



Active Document Versioning: from layout understanding to adjustment

Xiaofan Lin, Hui Chao, Greg Nelson, Elsa Durante
Imaging Systems Laboratory
HP Laboratories Palo Alto
HPL-2005-186
October 17, 2005*

variable data
printing, layout
adjustment, layout
understanding,
document
versioning

This paper introduces a novel Active Document Versioning system that can extract the layout template and constraints from the original document and then automatically adjust the layout to accommodate new contents. “Active” reflects several unique features of the system: First, the need of handcrafting adjustable templates is largely eliminated through layout understanding techniques that can convert static documents into Active Layout Templates and accompanying constraints. Second, through the linear text block modeling and the two-pass constraint solving algorithm, it supports a rich set of layout operations, such as simultaneous optimization of text block width and height, integrated image cropping, and non-rectangular text wrapping. This system has been successfully applied to a wide range of professionally designed documents. This paper covers both the core algorithms and the implementation.

* Internal Accession Date Only

Approved for External Publication

© Copyright 2006 SPIE. Published in SPIE Conference on Document Recognition and Retrieval XIII, 15-19
January 2006, San Jose, CA, USA

Active Document Versioning: from layout understanding to adjustment

Xiaofan Lin, Hui Chao, Greg Nelson, Elsa Durante

Hewlett-Packard Laboratories

1501 Page Mill Rd MS 1203, Palo Alto, CA 94304

Email: {xiaofan.lin, hui.chao, elsa.durante}@hp.com, greg@pernelson.org

ABSTRACT

This paper introduces a novel Active Document Versioning system that can extract the layout template and constraints from the original document and then automatically adjust the layout to accommodate new contents. “Active” reflects several unique features of the system: First, the need of handcrafting adjustable templates is largely eliminated through layout understanding techniques that can convert static documents into Active Layout Templates and accompanying constraints. Second, through the linear text block modeling and the two-pass constraint solving algorithm, it supports a rich set of layout operations, such as simultaneous optimization of text block width and height, integrated image cropping, and non-rectangular text wrapping. This system has been successfully applied to a wide range of professionally designed documents. This paper covers both the core algorithms and the implementation.

Keywords: variable data printing, layout adjustment, layout understanding, document versioning

1. INTRODUCTION

Automatic document layout technology is of great commercial interests because it can relieve or even eliminate the bottleneck of creating documents composed of highly customized text and image contents in end-to-end automated publishing solutions. It is also a very challenging technical problem since it involves 2-D optimization of positions and dimensions of multiple types of contents: images, texts, and vector graphics. Thus, there has been extensive research in this area. Jacobs et al [3] introduced an adaptive document layout system that automatically selects the best template for given contents. Purvis et al [5] formalized the creation of personalized documents as a multi-objective optimization problem and used a genetic algorithm to automatically assemble such documents. Johari et al [6] created a specialized pagination and layout system for yellow pages. Badros et al [2] proposed a constraint extension to Scalable Vector Graphics (SVG) to enable interactive graphics on the Web. Heydon and Nelson [7] developed the Juno-2 constraint-based drawing editor. Berkner et al [4] introduced a method to intelligently scale picture and text portions of an image by utilizing information available in the JPEG2000 file.

Obviously, automatic layout is a very wide area and no single algorithm can solve all of the challenges. One interesting sub-area is document versioning, whose goal is to automatically adjust an existing document layout to accommodate new contents. Adobe Acrobat’s Touchup function only allows very limited PDF text editing without changing the overall text line breaking. Another common approach used by commercial software packages such as Adobe InDesign [13] and QuarkXpress [14] is to define copy holes of fixed positions and sizes and then fill variable content into the copy holes. In order to avoid content truncation, each copy hole must be sufficiently large to hold the maximal possible amount of content. If the content varies significantly and there are multiple variable-data copy holes, the resulting layout may have unnecessary white spaces and cannot meet the required visual quality standard. To address this problem, some commercial products such as PageFlex [15] allow a user to manually design adjustable templates so that the positions and sizes of copy holes can adapt in the runtime according to variable content. However, this approach imposes heavy burden on the creation side: The graphic designer has to take into account all kinds of interactions between different objects to obtain rules governing adjustable templates. Although by training a graphic designer can easily figure out a

good concrete layout for a given set of content, he/she may find it difficult to formulate abstract layout rules and constraints that can work for different content combinations.

To deal with the above drawbacks of existing methods, we have created a comprehensive solution (see Figure 1) to document versioning by combining layout understanding and adjustment techniques. This system only requires standard documents such as PDF files as input. It then automatically extracts the text, image, and graphics objects from the input document and converts the original static document to an Active Layout Template (ALT), which uses variables to replace numeric constants in standard layout description languages such as XSL-FO and SVG. It also intelligently infers the constraints among the variables to govern the dimensions and positions of different objects. With the extracted ALT and the layout constraints, the layout adjustment engine can automatically modify the layout to accommodate new contents. Based on linear text block modeling and two-pass constraint solving algorithm, the layout adjustment engine supports a rich set of layout operations, such as simultaneous optimization of text block width and height, integrated image cropping, and non-rectangular text wrapping, as demonstrated in Figure 8.

Section 2 discusses various techniques on the layout understanding subsystem: layout extraction, constraint inference, and ALT generation. Section 3 is devoted to the layout adjustment subsystem: linear text block modeling, two-pass constraint solving, and layout generation. Section 4 summarizes this paper and points out future research directions.

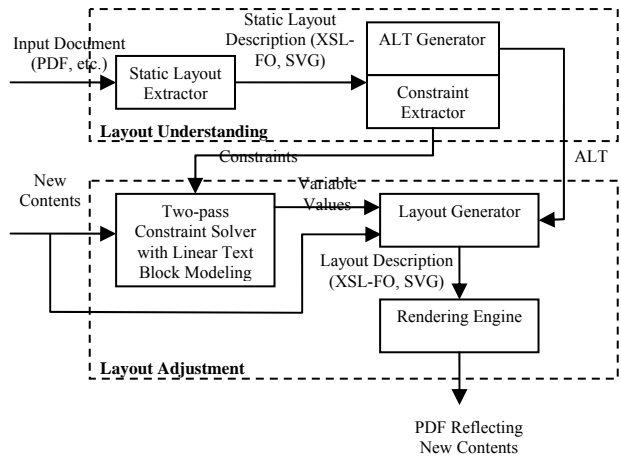


Figure 1: Comprehensive document versioning system

2. LAYOUT UNDERSTANDING

The template of a PDF page is extracted by first identifying logical blocks and associated layout styles. The outline and style of each logical block is described as a feature vector and expressed as an XML file. The layout relationships among different logical blocks are then analyzed and expressed as the constraints in another XML file. The following subsections discuss the layout extraction and constraint inference techniques respectively.

2.1 Static layout extraction

Although tagged PDF documents can have embedded logical structure information, most PDF documents are not tagged and thus do not express logical structure explicitly. As shown in Figure 2, a processing pipeline is built to automatically identify different logical blocks and associated styles. Overlaid blocks are very common in documents with rich graphical content. For example, a text block or a logo is embedded in an image. This makes the analysis more complicated. To minimize the potential effect of component overlay and the interference between different logical components, we first divide the page into three layers: text, image and vector graphics. Each layer then becomes an individual PDF document. To identify the logical blocks in each layer, we either directly analyze the PDF document or convert the PDF document into a bitmap image and perform image-based segmentation. The outline of each logical block is expressed as a polygon. For the identified polygon, the layout style within each outline is extracted and described as a style vector. For the text block, the vector includes the font name, font style, font size, line spacing, paragraph spacing, orientation, color, and block layering order. For the image block, the vector includes the

transformation matrix (for rotation and scaling purposes), compression format, layering order, and shape clipping path. For the vector graphics block, we identify the path objects within each component, convert them into SVG path objects, and create an SVG file for each vector graphics block. The style vector includes the outline of the block and laying order. Additional details for this work can be found in the reference [8].

Extensible Stylesheet Language Formatting Objects (XSL-FO) is used to store the final extracted layouts because it is an open XML-based document layout standard with strong support by open-source (for example, Apache FOP [10]) as well as commercial software. SVG objects can also be embedded in to XSL-FO to describe sophisticated text and graphics objects.

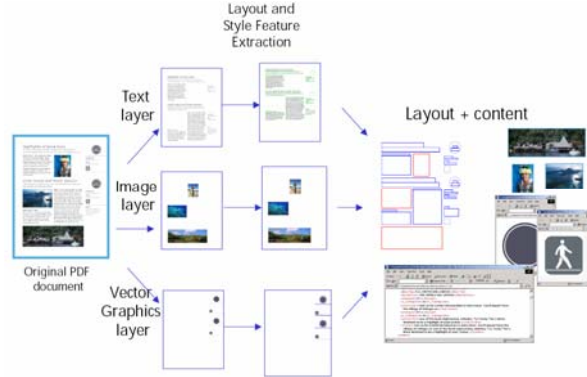


Figure 2: Template extraction processing pipeline

2.2 Constraint inference

The layout constraint rules are extracted to determine the position relationships among different logical blocks and relative to the page boundary. The constraints that describe how one block should be placed relative to other blocks include layering orders, gaps, overlaps, embedding relationships and alignments among blocks. The constraints that describe the overall page appearance include margins and bleeding effects. The alignments that run across page also affect the overall page look and feel. The extracted layout constraints are all linear equality or inequality constraints and they are prioritized using three levels of strengths: required, strong and weak.

2.2.1 Inter-block relationship constraints

Layering order determines the order of the logical blocks being placed on the page. This order is important when there are overlaid blocks. For example, if there is a text block sitting on the top of an image block, this order is preserved as a required constraint to govern which block should be rendered first.

A gap is the white space between two blocks. The gap relationship is considered only if the two logical blocks are adjacent with no other logical blocks located between the bounding rectangles of the two blocks. Although during document layout adjustment the gap between two adjacent blocks needs to be preserved and no intersection is allowed between the two originally separated blocks, a limited flexibility is allowed for the size of the gap during adjustment.

For two overlapping blocks, their layout relationship is locked. If the two blocks partially overlap, the constraint is set to fix the amount of the overlapping area. If one block is completely embedded in another block, the constraint is set to fix the ratio of the left margin to the right margin, and the ratio of the top margin to the bottom margin. In Figure 3, the bounding polygon of TB is embedded in the bounding polygon of ImgB, and thus the constraint should be:

`<constraint "rule=dT/dB= constant1" strength= "required">`

`<constraint "rule=dL/dR=constant2" strength= "required">`

where *constant1* and *constant2* are the ratios obtained from the original document.

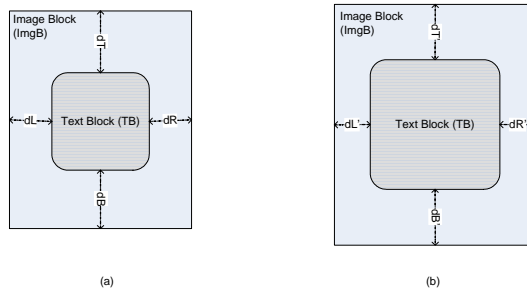


Figure 3 (a) The original document where a text block is embedded in the image block (b) A modified document where the layout relationship between the image and text block is preserved

Alignments also play an important role in the appearance of the document [17]. Left, right, center, top, bottom and middle alignments are considered. All alignments are originally set to be “required” unless there are multiple alignments. In case of multiple alignment relationships among logical blocks, the priorities of the constraints are set in accordance with a predefined reading order as follows:

- Top Alignment > Bottom Alignment > Middle Alignment*
- Left Alignment > Right Alignment > Center Alignment*

When there are conflicting constraints, the ones with lower priorities are reset to be “strong” instead of “required.”

2.2.2 Block-page relationship constraints

Page margins are the distances between the edges of a page and the edges of the minimal bounding rectangle of all non-bleeding blocks. Constraints are defined to make sure that all non-bleeding blocks stay within the page boundary. On the other hand, the margin values are flexible and set to be “weak” constraints with the original margins as references. For example, the constraint for the left margin would be:

```
<constraint rule=“left_margin=constant” strength= “weak”>
```

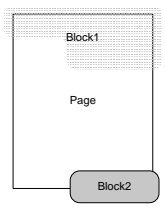


Figure 4: A page with two bleeding blocks.

A bleeding block has some of its edges extending beyond the page boundary. Correspondingly, constraints are set to preserve the bleeding effect. Strong constraints are set so that the distances between the edges of the page and the edges of the bleeding block are close to those on the original document. In Figure 4, the layout rule for Block 1 on the top should be:

```
<constraint rule=“Block1.top<page.top” strength= “required”>
<constraint rule=“Block1.top-page.top = constant” strength=“strong”>
```

Similar constraints are also generated for the left and right bleeding edges. Although most layout constraints are automatically inferred by the layout understanding subsystem, users or other intelligent programs are also allowed to add

or modify the constraints to satisfy special needs. For the document shown Figure 8, constraints governing are added by an automatic image cropping algorithm [11] to dictate how much the image at the bottom can be cropped.

2.3 Active Layout Template generation

Besides the core layout understanding algorithms, another important aspect is how to describe adjustable layouts. Although many standard formats, such as SVG, XSL-FO, are designed for static layouts, few formats are available for adjustable layouts. Badros et al have designed Constraint SVG (CSVG) [2] to support adjustable graphics by introducing variables into standard SVG. We extend this idea to convert almost any XML-based layouts to adjustable templates, called Active Layout Templates (ALT). In ALT, the numerical values of the attributes in XML layout documents are replaced with variables and expressions (see Figure 5). Special characters are introduced to avoid ambiguities and also to concatenate variables into expressions. In the current implementation, “##” means that the attribute value is generated for ALT purpose rather than as an ordinary string (for example, “absolute” in the following example). “!!” separates the symbol/expression from the unit (for example, “pt”). “+” is used in expressions to mean that the two sides of “+” should be added together.

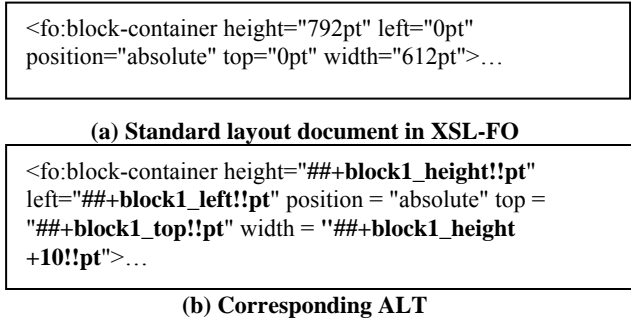


Figure 5: Conversion of standard layout document to ALT

In layout adjustment applications, the ALT is automatically generated on the basis of the original layout document using a converter. The converter parses the original layout XML file and replaces numerical values with expressions and variables according to the protocol introduced earlier. We have also defined a variable naming protocol. For example, if a block’s ID is “block1”, then the height will be named “block1_height” and the left coordinate will be named “block1_left”, etc. The same variable protocol is also followed by the constraint inference algorithm discussed in the previous subsection so that a variable cited in the ALT is always associated a variable used in the constraint file.

3. LAYOUT ADJUSTMENT

As shown in Figure 1, with the Active Layout Template and constraints from the layout understanding subsystem, the layout adjustment subsystem adjusts the original layout to accommodate the new contents. The next several subsections will describe the major techniques used in this subsystem. We will also discuss system implementation issues at the end of this section.

3.1 Linear text block modeling

As a key feature of the proposed system, individual text blocks are allowed to have variable widths in order to obtain the optimal layouts. However, such freedom in text block width poses a difficult technical challenge. When the text content in a rectangular block is fixed, the block’s width (w) and height (h) roughly follow a nonlinear relationship: $w * h = a$, where a is a constant. The exact relationship is even more complex. h is not a continuous function of w and instead follows a stepwise pattern, as shown in Figure 6.

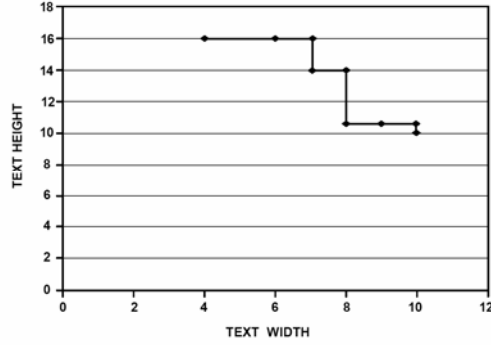


Figure 6: Non-linear relationship between the width and height of a rectangular text block, given the text content

This nonlinear relationship renders many efficient constraint solving methods such as Simplex useless. So our first strategy is to build linear models for the text block height-width relationships. If the adjustment is not too dramatic, we can use a single linear model to approximate the height-width function of a text block. Let us assume the original width of a text block is W_0 and the original height is H_0 . Then we attempt another width W_1 . By accessing the line-breaking function of the rendering engine, such as Apache FOP [10], we can get the new height H_1 . In this way the linear model between the width and height can be defined as:

$$H = H_0 + k * (W - W_0) \tag{1}$$

where $k = (H_1 - H_0) / (W_1 - W_0)$

In fact, the above linear model can also be applied to non-rectangular text blocks, which are very common in professional designs. Block A in Figure 8 is an example of such text wrapping.

3.2 Two-pass constraint solving

The linear models are approximate and cannot guarantee completely correct layouts. Thus, we run two passes of constraint solving. Using the linear models of the text blocks, the first pass decides the optimal width for each text block. Then through actual line-breaking, we can calculate the exact height for each text block. In the second pass, we fix the dimension of each text block and decide the final positions of the text blocks as well as the positions and sizes of the image blocks. Figure 7 shows the workflow.

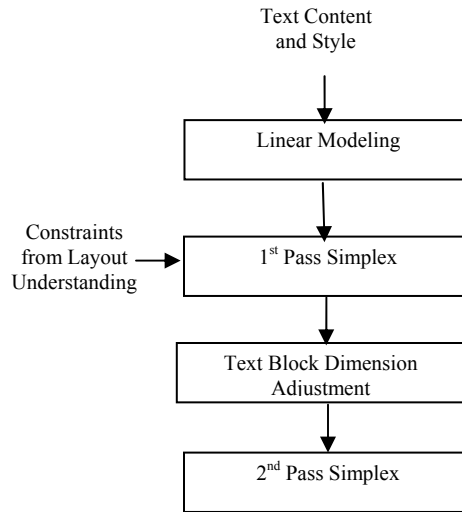


Figure 7: Workflow of two-pass constraint solving

The following example illustrates why the two-pass algorithm is necessary and how it works. The left side is the document containing the original contents. Then more text contents have been added to the two text blocks and the layout after the first pass is shown in the middle. As can be seen, this layout is roughly good with Block A expanded

horizontally and Block B expanded vertically. The only problem is that the bottom of Block A overlaps with the title text under it. This is due to the approximate nature of the linear models used in the first pass. After the second pass, the overlap problem has been corrected and the bottom image is cropped a little to make the extra room for the text content.

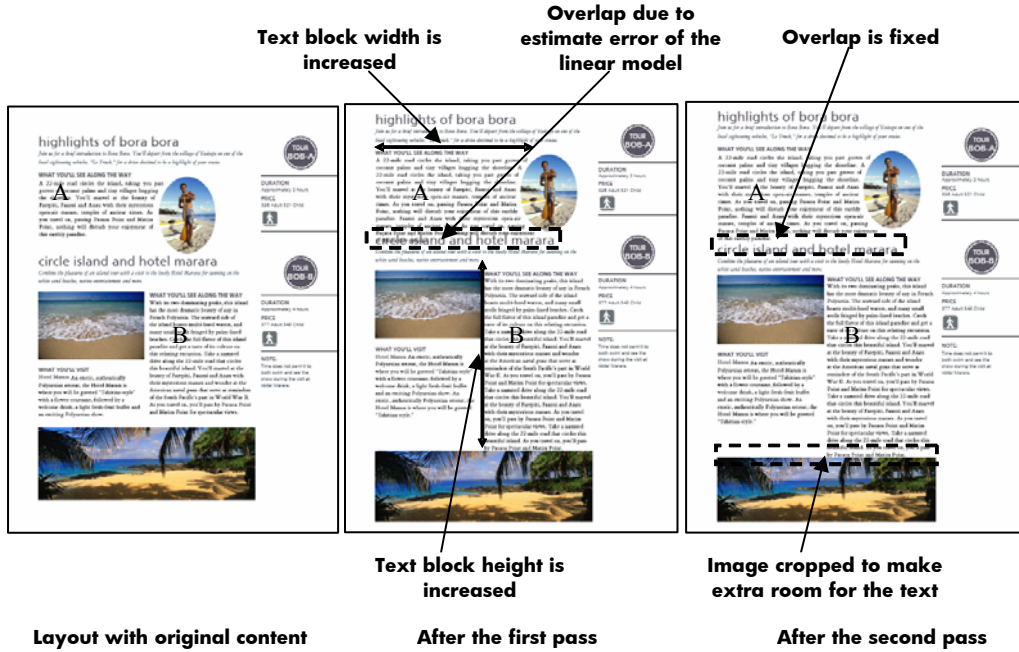


Figure 8: Example of the two-pass constraint solving algorithm

3.3 Constraint formulation

The constraints are formulated in the format specified by CSVG [2]. They are easy-to-understand infix expressions of equalities or inequalities. There are two parts of constraints in the first pass (see Figure 9). One part is the original constraints from the layout constraint inference component described in Section 2.2, and the other part is the approximate linear constraints based on the linear text block modeling. Figure 10 shows the solution after the first pass. Then we make corrections to the first pass results. As shown in Figure 10, based on the linear model, when the Δ width of the text block P0T5 ($P0T5_deltawidth$) is 48.72 (here Δ means relative value compared with that of the original layout), the height of P0T5 ($P0T5_height$) is 123.23. On the other hand, we can actually place the text into P0T5 under the condition that $P0T5_deltawidth$ is 48.72, and the height is calculated to be 162, which is different from the linear estimate. So we use height obtained from actual content placement to replace that based on linear model and generate the constraints for the second pass (see Figure 11). The constraint solver will produce the final solution (see Figure 12).

```

Original constraints:
<constraint rule="P0T2_left = P0I2_left" strength="required"/>
<constraint rule="P0T4_left = P0I2_left" strength="required"/>
<constraint rule="P0T5_left = P0I2_left" strength="required"/>
<constraint rule="P0I0_left >= P0T4_right" strength="required"/>
<constraint rule="P0I0_left >= P0T5_left" strength="required"/>...

Approximate linear constraints:
<constraint rule="P0T4_deltawidth = P0T4_right - P0T4_left - 186.48" strength="required"/>
<constraint rule="P0T5_deltawidth = P0T5_right - P0T5_left - 190.97" strength="required"/>
<constraint rule="P0T1_height = 13.93-P0T1_deltawidth*0.6966" strength="required"/>
<constraint rule="P0T2_height = 32.00-P0T2_deltawidth*0.0" strength="required"/>
<constraint rule="P0T3_height = 37.92-P0T3_deltawidth*0.9480" strength="required"/>
<constraint rule="P0T4_height = 11.85-P0T4_deltawidth*0.0" strength="required"/>
<constraint rule="P0T5_height = 189.00-P0T5_deltawidth*1.3500" strength="required"/>...

```

Figure 9: Sample CSVG file for the first pass Simplex

```

Text block dimensions:
P0T3_deltawidth = 13.63
P0T3_height = 25.00
P0T4_deltawidth = 0.0
P0T4_height = 11.85
P0T5_deltawidth = 48.72
P0T5_height = 123.23
P0T6_deltawidth = 0.0
P0T6_height = 11.85

```

```

Other variables:
P0T3_left = 510.53
P0T3_top = 61.56
P0T4_left = 56.32
P0T4_top = 80.91
P0T5_left = 56.32
P0T5_top = 94.15
P0T6_left = 468.34
P0T6_top = 111.99

```

Figure 10: Solution from Cassowary solver after the first pass: P0T5_deltawidth and P0T5_height comply with the linear constraint defined in Figure 9

```

P(Bi) related constraints:
<constraint rule="P0T2_left = P0I2_left" strength= "required"/>
<constraint rule="P0T4_left = P0I2_left" strength= "required"/>
<constraint rule="P0T5_left = P0I2_left" strength= "required"/>
<constraint rule="P0I0_left >= P0T4_right" strength= "required"/>
<constraint rule="P0I0_left >= P0T5_left" strength= "required"/>...

S(Bi) related constraints:
<constraint rule="P0T3_deltawidth = 13.63" strength= "required"/>
<constraint rule="P0T3_height = 18.96" strength= "required"/>
<constraint rule="P0T4_deltawidth = 0.0" strength= "required"/>
<constraint rule="P0T4_height = 11.85" strength= "required"/>
<constraint rule="P0T5_deltawidth = 48.72" strength= "required"/>
<constraint rule="P0T5_height = 162.00" strength= "required"/>
<constraint rule="P0T6_deltawidth = 0.0" strength= "required"/>
<constraint rule="P0T6_height = 11.85" strength= "required"/> ...

```

Figure 11: Sample CSVG file for the second pass Simplex: The linear constraint dictating P0T5_deltawidth and P0T5_height is now replaced by fixed values from actual text placement.

```

P0T3_left = 510.53
P0T3_top = 73.58
P0T4_left = 56.32
P0T4_top = 79.65
P0T5_left = 56.32
P0T5_top = 92.89
P0T6_left = 468.34
P0T6_top = 124.01 ...

```

Figure 12: Sample solution after the second pass Simplex

3.4 Layout generation

After the two-pass constraint solving engine has decided the optimal values for the variables that define object positions and dimensions, the layout generator can produce a standard layout document based on the Active Layout Template described in Section 3.3. The ALT is parsed using DOM or SAX APIs. Each attribute value is analyzed and the abstract variables and expressions are replaced with concrete numeric values from the layout adjustment engine and the result is a standard layout document in XSL-FO and SVG, which can be then rendered into PDF by Apache FOP [10] for printing and electronic distribution.

3.5 System implementation

The layout adjustment subsystem is implemented in Java and invokes a number of third-party Java packages (see Figure 13). Apache FOP [10] is an open-source software package that can parse and render XSL-FO document. In our system, it is extensively utilized in the text modeling and placement as well as the final PDF generation. We have chosen Cassowary solver [1] for the two passes of linear constraint solving because of its several attractive features. First, it supports non-required constraints, such as “strong” and “weak” constraints, in addition to the conventional required constraints. As described in Section 2, this feature is very convenient for layout adjustment purpose. Second, it supports efficient incremental constraint solving. It can be very useful to increase the processing speed of high-volume VDP applications by obtaining a new layout from the existing states of the solver rather than from scratch. Third, Cassowary solver has demonstrated impressive robustness and stability throughout our experiments.

In order to reduce the overhead of loading the engine and thus accelerate the layout adjustment, we have implemented the system in client/server mode. The heavyweight layout adjustment engine, including FOP and Cassowary solver, is started as a server in advance. It then listens to a particular TCP socket for incoming requests and executes the two-pass constraint solving algorithm. The layout adjustment client a very lightweight shell program that simply sends requests to the layout adjustment engine through TCP sockets. Under the client/server mode, it only takes a 2.8GHz Windows XP computer 6 seconds to adjust the document shown in Figure 8. The original execution time is 17 seconds without the client/server mode. In addition, because TCP/IP is a standard protocol supported by different programming languages, this approach also makes the layout adjustment engine accessible to client programs in various languages, such as C, Java, and C#.

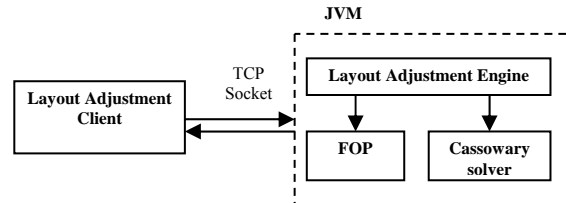


Figure 13: The client/server implementation of the layout adjustment subsystem

Besides, a Web interface has been designed to make the editing easier. The user can simply click on a text block or image block and then change the content. Then the updated contents are sent to the layout adjustment engine. After the new layout is produced, the browser will automatically refresh to display the new layout. Figure 14 is a snapshot of this interface. This system has been successfully applied to a wide range of professionally designed documents. Besides the vacation brochure example shown in Figure 8, Figure 15 is another example featuring a marketing flyer. The SVG-based black frames are expanded to accommodate extra text. The circular image and the text surrounding it move up together.

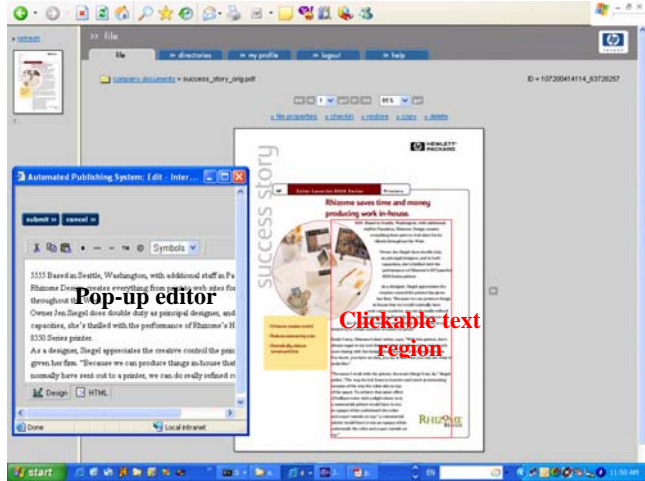
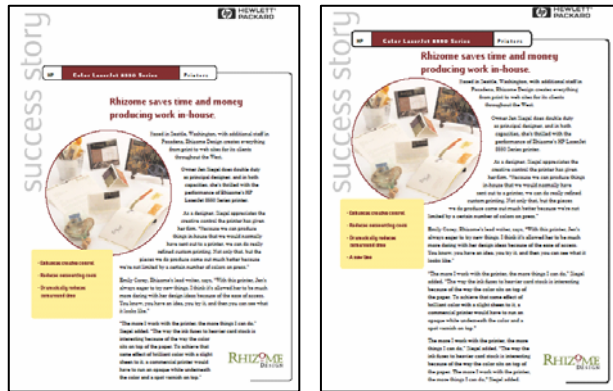


Figure 14: Web user interface of the system



(a) Original document (b) After layout adjustment

Figure 15: Another example of layout adjustment

4. CONCLUSIONS

Through several innovations such as layout template and constraint extraction, linear text block modeling, two-pass constraint solving algorithm, the proposed Active Document Versioning system provides a robust and generic solution to relieve the bottleneck on the creation side of end-to-end automated publishing pipeline. It supports a rich set of layout adjustment operations in professional graphic designs, such as simultaneous optimization of text block width and height, integrated image cropping, and non-rectangular text wrapping. On the other hand, there are a number of promising future research topics on both layout understanding and adjustment:

As with any task related to artificial intelligence, layout template and constraint extraction is not and will not be perfect. There are a number of other factors that also have a lot of influence on the look and feel of the page, such as the repetitive patterns, colors, visual balance, and they are yet to be studied. Also, sometimes the interpretation of a graphic design's intent is quite subjective. So we should allow the user to edit the automatically discovered rules through a friendly graphic user interface.

The current experimental system runs on a single machine. In the high-throughput production environment, it is imperative to distribute the layout engine on multiple machines. A number of technical questions then need to be answered: How to do the load balancing? How to synchronize the processes across different computers? What distributed computation protocol (CORBA, Web Service, etc.) should be adopted? This is another direction of our future research.

ACKNOWLEDGMENTS

The authors are grateful to Greg Badros and Alan Borning for their help with the Cassowary solver. Steven Simske has kindly provided us with the software of bitmap image analysis. Anna Durante, Gary Vondran, and Henry Sang have supported this research from the start.

REFERENCES

1. G. Badros and A. Borning, "The Cassowary Linear Arithmetic Constraint Solving Algorithm: Interface and Implementation," *Technical Report UW-CSE-98-06-04*, University of Washington, Seattle, Washington, June 1998.
2. G. Badros, J. Tirtowidjojo, et al, "A Constraint Extension to Scalable Vector Graphics," *Proceedings of Tenth International World Wide Web Conference*, Hong Kong, May 2001.
3. C. Jacobs, W. Li, et al, "Adaptive Grid-based Document Layout," *ACM Transaction on Graphics*, vol 22 no 3, 2003, pp 838-847.
4. A. Berkner, EL. Schwartz, "SmartNails: Display- and Image-dependent Thumbnails," SPIE Conference on Document Recognition and Retrieval XI, pp. 54-65, San Jose, USA, January 2004.
5. L. Purvis, S. Harrington, et al, "Creating Personalized Documents; An Optimization Approach," *ACM Conference on Document Engineering*, 2003, pp 68-77.
6. R. Johari, J. Marks, et al, "Automatic Yellow-Pages Pagination and Layout," *Mitsubishi Electric Research Laboratory Technical Report TR-96-29*, 1996, <http://www.merl.com>.
7. A. Heydon and G. Nelson, "The Juno-2 Constraint-based Drawing Editor," *Technical Report 131a*, Digital Systems Research Center, Palo Alto, USA, December 1994.
8. H. Chao and J. Fan, "Layout and Content Extraction for PDF Documents," *International Workshop on Document Analysis Systems*, Florence, Italy, September 2004.
9. <http://www.w3.org/TR/xsl/>
10. <http://xml.apache.org/fop>
11. P. Cheatle, "Automated Cropping of Electronic Images," *US Patent Application 20020191861*.
12. <http://www.cs.sandia.gov/opt/survey/mip.html>
13. <http://www.adobe.com>
14. <http://www.quark.com>
15. <http://www.pageflex.com>
16. H. Chao and X. Lin, "Layout and Content Extraction from PDF Documents," *7th International Conference on Document Analysis and Recognition*, August 2005.
17. S. Harrington, J. F. Naveda, et al, "Aesthetic Measures for Automated Document Layout," *ACM Conference on Document Engineering*, 2004.