Toward Automated Document Reformatting: A SGML Markup System

Julia C. Lee[1]
Argonne National Laboratory
9700 S. Cass Ave DIS/900
Argonne IL 60439-4832
lee@dis.anl.gov

Craig E. Swietlik[1]
Argonne National Laboratory
9700 S. Cass Ave DIS/900
Argonne IL 60439-4832
swietlik@dis.anl.gov

Abstract

A SGML markup system is presented. One major obstacle for the SGML to gain more application is the prohibitively high cost of the markup process. The system we are to present adopts an incremental design approach. This approach helps to "divide and conquer' each specific problem encountered during the markup process and ensures that the system converges to a almost-fully automated markup system. The major software components of the system are described. Some selective algorithms are also introduced.

Key Words: Multimedia, Hypertext, SGML, Heuristic Approach, Text Processing System.

---

1. Introduction

SGML is a international standard markup language for processing text and other types of information [1]. The major initial purpose of the standard was to allow for an easy exchange or reformatting of textual information [2]. However, researchers are still trying to explore the full set of application opportunities that the SGML standard can offer. One concrete example is the HTML markup used by the World Wide Web (WWW) for presenting multimedia information [3]. Markup language can preserve much of the physical and logical context of the information. In turn, contextual information can support greater levels of control in the querying process of text-retrieval.

One major obstacle for the SGML to gain more application is the prohibitively high cost of the markup process. Generally, documents found in text-retrieval systems vary greatly in both their physical appearance and their logical structure. It is, therefore, difficult to automate the process of marking the documents. The emerging tools for increasing the degree of markup automation are of two types: interactive and batch. Interactive markup utilities are usually augmented with a visual text-editor. Batch markup takes documents and some other help file(s) as input file and outputs the marked documents with certain types of manual pre-processing and/or post-processing [4]. Though the degree of automation provided by these tools varies greatly, all require manual pre-processing, interactive or post-processing steps.

We are proposing a new markup system or environment which tries to maximize the degree of markup automation. We have adopted an "incremental" development strategy. The environment is modified and improved as more documents are processed. Manual intervention may never be completely eliminated from the mark up process, yet measurable convergence toward full automation should be possible.

We use "interactive" as well as "batch" approaches, but our "interactive" is different from the one mentioned above. Users are not asked to interact with the text-editor. Instead, we let the different components of the system interact themselves based on a set of heuristic rules. Information pertaining to the construction of the documents is extracted by system components and modified or used by other components. The information of document construction may, in fact, feed-back to the components which creates them initially. The batch approach in our system is the same as found in other markup tools. Since the action of the subsequent steps depend on the result of the previous steps and the condition of the documents, this system is a heuristic approach from a large perspective.

We also use an algorithmic approach. The algorithms for

each of the components are carefully designed. In particular, finite state automata, regular expressions and double linked dynamic lists play important rolls in the construction of the system.

In Section 2 we introduce the major software components of the system. Section 3 describes some detail of the "incremental and heuristic" approach. Section 4 introduces some selective algorithms used in the software components. Section 5 is the closing remark.

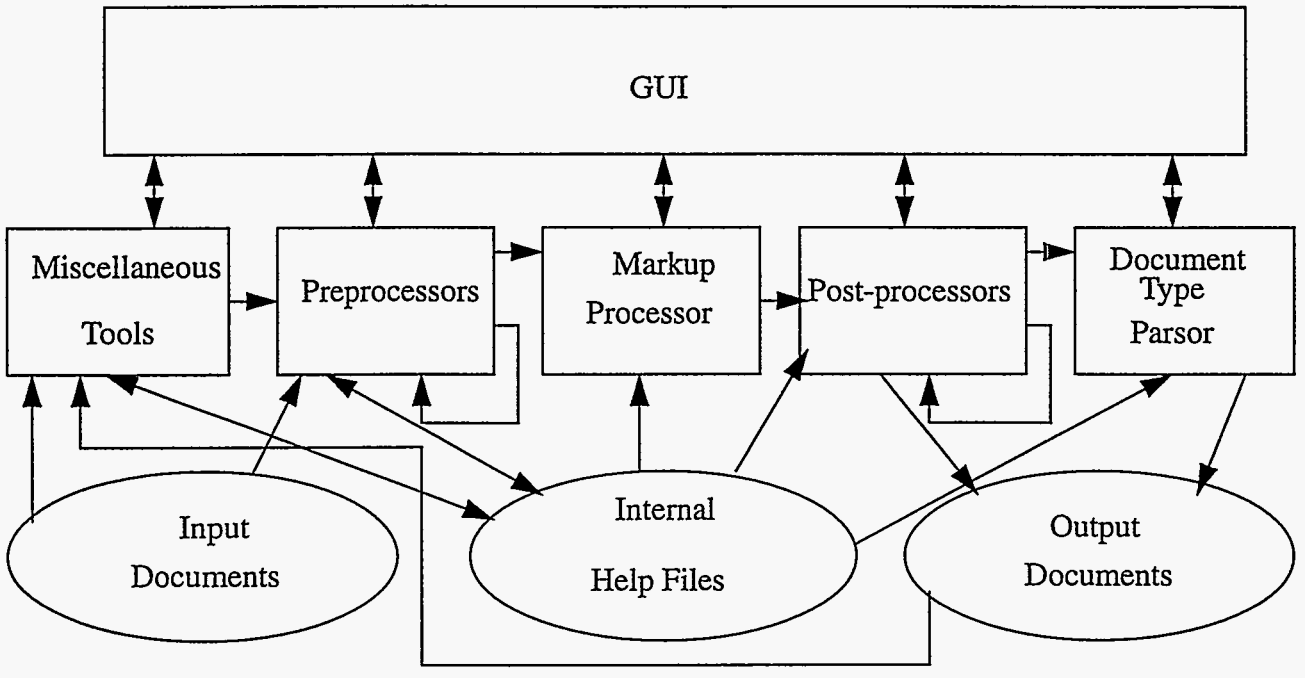

Figure 1. Software components of the markup system

2. Software components

The major software components are depicted in Figure 1. above. The system includes a graphical user interface (GUI) which will allow users to choose from different classes of input documents and different status of the documents. The GUI also provides a convenient and efficient way for verifying the markup created and for checking the internal help files needed by other system's components

Other components include: (1) GUI (described above), (2) Miscellaneous tools such as a text editor which can be used for examining or manually modifying the input/output documents and the internal help files. Some of the tools are off-the-shelf tools. We merely integrate them into our system. Some of them we created to meet the specific needs of the system. (3) A group of preprocessor which preprocessing different types of documents and partially create the system's internal help files. They also modify or "repair" the format of the input document making them easier to be processed by the markup processor. (4) The marking-up processor marks the documents with predefined tag symbols with the help of the internal help files. One of the internal file is to map the symbol used by the document definition to the symbols used by the markup processor [1][5][6]. (5) A group of post processor which adds additional information to the marking up such as attributes and document formatting information, etc. (6) The parsor and a group of utility programs verifies the correctness of the markup and restructure, and reformats or decomposes the marked document into smaller logical or physical components according to the choice of the user.

As one can see in Figure 1. all the components except the GUI are invoked by the GUI for a user friendly environment.

3. The Incremental and heuristic approach

All the components are designed such that they are easy to be modified and that new features are easy to be added.This is one of the criteria for an incremental design strategy.

Other criteria for incremental design and implementation include making each individual part of the components as small as possible. Each small part performs a specific task. This gives a large degree of flexibility for "assembling" parts into components with different features and serving different types of documents.

Individual parts are designed and implemented when specific requirements or different types of documents are encountered.When a different type of documents is encountered, a new group of parts is formed to process this type of documents. The group may include both existing parts and

new parts to be designed and implemented.There are tools for "assembling" the different parts together.

While each component is designed with a clear deterministic algorithm, the entire system is based on a heuristic approach. For most of the documents we have processed there were no definite processing steps known at the beginning of the process. The result of the previous steps could give a feasible choice for next step. Manual interventions are also involved from time to time to resolve some "abnormal" situations.

To be more specific on the heuristic approach, we describe how one of the functions or tasks of the preprocessors is carried out. One tool is used for generating a initial "Logical Structure Definition File" (LSD File) based on the percentage of the capitalized characters in the lines of a document. The initial LSD files are normally erroneous containing a large number of misplaced lines. A group of other tools are invoked based on certain clues contained in the initial LSD File. For example if a capitalized "Acronym" is found in the initial LSD file, it is likely that a large number of misplaced lines will follow this line. A tool is designed for cleaning these lines. A "manual" observation or a "grep" of the word can be used to determine whether or not this particular tool needs to be invoked. Some other tools may be more "general" and can be always invoked for clean up. The finalization of the LSD file is done manually since it is a important internal help file used by the markup and the post-processing processes.

One may be wondering about the efficiency or value of the tools since each tool can only do a very limited task and may ask why we still need these "trivial" small tools when manual processing is still required. To answer these questions, we would like to point out the following important facts:

(1) Although there will still be manual involvement, the amount of manual involvement makes a big difference. If for one document the original manual processing needed 8 hours and if manual processing reduced the time to 2 hours after the tools are used, a four-time efficiency is gained. This could result in months or years of savings in man power and tens or hundreds thousands of dollars in costs when a large amount of documents needs to be processed.

(2) It is almost impossible to design one algorithm to automate the SGML markup process. However, the heuristic integration of the small tools designed and implemented incrementally could gradually reduce the amount of manual intervention and make the system converge to a fully or almost-fully automated markup processing system.

This type of combination of formal and informal (or heuristic) methods has been adopted by many system implementation in information processing and/or computer system practice.

4. Selection of algorithms

The functions of the parts (or tools) in our system are very much diversified. The algorithm of each part is very different. Some of them are pretty straight forward. Some of them took much thought to design.

We would like give a few examples in the following. If the algorithm falls into a certain well-known category, we just point out the well-known category and briefly describe the specific variation in our system.

4.1 Finite state machine

In many cases the same function can be carried out by different algorithms. The design of an algorithm is not only the embodiment of the effort of the designers but also the style of the designers. In many cases we are in favor of a finite state machine approach since many seemingly "obscure" algorithms can be converted into a more clear finite state machine with a certain number of states. In fact almost all the conditional branch statement in a program language can be replaced by a "switch" statement. It is, of course, not the scope of this paper to prove this claim. Finite state machine is a clear way to formalize a software process [7], but we do not mean that it is simple. A finite state machine can be fairly complex involving large number of states, substates and state transition conditions.

A typical finite state machine approach in our system is the "markup" process. The markup process in our system is a "large finite state machine". It has over 120 states and substates and over 400 possible state transition and associated conditions. To define, analyze and to test all these possible transitions is a challenging job. Structuralization and top-down decomposition are used in our design process.

Figure 2 is a small part of the state transition diagram for our markup process. This part illustrates how the ordered lists are handled by our markup process. The oval shapes represent states and sub states. The arrows represent state transitions. Conditions are marked aside of the transitions. Squire shapes are for sketching the corresponding actions to be taken under that state transition.

4.2 Regular expression and context-free grammar

Regular expressions are used through out the tools sets of our system for pattern-matching. It is also used in Document Type Definition(DTD) files for SGML marking and parsing. In fact the SGML standard is defined by using generic regular expressions [1][5][6].

The following is a segment of a DTD file used by our system:

```
<!--   ELEMENTS   CONTENTS          -->
<!ELEMENT dc  - - ((#PCDATA)?, (eas? | fns?)+) >
<!ELEMENT fns       - o (ffm,fbd,fbm?)   >
<!ELEMENT ffm       - o (hd,org?,act?)  >
<!ELEMENT org       - o (#PCDATA)  >
<!ELEMENT act       - o (#PCDATA)  >
<!ELEMENT fbd       - o (s+)  >
```

```
        ELSE
            PRINT line to output;
        END IF;
    END LOOP if EOF is reached;
END.

DO_TABLE
BEGIN
    READ parameters of the table (e.g. max_column
        and formatting information);
    LOOP on each line of the document after start-table
            marker
        CHECK for commented lines
        IF commented line found THEN
            PRINT the line into output;
        END IF;
        CHECK each line for blank line;
        ELSE IF this_blank == 1 AND last_blank == 0
        AND column counter == max_column THEN
            FORMAT the multiple-text-line table row
                stored in the row_content array;
            PRINT the table row into output
            SET column counter to 0;
        IF this_blank == 1 AND last_blank == 0 AND
        column counter != max_column THEN
            SET text_line counter to 0;
            INCREASE column counter;
        ELSE IF this_blank == 0 AND last_blank == 1
        THEN
            STORE the text line into the row_content
            array for current row and current column;
            IF column counter == 0 THEN
                SET column counter to 1;
            END IF;
        ELSE IF this_blank == 1 AND last_blank == 1
        AND start_blank_row == 0 THEN
            SET start_blank_row to 1;
        ELSE IF this_blank == 1 AND last_blank == 1
        AND start_blank_row == 1 THEN
            INCREASE column number;
        ELSE
            STORE the text line into the row_content
            array for current row and current column;
            INCREASE text_line counter;
        END IF;
    END LOOP if end_table marker is found;
END;
```

The above algorithm is only for one type of the many table-formatting/re-formatting tools.

5. Remark

We have presented a SGML markup system. We described the major components and the design strategy of the system. We also selected some interesting algorithms to give a more detailed view of the components in the system. We feel that the design and implementation strategy we adopted could lead to convergence of the system to a fully or almost-fully automated markup system, and manual intervention of the markup process may be eliminated little by little although may not be completely eliminated.

This system will increase the productivity of the markup process a great deal. We have experienced the increase by using the current implementation of the system in our document processing practice.

The work discussed in this paper was partially supported by the U. S. Department of Energy under contract W-31-109-Eng-38.

References

[1] Department of Commerce/National Institute of Standards and Technology; Standard Generalized Markup Language (ISO 8879-1986(E); Federal Information Processing Standards Publication; Sept. 26, 1988. 1-58.

[2] Department of Commerce/National Institute of Standards and Technology; Annex A - Introduction to Generalized Markup (ISO 8879-1986(E); Federal Information Processing Standards Publication; Sept. 26, 1988. 59-65.

[3] T. Berners-Lee, R. Cailliau, A. Luotonen, H. F. Nielsen, and A. Secret; The World-Wide- Web; Communications of the ACM; Vol.37, No.8; August 1994; 76-82.

[4] Interleaf Inc. "Avalanche FastTAG" User's Guide; "Avalanche FastTAG" Manual; Interleaf, Inc.;1995; 1-1 - 6-6.

[5] Department of Commerce/National Institute of Standards and Technology; Annex B Basic Concepts (ISO 8879-1986(E); Federal Information Processing Standards Publication; Sept. 26, 1988.66-93

[6] Department of Commerce/National Institute of Standards and Technology; Annex C - Additional Concepts (ISO 8879-1986(E); Federal Information Processing Standards Publication; Sept. 26, 1988.94-109.

[7] J.E. Hopcroft, J.D. Ullman; Introduction to Automata Theory, Languages and Computation;