ELECTRONIC PUBLISHING, VOL . 4(1), 3–26 (MARCH 1991)

Processing SGML documents

JOS WARMER[1]
PTT Research
DR Nehir Laboratories
Leidschendam, The Netherlands

HANS VAN VLIET
Faculteit Wiskunde en Informatica
Department of Mathematics and Computer Science
Vrije Universiteit, Amsterdam

SUMMARY

SGML (Standard Generalized Markup Language) is an ISO Standard that specifies a language for document representation. The main idea behind SGML is to strictly separate the structure and contents of a document from the processing of that document. This results in application-independent and thus reusable documents. To gain the full benefit of this approach, tools are needed to support a wide range of applications. The Standard itself does not define how to specify the processing of documents. Many existing SGML systems allow for a simple translation of an SGML document, which exhibits a 1–1 correspondence between elements in the SGML document and its translation. For many applications this does not suffice. In other systems the processing can be expressed in a special-purpose programming language. In this paper the various approaches to processing SGML documents are assessed. We also discuss a novel approach, taken in the Amsterdam SGML Parser. In this approach, processing actions are embedded in the grammar rules that specify the document structure, much like processing actions are embedded in grammars of programming languages that are input to a parser generator. The appendix contains an extended example of the use of this approach.

KEY WORDS SGML Parser generators Application generators Data translation Structured documents Reusability

1 INTRODUCTION TO SGML

SGML is a language that is used at two levels. At the first level, SGML is used to write document type definitions. A document type definition (DTD) is a grammar that defines the structure of a document. An example of a DTD is given in Figure 1.

```
<!doctype memo [
<!element memo                - O (sender, receiver, body)>
<!element sender              O O (person) >
<!element receiver            O O (person)+>
<!element person              O O (forename, surname)>
<!element forename            O O (#PCDATA)>
<!element surname             O O (#PCDATA)>
<!element body                O O (#PCDATA)>
]>
```

1. A simple document type definition

[1] The work covered in this paper was carried out while Jos Warmer was employed at the Vrije Universiteit, Amsterdam.

0894–3982/91/010003–24$12.00
Received 10 January 1990
Revised 18 October 1990

The first line specifies that this DTD describes a document type called *memo*. The second line says that a *memo* consists of a sequence-group with three consecutive elements: *sender*, *receiver* and *body*. *sender* consists of one *person*, *receiver* consists of one or more *person*s. *person* is a *forename* followed by *surname*. All other elements are defined as PCDATA, which means they consist of data characters. A line starting with `<!element` is called an element declaration. Following `<!element` is the name of the element being defined. The next two characters indicate whether the **starttag** and/or **endtag** may be omitted in the actual document. The last part of an element declaration is called the **content model**. The starttag and endtag are used to delimit elements in an actual document (see also Figure 2). For an element X, they are denoted by `<X>` and `</X>`, respectively.

The DTD describes the logical structure of a document, using a formal grammar. Note that it does not say anything about the layout or the meaning of the elements of a document. It only names the structural elements and their mutual relationships.

This strongly resembles the use of grammar formalisms, such as (E)BNF, that are used to define the syntax of programming languages. The DTD of Figure 1 thus defines the syntax of the language *memo*.

At the second level, SGML is used to describe a particular document. In the document, the structure is indicated by starttags and endtags. This structure must satisfy the formal structure described in the DTD. An example document according to the DTD in Figure 1 is given in Figure 2. The indentation in Figure 2 is not mandatory, but is used to elucidate the structure.

```
<memo>
  <sender>
    <person>
      <forename>Jos</forename>
      <surname>Warmer</surname>
    </person>
  </sender>
  <receiver>
    <person>
      <forename>Sylvia</forename>
      <surname>van Egmond</surname>
    </person>
    <person>
      <forename>Hans</forename>
      <surname>van Vliet</surname>
    </person>
  </receiver>
  <body>
    Tomorrow’s meeting will be postponed.
  </body>
</memo>
```

Figure 2. A document marked up with SGML

If we use the (programming) language analogy once more, Figure 2 gives an example sentence in the language *memo*.

SGML offers numerous other features. For example, starttags or endtags may be omitted in many cases, provided this does not render the resulting document ambiguous.

These other features are not relevant for the present discussion. The interested reader is referred to [1] for a more elaborate introduction to SGML and to [2] for more details.

Since a DTD is a formal description, the process of verifying the conformance of the structure of a document to its DTD can be carried out automatically. A program that checks whether a document conforms to a DTD is called an SGML parser. Such an SGML parser can be generated from a DTD.

In the (programming) language analogy, we then have an SGML parser-generator. It takes the formal specification (DTD) of a language and generates a parser for that language.

If a document is correct with respect to its DTD, it is said to be a conforming SGML document. The Standard does not define how a conforming SGML document is to be processed further. Consequently, current SGML systems do so in rather different ways. The main approaches taken will be discussed in Section 3.

2 PROCESSING SGML DOCUMENTS

In this section, we give a model of processing SGML documents. This model closely resembles the traditional model of processing programs written in a programming language.

Most processing systems for programs have an identical structure. This structure is depicted graphically in Figure 3.

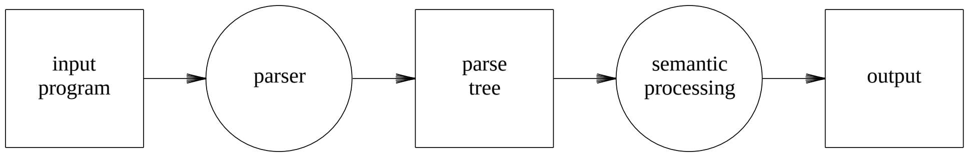

Figure 3. General structure of a compiler

The task of the parser is to check whether the input is syntactically correct and to build the parse tree. After the parser has done its task, the other part of the system will perform the semantic processing.

For different systems, the semantic processing will be quite different. In a compiler, the semantic processing can be split into several subtasks such as typechecking, optimization and code generation. The output at the right-hand side can be machine code for various processors. If the system is a cross-reference generator or a flow graph generator, the output will be a cross-reference listing or a flow graph respectively. All these different semantic processors may use the same parser.

An SGML processing system has the same general structure:

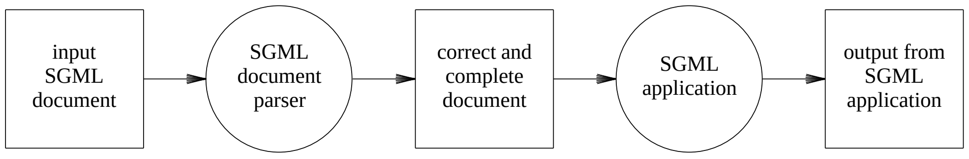

Figure 4. General structure of an SGML document processing system

An SGML parser, as defined in the Standard, has the same task as the parser for programming languages. It only checks the conformance of an SGML document to its DTD and it performs no further semantic processing. The output of most SGML parsers includes the so-called **complete** document. This is the document in which all starttags and endtags are present. At this stage, the document is known to conform to the corresponding DTD. The internal structure of this complete document corresponds to the parse tree in systems for programming languages.

As with programs, the complete document is not the end-stage in processing a document. It merely serves as an intermediate product, in which the correctness of the document has been assessed. Subsequently, the document has to be further processed. This is labelled **SGML application** in Figure 4. The SGML application may generate code for various formatters like *troff*, TEX or Scribe, but it can also produce an index or a list of figures. Again, these different semantic processors may use the same parser. What concerns us here is the node labelled **SGML application**.

Current SGML systems almost always offer some formalism for expressing the semantics of the node labelled **SGML application**. Such a formalism is called an **application interface**. These application interfaces can roughly be categorized into three groups:

Mapping: All starttags and endtags can be mapped onto some string. This is suitable only for 1–1 translation of SGML documents. That is, the structure of the output document is the same as the structure of the input document.

YAPL: YAPL stands for Yet Another Programming Language: a special-purpose language is used to express the processing of the document. This language is suitable only for translation of SGML documents. A wider range of translations is possible, though, because of the programming facilities provided.

General: A general-purpose language is used to express the processing of the document. This is suitable for any kind of SGML application.

2.1 Mappings

A mapping is a formalism in which all starttags and endtags can be mapped onto some string. The input SGML document is translated to an output document in a format that can be processed by an external application, like *troff*, TEX, etc. Mappings are easy to use and easy to implement, but the range of applications one can describe is very limited. The main advantage is that almost anyone can use them immediately, without learning a complete language. The Amsterdam SGML Parser [3], for example, also has a simple back end in which the starttag and endtag of each element can be replaced by some string. This back end is very easy to use and is powerful enough for simple applications. With this back end we are able to translate SGML documents to *troff*. Some SGML-like editors use a similar scheme to define the appearance of the different elements. In Figure 5 an example of a mapping table for the Amsterdam SGML Parser is given.

In the Standard, a so-called link mechanism is defined, which offers an alternative means to specify such a translation. Some SGML systems use the link mechanism for specifying the further processing of documents.

```
<memo>          "Memorandum:\n\n"
<sender>        "From: "
<receiver>      "To: "
</person>       "  "
</forename>     " "
<body>          "\n"
```

Figure 5. Example mapping table for Amsterdam SGML Parser

In many cases we need more than a simple 1–1 translation. The following list gives several cases where a more powerful translation scheme is needed.

- The order of elements of the SGML document might be different from the order in which the external application expects them.
- When replacing an element's starttag, the context might be essential. E.g., an item might occur in both an ordered and an unordered list. The replacement could be different for each context.
- One needs to know whether an element is the first or the last of a sequence of elements of the same type.
- After a sequence of elements, one needs to take some action.
- Different parts of the SGML document might go to different output streams.
- Specific parts of the document might have to be ignored.
- The data content of specific elements might have to conform to some specific syntax, that cannot be described in SGML. E.g., an ISBN number has a very specific syntax and an application may wish to check conformance to this syntax.
- The contents of certain constructs must be checked by a spelling checker.

As we want to use SGML documents in more contexts, the need for a more powerful backend grows.

2.2 YAPL

Many SGML systems offer the YAPL application interface. They use a special purpose language, meant for one application domain. At each starttag and endtag in the document we specify the YAPL-function to be called. Examples of this approach can be found in [4] and [5], where YAPLs are defined for formatting SGML documents, and in [6], where a YAPL is defined for database processing. As is explained in [5], where the language METAFORM is defined, YAPLs contain few basic programming facilities, since they are oriented towards domain experts rather than to expert programmers. These languages should be relatively easy to learn and use in the domain for which they are defined. Still, one has to learn a new language, which is a disadvantage. YAPLs are more powerful than mappings, because of the programming facilities provided. Still their power falls short for some applications. Most YAPLs solve some, but not all, of the problems mentioned at the end of the previous section.

Figure 6 is a small example of a function in the METAFORM language, defined in [5]. This function will be called at the start of element *person*. It prints “To:” if *person* is a descendant of *receiver* and it prints “From:” if *person* is a descendant of *sender*. The variable `GIPARENT` contains the name of the parent of the current element.

```
BEGIN_GI person
    BEGIN_ATT SYS.ATT
        IF $GIPARENT OF "sender" THEN
        BEGIN
            WRITE("From: ");
        END
        ELSE
        IF $GIPARENT OF "receiver" THEN
        BEGIN
            WRITE ("To: ");
        END
    END_ATT
END_GI
```

Figure 6. Example of a function in METAFORM

As argued in [3], we are reluctant to use this method. The number of programming languages is already too large as of now. For that reason we have chosen the simple mapping approach in the first place.

To avoid the YAPL-trap, the logical choice would be to use some existing general purpose language. This brings us to the third approach.

2.3 General

The third type of application interface is the most general. It behaves like the second, but instead of YAPL a general purpose language is used. Examples of SGML systems that offer this application interface are the IBM SGML Translator [7], MARK-IT [8], PISA [9], MARKUP [10] and the Amsterdam SGML Parser [3]. All these systems use the C language.

This formalism is the most flexible and the most complex to use. It gives much more control to the programmer than the other formalisms. The advantages of C over YAPLs are numerous. C is well-known. The full power of a general purpose language becomes available. One may use all kinds of existing C libraries and tools. The syntax and semantics of YAPL need not be defined. They need not be implemented either.

There are also several disadvantages. Application programmers must use a general purpose language, which often isn't very suitable for the domain. C is, for example, not the best language for text-processing. Furthermore, domain-experts often do not have enough programming skills to use a general purpose language. Therefore, if only a simple translation is needed, it is often easier to use a mapping or YAPL system.

Within the general application interface, two different methods can be distinguished. The method used by the Amsterdam SGML Parser is **grammar-driven**. All other systems mentioned at the start of this section are **event-driven**.

These methods are described in the following sections, followed by a discussion of their merits.

2.3.1 Event-driven application interface

The main idea in event-driven application interfaces is that an SGML document can be seen as a sequence of so-called **events**. At each event a C-function is called. This is either some default function or a function defined by the application programmer. Events are, for example, the occurrence of a starttag/endtag/processing instruction or the occurrence of data. The correspondence between events and C functions is usually given by a table. The following discussion will take PISA as an example:

```
/*
ELEMENT     STARTTAG FUNC.     ENDTAG FUNC.     CONTENT FUNC.
-------     --------------     ------------     -------------
*/
"sender",   start_sender,      end_sender,      content_sender,
"receiver", start_receiver,    end_receiver,    content_receiver,
"body",     start_body,        default_end,     default_content,
"person",   default_start,     default_end,     default_content,
"memo",     default_start,     default_end,     default_content,
"forename", start_forename,    end_forename,    content_forename,
"surname",  start_surname,     end_surname,     content_surname,

/*
ENTRY TYPE     FUNCTION
----------     --------
*/
DATA,          default_data,
```

Figure 7. Specifying processing functions in PISA

In PISA, the C functions to be called at the start and end of an element, are specified in tables as in Figure 7.

According to the tables in Figure 7, at the starttag of element *sender* the C function `start_sender` will be called. At the endtag of *sender*, `end_sender` will be called. The contents of *sender* will be handled by `content_sender`. For each of the events a system-defined default function, like `default_end` or `default_data`, can be used.

Consider, for example, the document from Figure 2, where the output should look like Figure 8.

```
Memorandum:

To: van Egmond, Sylvia;van Vliet, Hans
From: Warmer, Jos

Tomorrow’s meeting will be postponed.
```

Figure 8. Output for document from Figure 2

In the output the *sender* follows the last *receiver* and the *forename* follows the *surname*. Some newlines and some default text are added. Note that the order of the elements in the output differs from that in the SGML document. This is impossible to achieve with a mapping interface, and with most YAPL interfaces. Figure 9 shows possible C code for some of the functions mentioned in Figure 7 to achieve the output described above.

In order to achieve this result, the processing function for the contents of *person* has to save the contents in some global variable if it is inside *sender*. On the other hand, this

```
static char* saved_forename, *saved_sender;

content_forename(in, out)
char   *in;
char  **out;
{
    saved_forename = in;
}

content_surname(in, out)
char   *in;
char  **out;
{
    *out = allocate( strlen(saved_forename) + 2 + strlen(in) );
    sprintf(*out, "%s %s", surname, in);
}

       person_content(in, out)
       char   *in;
       char  **out;
       {
           if( string_equal(nodename(parent(current)), "sender") ){
               saved_sender = in;
           } else {     /* parent(current) == "receiver" */
               if( siblet(current) != NULL ){ /* no sibling of same type */
                   *out = allocate( strlen(in) + 3 );
                   sprintf(*out, "%s, ", in);
               }
           }
       }

       receiver_start(out)
       char   **out;
       {
           *out = "To: ";
       }

receiver_end(out)
char   **out;
{
    *out = allocate( 1 + strlen(saved_sender) + 1
                        + strlen("From: ") );
    sprintf(*out, "\nFrom: %s", saved_sender);
}
```

```
start_memo(out)
char   **out;
{
    *out = "Memorandum:\n\n";
}


start_body(out)
char   **out;
{
    *out = "\n";
}
```

Figure 9. C Application code for PISA

same procssing function has to output its contents if it is inside *receiver*. Subsequently, it has to check whether another *person* follows. If not, the saved contents of *sender* must be output.

As the example shows, several standard functions are available to check the environment in the document-tree of an element. The variable `current` denotes the current element and the function `parent` returns the parent of its argument. The function `nodename` returns the name of its argument. The function `siblet` returns the next sibling element of the same type as its argument. If it returns NULL, there is no such element. In this way it is possible to check whether `current` denotes the last occurrence of a repeated element.

2.3.2 Grammar-driven application interface

The Amsterdam SGML Parser uses the same general approach as described in the previous section, but its interface to the application programmer is quite different. As seen in the example in Figure 11, C actions can be incorporated in the DTD. These C actions define the processing. Their position is not restricted to the start and end of an element; they are allowed to be placed anywhere in an element declaration. The processing at the start and end of a sequence, for example, can be expressed at the corresponding place in the DTD.

```
<!element chapter       O O  (heading, pp*, appendix)>
```

Figure 10. Sample element declaration

This allows one to view the DTD-grammar as a program. Each element declaration behaves like a procedure. The element declarations can have user-defined parameters and local variables. The generic identifiers in the content model of an element behave like procedure calls. The complete document type definition, including the C code and parameters, behaves like a C program. This allows for a quick understanding of the interface by the programmer.

C actions can be incorporated in the DTD before and after each element, group and occurrence indicator. That is, at each position in the content model. In Figure 11, for example, the *pp* sequence is surrounded by three C actions. The first action is performed before the *pp* sequence, the second action is performed after each *pp*, and the third action

```
<!element chapter O O (heading,
                      {
                        /* processing before pp-sequence */
                      }
                      pp(/* parameters */)
                      {
                          /* processing for each pp */
                      }*
                      {
                          /* processing after pp-sequence */
                      },
                      {
                          /* processing before appendix */
                      }
                      appendix)>
```

Figure 11. Sample element declaration with C actions in the Amsterdam SGML Parser

after the complete *pp* sequence. If the *pp* sequence is empty, none of the three actions are performed. However, the action before *appendix* is performed always, because *appendix* must occur.

This scheme is borrowed from parser-generators, notably LLgen. As described in [11], such an interface is easy to use for the application programmer, because the grammar including the C actions can be seen as a normal recursive C program. To the knowledge of the authors, the Amsterdam SGML Parser is the first and only SGML parser that follows this approach.

This interface gives more control and freedom to the application programmer than the event-driven interface. Communication between elements is done through parameters, instead of global variables. This goes for communication to lower as well as communication to higher levels.

The example from the previous section, expressed in the Amsterdam SGML Parser C interface format, is given in Figure 12. A more complete example is given in Appendix A.

```
<!element memo
{
    char* sender;
    char* receiver;

    printf("Memorandum:\n\n");
}                              - O (sender(&sender),
                                    receiver,
                                    {
                                     printf("From: %s\n", sender);
                                    }
                                       body)>


<!element sender(char **person)  O O (person(person)) >
```

```
<!element receiver
{
    char* person;
    int   first = TRUE;

    printf("To: ");
}                                  O O (person(&person)
                                     {
                                         if( first ){
                                             first = FALSE;
                                         } else {
                                             printf("; ");
                                         }
                                         printf("%s", person);
                                     })+
{
    printf("\n");
}
>

<!element person(char** person;)
{
    char  *forename, *surname;
}                                  O O (forename(&forename),
                                        surname (&surname))
{
    *person = allocate( strlen(forename) + 3
                              + strlen(surname) );
    sprintf(person, "%s, %s", surname, forename);
}
>


<!element forename(char** name)  O O (#PCDATA(name))>
<!element surname (char** name)  O O (#PCDATA(name))>
```

Figure 12. C Application code for Amsterdam SGML Parser

2.3.3 Differences between event-driven and grammar-driven interfaces

This section shows some of the major differences between event- and grammar-driven application interfaces. First, some individual differences will be highlighted, followed by a general conclusion.

- In the event-driven interface the application programmer gets control only at a limited number of predefined places in the document. This makes it, for example, cumbersome to reverse the order of two elements.
- In the event-driven interface the number and type of the parameters that can be communicated by the application's C processing functions are predefined. In PISA, for example, the only parameter that can be passed to the parent level is a character string. A problem arises if the functions in the application program need to communicate information that does not fit within these predefined types.

  Furthermore, communication to a lower level can only be made through global variables. This is in contrast with the concept of structured programming. It is clear that such will reduce the readability and increase the complexity of an application.

- Processing of an element may depend on both local information and on information from ancestors.

  Whether a *person* should print itself, or should be saved, depends on its parent in the document. In the event-driven interface this decision is made explicitly inside the function that performs *person* processing. That is, *person* must know about its environment.

  Furthermore, the transposition of the *forename* and the *surname* is local to *person*. In PISA, this transposition is done within the *surname* processing function. Thus, processing actions are located at illogical places because of scoping and parameter restrictions.

  In the grammar-driven interface, on the other hand, processing that depends on knowledge local to *person* is done inside *person*, while processing that depends on the parent is done by the parent. This conforms better to what the programmer has in mind. It also conforms to the concept of object-oriented design, in which each object (element) can process itself, independent of its environment.

- Suppose a new element, *carbon*, is added to the DTD, and this element uses *person*.

  ```
  <!element CARBON  - - (person)+ >
  ```

  In the grammar-driven application interface, the processing of *person* need not be changed. With the event-driven interface the processing function of *person* often requires changes in this case. Because of this, an application that uses the grammar-driven application is easier to maintain.

- In the grammar-driven interface changing the names of elements does not affect the application code. If, in the example, the name of *sender* is changed into *from* and the name of *receiver* into *to*, then with the event-driven interface code has to be changed, while the grammar-driven interface remains the same.
- If the content model of an element, i.e. the right-hand side of an element declaration, changes, it seems that the grammar-driven interface is more difficult to maintain, because the code is integrated with the element declaration. However, the application code of the event-driven interface also has to change, because it makes assumptions about the structure of elements. If, in the example, *sender* and *receiver* are transposed in the DTD, then the functions `person_content` and `receiver_end` have to change. In fact, it is probably more difficult to maintain the event-driven application, because the dependent functions are more difficult to find and there may be many such functions.

Most of the advantages of the grammar-driven interface are due to the fact that information about the environment is implicitly communicated. Names of elements are almost never needed inside applications, as is shown in the example. In the event-driven applications, the environment must always be checked explicitly, which involves checking for element names, checking for siblets, etc. Standard functions like `parent` or `siblet` are not needed in the grammar-driven interface.

An advantage of the event-driven interface is that is is easier to create multiple applications for one DTD. The DTD need not be changed or touched, only a new set of functions must be defined. In the grammar-driven interface, a copy of the DTD must be made and new actions must be put inside this DTD. So we get two copies of the DTD, which potentially gives consistency problems. However, as shown in the appendix, a careful programmer can overcome this problem. In general, this problem needs a better solution, though. A feasible solution might make use of existing revision control systems or it might use a web-like [12] architecture.

3 IMPLEMENTING THE GRAMMAR-DRIVEN INTERFACE

The grammar-driven method is far from new, but to our knowledge it has not been applied to SGML systems or other structured document processing systems yet. It uses a technique well-known in compiler writing. Many widely used parser generators allow for embedded actions inside the grammar rules. This is seen in, for example, Yacc[13] and LLgen.

The Amsterdam SGML Parser, as described in [3], is an SGML parser generator. The **generator** reads and analyses the document type definition and generates a so-called **document-parser** (see Figure 13).

In fact, the **generator** does not directly generate the **document-parser**. Instead, it generates input for the existing parser generator LLgen. LLgen then produces the actual **document-parser**. We have chosen this approach because it allows us to reuse an existing parser generator, instead of writing one ourselves. We have extended this parser generator so that it recognizes actions in C that are embedded within the SGML document type definition.

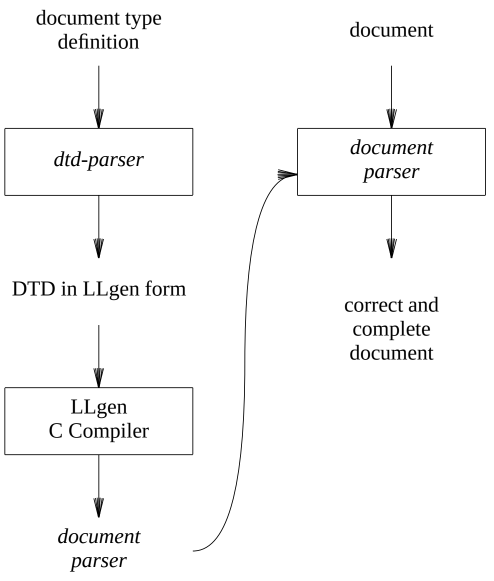

Figure 13. Structure of the Amsterdam SGML Parser

An LLgen grammar may contain embedded actions in the C programming language. For the C actions recognized within the SGML document type definition we have adopted a scheme identical to that of LLgen. In this way the **generator** only has to recognize the C actions within the document type definition. Subsequently the **generator** is able to simply copy the actions from the SGML document type definition into the generated LLgen code for the document parser. Now LLgen takes care of generating the **document-parser**, including the C actions (Figure14). Because we use the same scheme as in LLgen, implementation of this mechanism is easy.

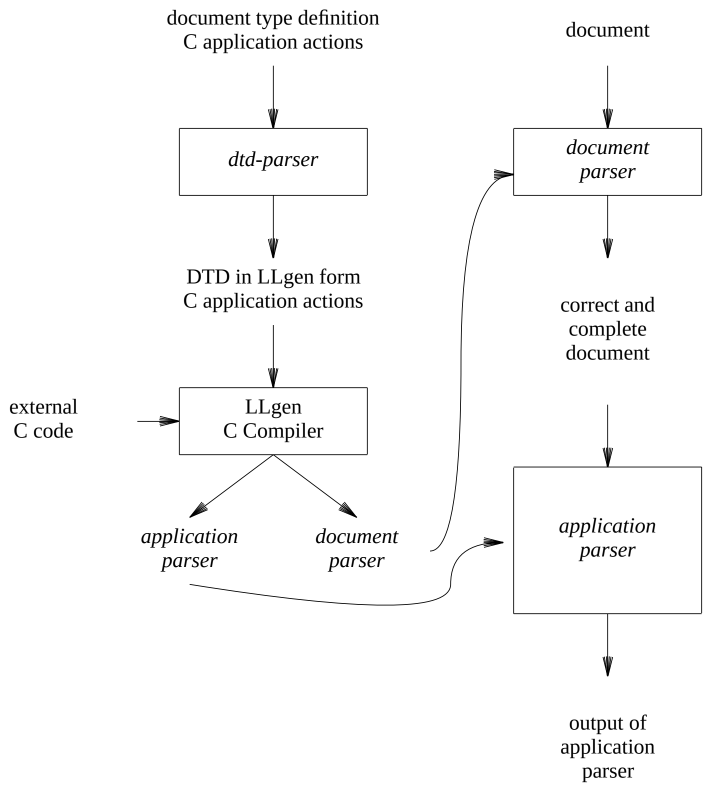

Figure 14. Structure of the Amsterdam SGML Parser, including C interface

In the scheme described above, there is a problem whenever a bug occurs. It is difficult to find out whether the bug resides within the SGML parser or within the application code. It is even possible that the application programmer interferes with the SGML parser itself. Therefore we have implemented the scheme in two passes. The **generator** generates an SGML **document-parser**. The **document-parser** parses and validates the SGML document and stores the complete document in an easy to read internal format. The **generator** also generates a second parser, which contains the code from the application programmer. This second parser uses the internal format stored by the **document-parser** as its input. Because the document is validated by the **document-parser**, no parsing errors can occur. Each error in the second parser must be an application error. In this way the SGML parser and the application code are strictly separated.

Another problem that might occur if everything is done in one pass, is that the document might prove incorrect at some point during parsing. The application has already performed some processing and it must make sure that the processing is reversible. In this two-pass approach, the document is known to be correct when the application starts processing, so this problem will not occur.

Another advantage of this approach is that the document parser has to parse the SGML document only once. The stored internal format can be used by several applications.

The mechanism has been only partly implemented as yet. SGML AND-groups are not handled. An AND group is a construct of which each element must occur, though the order is not fixed. If, for example, A is defined as:

```
<!element A  - - (b & c & d)>
```

then the following documents are valid:

```
<a><b>...</b><c>...</c><d>...</d></a>
<a><b>...</b><d>...</d><c>...</c></a>
<a><c>...</c><b>...</b><d>...</d></a>
<a><c>...</c><d>...</d><b>...</b></a>
<a><d>...</d><b>...</b><c>...</c></a>
<a><d>...</d><c>...</c><b>...</b></a>
```

We have never encountered a real need for AND groups, although there is no fundamental problem in implementing this feature.

The SGML ANY construct is not supported either. An element may have a content model ‘ANY’:

```
<!element whatever    - - ANY>
```

This means that each element defined in the DTD may occur. Because the elements are not named explicitly, there is no way to specify the parameters within the ANY construct. However, each ANY construct can be rewritten as an equivalent OR group of all elements in the DTD, which can be handled by the Amsterdam SGML Parser.

Inclusions are handled, but in the current implementation they cannot have parameters. An inclusion is an element that may occur anywhere within some given element. They can be called everywhere and thus it is impossible to specify the parameters.

During application development, it became evident that a general-purpose library of flexible string-handling and output functions would be very useful. Such a library has been implemented, although it still is relatively small.

4 CONCLUSIONS

Because documents in SGML are described by context-free grammars, there is a strong resemblance to programs. This analogy can be used to borrow well-defined techniques from compiler technology and apply them to structured document processing.

LL(1) grammars have the conceptual and practical advantage that they allow one to view the grammar as a program; this allows a more natural positioning of semantic actions and a simple attribute mechanism ([11] page 29). The document type definition grammar used in SGML is almost LL(1) and the application interface of the Amsterdam SGML Parser has the same advantages.

Many application interfaces are being developed and it is not clear which approach is the best to use. Many of the approaches seem to be tailored to solve one specific problem. By proposing a new type of application interface we hope to stimulate the discussion about SGML application development. We are convinced that, at least for complex applications, the mechanism used by the Amsterdam SGML Parser has great advantages over the other mechanisms described.

REFERENCES

1. D. W. Barron, ‘Why use SGML?’, *Electronic Publishing, Origination Dissemination and Design (EPODD)*, **2** (1), 3–24 (1989).
2. Martin Bryan, *SGML: An Author’s Guide to the Standard Generalized Markup Language,* Addison-Wesley, Wokingham, 1988.
3. J. Warmer and S. van Egmond, ‘The implementation of the Amsterdam SGML Parser’, *Electronic Publishing, Origination Dissemination and Design (EPODD)*, **2** (2), 3–28 (1989).
4. C. Smith, ‘Formatting SGML documents: essence and solution’, *SGML Users’ Group Bulletin*, **2** (1), 4–8 (1987).
5. Le Van Huu and E. Terreni, ‘A language to describe formatting directives for SGML documents’, *LNCS-TeX for Scientific Documentation,* 98–118 (1986).
6. J.P Gaspart, ‘Use of the SGML Parser at the office for official publications of the European Communities (OPOCE)’, *SGML Users’ Group Bulletin*, **2** (1), 29–36 (1987).
7. IBM, *IBM SGML Translator DCF Edition,* IBM Product Nr 5684-025, September 1988.
8. Sobemap, *The Mark-It Manual, version 2.0,* Sobemap S.A., Brussels, June 1988.
9. L. van Dam and E. van Loenen, ‘A programmers interface for SGML applications (PISA)’, CERN Internal Report (June 1989).
10. Lynne A. Price and Joe Schneider, ‘Evaluation of an SGML Application Generator’, *Proceedings of the ACM Conference on Document Processing Systems,* 51–60 (1988).
11. C. J. H. Jacobs and D. Grune, ‘A programmer-friendly LL(1) Parser Generator’, *Software — Practice and Experience*, **18** (1), 29–38 (1988).
12. Donald E. Knuth, ‘Literate programming’, *The Computer Journal*, **27** (2), 97–111 (1984).
13. S.C. Johnson, ‘Yacc: yet another compiler compiler’, Comp. Sci. Tech. Report No. 32, Bell Laboratories (1975).
14. IEEE, ‘IEEE Recommended Practice for Software Design Descriptions’, Std 1016 (1987).

APPENDIX: USAGE IN SOFTWARE SPECIFICATIONS

Consider an environment where software specifications are written in a strict format, such as the one described in [14]. Using SGML, we may devise a DTD which defines the structure of such software specifications. Through an appropriate backend, say one using *nroff*, we may then produce formatted hard-copy versions of these software specifications.

At the implementation stage, part of the software specification may be reused: routine headers, pre- and postconditions, and the like, will recur almost verbatim in the eventual program text. If the implementation language is Modula-2, this information will recur twice: once in some definition module, and once in the corresponding implementation module. If the implementation language is C, something similar holds, with a somewhat different syntax.

The syntax of *nroff*, Modula-2 and C is sufficiently different to make it impossible to translate the SGML description into any of them using a 1–1 backend. The example below shows how routines can be described within the IEEE specification framework. Using the approach of embedded C actions discussed in this paper, the different output formats can be generated quite easily. Besides *nroff* code for the textual specification, a set of Modula-2 definition and implementation modules, or a set of C header (.h) and source (.c) files, can be generated for each component in the specification.

This example is part of a DTD used in our software engineering course. In this course, each group of students has to write a specification, following the IEEE format. At a subsequent stage, each group has to implement a specification written by some other group. For the implementation language, they are free to choose either Modula-2 or C. The tools provided allowed them to generate code skeletons in either language, from the specification given.

In Figure 15 the DTD for a routine specification is given. Figure 16 contains the same DTD, with embedded C actions.

```
<!doctype spec [

<!element spec      - -   (routine)* >

<!attlist routine   name  CDATA  #REQUIRED
                    type  CDATA  ""       >
<!element routine   - -   (param*, pre, post, explain?) >

<!element pre       - -   CDATA >
<!element post      - -   CDATA >
<!element explain   - -   CDATA >

<!attlist param     inout (reference, value)     value
                    name  CDATA                  #REQUIRED
                    type  CDATA                  #REQUIRED>
<!element param     - - EMPTY >
]>
```

Figure 15. DTD for routine specifications

```
<!doctype spec
{{
/*
 *  After the ‘doctype’ declaration one can put some C code used
 *  for include, define and extern declarations
 */
#include "list.h"

extern char* att_value();
}}
[
<!element spec     - -   (routine)* >

<!attlist routine  name  CDATA  #REQUIRED
                   type  CDATA  ""       >
```

```
<!element routine
{{
    char*    proc_name;
    char*    return_type;
    P_List   param_var   = list_create();
    P_List   param_types = list_create();
    P_List   param_names = list_create();
    char*    param_name  ;
    char*    param_type  ;
    int      variable    ;
    char*    pre;
    char*    post;
    char*    explain;

    proc_name    = att_value("name");
    return_type  = att_value("type");
}}
                   - - (param  (&variable, &param_name, &param_type )
                            {{
                                  if( variable ){
                                      list_add(param_var, 1);
                                  } else {
                                      list_add(param_var, 0);
                                  }
                                  list_add(param_types, param_type);
                                  list_add(param_names, param_name) ;
                            }}*,
                        pre      (&pre),
                        post     (&post),
                        explain  (&explain)?
                       )
{{
    process_routine(proc_name, return_type, param_var, param_types,
                    param_names, pre, post, explain);
}}
>

<!element pre     (char** s;)  - - CDATA(s) >
<!element post    (char** s;)  - - CDATA(s) >
<!element explain (char** s;)  - - CDATA(s) >

<!attlist param   inout  (reference, value)     value
                  name   CDATA                  #REQUIRED
                  type   CDATA                  #REQUIRED>
<!element param
(
    int*   variable    ;
    char** param_name  ;
    char** param_type  ;
)
```

```
{{
    (*variable)   = ! strequal("value", att_value("inout"));
    (*param_name) = att_value("name");
    (*param_type) = att_value("type");
}}
                                    - - EMPTY >
]>
```

Figure 16. The same as Figure 15, but with embedded C-actions

Three external C functions `process_routine` were written, which write the output in one of the three different formats.

The example document in Figure 17 can thus be converted into the outputs shown in Figures 18, 19, 20, 21 and 22.

```
<spec>
    <routine name="CreateStack" type="StackType">
        <pre>
            None.
        </pre>
        <post>
            An empty stack is created and returned.
            The returned stack exists.
        </post>
        <explain>
            New memory must be allocated for a stack.
            If no memory is available, the program is aborted.
        </explain>
    </routine>

    <routine name="DestroyStack">
        <param inout="reference" name="stack" type="StackType">
        <pre>
            Parameter ’stack’ should be an existing stack.
        </pre>
        <post>
            The memory used by ’stack’ is freed and ’stack’
            does not exist anymore.
        </post>
        <explain>
        Watch out when there is more than one reference to the stack.
        </explain>
    </routine>
</spec>
```

Figure 17. Example routines in SGML

```
.nr LL 80m
.nr LT 80m
.LP
.sp 1
.nf
FUNCTION CreateStack() : StackType ;
.fi
.nf
.br
.in +6
.ti -6
Pre :\     None.
.in -6
.fi
.nf
.br
.in +6
.ti -6
Post:\     An empty stack is created and returned.
    The returned stack exists.
.in -6
.fi
.nf
.br
.in +8
.ti -8
Explanation:\     New memory must be allocated for a stack.
    If no memory is available, the program is aborted.
.in -8
.fi
.sp 1
.nf
PROCEDURE DestroyStack(VAR stack : StackType)
.fi
.nf
.br
.in +6
.ti -6
Pre :\     Parameter ’stack’ should be an existing stack.
.in -6
.fi
.nf
.br
.in +6
.ti -6
Post:\     The memory used by ’stack’ is freed and ’stack’
    does not exist anymore.
```

```
 .in -6
 .fi
 .nf
 .br
 .in +8
 .ti -8
 Explanation:\     Watch out when there is more than one reference
 to the stack.
 .in -8
 .fi
```

Figure 18. Nroff generated from SGML routines

```
DEFINITION MODULE stack;
(*
 *  Modula2 definition module generated from specification
 *)

(*
 *  procedure declarations
 *)

PROCEDURE CreateStack(): StackType;
(*
   PRE:
             None.
   POST:
             An empty stack is created and returned.
             The returned stack exists.
   EXPLANATION:
             New memory must be allocated for a stack.
             If no memory is available, the program is aborted.
*)

PROCEDURE DestroyStack(VAR stack : StackType);
(*
   PRE:
       Parameter ’stack’ should be an existing stack.
   POST:
       The memory used by ’stack’ is freed and ’stack’
       does not exist anymore.
   EXPLANATION:
       Watch out when there is more than one reference to the stack.
*)



END stack.
```

Figure 19. Modula2 definition module generated from SGML routines

```
IMPLEMENTATION MODULE stack;
(*
 *  Modula2 implementation module generated from specification
 *)

(*
 *  procedure declarations
 *)

PROCEDURE CreateStack(): StackType;
(*
   PRE:
             None.
   POST:
             An empty stack is created and returned.
             The returned stack exists.
   EXPLANATION:
             New memory must be allocated for a stack.
             If no memory is available, the program is aborted.
*)

BEGIN
    (* implementation *)
END CreateStack;

PROCEDURE DestroyStack(VAR stack : StackType);
(*
   PRE:
       Parameter ’stack’ should be an existing stack.
   POST:
       The memory used by ’stack’ is freed and ’stack’
       does not exist anymore.
   EXPLANATION:
       Watch out when there is more than one reference to the stack.
*)

BEGIN
    (* implementation *)
END DestroyStack;



END stack.
```

Figure 20. Modula2 implementation module generated from SGML routines

```
/* MODULE stack */
/*
 *  C header file generated from specification
 */
#ifndef stack_H   /* avoid multiple inclusions */
#define stack_H


/*
 *  procedure declarations
 */

extern  StackType   CreateStack();
/*
   PRE:
             None.
   POST:
             An empty stack is created and returned.
             The returned stack exists.
   EXPLANATION:
             New memory must be allocated for a stack.
             If no memory is available, the program is aborted.
*/

extern  void   DestroyStack(/*  StackType *stack */);
/*
   PRE:
       Parameter ’stack’ should be an existing stack.
   POST:
       The memory used by ’stack’ is freed and ’stack’
       does not exist anymore.
   EXPLANATION:
       Watch out when there is more than one reference to the stack.
*/


#endif stack_H
```

Figure 21. C header file generated from SGML routines

```
/* MODULE stack */
/*
 *  C source file generated from specification
 */
#include "stack.h"

/*
 *  procedure declarations
 */

StackType   CreateStack()
/*
   PRE:
             None.
   POST:
             An empty stack is created and returned.
             The returned stack exists.
   EXPLANATION:
             New memory must be allocated for a stack.
             If no memory is available, the program is aborted.
*/

{
    /* implementation */
}

void   DestroyStack(stack)
StackType     *stack;
/*
   PRE:
       Parameter ’stack’ should be an existing stack.
   POST:
       The memory used by ’stack’ is freed and ’stack’
       does not exist anymore.
   EXPLANATION:
       Watch out when there is more than one reference to the stack.
*/

{
    /* implementation */
}
```

Figure 22. C source file generated from SGML routines