Year: 1998

Markup of a Test Suite with SGML

Volk, M
Markup of a Test Suite with SGML

Abstract

Recently, there have been various attempts to set up a test suite covering the syntactic phenomena of a natural language (cp. [Flickinger et al. 1989], [Nerbonne et al. 1993]). The latest effort is the TSNLP project (Test Suite for Natural Language Processing) within the Linguistic Research and Engineering (LRE) framework sponsored by the European Union (cp. [Balkan et al. 1994]). These test suites are meant for the testing of NLP software regarding their coverage of syntactic phenomena. [Volk 1995] showed that a well-organised test suite can also be used to support incremental grammar development and grammar documentation. The key issues in the organisation of a test suite are the ease of extensibility and interchangeability as well as the avoidance of redundancy. We have implemented a test suite, which is optimized for the avoidance of redundancy and we report on the trade-off for extensibility and interchangeability.
Markup of a Test Suite with SGML

Martin Volk
University of Zurich
volk@ifi.unizh.ch

March, 26 1996

(Published in: John Nerbonne (ed.): Linguistic Databases.

1 Introduction

Recently, there have been various attempts to set up a test suite covering the syntactic phenomena of a natural language (cp. [Flickinger et al. 1989], [Nerbonne et al. 1993]). The latest effort is the TSNLP project (Test Suite for Natural Language Processing) within the Linguistic Research and Engineering (LRE) framework sponsored by the European Union (cp. [Balkan et al. 1994]). These test suites are meant for the testing of NLP software regarding their coverage of syntactic phenomena. [Volk 1995] showed that a well-organised test suite can also be used to support incremental grammar development and grammar documentation. The key issues in the organisation of a test suite are the ease of extensibility and interchangeability as well as the avoidance of redundancy. We have implemented a test suite, which is optimized for the avoidance of redundancy and we report on the trade-off for extensibility and interchangeability.

We define a test suite as a collection of syntactically well-formed natural language sentences and contrastively selected ungrammatical sentences (termed: non-sentences). The non-sentences can be used to check for overgeneration of the grammar. The collection is built according to the following principles:

1. The vocabulary for the sentences and non-sentences is derived from a controlled set of words.

2. The sentences and non-sentences are annotated with respect to their syntactic properties. The annotation aims at being as theory-neutral as possible.

3. The sentences differ among each other in at least one feature. The same holds for the non-sentences.

4. A non-sentence is identical to a sentence up to one feature. This particular feature is the reason for the ungrammaticality.

The vocabulary restriction (principle 1) guarantees that problems encountered in parsing a test sentence do not derive from word form problems, i.e. that they are truly syntax problems rather than morphology problems. The annotation of test suite
entries (principle 2) is needed to document why a test sentence was entered into the test suite and to maintain the test suite’s extensibility. The minimal difference (principle 3) ensures that the redundancy in testing is kept minimal. Principle 4 governs the selection of non-sentences such that they are only minimally deviant from a grammatical item.

The simplest way of storing the sentences in a test suite is by sequentially listing them in a file. The annotation can then simply be added after a special symbol. The obvious disadvantage is that this becomes hard to understand and control after few sentences. Furthermore, it is difficult to assign the various non-sentences to a related well-formed sentence in such an unstructured setup. Example 1 shows some sentences taken from a test suite for German relative clauses (see [Krenn and Volk 1993]) with simple annotations demonstrating that such a format is inappropriate because of the arbitrariness of the annotations.

Der Mann, der den Jungen sieht, ist zufrieden.
// Relative Pronoun: 'der'
*Der Mann, die den Jungen sieht, ist zufrieden.
// Problem: Agreement mismatch
*Der Mann, der der Junge sieht, ist zufrieden.
// Problem: Wrong function
Der Mann hat das Haus gesehen, das gross ist.
// Position: moved
*Der Mann hat, das gross ist, das Haus gesehen.
// Problem: Wrong position

Example 1

Since a test suite is meant to support testing of NLP software, it is necessary to ensure a uniform and system independent representation language. The language must provide for structuring the test suite and for annotating the sentences, while at the same time allowing a database view on the collection. We propose using SGML (Standard Generalized Markup Language) as representation language since it is “a database language for text” ([Goldfarb 1990]). SGML is an abstract representation scheme not bound to any hardware or system platform. It is thus superior to any database system in view of its interchangeability. Various ways of employing SGML for the markup of a test suite will be compared in this paper.

In order to demonstrate the feasibility of using SGML for a test suite, we have collected a test suite of 350 German sentences and represented them with SGML. The test suite is integrated into the GTU system (German: Grammatik-Testumgebung grammar test environment), a tool for the development and testing of natural language grammars. GTU was prototyped on DOS-PCs and reimplemented under UNIX using SICStus-Prolog (see [Volk et al. 1995] and section 3 of this paper).

2 Different SGML markup schemes

SGML was developed as a markup language for textual documents ([Goldfarb 1990]). It was meant for the markup of different types of documents allowing their interchange across implementations and systems. In 1986 it was approved as ISO standard 8879.
But it has only gained widespread acceptance in the early 1990s as tools for working with SGML became available. Recently, there has been an increasing interest in SGML due to the interest in HTML (which is an instance of SGML) in the World Wide Web.

SGML allows to define tag sets for different types of documents. It becomes most useful when the tag sets are standardized or at least interchanged together with the documents. The Text Encoding Initiative (TEI) has defined SGML tag sets for various document types such as prose, verse, drama, spoken language transcriptions, dictionaries, terminological databases and many more ([Sperberg-McQueen and Burnard 1994]). Because a test suite is a rather specific document type there exists no standard tag set. One can at most borrow from tag sets such as defined by the TEI for general header information (tags for author, title, funder, revision history etc.) and for units such as sentence or intra-sentential phrases. But even the TEI tag set for corpora that can be assumed to come closest to a test suite is insufficient since it is concerned with corpora details not relevant to test suites (such as contextual information) and lacks test suite specific details (such as special annotations). One could, of course, embed any of the SGML tagging schemas that are introduced in this paper into the TEI schema with the modification provisions specified therein. But this is solely a technical move and can therefore be omitted here.

In this paper we discuss various approaches of defining SGML tag sets for a test suite. We evaluate every approach with respect to

**Extensibility** How difficult is it to add more sentences to the test suite and to keep it consistent?

**Interchangeability** How difficult is it to port the test suite to another system and to use it for testing another grammar?

**Redundancy** What amount of redundant annotation does the test suite’s representation format require?

The goal is to maximize the first two criteria while minimizing redundancy.

For a document to be processable by an SGML parser there must be three units of information (all of whom are typically ASCII files):

1. The **SGML document** contains the text with its markup tags. In our case this is the file holding the test sentences and annotations with the markup.

2. The **Document Type Definition** (DTD) defines which tags and attributes are allowed or required in a given SGML document. A DTD can be seen as a context-free grammar stating how an SGML document is to be marked up.

3. The **SGML declaration** defines parameters for the tag set such as maximum name length for tags and their attributes or the character sets used in a document.

We will concentrate on the conceptual level and will therefore only be concerned with the SGML document and its DTD. The SGML declaration is processing specific and can thus be omitted here.
2.1 Non-hierarchical markup

Using SGML we can define tags that account for the definition of a test suite and help in structuring it. The objects in the test suite are the sentences and their annotations as pairs of feature name and feature value. In addition we must assign non-sentences to a sentence and we must document the reason for their ungrammaticality. The following DTD shows a possible definition.

```xml
<!DOCTYPE flatTS1 [
<!ELEMENT flatTS1    - - (entry+) >
<!ELEMENT entry      - - (sentence, 
                      (feature, value)+, comment?, 
                      (nonsentence, problem)+ ) >
<!ELEMENT sentence   - 0 (#PCDATA) >
<!ELEMENT feature    - 0 (#PCDATA) >
<!ELEMENT value      - 0 (#PCDATA) >
<!ELEMENT comment    - 0 (#PCDATA) >
<!ELEMENT nonsentence - 0 (#PCDATA) >
<!ELEMENT problem    - 0 (#PCDATA) >
<!ATTLIST entry      sID   ID #Required >
]
```

Example 2

This DTD defines a test suite as a sequence of elements of type entry. Every entry consists of a sentence, one or more feature-value pairs, an optional comment and one or more nonsentences together with a problem field. All of these elements are defined to take strings as content, i.e. parsable character data (#PCDATA). In addition an entry gets a unique identifier which is realized via the attribute sID.

This format allows an entry as shown in example 3. The end tags for most of the elements have been omitted since the strong sequentiality of the elements makes it obvious where an end tag has to be inserted by an SGML parser. The example shows a German sentence which contains a relative clause (translating as The man who sees the boy is content). The entry also contains two non-sentences that were derived from the sentence. Every problem field contains information on why the respective non-sentence is ungrammatical.1

```xml
<entry sID=s1>
  <sentence> Der Mann, der den Jungen sieht, ist zufrieden.
  <feature> Type of relative pronoun <value> 'der'
  <feature> Case of relative pronoun <value> nominative
  <feature> Type of reference word <value> noun
  <feature> Position of relative clause <value> direct
```

1Note that most of our examples cannot be processed using a standard SGML declaration. One needs to adapt the NAMELEN variable and to extend the characters permitted in attribute names to include the apostrophe.
The advantage of such a format is its transparency and therefore its straightforward interchangeability. The test suite can be given to other researchers and they can easily extract the information they desire. Moreover, the order of entries in the test suite is not of any significance and therefore the test suite can be easily extended. The major shortcoming of this format lies in the amount of redundant annotations that needs to be copied from entry to entry. Imagine an entry that needs identical annotation up to the feature “Type of relative pronoun”. It would require all the other annotations to be copied in its entry.

### 2.1.1 Using SGML attributes with a default value

A first step towards reducing this redundancy lies in the use of a default mechanism that inserts default values for the annotation features if nothing else is specified. That way one would only need to specify the features specific to a given sentence. SGML provides a default mechanism for attributes, attributes being properties of SGML elements. The modified DTD for the test suite could be defined as in example 4.

```xml
<!DOCTYPE flatTS2 [

<!ELEMENT flatTS2    -- (entry*) >
<!ELEMENT entry       -- (sentence, comment?,
                              (nonsentence, problem)+ ) >
<!ELEMENT sentence    - 0 (#PCDATA) >
<!ELEMENT comment     - 0 (#PCDATA) >
<!ELEMENT nonsentence  - 0 (#PCDATA) >
<!ELEMENT problem     - 0 (#PCDATA) >

<!ATTLIST entry
   sID ID  #Required >
<!ATTLIST sentence
   relprontype ('der' | 'welcher' | 'was' | 'wo')  'der'
   relproncase (nom | gen | dat | acc)   nom
   refwordtype (noun | name | pronoun | adverb)  noun
   position    (direct | pre | moved)      direct
   number       (simple | embedded | conjoined) simple
   function     (attribute | subject | object)  attribute
]>
```
The definitions for the elements feature and value have been eliminated. Instead, all features are added as SGML attributes to the element sentence. The features (and some values) have been replaced by abbreviations corresponding to the features in example 3. This format has the additional advantage that an attribute's domain can be enumerated (middle column of the attribute list) and a default value can be defined (rightmost column). For example, the feature relproncase stands for "case of the relative pronoun" and has as its domain nominative, genitive, dative, and accusative with the default set to nominative. With the domain information the SGML-parser can check whether a given value is valid according to the DTD thus providing a free consistency check for the test suite's feature system. In the SGML document we now have to specify only those features whose values deviate from the default values. Example 5 is identical to example 3 except for a different type of relative pronoun.

<entry sID=s2>
  <sentence relprontype='welcher'>
    Der Mann, welcher den Jungen sieht, ist zufrieden.
  </sentence>
  <nonsentence>
    Der Mann, welche den Jungen sieht, ist zufrieden.
  </nonsentence>
  <problem> agreement mismatch between relative pronoun and reference word
  </problem>
  <nonsentence>
    Der Mann, welcher der Junge sieht, ist zufrieden.
  </nonsentence>
  <problem> wrong form or function of relative pronoun
</entry>

Example 5

The SGML parser will turn this entry into a fully specified entry in the following format where the attributes are given before the element.

Attr SID = S2
(ENTRY
Attr RELPRONTYPE = WELCHER
Attr RELPRONCASE = NOM
Attr REFWORDTYPE = NOUN
Attr POSITION = DIRECT
Attr NUMBER = SIMPLE
Attr FUNCTION = ATTRIBUTE
(SENTENCE Der Mann, welcher den Jungen sieht, ist zufrieden.
)SENTENCE

---

2 The meanings of the other features and their domains are documented in [Krenn and Volk 1993].
3 Der and welcher are synonymous relative pronoun forms, the former is much more frequently used ([Grebe 1973] claims that der and its corresponding feminine and neuter forms account for 85% of all relative pronoun occurrences).
4 This output was produced by the sgmls-parser, a validating SGML parser available in the public domain. The output was slightly modified to facilitate readability.
Example 6

In the parser output every attribute for the sentence is present with its default value except for the attribute relprontype which has its value as specified in the SGML document. Please note that this format cuts back on redundancy only where default values are appropriate (which is the case in a substantial number of entries). In addition, the default attributes are assigned to every element of type sentence. If the test suite contains sentences (i.e., entries of type sentence) that have nothing to do with relative clauses, these default values will still be assigned. It is not possible to assign an attribute only if some other attribute is present. Attributes are independent of each other. To remedy this problem one needs to define a special tag for every syntactic phenomenon that has its own set of attributes. Instead of having a tag for sentence we have to have a tag for sentence-with-relative-clause, sentence-with-coordinated-clauses, sentence-with-complex-np and so on. The attributes defined in example 4 for sentence will then be defined for sentence-with-relative-clause only. Using the SGML default mechanism thus cuts back on the redundancy of the annotation system but results in an unacceptable proliferation of special purpose tags.

2.1.2 Using SGML attributes with a "Current" value

Another means of using inheritance within SGML attributes is the use of the #Current value instead of the default value. Specifying #Current says that the value of an attribute needs to be given at the first occurrence of the element and that it will be copied to every subsequent occurrence until a new value is explicitly given which is copied from there on. The attribute list for sentence in example 4 could be turned into the format in example 7.

```xml
<!ATTLIST sentence
  relprontype ('der' | 'welcher' | 'was' | 'wo') #Current
  relproncase (nom | gen | dat | acc) #Current
  refwordtype (noun | name | pronoun | adverb) #Current
  position (direct | pre | moved) #Current
  number (simple | embedded | conjoined) #Current
  function (attribute | subject | object) #Current
>
```
If we sort the test suite entries according to the first attribute, we have to specify the attribute only once for every value which will then be inherited by all subsequent entries. Of course the same limitations mentioned for the default value also apply to the \texttt{Current} value. Since the attributes are specific to sentences with relative clauses, one has to make sure that they are not copied to other test suite entries. In addition, using \texttt{Current} forces the test suite developer into carefully ordering the entries. Extending the test suite thus becomes an error prone process. For this reason the usage of \texttt{Current} is generally discouraged (cp. [Rieger 1995] p.134).

Another approach to reducing redundancy consists of inheriting neither default nor current values but all general values through an inheritance hierarchy. For this we need something resembling an object-oriented class system where properties are inherited from one level to another unless otherwise specified. But switching to a real object-oriented framework would undermine our basic assumption of keeping the test suite in a format for easy interchangeability. We therefore have experimented with describing an inheritance tree with SGML.

2.2 Layered Markup

Organizing the test suite into a tree structure requires, above all, to identify the core syntactic phenomena that serve as the nodes in the tree. Let’s assume that we want to model a tree as depicted in figure 1. The nodes represent syntactic phenomena like main clauses, subordinate clauses or relative clauses. A node in the tree inherits all features from the phenomena on the path that connects it to the root of the tree. Thus, a node is more general than its daughter nodes. This inheritance obviously provides a means to compact the representation.

![Figure 1: Simple test suite tree](image)

There are two principally distinct ways of expressing a tree structure in SGML. One is by using the built-in reference links as edges between the nodes. The other is mapping
the nested structure generically or directly into the DTD.\(^5\)

### 2.2.1 SGML tree structure via reference links

Reference links in SGML are special purpose attributes to indicate references within a document. A reference consists of a pointer and a designated goal. In the DTD the pointer is marked with the IDREF value whereas the reference goal is marked with ID. Attaching these values to two attributes called nextnode and nodeID for an element node provides the means to point from one node to another. However, this allows only to point from a node to exactly one other node. But in order to account for a tree structure we need to point to more than one node. We therefore introduce an empty element within node called nodelink which can occur more than once. This empty element holds nothing but an IDREF pointer to another node. The DTD will then look like example 8. The definition for entry has been omitted. It can be assumed to be the same as in example 2.

```xml
<!DOCTYPE layeredTS1 [

<!ELEMENT layeredTS1 (node+, entry+)>
<!ELEMENT node (name, feature, value, (nodelink+ | entrylink+))>
<!ELEMENT name (#PCDATA)>
<!ELEMENT feature (#PCDATA)>
<!ELEMENT value (#PCDATA)>
<!ELEMENT nodelink EMPTY>
<!ELEMENT entrylink EMPTY>

<!ATTLIST node nodeID ID #Required>
<!ATTLIST nodelink nextnode IDREF #Required>
<!ATTLIST entrylink nextentry IDREF #Required>
]
```

Example 8

The DTD in example 8 states that a layered test suite consists of a sequence of nodes followed by a sequence of entries. Every node is required to have a name, a feature, and a value element as well as at least one link to another node or at least one link to an entry. The links are empty elements consisting only of the attributes nextnode and nextentry respectively. The nodes and entries in example 9 model part of the tree in figure 1.

```xml
<node nodeID = n2>
<name> subordinate clause
<feature> finite verb position
<value> final
<nodelink nextnode = n21>
```

\(^5\)Note that in a similar manner the TEI Guidelines provide both built-in reference links and a nested generic structure for the representation of tree structures (cp. [Sperberg-McQueen and Burnard 1994] section 21.2 and 21.3).
Node n2 subsumes subordinate clauses. In the example it has links to the nodes n21, n22 and n23, which are nodes for 'dass'-clauses, relative clauses and whatever other subordinate clause one wants to specify. n2 is defined by the feature "finite verb position" which has the value "final" which stands for the fact that in German subordinate clauses the finite verb is at the end whereas in German main clauses it is at the second position. This feature is meant to be inherited to all nodes below n2. There is an inheritance chain from n2 to n22 to n221 to the test suite entry s1. This entry contains a relative clause starting with a "der" relative pronoun and will inherit all the features specified in the chain for "finite verb position", "conjunction type", and "type of relative pronoun".

The format in example 9 is very transparent but it puts the burden of organizing the inheritance completely on the software developer, because SGML does not define any inheritance along the lines of reference links. Additional software is needed that takes the output of the SGML parser, traces the inheritance links, collects the features and values, and adds them to the entries.

This approach has been implemented within the GTU project at the University of Koblenz. A tree with 35 nodes has been set up and 350 sentences were attached as entries to this tree. A C-program was added to ensure the inheritance of features and to
transform the output into a Prolog database so that the test suite can easily be accessed from GTU. In addition the tree was mapped into GTU as an ordering mechanism in accessing the test suite. The user can select a particular node and is presented with all the sentences subsumed by it (see section 3).

2.2.2 SGML tree structure via generic nested tags

The tree structure of an inheritance tree can not only be represented via reference links but also by nested tags. Since SGML allows the recursive definition of tags we can define a node as consisting of name, feature, and value followed by either some (other) nodes or some entries. The DTD for this format is shown in example 10.

```xml
<!DOCTYPE layeredTS2 [
  <!ELEMENT layeredTS2 - - (node)>
  <!ELEMENT node - - (name, feature, value, (node+ | entry+))>
  <!ELEMENT name - 0 (#PCDATA)>
  <!ELEMENT feature - 0 (#PCDATA)>
  <!ELEMENT value - 0 (#PCDATA)>
]>]
```

Example 10

Compared with the nested structure in example 8 we can now do without the reference links since the mother-child relation between nodes is defined by their nesting. Within the SGML document we have to keep track of the nesting of nodes which is bothersome if not supported by special editing tools. The entries are within the most deeply embedded nodes.

```xml
  <node>
    <name> ... <feature> ... <value> ... </node>
  </node>
  <node>
    <name> ... <feature> ... <value> ... </node>
  </node>
  <node>
    <name> ... <feature> ... <value> ... </node>
  </node>
  <entry> ... </entry>
  <entry> ... </entry>
  <node>
    <node>
      <name> ... <feature> ... <value> ... </node>
    </node>
    <node>
      <name> ... <feature> ... <value> ... </node>
    </node>
    <entry> ... </entry>
    <entry> ... </entry>
  </node>
</node>
```

Example 11

As in the example 8/9 the inheritance of annotation features has to be performed by add-on software. The traversal of the tree as defined by example 10 is easier since a
mother node directly precedes all children nodes in the linear sequence within the SGML document whereas in example 9 the inheritance program needs to follow the reference links which can result in jumps to anywhere in the file.

### 2.2.3 SGML tree structure via specific nested tags

Finally, it is possible to use the linguistically motivated node names directly as SGML element names. In this way we get an explicit coding of the inheritance tree in the DTD.

```xml
<!DOCTYPE layeredTS3 [
  <!ELEMENT layeredTS3 - - (main-clause, sub-clause) + (feature, value)>
  <!ELEMENT main-clause - - (transitive-clause, intransitive-clause) >
  <!ELEMENT sub-clause - 0 ('dass'-clause, relative-clause) >
  <!ELEMENT relative-clause - 0 (d-rel-clause, w-rel-clause) >
  <!ELEMENT 'dass'-clause - 0 (entry+) >
  <!ELEMENT d-rel-clause - 0 (entry+) >
  <!ELEMENT w-rel-clause - 0 (entry+) >
]
```

Example 12

In this format a subordinate clause is a 'dass'-clause or a relative-clause (among other things not listed here). The feature-value pair is given as an included element in the top level tag and is therefore allowed in all subelements. The resulting SGML document is very short since most of the information is already in the DTD. An entry in such an SGML document consists only of the sentence, the non-sentences and the problem field. The obvious disadvantage of this approach is that for any extension of the test suite related to a new phenomenon the DTD must be modified. A new element has to be inserted into the DTD and its attributes have to be defined. It is easy to imagine how complex a DTD we can expect when covering more than a dozen syntactic phenomena. Even simple extensions can become a major project requiring SGML experts to work on the DTD. The principle of defining a markup language for easy extensibility is thus violated and interchangeability can no longer be entertained.

### 3 The test suite format in the GTU system

GTU is a workbench for the development and testing of unification based grammars. It supports three different grammar formalisms (LFG, ID/LP, DCG) and two different parsing modes (top-down and bottom-up chart parsing). GTU has been interfaced to huge lexical resources (the CELEX database and the Gertwol analyzer). It comes with special purpose grammar editors and a hypertext help system. GTU was originally designed as a tutoring system for linguistics students. Therefore special care has been taken in the design of a robust and user friendly interface including visualization and
automatic comparison of parsing results. Natural language sentences can be manually entered and fed to one of the parsers or they can be selected from the test suite.

GTU contains a test suite in a format similar to example 8/9, that is, it has an inheritance tree with reference links. The sentences are attached to the tree's leaves and inherit all the features from the leaves. In contrast to example 8/9 it is legitimate in the GTU test suite to attach a sentence to more than one leaf. This explains why we could not use the solution with nested tags as given in example 10/11 unless we added sentence identifiers and reference links. In GTU, a test sentence can be a representative for different phenomena. E.g. a sentence like Der Mann sieht den Jungen an. (The man looks at the boy,) will be attached to the “separable prefixes” node (it contains a separated verb prefix) and to the “transitive verbs” node. In order to minimize redundancy we have attached the GTU test suite to a tree structure as depicted in figure 2.

![Figure 2: Part of GTU's test suite tree](image)

When building up the test suite, we first had to design the test suite tree and a corresponding DTD. The tree includes nodes for noun phrases, prepositional phrases, verb groups, coordination, sentence types and so on. For all these phenomena sentences were collected from grammar books and normalized until they conformed to the requirements of the test suite definition. Nonsentences had to be made up according to the principle that they differ from a given sentence in exactly one feature. In a next step the tree itself and all sentences were annotated with SGML tags.

The complete test suite document was validated with an SGML parser. This ensures that all the sentences were tagged in conformance with the DTD. The parser output is again a structured SGML document with all our test sentences. We translated the parser output (via lex and yacc) into a Prolog database. Since we envisioned our test suite to grow to several thousand sentences we were looking for a mechanism that could handle a large database efficiently. We chose the SICStus-Prolog external database facility which comes with indexing schemes while keeping the data on disk storage (unlike regular Prolog data that are loaded into the knowledge base in working memory).

4 Accessing the test suite

So far we have looked at the advantages and drawbacks when building up and extending a test suite. We still have to consider what the different representation formats mean for accessing the information. Retrieving information from a database can in principle
be done in two different ways by either using a database query language or by using task specific menus. Database languages are more powerful and flexible while menus allow faster access for predefined purposes and are easier to handle. SGML documents can in principle be translated into relational databases. But because of the recursive structure in SGML documents this is very tedious. Therefore, [Böhm et al. 1994], on examining the relation between databases, SGML documents and information retrieval, concluded that object-oriented databases are a better choice.

It is obvious that a database query language can be used to access an SGML test suite only if the tags are sufficiently general. (If the tags were specific, how would the user know what to ask for?) For instance, it is better to have `<sentence>` rather than `<sentence-with-relative-clause>`. Representing syntactic features as SGML attributes (as in example 4 but without defaults) rather than as SGML elements (as in example 2) helps in keeping the annotations consistent and compact which is also important for using a query language. For instance, it is more compact to query for the attribute `relpronType` rather than for the element `feature` with the content *Type of relative pronoun*. So one can imagine a database query to a test suite with non-hierarchical markup and attributes as

```
find <sentence> where relpronType = 'der'
```

Note that the validity of a query can be decided before searching the database by checking in the DTD whether `relpronType` is a valid attribute for `<sentence>` and whether `der` is a valid value for this attribute. This fits in nicely with findings by [Böhm et al. 1994]: “The key for applying IR retrieval techniques to SGML documents has to be found in using the DTD to guide the retrieval process.” When elaborate and user friendly query languages for SGML documents become available this will be the way to go for a test suite with SGML markup.

In the GTU project we did not have access to any such software and therefore decided to implement specific menus. For every node in the test suite tree the interface provides a screen display of all subsumed sentences. The sentences can be selected one at a time or in groups and fed to the GTU natural language parser with a mouseclick. The user interface also allows the selection of multiple nodes at the same time to obtain the intersection of the sentence sets subsumed by the selected nodes. For instance, we can get all the sentences that have a separable prefix verb and a direct object.

Experience has shown that it is important to have groups of sentences that can be tested repeatedly be it for a grammar in different development phases or for grammars in different formalisms. Test results can only be compared if sentence groups are stable. Therefore it is important that groups of sentences can be saved for future usage. For details on GTU see [Volk 1995]; the development of GTU’s test suite is documented in [Fitschen and Pieper 1994].

5 Summary and conclusion

A test suite should be easy to extend and to interchange. Ease of extension requires that a transparent representation format is used. Ease of interchange requires that a system and platform independent format is used. SGML fulfills both of these requirements at first sight. But there are some drawbacks if one wants to minimize redundancy in annotating the test suite entries.
Using inheritance in the way it is built into SGML, such as default values or \$Current\ values for attributes, results in problems of keeping the attributes to the relevant elements which can only be resolved with a proliferation of element names. A different approach consists in forcing hierarchic structure into the SGML markup by means of reference links or nested tags. But then the inheritance mechanism has to be performed by add-on programs which are interpreting SGML markup in an idiosyncratic way.

**The lesson:** If extensibility and interchangeability are considered most important then one has to keep it simple, and a format like in example 2 is the best choice. This should, however, be accompanied with features represented as SGML attributes in order to get a consistency check by the SGML parser. Redundancy must then be taken into account and should be hidden from the user with the help of appropriate editing and viewing tools. Validating SGML parsers are freely available, and can be used to perform checks on the markup.

On the practical side: In the future we would like to have tools that provide for efficient browsing and retrieval from SGML databases. This would considerably ease the usability of a test suite tagged with SGML.

**Acknowledgement:** The work reported in this paper is in part joint work done with Arne Fitschen and Stefan Pieper at the University of Koblenz. In addition, I would like to thank Michael Hess and Dominic Merz for comments on earlier versions of this paper.

**References**


