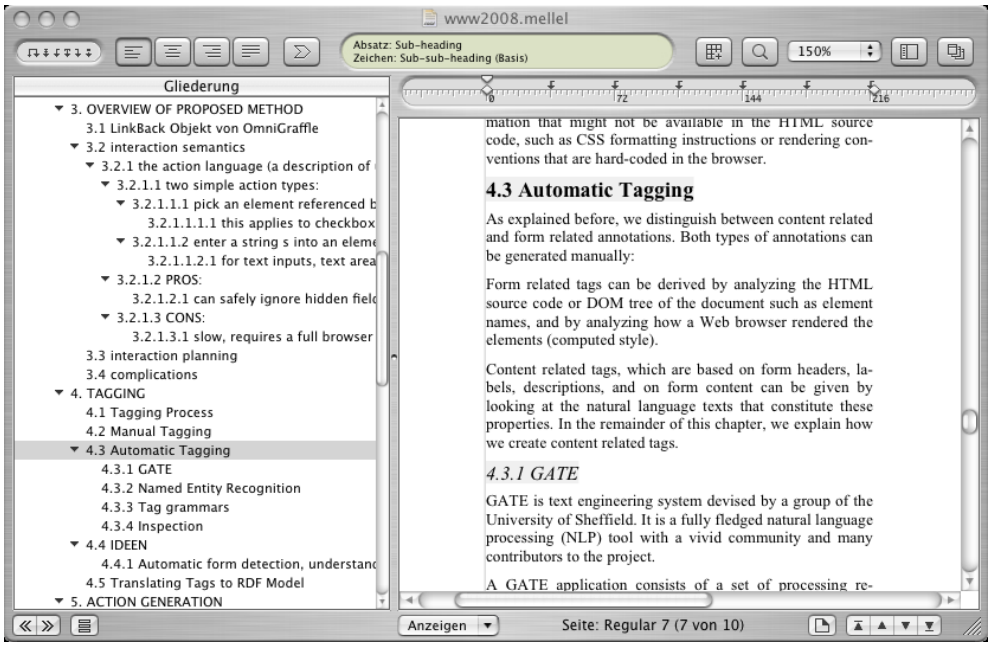

Figure 5: The *Mellel* word processor provides a suitable user interface for the composition of longer, structured texts

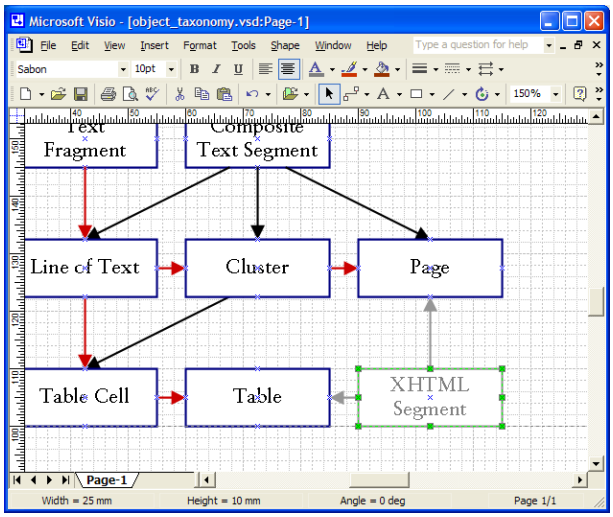

Figure 6: Example of the use of *restrictions* in the editing of a diagram in *Visio*. Arrows and lines can only be joined to rectangles at the predetermined connection points

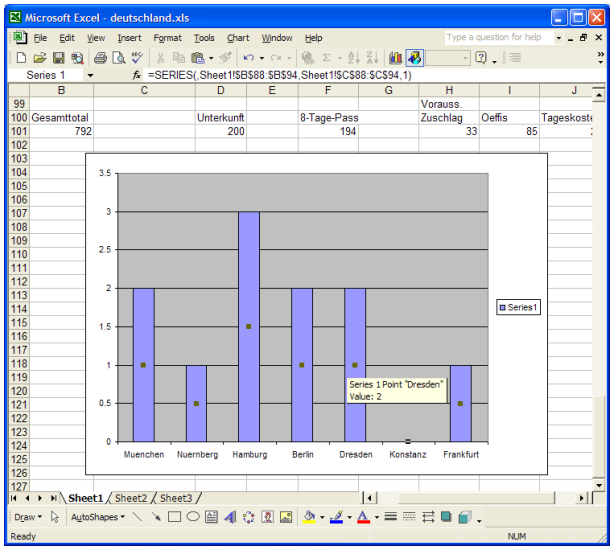

Figure 7: Another example of the use of *restrictions* in the manipulation of an Excel chart

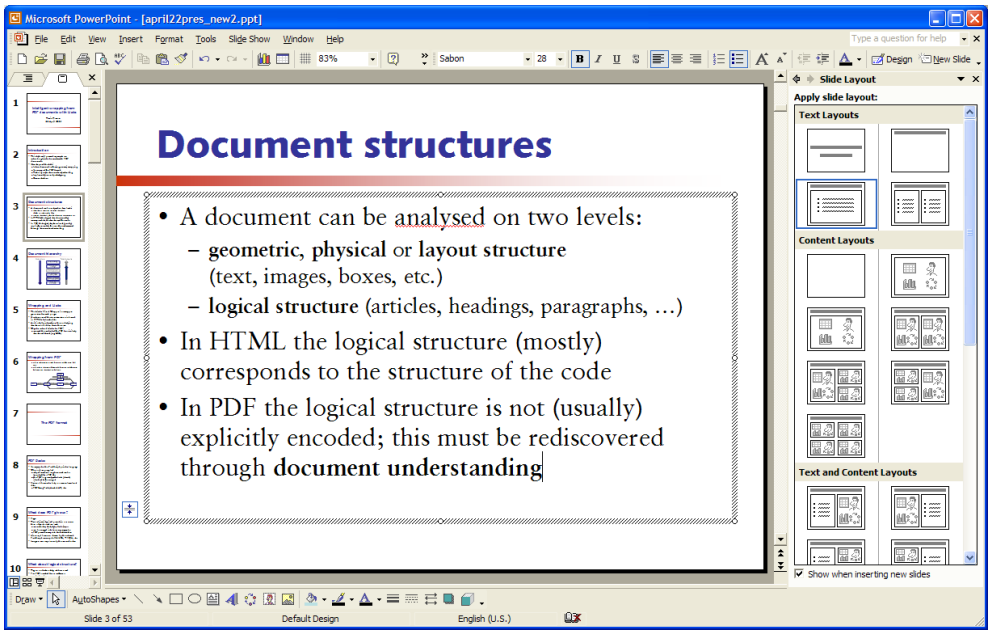

Figure 8: Example of the *PowerPoint* user interface showing what could be interpreted as a very basic version of the *flexible layout* concept as well as *restrictions* in action. When the text box is overfull, the text size is automatically reduced. The predefined layouts on the right-hand side allow from a limited set of predefined layouts to be chosen, and objects within the page can be arranged interactively.

Vector graphics

Vector graphics software, such as *Adobe Illustrator* and *CorelDraw*, also deserves a mention, as it shares many similarities with DTP software, and therefore presents a further overlap in functionality. Additionally, the "page" metaphor is commonly used, although it may refer to an abstract bounding box rather than an actual page size. Therefore, I believe it would also make sense for vector graphics functionality to be incorporated into DTP in the same way that was suggested for word processing in Part 1 of this paper. This would allow the same tools to be used to manipulate page layout items (lines and boxes) as are used to edit the graphics themselves. As most vector graphics usually occur as separate, encapsulated objects on the page, it is important that this separation be retained, even if the tools used to edit them are the same. This separation could be achieved by multi-level object grouping, with the editing of such objects possibly even occurring in a separate window, ensuring that they are dealt with as single objects on the page layout level.

Bitmap graphics, in contrast, have a completely different nature. I do not believe that the "page" metaphor is suitable for images that are totally constrained by a square grid of pixels. Such images will always need to be edited separately.

Part 3: Non-HCI issues

Portability

Most off-the-shelf document authoring software has not made it easy to move a document between different computers or use its file format for distributing documents. Often, there are differences between the file formats of differing versions of the same software. And although later versions can usually open most files created by earlier versions of the software, this does not work at all the other way round. If exactly the same fonts are not present, formatting is affected. In Microsoft Word, formatting can even be affected by changing the printer driver. With the increase in interaction between applications proposed in Part 2 of this paper, there are many more potential points of failure, and it is therefore crucial that the *portability* problem be solved once and for all.

The Adobe Portable Document Format (PDF) was successful precisely because it filled this niche, providing a robust page layout regardless of computer or printer. But this format is not (in any significant way) editable. Here are a few suggestions that should be addressed in the creation of a portable file format:

- File formats should be structured so that even earlier versions of the software can read documents saved in later versions. Most of the time, the extra functionality introduced in newer versions is not used anyway.

- Layout must be robust. Ideally, all fonts would be automatically embedded in the file, unless the user specifically opts not to do this[9]. In such a case, the font metrics should still be available to ensure that a substitute font can be synthesized[10]. PDF does this already.

- The above point may not be so crucial if a flexible layout engine is employed; the document could be described just with its constituting text blocks and their dimensions; the layout engine will then be responsible for ensuring that everything fits together, regardless of which font is used.[11]

- Similarly, any linked images should also be locally saved (within the file), again unless the user specifically opts not to do this. In this case, the user should be warned before saving that the resulting file will not be complete.

- Any embedded objects should include a rasterized copy in case the originating program no longer becomes available. (I believe Microsoft Office does this already; this could be the cause of the formatting irregularities described earlier.) Any new fonts present in these objects should also be embedded in the document. If the original program is not available, it should still be possible to edit the rasterized copy using a basic vector or bitmap editing program.

- Last, but not least, file formats need to be fully documented and, ideally, open too.

Versioning

Finally, a small note about versioning. The "Undo/Redo" feature has already come a long way in recent years, allowing the user to step back and forth between multiple undo levels. This is important, as it upholds an important HCI principle: *forgiveness*. This encourages the user to experiment with new ideas and increases productivity. However, I believe that this is not enough; anyone who has ever worked on along document will certainly remember having dozens of versions scattered around on the hard disk, so that certain, older ideas could be later revisited if necessary. Ideally, the entire document history would be encapsulated in a single file, simplifying organization, saving disk space, and allowing the user to

[9]This could be for copyright reasons, or in order to minimize storage space in situations where the actual font used is not important.

[10]More information on font synthesis and parameterized fonts is available at `http://www.tamirhassan.com/typography.html`

[11]Still, in order to maintain appearance, it is important to include some metadata about the original typeface so that an appropriate subsitute can either be synthesized or chosen from the list of fonts available on the system.

search and recover any previously deleted material should he discover that he did want it after all.

As this is a separate topic in itself, I will not go into further detail, but rather point the user to two developments that aim to address this problem; the first is *Google Documents*[12], which features an online document store with revision tracking; the second is *Scrivener*[13], shown in Figure 9, a Mac word processor aimed at novelists, which allows the easy addition of ideas onto a pinboard, which can be shuffled later into complete chapters. Even later versions of Microsoft Word include a little-known feature that allows multiple versions of a document to be saved in a single file.

A further question arises whether such versioning functionality should be incorporated at the application level or at the operating system level. Clearly an operating system-level integration would reduce redundancy and provide a more consistent interface across the entire desktop. With text-based formats, this is already possible using *CVS* or *Subversion*, but the setup of such a system is beyond the average user's capabilities and would only enable text to be searched for. With the improvements in interaction between applications and the OS proposed in this paper, a GUI-based, object-based versioning and revisioning system could be the next logical step.

Summary

The problem with document authoring is that is has now reached a stage where it is considered by most to be "good enough". In fact, it has been this way for the last decade or so. Today's off-the-shelf word processing packages still bear much resemblance to their historical development which—even including the move to GUIs—can be seen as being evolutionary rather than revolutionary. Moreover, whilst many advanced users know the cause of their frustration, it is not so obvious how the software could be improved to avoid such problems altogether.

This paper has analysed the present situation from an HCI perspective and, I hope, has offered the reader a fresh look at how the situation could be improved and how document authoring systems of the future—and even GUIs in general—could function. The following questions have been proposed, which I hope will offer stimulus for further discussion and eventually development in this area:

- How can we combine word processing and DTP to create a program that is equally good at drafting as it is at page layout? Should powerful vector graphics functionality also be combined here?

[12]Google Documents, `http://docs.google.com`
[13]Scrivener, `http://www.literatureandlatte.com/scrivener.html`

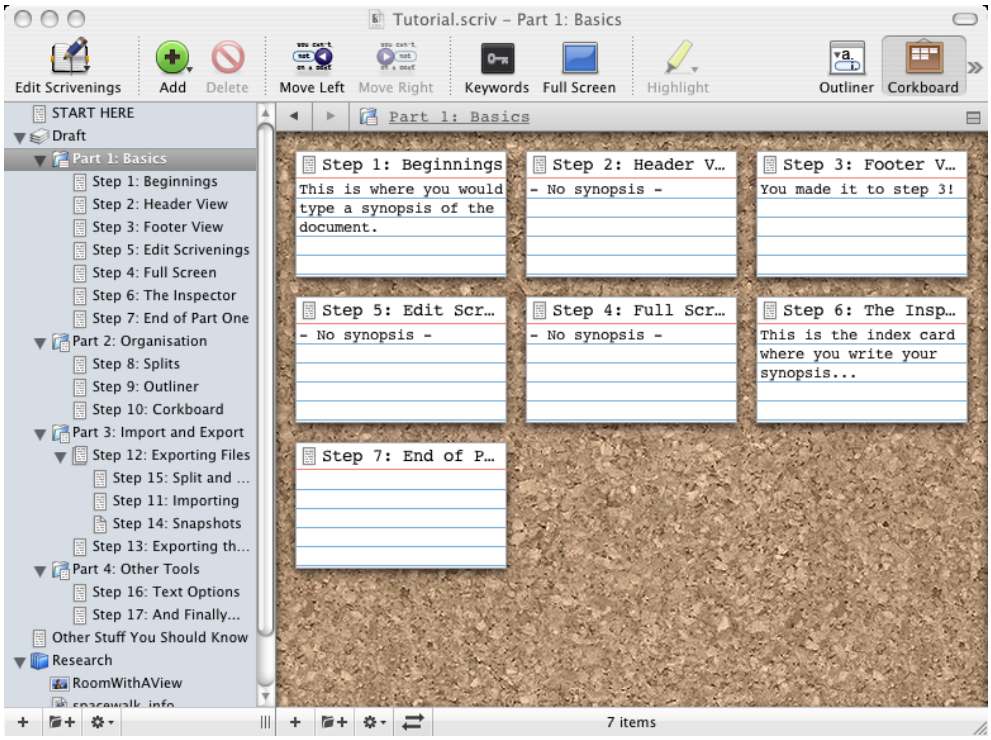

Figure 9: Example of *Scrivener*'s pinboard user interface

- How can we integrate LaTeX into a user-friendly, fully WYSIWYG GUI?
- How can we further develop the *flexible layout* concept, as used by e.g. the LaTeX layout engine, to allow interactive manipulation of page layout?
- How can we improve interaction between applications and reduce redundancy on the desktop? Should the physical boundaries between applications be removed completely?

Additionally, in Part 3, we briefly visited the problems of portability and revisioning/versioning relating to current document authoring software.

In more recent times, document authoring software has been driven primarily by commercial interests. For the market leader, it is very risky and non-profitable to change the current status quo. As an example, even the new "Ribbon" toolbar in the latest version of *Microsoft Office* has alienated many users. Furthermore, the suggestions posed in this paper could be seen to evolve around a central theme: *interoperability*. In the current commercial software environment, this is indeed a dirty word. The open source community, which is probably the biggest competitor, now has the opportunity to bring about such a change, but unfortunately seems to spends all its resources on attempting to achieve interoperability and compatibility with existing commercial software instead. This seems to be a futile effort.

I believe it's time that we start to rethink the design of such software from the user's perspective, and this paper is my part in an effort to start the ball rolling. The questions posed in this paper shed light on interesting topics worthy of further investigation. If we address these issues, I believe we would have the potential to produce much better software, making existing programs and approaches obsolete and ensuring a better computing experience for all.

This is Version 1.1.1a of this paper, as of 23 November 2007.

After much deliberation, it was decided to only use the male pronouns he *and* his *where appropriate for reasons of clarity. These should, of course, be interpreted in the gender-neutral sense.*

This paper has been typeset using LyX. *I prepared the initial draft in* OpenOffice Writer, *and also spent some time playing around with both* LyX *and* TeXmacs *before finally deciding upon* LyX. *Now that I am familiar with* LyX, *I am pleasantly surprised with it, and would now rank it as first choice for preparing my upcoming doctoral thesis. I was, however, less than impressed when* LyX 1.4.2 *completely refused to load a document saved in* LyX 1.5.2. *:(This was on a workstation on which I didn't have administrative privileges, and therefore couldn't upgrade either.*

Thanks to all who offered help in the form of proof-reading and feedback!