ELECTRONIC PUBLISHING, VOL. 3(2), 93–98 (MAY 1990)

On improving SGML

MICHAEL J. KAELBLING
Siemens AG, ZFE IS EA 11
Corporate Applied Computer Sciences
Otto-Hahn-Ring 6
8000 Munich 83, FRG

SUMMARY

Several improvements are suggested to the syntax of SGML, the recent international standard for the description of electronic document types. These improvements ease processing by existing tools, remove ambiguity cleanly, and increase human usability. They also indicate some guidelines that should be followed in the design and specification of computer-software standards. By following accepted computer-science conventions for the description of languages the design of a standard may be improved, and the subsequent implementation task simplified.

KEY WORDS Ambiguity LALR(1) Grammars Language Definitions Parsing SGML Standards

INTRODUCTION

The Standard Generalized Markup Language (SGML) is an international standard for the description of document types. Since its adoption in 1986 by the International Organization for Standardization (ISO) as ISO 8879 [1], SGML has become the fastest-selling standard in ISO history [2]. It is used by industry in production systems [3, 4], and it is used by academic and industrial research efforts as a basis for developing systems [5, 6]. SGML is a valid subject for analysis by computer scientists because: it is intended for computer-based processing, it is used in industry and academia, and it is increasingly popular. Furthermore, observations made on improving the SGML standard can serve as useful guides to the creation of other standards.

In this paper, changes will be suggested for improving the technical quality of SGML's specification. As the grammar describing SGML has over 200 productions and many pages of explanation, elaboration and clarification, it is inappropriate to present a detailed description here. Therefore the interested reader is referred to the standard itself for a detailed description of SGML. The complete description, however, is not necessary for an understanding of the points to be made—instead, we will use simplified descriptions and excerpts from the standard. It will be shown that some of the standard's features make the document types it describes difficult to process with existing tools, or unnecessarily restrictive of a user's options; improvements will be suggested to overcome these problems. We will discuss how the SGML specification may be changed to allow for processing by existing tools, for clean removal of ambiguity, and for increased human usability.

IMPROVING THE SYNTAX OF THE STANDARD

Since SGML is a language intended for computer-based systems, it is reasonable (in

0894–3982/90/02093–06$5.00

Received 16 March 1988
Revised 18 May 1990

the absence of convincing argument to the contrary) that it should follow established conventions within the realm of computer science. By following accepted methods of notation and structural configuration, languages can be processed by existing tools following well-understood theories and techniques, thus offering considerable savings to development efforts. These savings are realized by automatically performing tedious, error-prone calculations correctly and by checking and clarifying the formal descriptions of the languages.

Parsing, the process of recognizing languages, is a major concern of computer science, and has been the subject of considerable study [7]. There are several common techniques of parsing, each with known properties and limitations. Some of these techniques are so well established that programs exist that can automatically generate parsers for conforming language descriptions. The use of these parser generators, or compiler-compilers, goes a long way to ensuring the rapid production of reliable, efficient parsers. Furthermore, the theoretical underpinnings of parsing serve to identify ambiguity that must be removed from a language description.

It is not within the purview of this paper to painstakingly examine and report on the entire SGML specification, but a discussion of several major points should give the reader a feeling for the whole. Therefore, some observations will be made on easing the processing of the grammar by existing tools, and a brief discussion will follow on how to further improve a part of the specification syntax dealing with SGML attributes.

Use an established model and process with existing tools

Although the standard specifies a grammar that is not LALR(1),[1] we believe it is both possible and advantageous to specify SGML with an LALR(1) grammar. In addition to our own experience, the success of the Amsterdam SGML parser [8], which is based on the less powerful LL(1) technique, indicates that an LALR(1) grammar for SGML is possible. By specifying SGML as an LALR(1) grammar, developers would reap the time-saving, reliability and design benefits associated with the use of existing tools for the processing of such grammars. We will now discuss how the SGML syntax can be changed to allow for processing by existing tools.

The processing of a string begins with lexical analysis. The current convention is to enable the lexical analyzer (scanner) to return tokens without concern for left- or right-context. SGML has characteristics that make this direct[2,] approach difficult. For example, SGML has overlapping token classes: ‘1130’ can be either a ‘number’, or a ‘number-token’, or a ‘name-token’, depending on the context. Some scanner generators (like Lex [9]) can, with added effort, produce indirect lexical analyzers; others (like GLA [10]) cannot. Processing of strings continues with syntactic analysis. There are several general-purpose tools that generate syntactic analyzers (parsers). Yacc [11], perhaps the most commonly available generator, and others, like Xerox’s PGS [12], generate LALR(1) analyzers.

[1] LALR(1) is the name of a parsing technique that uses a single look-ahead to guide a left-to-right parse of a string. This technique is a compromise between the power of full LR(1) parsing, and the desire to limit the size of a parser.

[2] Using the terminology of [7], a *direct* lexical analyzer is told where to look for the next token; an *indirect* lexical analyzer must also be told what type of token to look for next.

SGML, as currently specified and amended [13], cannot be parsed by LALR(1) techniques. SGML is specified in a type of extended Backus–Naur form that can readily be converted to standard Backus–Naur form (BNF) [14]. After a mechanistic conversion to BNF, we attempted to use Yacc to generate an LALR(1) analyzer for SGML, and were warned of hundreds of conflicts.[3]

We took the grammar as specified, and rewrote it as an LALR(1) grammar by defining non-overlapping token classes and simplifying the production rules. Our version has 19 shift/reduce conflicts (for which the Yacc default of shifting is the correct choice) and zero reduce/reduce conflicts. This rewrite allowed Yacc to produce a parser for SGML specifications.

name group = **grpo**, *ts*∗, *name*, (*ts*∗, *connector*, *ts*∗, *name*)∗, *ts*∗, **grpc**

Figure 1. A sample SGML rule containing separators

The main problem with the ISO standard's specification of the grammar is the role and placement of separators. The separators, which the standard says are to be ignored, are written into the very productions that are to ignore them. Figure 1 is a sample SGML rule containing separators. The *ts*'s are the separators; the commas indicate concatenation, and the asterisks indicate zero-or-more instances. The presence of separators in the rule causes a problem for single-look-ahead parsers. Consider the following string (the separators are explicitly indicated): '**grpo** `name1` *ts* | *ts* `name2` *ts* **grpc**'. When the parser considers '`name2`' it does not know if it should continue accepting names. It asks for a look-ahead, but unfortunately this look-ahead does not resolve the dilemma. The *ts*-token the parser gets as its single look-ahead could either precede a connector and another name, or it could precede the **grpc**. A parser would require a second look-ahead (or possibly more) to correctly continue.

name group = **grpo**, *ts*∗, *name*, *ts*∗, (*connector*, *ts*∗, *name*, *ts*∗)∗, **grpc**

— or —

name group = **grpo**, *name*, (*connector*, *name*)∗, **grpc**

Figure 2. Improved versions of the rule in Figure 1.

Two ways to avoid this difficulty are shown in Figure 2. The rule at the top of the figure has been written in a way that leaves the language unchanged, but allows for LALR(1) parsing. The rule at the bottom of the figure will also accept the same language, if the lexical analyzer is allowed to expand the *ts* separators and skip over the whitespace. This latter approach has the advantage of reducing the size of the parser that accepts the language, and increasing its speed.

[3] A conflict occurs when the analyzer generator must arbitrarily decide what action to take in an ambiguous situation. A shift/reduce conflict occurs when a choice must be made between continuing with a production rule, or accepting a different rule. A reduce/reduce conflict occurs when a choice must be made between several equally acceptable rules. We were warned of over 500 shift/reduce conflicts, and over 600 reduce/reduce conflicts for a grammar with 555 rules.

Remove ambiguity cleanly

In addition to being difficult to process, the grammar of the SGML specification is ambiguous. Figure 3 is a simplified model of the ambiguous, SGML syntax for an attribute list. The simplifications serve to highlight the problem of ambiguity, and in no way contribute to it. A brief examination of this syntax will make the problem apparent: in a list of names, the attribute names cannot be syntactically distinguished from the attribute values. In this section we will see how simple modifications to SGML can cleanly remove ambiguity and increase human usability.

```
rule 1a <attr list> ::= <attr list> <attr spec>
rule 1b                 | empty
rule 2a <attr spec> ::= <name> = <name list>
rule 2b                 | <name>
rule 3a <name list> ::= <name list> <name>
rule 3b                 | <name>
```

Figure 3. A simplified BNF version of the SGML standard's grammar for attribute lists

The SGML standard attempts to offer relief from this dilemma by requiring that *rule 2b* can only be used when the name is one of an enumerated type. A further prohibition insures that the name is valid for only one of the attributes for a given tag, i.e., no two enumerated types have the same items. For example, if '`memo`' is a tag with an attribute '`status`' that can be either '`draft`' or '`final`', then a second enumerated-type attribute, say '`notice`', cannot use either of those words. One could not say that '`notice`' can be either '`first`', '`second`', or '`final`' since '`final`' is reserved for '`status`'; one would have to substitute a word like '`last`'. Such a restriction on vocabulary conflicts with human usability since it is possible that unusual or unnatural words will have to be substituted for familiar words.

Furthermore, the steps taken by SGML to avoid ambiguity are not sufficient. Assume that the '`memo`' tag has a third attribute, '`keywords`', of type 'list-of-names' (which is not an enumerated type). Now, consider the string '`<memo keywords = ISO first draft>`'. Is '`first`' a keyword, or is it implying '`notice = first`'? Is '`draft`' a keyword, or is it implying '`status = draft`'? Ambiguities of this type are not addressed by the standard. One solution would be to forbid the use of reserved words in the '`keywords`' list, but such a solution is awkward and, again, counter to the goal of human usability. SGML allows the use of attribute value literals to disambiguate such strings, but the standard does not indicate what action to take should an author with imperfect knowledge of all attributes and reserved words neglect to disambiguate such strings. Furthermore, it is undesirable to burden an author with a task that is otherwise easily accommodated.

A solution to the ambiguity problem is proposed here to demonstrate how simple modifications in the syntactic and semantic specification of SGML can greatly enhance automatic processing and human usability. First, allow *rule 2b* only when it is unambiguous which attribute takes that value. This would allow for the reuse of natural words. Second, respecify the syntax for attribute lists along the lines of the grammar rule shown in Figure 4. The slight change at the end of *rule 1a′* makes a huge difference

in removing ambiguity from the language for attributes, and it allows lists to contain whatever words one chooses. For example, the string '`<memo keywords = ISO first; draft; >`' is easily parsed.

rule 1a′ `<attr list> ::= <attr list> <attr spec> ;`

Figure 4. An improved grammar rule for attribute lists

CONCLUDING REMARKS

The specification of a standard should reflect the state of the art. To this end, the grammar specifying SGML should be rewritten to allow for automatic processing by common tools and techniques. Processing by existing and generic tools will lead to correct and efficient implementations produced with a minimum of effort. Had automatic tools been used to process the initial specification of SGML, unnecessary ambiguities would have been more readily discovered.

To avoid the problems of unreachable productions, undefined productions, and ambiguity, the next edition of the SGML standard should be verified by applying known tests and techniques. Such testing is not burdensome given the availability of tools that perform these functions. This advice applies as well to all standards intended for computer-based processing.

ACKNOWLEDGEMENTS

This work was supported, in part, by the Applied Information Technology Research Center, and by an equipment grant from Xerox Corporation to the Chameleon research group at The Ohio State University. Thanks go also to Frank Glandorf and to R. Hayter for careful readings of a preliminary version of this paper.

A special acknowledgement goes to A. E. K. Sobel, Ph.D.

REFERENCES

1. *Information Processing—Text and Office Systems—Standard Generalized Markup Language (SGML)*, ISO 8879-1986 (E), 1st edn., International Organization for Standardization, October 1986.
2. J. M. Smith, 'SGML Update III', *SGML Users' Group Bulletin*, **2** (1) 48–49 (1987), National Computer Centre, Oxford Road, Manchester M1 7ED, United Kingdom.
3. *Standard for Electronic Manuscript Preparation and Markup*, Association of American Publishers, Washington, DC, 1986.
4. Special Issue devoted to the Association of American Publishers' Electronic Manuscript Standard, *Electronic Publishing Business*, **4** (8), 1–32 (1986).
5. D. D. Chamberlin, H. F. Hasselmeier and D. P. Paris, 'Defining Document Styles for WYSIWYG Processing', Proceedings of International Conference on Electronic Publishing (EP88) Nice 1988 ed. J. C. van Vliet, pp. 121–137.
6. S. A. Mamrak, M. J. Kaelbling, C. K. Nicholas and M. Share, 'A Software Architecture for Supporting the Exchange of Electronic Manuscripts', *Communications of the ACM*, **30** (6), 408–414 (1987).

7. A. V. Aho and J. D. Ullman, *The Theory of Parsing, Translation, and Compiling*, Prentice-Hall, Inc., 1972.
8. J. Warmer and S. vanEgmond, 'The Implementation of the Amsterdam SGML Parser', *Electronic Publishing—Origination, Dissemination and Design*, **2** (2), 65–90 (1989).
9. M. E. Lesk and E. Schmidt, 'Lex: A Lexical Analyzer Generator', Technical Report CSTR#39, Bell Laboratories, October 1975.
10. V. P. Heuring, 'Compiler Construction: The Automatic Generation of Fast Lexical Analyzers', Technical Report SEG-85-1, University of Colorado, 1985.
11. S. C. Johnson, 'Yacc: Yet Another Compiler Compiler', Technical Report CSTR#32, Bell Laboratories, 1975.
12. *XDE User Guide, Appendix D: Parser Generator System*, Xerox Corporation, 1986.
13. *Information Processing—Text and Office Systems—SGML Amendment 1 (Final Text with Ballot Comments Resolved)*, ISO 8879-1986 (E) Amendment 1, International Organization for Standardization.
14. P. Naur (ed.), 'Report on the algorithmic language ALGOL 60', *Communications of the ACM*, **3** (5), 299–314 (1960).