DATA STRUCTURES FOR PAGE READERS

Henry S. Baird

David J. Ittner

AT&T Bell Laboratories
600 Mountain Avenue, Room 2C-322
Murray Hill, NJ 07974-0636 USA
henry.baird@att.com

ABSTRACT

Software-engineering aspects of an experimental printed-page reader are described, with emphasis on data-structure choices. This reader performs a wide variety of tasks, including geometric layout analysis, symbol recognition, linguistic contextual analysis, and user-selectable output encoding (*e.g.* Unicode). We have implemented a single common data structure to support all these tasks. It embraces iconic, geometric, probabilistic, and symbolic data, and can represent the full physical document hierarchy as well as many partial stages of analysis. We illustrate the evolution of this data structure, in the course of reading a page, from purely iconic to purely symbolic form. The data structure can be snapshot in machine- and OS-independent peripheral files. Careful agreement on its semantics allows it to be used as a 'blackboard' in an elegant exploitation of generic UNIX multi-processing. One partly unresolved issue is the best representation for ambiguities spanning several levels of the document hierarchy.

1. Introduction

We have previously described algorithms and heuristics used in the component subsystems of our experimental machine-print page reader[1,2,3,4,5]. We have also demonstrated its versatility [6,7]. In this paper we consider software-engineering features of its design, with special attention to data structure (d/s) choices.

Looking back several years, it seems to us that much of the influence of our research software on AT&T engineering practice has been due to certain d/s choices which have had strongly positive effects on systems integration and evolution, allowing our experimental code to be applied to new tasks with relative ease. They have also supported easy portability and debugging. Most surprisingly perhaps, they have recently permitted an elegant non-trivial exploitation of multi-processing. Although these d/s choices were often made almost as an afterthought, at times when our attention was focused on algorithm design, in retrospect they seem to have been among the most beneficial. For this reason we feel that our experience may be useful to researchers and engineers building similar systems.

Published in A. L. Spitz & A. Dengel (Eds.), *Document Analysis Systems*, World Scientific, Singapore, 1995. (Expanded version from *Proc., IAPR 1994 Workshop on Document Analysis Systems*, Kaiserslautern, Germany, October 18-20, 1994.)

As our page reader evolved, we saw that certain characteristics of data structures were particularly desirable. Some were obvious enough: for example, data types of great variety are needed, including iconic (graphic), geometric, probabilistic, and symbolic, to support algorithms adapted from many fields of research including image processing, computational geometry, statistical decision theory, numerical analysis, and computational linguistics, to name only a few. Quite early we decided that all these data types should be integrated within a single d/s. Results to be communicated among component subsystems often represent intermediate incomplete stages of analysis; thus we wanted the d/s to express, not only the full document hierarchy (pages owning blocks owning text lines, words, characters, etc), but also arbitrary partial subsets of the hierarchy.

To encourage other R&D colleagues to share results, we wanted to be able to "snapshot" the d/s at any stage of analysis and communicate it to people working on different machines and varieties of UNIX operating systems. Thus we felt the need for the main memory d/s to have an isomorphic, machine- and OS-independent peripheral file representation (*i.e.* the d/s should be *persistent*).

Of course, we also wanted our large software system to be (relatively) easy to understand, extend, and maintain. We believe this is most likely to occur when a system can be decomposed into modules having well-defined input and output conditions, clear and unambiguous semantics, and no side-effects. Such a programming discipline can be encouraged (although not ensured) by providing a single well-documented d/s with clear semantics in which all the I/O conditions can be represented. For this reason, we strove to define a data structure that was — in our eyes at least — simple, regular, and unvarying: *simple* in that it consists principally of elementary data types with unambiguous semantics; *regular* in that it allows no special cases or exceptions; and *unvarying* in that it offers no more than one way to represent the same thing. In this we have been successful, in that our colleagues have found it fairly easy to agree on the semantics of the d/s. In order further to discourage undisciplined improvisation, we have provided a family of subroutines for each record in the d/s, named and implemented regularly in an object oriented style (written in C), supporting allocation and freeing, insertion and removal (within lists), reading and writing to files, etc.

This strategy has encouraged the development of cleanly isolated functions which can be executed in a variety of sequences and combined safely with functions developed by colleagues working elsewhere. As we have described in other papers, the resulting system has been applied to physical segmentation and symbolic interpretation (recognition) of complex textual documents.

We are not aware of previously published discussion of page-reader data structures in our concrete sense. Architectural issues in page readers have been addressed in [8,9,10], more abstractly than we do. Isolation of functions and modular programming were emphasized by the structured programming movement [11].

We describe the software engineering environment in Section 2. Sections 3 gives the motivation for our representation of images, geometry, probabilities, and symbolic codes. Section 4 introduces the physical document hierarchy, and Section 5 illustrates its evolution during analysis. Section 6 introduces two isomorphic representations of the data structure, in main memory and in peripheral files. Section 7 gives a detailed discussion of their use in multi-processing. Sections 8 and 9 briefly touch on the representation of ambiguity and graphical editing. Section 10

summarizes the main results.

2. Application and Engineering Environments

The family of programs making up the page reader are all written in the C programming language and run under various flavors of the UNIX® operating system. Our main development machine is a Silicon Graphics Computer Systems Challenge XL running IRIX Version 5, although the software automatically configures itself for other machines and environments including SUN workstations running SunOS or Solaris, PCs running variants of UNIX, and VAX hardware running Research UNIX. We frequently need to ship document images, after various stages of processing, among all these environments; obviously, a machine- and OS-independent peripheral data structure is critical.

We can accept document images in most of the major formats for bilevel images, including TIFF, CCITT (Group 3 or Group 4), and Sunraster files, under various compression schemes.

The system is not restricted to any symbol set or target language, therefore the data structure must support the internal identification of symbols from *any* writing system. Our laboratory is equipped to display, edit, and print the full Unicode[13] character set, but to preserve portability we adhere to a conservative policy during program development, in which all program sources and tabular data needed to build the system are written in 7-bit ASCII. Special needs of symbol recognition precluded choosing Unicode as the internal code for symbol classes (*e.g.* ligatures do not have code points), so we represent them by 7-bit ASCII strings — more on this later. Encoding of text at any external interface, say to Unicode or JIS, is under user control, specified in 7-bit ASCII tables.

3. Basic Data Types

In this section we give a brief introduction to our representations of four basic types of data: images, geometry, probabilities, and symbol codes.

3.1. *Images (Iconic Data)*

Bilevel images in our system are treated as finite black foreground objects on an infinite white background. Black and white are not treated symmetrically: only black pixels are explicitly stored. As physical segmentation proceeds, black pixels may be moved about and connected components may be cut into pieces. No black pixel is ever discarded — it is assigned to some part of the hierarchical d/s — but, as we will see, it may be duplicated in order to represent alternative segmentations.

The spatial sampling rate (digitizing resolution in pixels/inch (ppi)) is assumed to be fixed within each page, but may vary arbitrarily from page to page; thus it is stored in the page record. The sampling rate is generally assumed in our system to be the same horizontally as vertically, but some algorithms are able to cope with different rates.

Within main memory, images are stored as horizontal runs of black pixels, which are organized into 8-connected components at an early stage of processing. Operations performed on runs of pixels are generally more efficient, and hence run faster, than if they operated on individual pixels. When images are written to peripheral

files, black connected components are encoded using the CCITT Group 4 encoding. In practice, this produces a factor of 8 reduction in file size over encoding runs; the computation to convert between encoding methods is modest.

3.2. *Geometry*

As skew and shear angles (of pages, blocks, and text lines) are detected, we often correct the artwork immediately by pseudo-rotation and pseudo-shearing and then store the angles in the d/s so that — in principle, never yet in practice — the transformation can be inverted.

Subregions of the image — for example, text blocks — are defined implicitly by the black pixels they contain. Thus, we do not need a mechanism for defining regions geometrically — by, *e.g.*, boundaries. This allows considerable freedom in layout analysis, where in particular blocks of text are not restricted to those bounded by orthogonal polygons.

Future extensions to grey-scale and color may force reconsideration of this policy. We plan to investigate the use of 'basis' regions, defined either geometrically or as bilevel images, which are attached to artwork to delimit the area over which the artwork is defined.

Explicit orthogonal bounding boxes (shrink-wrapped about their black pixels) are carried along with each major data item in the hierarchy. These are not, of course, intended to define the contents of the region — it is permitted, for example, for boxes of distinct regions to overlap — but rather as a rough indication of their extent. Often, the bounding boxes alone are sufficient to support a critical computation: for example, they are the primitive data item in block-finding.

3.3. *Probabilities, Confidence, and Evidence*

Several stages of analysis — notably classification — compute lists of alternative interpretations labeled with approximate probabilities. Oftentimes these 'probabilities' are very far from well-behaved from a theoretical point of view: they may, for example, not always sum to one. This occurs in classification, to take one example, because we wish them not only to exhibit good rank order (with the most probable class highest), but also to possess a reliable reject threshold (so that values below threshold indicate malformed images). Perhaps it would be better to call them, generically, 'confidence scores'. Whatever their properties and purposes, we generally represent them as real numbers in the range [0,1].

3.4. *Symbolic Codes*

Symbol classes need names. Unicode provides a unique code point for most characters in the living languages, so it might seem a good choice. However, several considerations have led us to reject Unicode as a coding convention *within the page reader*, while we still provide full support for Unicode as an input and output encoding selectable by users. First, Unicode does not provide code-points for ligatures (*e.g.* 'ﬁ') and other commonly merged combinations of characters which it is convenient to treat within the reader as elementary shapes. Second, it is often an engineering necessity to use distinct classes for dissimilar *shape variants* of character classes (*e.g.* 'a' and 'ɑ'). Third, Unicode is not yet well supported as a text encoding — by

editors, printers, displays, etc — in most software development environments, so that requiring Unicode support is an obstacle to portability.

Thus we have chosen a conservative strategy of declaring class names to be short 7-bit printable ASCII strings. For example, "a" for 'a', "ast" for '*' (asterisk), "c-cd" for 'c' (c-cedilla). Shape variants are indicated by suffixes: *e.g.* "a.0" for 'a' and "a.1" for 'ɑ'.

Users can achieve arbitrary input/output code translations to/from our internal class names, to yield Unicode, JIS, and other encodings. This requires use of a 7-bit printable ASCII translation table. Writing and referring to these tables is, arguably, a bit awkward for naive users; but we do not know a better way that provides as much versatility.

4. The Physical Document Hierarchy

By the physical document hierarchy we mean the well-known nested decomposition of document into pages, of each page into any number of blocks of text, blocks into text-lines, and so forth as in this schematic:

```
document
 page
  block
   textline
    word
     char (symbol)
      interpretation:
       class + confidence score
      image:
       connected component
        run
         pixel
```

This shows the fullest elaboration of the hierarchy, when all of the parts of the document have been extracted. Of course, this evolves through several stages of analysis, as we shall see in the next section.

5. Evolution of the Data Structure

We now illustrate the evolution of the d/s at certain stages of analysis. For reasons of space, we omit many stages and many details.

5.1. *Page Skew and Shear Analysis*

The input to this stage is a document consisting of one or more pages, each of which contains a rectangular image, already expressed as a set of black connected components; each page is also labeled with its box and the spatial sampling rate (resolution). This is indicated schematically as follows:

```
doc
 page + resn, box
  image (set of black connected components)
```

The output is skew- and shear corrected and labeled with the angles:

```
doc
 page + resn, box (new), skew, shear
  image (skew- & shear-corrected)
```

Note that the box may have been updated, since it is now shrink-wrapped around artwork that may have been moved.

5.2. *Block Finding*

The input to this stage is the output shown immediately above. The output d/s is now enriched to contain text blocks:

```
doc
 page + resn, box, skew, shear
  image (photo, line-graphics)
  block + box
   image
```

Note that the page may also own an image that is not contained in any block: in practice, this is artwork that was set aside by the block-finding algorithm on the grounds that it appeared to be either too large or too small to be text. This illustrates the policy that images that are not selected for inclusion in a subrecord (here, a block) of a record (here, a page) remain owned directly by the original record.

5.3. *Resegmentation of Symbols Within Words*

We jump ahead to an interesting stage whose input contains pages, blocks, text lines, words, and symbols which have been classified:

```
doc
 page + resn, box, skew, shear
  block + box
   textline + box
    word
     char
      image
      code + score
```

The output expresses the results of generating alternative segmentations of each word into characters:

```
doc
 page + resn, box, skew, shear
  block + box
   textline + box
    word
     char
      image
      code + score
     word (resegmentation alternatives)
      char
       image
       code + score
```

Note that the alternative resegmentations are expressed by allowing a `word` record

to own a set of one or more `word` records. Similar recursive representations are allowed for text lines and blocks, but have not yet been exercised in practice.

5.4. *Summary*

These few examples illustrate several features of the data structure:

1) results of each stage of analysis are stored in the d/s so that, in principle, the analysis can be inverted;
2) records can own not only their normal subrecords (*e.g.* pages owning blocks) but also less completely analyzed parts (*e.g.* images), at the same time; and
3) records can recursively own records of their own type, permitting the expression of alternative analyzes (*e.g.* segmentations).

6. Representations: Internal and External

We have implemented two isomorphic forms of the d/s. In main memory it is a linked-list hierarchy served by a systematic and regular set of functions in an object-oriented programming style (in C). Thus all functions see the same data structure no matter what program they run in. In peripheral files the d/s is a byte-stream, machine- and OS- independent, so that it can be written to and read from files and UNIX pipes, moved around by `ftp`, `uucp`, etc. All programs therefore see the same file structure no matter what shell script they run in.

We have provided a family of I/O functions which can read/write the entire hierarchy, or optionally only selected parts of it, from/to files. This is quite general and unconstraining, so that it can be used to 'snapshot' intermediate forms of the d/s. These intermediate forms are often helpful in debugging, archiving interesting results, or communicating results to workers at other locations. They have become essential tools in collaborations between geographically separated teams, such as research and development.

7. Exploitation of Multi-processing

One of the recent advances in computing hardware is symmetric multi-processing. Document image analysis, and in particular symbol recognition, is an ideal application for multi-processing through parallelization. We have extended our experimental page reader to take advantage of multi-processors while making minimal demands of the underlying operating system or hardware.

A schematic diagram of a portion of the reader is shown in Figure 1. Geometric layout analysis, including connected components analysis, skew- and shear-correction, segmentation into text blocks, determination of text-line orientation, segmentation into text lines, and finally into symbols, is performed first. As described in Section 5, this processing results in a page owning possibly multiple blocks, which own text lines, which in turn own symbols.

This incomplete hierarchy is passed to the next processing stage which performs classification of symbols by shape, inference of text size and baseline, segmentation of lines into words by spacing, and shape-directed resegmentation of symbols to handle touching or broken characters. The hierarchy, now enriched with possibly multiple segmentations of words (if appropriate for the language), and a short list of interpretations for each symbol, is passed to the contextual analysis stage which exploits

punctuation rules, simple morphology, and spelling checks to prune alternatives.

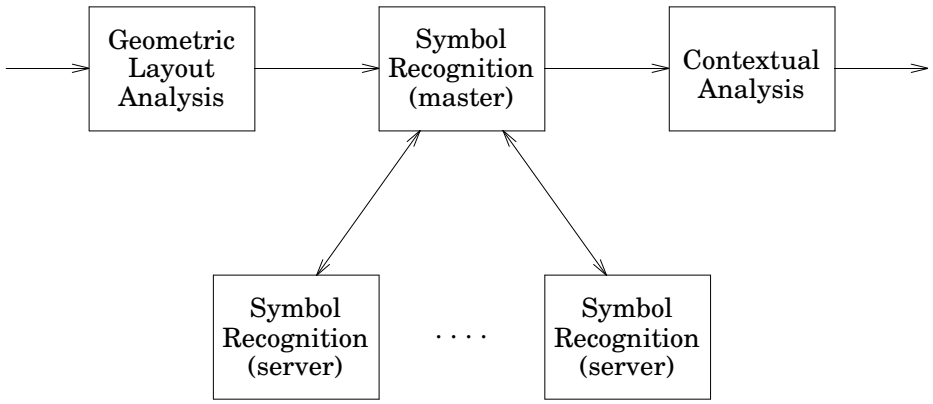

Figure 1. Schematic of a portion of our experimental page reader. Character classification is distributed by the master to multiple instances of identical software acting as servers. The same data structure is used within, and exchanged between, all software units.

Typically the majority of the computation in this simple feed-forward pipeline is spent in the symbol classification stage; our focus has therefore been to parallelize symbol classification in order to improve overall throughput. As illustrated in Figure 1, the task of symbol classification is broken up by the "master" symbol recognition software unit and distributed to multiple instances of the same unit, now acting as servers. Rather than defining a special set of messages to be passed between the recognition processes, we exploit the same data structure used in the main pipeline. This has greatly simplified the implementation, caused minimal change to our pre-existing control flow, and has allowed us to concentrate on the more interesting issues, such as the granularity of parallelization.

7.1. *Implementation*

Within our page reader the tasks of geometric layout analysis, symbol classification, and contextual analysis is each performed by a separate UNIX process. For these experiments, the classification process was enhanced to optionally duplicate itself, setting up a pair of UNIX pipes for pairwise communication between the master process and each server.

Symbol recognition naturally lends itself to parallelization at various levels. For example, individual symbols can be distributed to servers for recognition by shape. Another natural alternative is to send all symbols owned by a text line or block in a single transaction. In general, the smaller the unit of work, the larger the number of transactions required and the more demands made of communications (*e.g.* very low latency) in order to keep the servers busy. The larger the unit of work, the fewer the number of transactions, however, large transactions may also make it difficult to keep all servers fully utilized. For example, if entire blocks are distributed to servers, and one block is much larger than the others, all but one of the processors will finish its work and go idle while the large block is completed (we always process

a page to completion before going on to the next page).

We wanted to experiment with various strategies for parallelization without having to greatly disturb our existing control flow and I/O conventions. Fortunately, the existing data structure supported this application nicely. Regardless of the level of distribution, the master process writes the appropriate partial hierarchy to the server. For example, in order to send a text-line's worth of symbols to the server, only the necessary context, that is, a page owning a single text line of symbols, is transmitted.

The work to be performed by the server is implicit in the data structure. Continuing the text-line example, the server recognizes that none of the symbols in the text line have been classified — so it does so. Then, the server recognizes that it cannot segment the symbols into words since the necessary context is not available (we typically use statistics gathered over the entire block), so it writes the enriched hierarchy to its standard output. The master reads the partial hierarchy from the pipe, breaks off its subtree containing the text line and attaches the new subtree. After all text lines in a block have been processed in this way, the master gathers statistics, segments text lines into words, and may distribute the text lines for a second pass necessary to clean up broken or touching characters (we bound this work to within a word, and therefore it cannot be done until words are found).

This logic, while seemingly complex, fits nicely into the existing control flow. A few "hooks" provide a clean separation between control logic and the multi-processing glue. During initialization a single routine is called which spawns duplicate processes and opens the pipes. There is a single run-time option indicating how many servers to spawn. This option is consumed by the master, although passing it to a server would produce multi-level servers with no additional logic. Two calls were added to the main logic at each level of the hierarchy; for example, at the block level:

```
for (each block) {
    if (MP_block(block)) continue;

   ... process this block locally ...

}
MP_end_blocks();
```

and at the text-line level:

```
for (each textline) {
    if (MP_textline(textline) continue;

    ... process this text line locally ...

}
MP_end_textlines();
```

The multi-processing routines such as `MP_block()` decide whether to send the item to a server. If so, it returns 1 and the block is skipped by the master; otherwise, it returns 0 and the block is further processed locally. That local processing will lead to the second block of pseudo-code, at which time the individual text lines

of the block may be distributed to the servers. The routines such as `MP_end_blocks()` waits until all blocks have been returned by servers. All transaction queuing and server management is hidden behind calls of this type; the multi-processing routines total 730 lines of C code.

Because this implementation makes such few demands of the underlying operating system — a `fork()` and `exec()` system calls and some method of streaming communications such as provided by UNIX `pipes` — the software has been run essentially unchanged on Silicon Graphics hardware with its System V flavor of UNIX, a SUN Microsystems 670 server with SunOS, a SUN 1000 system running Solaris, and even running on coprocessors with very little operating system support.

7.2. *Experimental Results*

The first 30 document images, in collating sequence, from the University of Washington English Document Image Database I [12] were used as the test set (those images containing no upright text were excluded). These pagers are an assortment of technical journals and reports printed in a variety of page layouts. Geometric layout analysis[7] was performed, producing a number of variable size blocks, each owning text lines which in turn own individual symbols for recognition.

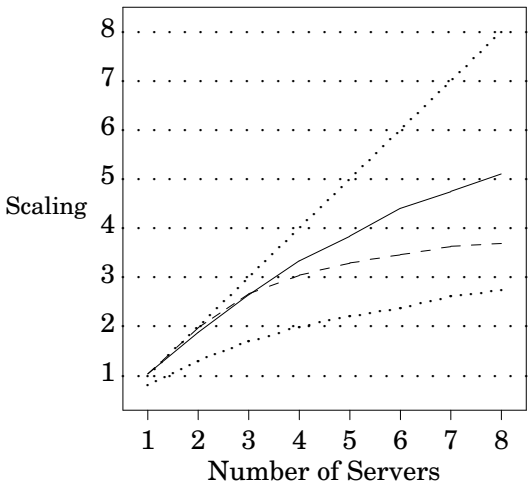

Figure 2. Scaling as recognition servers are added. Base time is elapsed time using 0 servers. The diagonal dotted line shows perfect scaling. Lower curve is when all parallelization done at the text-line level, middle curve at the block level, and upper curve when a hybrid strategy is used.

The graph in Figure 2 shows the scaling achieved as servers are added. The base time was taken to be the elapsed time required to process a page using **0** servers (*i.e.* all processing done locally by the master). The scaling was then computed as the base time divided by the elapsed-time taken for symbol classification using N servers; each datum is an average over the 30 page test set. These experiments were performed on a lightly-loaded Silicon Graphics Computer Systems Challenge XL, with 8 150MHz R4.4k processors, running IRIX release 5.1. Similar results were observed on the other platforms.

The lower curve of the graph shows the scaling results when parallelization is performed at the text-line level. There is a significant loss in efficiency even for one server due to server idle time for each of the relatively large number of transactions (only one transaction is outstanding to each server, creating some idle time after each text line). Efficiency also degrades as we add servers; for example, only a factor of 2 in throughput is achieved when using 4 servers, and only a factor of 2.7 when using 8 servers. The worst case occurs when a page is decomposed into a large number of small blocks, say each owning one or two text lines. Since a block is processed to completion before going to the next (a limitation of our simple control flow), several of the servers go idle.

The dashed curve of Figure 2 shows the scaling results when parallelizing at the block level only. Here the efficiency when going from 0 to 1 server is near 1; this indicates the overhead in reading/writing the intermediate dim data structure is minimal. Scaling begins to degrade drastically at about 4 servers. This decline in efficiency can be seen when a page is decomposed such that 1 block is much larger than the others. In this case all other blocks are processed and all but one of the servers go idle while the large block is completed; as the number of processors increase, the relative efficiency of each is reduced. We try to minimize this idle time by sorting queued transactions descending on the number of symbols in the subtree.

The upper curve in the graph shows the scaling results when applying very simple methods of selecting the granularity based on the page decomposition. Specifically, any block which contains more than $1/N$ of the total number of symbols on the page is decomposed and parallelized at the text-line level. Those blocks containing fewer than $1/N$ are queued, descending on the number of symbols. These whole blocks may then be used to fill idle time which would otherwise result as a long text line contained in another block is completed. Scaling using this hybrid strategy is much improved, particularly when more than 4 servers are used.

While the hybrid strategy produces scaling much improved over parallelizing at the block or text-line levels, there is still much room for improvement (*e.g.* only a scaling factor of 5.1 using 8 processors). Most of the remaining server idle time comes at the end of the page as the last task is completed by one server while the others are idle. Reducing this idle time further will likely require more complicated scheduling algorithms. For example, we currently make a decision on the parallelization level by examining the page up front, statically. A dynamic scheduling algorithm which takes into account actual processing times would undoubtedly make better use of the servers.

8. Representations of Ambiguity

A general-purpose data structure for document image analysis must provide a mechanism for representing ambiguity. We do not feel completely comfortable with our current methods for this task. As mentioned previously, we may resegment artwork owned by a word and carry multiple segmentations forward, each with the complete artwork. There is also the general issue of implicit versus explicit representation of segmentations. For example, the algorithms for resegmentation construct a lattice in which segmentations are implicit, yet we must convert to the explicit representation of the document hierarchy. We typically limit the number of such segmentations, discarding potentially useful information.

9. Graphics

We have built an editor for dim-files with its graphics displayed using the X Window System. This editor supports traversal of the document hierarchy at any level, while examining alternatives, confidence scores, and labeling. By default, nodes are visited in reading order (to the level established by the page reader), although they can be visited based on spatial properties, or by symbolic and logical labeling. Image artwork is displayed in various contexts, for example, within its block, text line, and word. The graphics are sufficiently general to cope with complex layouts, such as a mixture of horizontal and vertical text lines on the same page.

10. Discussion

We have described a document-image data structure (d/s), that expresses the geometric document image hierarchy (pages containing blocks of text-lines with words of symbols). The d/s can express both full and partial (incomplete) stages of analysis, and can express recursion (*e.g.* alternative resegmentations). It is complete, in that image, geometry, probabilities, symbolic codes, logical labels, etc, are all expressible in it. It is simple and regular in that there is no duplicated data, and no support for more than one expression of the same thing. We have implemented two isomorphic forms of the d/s. In main memory it is a linked-list hierarchy served by a systematic and regular set of functions in an object-oriented programming style (in C). Thus all functions see the same data structure no matter what program they run in. In peripheral files the d/s is a byte-stream, machine- and OS-independent, so that it can be moved around by `ftp`, `uucp`, etc. All programs see the same file structure no matter what shell script they run in.

This d/s has proven helpful in both research and development. Prototyping, systems integration, and evolution have all been aided. It supports rapid prototyping since it obviates the design of a special d/s from scratch each time. The simplicity of the d/s encourages disciplined design, yielding self-contained tools with well-defined semantics, and code that is easier to understand and maintain.

The result has been to allow the easy mixing and matching of tools in new combinations, for rapid attacks on new problems. Developers have the freedom to experiment with new tools in separately compiled programs, and later integrate them into a single program, with no code changes inside functions. The d/s can serve as a kind of 'blackboard' for passing intermediate results among tools, indicating, by its incomplete state, which action should occur next. This, in turn, allows an elegant and general exploitation of UNIX multi-processing.

11. Acknowledgement

Stimulating conversations with Tin Kam Ho and Larry Spitz are much appreciated. Greg Fabella wrote much of the multiprocessing control code and ran experiments on the SUN hardware.

12. References

[1] H. S. Baird, *Global-to-Local Layout Analysis,* in R. Mohr, T. Pavlidis, & A. Sanfeliu (Eds.), Structural Pattern Analysis, World Scientific, Singapore, 1990, pp. 181-196.

[2] H. S. Baird, S. E. Jones and S. J. Fortune, *Image Segmentation by Shape-Directed Covers,* Proc., IAPR 10th Int'l Conf. on Pattern Recognition, Atlantic City, NJ, 17-21 June, 1990, vol 1, pp. 820-825.

[3] H. S. Baird and R. Fossey, *A 100-Font Classifier,* Proc., 1st Int'l Conf. on Document Analysis and Recognition (ICDAR'91), Saint-Malo, France, 30 September - 2 October, 1991, pp. 332-340.

[4] H. S. Baird, *Background Structure in Document Images,* in H, Bunke (Ed.), Advances in Structural and Syntactic Pattern Recognition, World Scientific, Singapore, 1992, pp. 253-269.

[5] D. J. Ittner, *Automatic Inference of Textline Orientation,* 2nd Annual Symposium on Document Analysis and Information Retrieval, Las Vegas, Nevada, April, 1993.

[6] H. S. Baird, *Anatomy of a Versatile Page Reader,* IEEE Proceedings, July, 1992.

[7] D. J. Ittner and H. S. Baird, *Language-Free Layout Analysis,* Proceedings, 2nd Int'l Conf. on Document Analysis and Recognition, Tsukuba Science City, Japan, October 20-22, 1993, pp. 336-340.

[8] A. Dengel, *ANASTASIL: A System for Low-Level and High-Level Geometric Analysis of Printed Documents,* in H. Baird, H. Bunke, & K. Yamamoto (Eds.), Structural Document Image Analysis, Springer-Verlag, New York, 1992, pp. 70-98.

[9] J. Kreich, A. Luhn, & G. Maderlechner, *An Experimental Environment for Model-Based Document Analysis,* Proc., 1st Int'l Conf. on Document Analysis and Recognition, Saint-Malo, France, September 30-October 2, 1991, pp. 50-58.

[10] Y. Chenevoy & A. Belaid, *Hypothesis Management for Structured Document Recognition,* Proc., 1st Int'l Conf. on Document Analysis and Recognition (ICDAR'91), St.-Malo, France, 30 September - 2 October, 1991, pp. 121-129.

[11] D. L. Parnas, *On the Criteria to Be Used in Decomposing Systems into Modules,* in E. N. Yourdan (Ed.), Classics in Software Engineering, Yourdan Press, New York, 1979, pp. 141-150.

[12] I. T. Phillip, S. Chen, J. Ha, and R. M. Haralick, *English Document Database Design and Implementation Methodology,* Proceedings, 2nd Annual Symposium on Document Analysis and Information Retrieval, Caesar's Palace Hotel, Las Vegas, Nevada, April 26-28, 1993, pp. 65-104.

[13] The Unicode Consortium, *The Unicode Standard, Vols 1 & 2,* Addison-Wesley, Reading, Massachusetts, 1992.