University of Helsinki
Department of Computer Science
Series of Publications C, No. C-1997-36

TranSID: An SGML Tree Transformation Language

Jani Jaakkola, Pekka Kilpeläinen, and Greger Lindén

Helsinki, May 1997

Report C–1997–36

University of Helsinki
Department of Computer Science
P.O.Box 26 (Teollisuuskatu 23)
FIN-00014 University of Helsinki, FINLAND

TranSID: An SGML Tree Transformation Language

Jani Jaakkola, Pekka Kilpeläinen, Greger Lindén,

Department of Computer Science, University of Helsinki
Report C-1997-36
May 1997
14 pages

Abstract

We present a powerful document transformation language called TranSID, which is targeted at structured (SGML) documents. The language is based on a powerful model where the entire input document tree may be referenced during the transformation process. The evaluation is performed in a bottom-up manner. A language evaluator has been implemented which runs in Unix environments.

CR Categories and Subject Descriptors:

I.7.2	[Text processing]: Document Preparation.
J.7	[Computers in other systems]: Publishing.

General Terms: Design

Keywords: structured document transformations, SGML, tree transformation languages

Contents

1 Introduction

In the world of document preparation, text transformation is an everyday issue. Documents need to be produced and presented in different media such as paper, CD-ROM, and screen. Documents also need to be transformed between different text preparation systems depending on the needs of the users. The transformation needs have caused the creation of a plethora of more or less ad hoc transformation tools for transforming a document from one representation into another.

On the other hand, the idea of a data-centered information pool is well acknowledged. It would be sufficient to update only one particular master document and then propagate the changes to all other representations of the same document. Such a processing framework is supported by the Standard Generalized Markup Language (SGML) [ISO86]. The idea of SGML is to mark the structure of documents explicitly. Main recognizable components called *elements* impose a hierarchical (or tree-like) structure on documents. The contents of elements, which may contain further elements, are surrounded in the document text by a start tag and an end tag. Start tags may contain additional data items called *attributes*, which pertain to the corresponding element. Document instances may also contain *entities* and *processing instructions*. Entities are used as place holders for further document contents, e.g., external graphic files. Processing instructions are used for passing information, e.g., explicit formatting commands to a processing application. The overall structure of a set of documents is described in a document type definition (DTD) which defines what kinds of tags may be used and how they may relate to each other.

There are several translation engines targeted specifically at SGML applications on the market today. We distinguish between *conversion* tools for transforming documents *into* SGML documents (aka *up-translation*), and *transformation* tools for transforming SGML documents into SGML or other formats (aka *cross-translation* and *down-translation*). Transformation tools, such as Balise [AIS96], MetaMorphosis [MID95], OmniMark [Exo93], and CoST [Har93], use as their front end an SGML parser that reads and checks the SGML document before it is transformed.

We divide transformation tools into two categories. *Event-based transformers* (e.g., OmniMark) use a sequential evaluation strategy. They transform the SGML document at the same time when it is entered and parsed. This strategy is usually efficient, at least memory-wise, as the document never has to be entirely read into main memory. Event-based transformers use the ESIS [Gol90] output of an SGML parser as their input. The ESIS output consists of all 'events' or structural parts in the document, such as the start and end tags as well as the content between tags.

Tree-based transformers (e.g., MetaMorphosis) construct an internal representation, usually a tree, of the SGML document. A tree-based transformation (see, e.g., [KPPM84]) lets the user refer to any part of the document (tree) at any time during the transformation. This strategy is more powerful than the event-based one, e.g., it is easier to reorder document parts or to handle forward and backward references. Some transformers (e.g., Balise) combine the event-based and the tree-based strategies and let the user choose which one is appropriate at a certain moment during the transformation.

The TranSID language is a tree-based transformation language. The language is targeted at SGML transformations, but the underlying technique is independent of the representation format. The transformation has full access to the entire parse tree of the input document. Design goals of the language include declarativeness, simpleness and implementability with reasonable effort. Special features include a *bottom-up evaluation* process. Bottom-up evaluation is a declarative way of defining some transformations that would be awkward to define in a top-down manner. (See the example in Section 6.) The TranSID language also includes high level declarative commands that free the user from low-level programming. We have implemented an interpreter and an evaluator for TranSID which are fully operational in Unix environments [JKL96a, JKL96b]. TranSID has been developed in a research project called Strcutured and Intelligent Documents (SID)[1].

The Document Style Semantics and Specification Language Standard (DSSSL, [ISO96]) defines a related transformation language. DSSSL is, however, in its entirety quite complex as it covers both tree transformation and document formatting. TranSID is mainly concerned with tree transformation even if some simple formatting is possible. Also, no complete implementations of DSSSL exist yet — only a partial implementation of the DSSSL style language has been developed [Cla96a].

In the rest of this paper we present the TranSID language and its implementation. We start by giving a short explanation of the data model and by defining the semantics of the transformation language. We then go on to show some extensive examples of its use. We conclude by giving an overview of the implementation and planned extensions to the language.

2 Overall control and data model

A transformation engine for the TranSID language has been implemented. The transformation process is similar to the grove transformation process of the DSSSL standard [ISO96]. The basic environment consists of an SGML parser, a TranSID parser, a transformer and a linearizer (Figure 1).

A TranSID transformation starts by parsing an SGML document and constructing an internal document tree. We use the SP parser [Cla96b] for parsing the document. The tree transformation is specified in a TranSID program that is parsed by its own parser. An internal rule base is formed of the TranSID program. It may contain rules for transformation and linearization as well as some import declarations. The import declarations guide the SGML parser in building the source tree. For example, the declarations state where entities should be expanded. The transformation is performed by the transformer process which traverses the constructed parse tree and applies the transformation rules to build a corresponding target tree. The linearizer may still perform minor conversions to the target tree. It may output the target tree as an SGML document, or some other specified output, e.g., a stripped (of tags) ASCII version or a HTML document. There may also be several input and output documents.

[1]The project is funded by the Finnish Technology Development Center (TEKES) and seven Finnish enterprises (The Aamulehti Group, Oy Edita Ab, the National Board of Education, WSOY, Helsinki Media, Lingsoft, Inc., and MTV3).

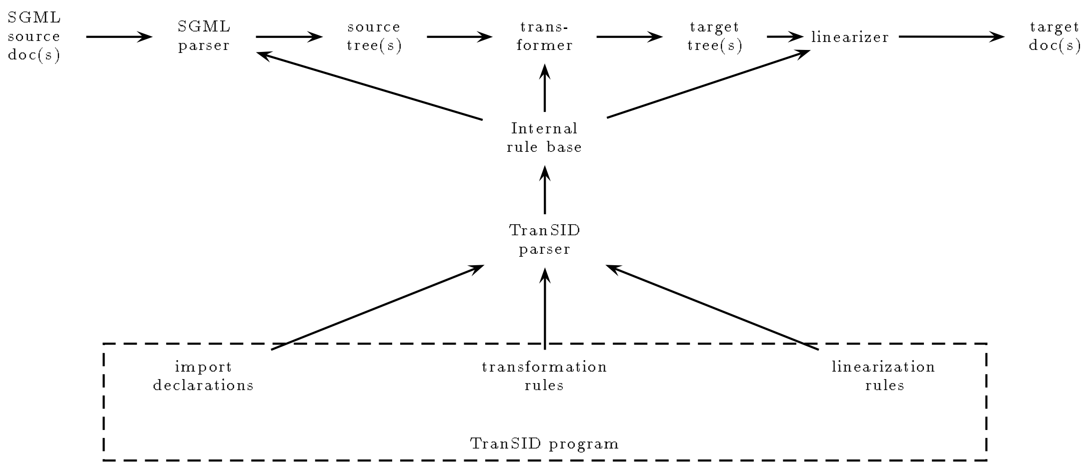

Figure 1: The TranSID transformation process

3 Semi-formal semantics

We present a semi-formal semantics for TranSID transformations. These definitions describe the overall semantics of TranSID, i.e., how a TranSID program specifies a mapping from source trees (or forests) to target trees (or forests). Detailed examples of transformation rules and expressions that can be used in them are given in the subsequent sections.

During a TranSID execution there is always a *current node* at the focus of control. Intuitively, the current node is the node that is being transformed. The evaluation proceeds bottom-up: the descendants of the current node belong to the result forest, but its siblings and ancestors are in the source tree (Figure 2).

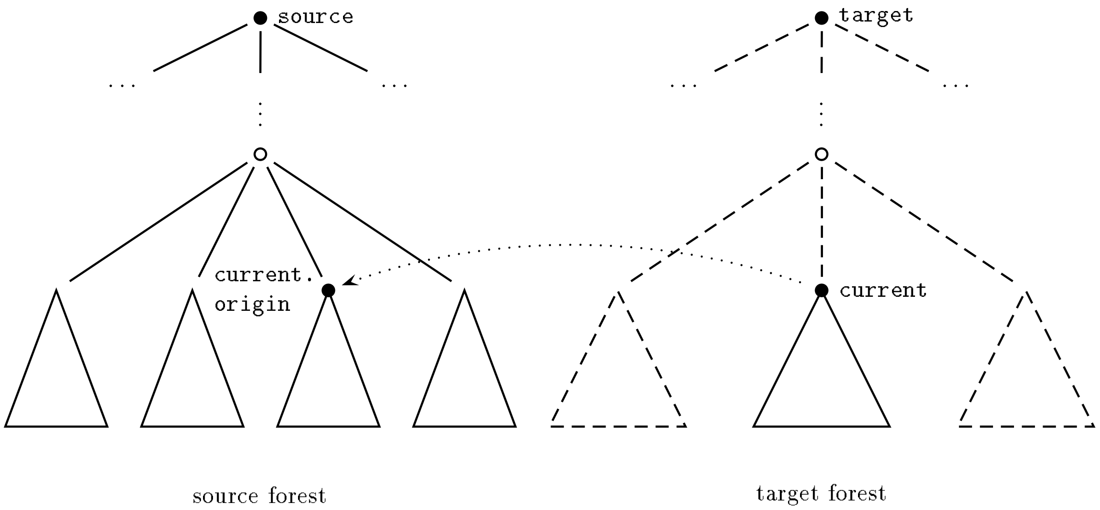

Figure 2: Source and target forests of a transformation process. Structures reachable from the `current` node are marked with solid lines, unreachable or yet uncreated ones with dashed lines.

A *TranSID program* $\mathcal{P}$ is a sequence of transformation rules $(\mathcal{R}_1, \ldots, \mathcal{R}_k)$, where each *rule* $\mathcal{R}_i$ is a pair $(\mathcal{S}_i, \mathcal{T}_i)$ consisting of a *source clause* $\mathcal{S}_i$ and a *target clause* $\mathcal{T}_i$. The source clause is a predicate on the subtree rooted by the current node. If source clause $\mathcal{S}_i$ is satisfied by the node, we say that the corresponding rule $\mathcal{R}_i$ *matches* the subtree rooted by the current node. The result of a rule $\mathcal{R}_i$ on a tree T is denoted by $\mathcal{R}_i(T)$, and it means the forest resulting by applying the target clause $\mathcal{T}_i$ on T. This application may involve insertions of new structures and selection and combination of tree components relative to the root of T. Again, we refer to the rest of the article for concrete examples.

Let $\mathcal{P} = (\mathcal{R}_1, \ldots, \mathcal{R}_m)$ be a TranSID program. We denote the result of applying $\mathcal{P}$ on a tree or a forest T by $\mathcal{P}(T)$, and define it as follows:

1. If T is a tree that matches no rule in $\mathcal{P}$, then $\mathcal{P}(T) = T$.
2. If T is a forest $(T_1, \ldots, T_n)$, then $\mathcal{P}(T) = (\mathcal{P}(T_1), \ldots, \mathcal{P}(T_n))$, i.e., the result is obtained by concatenating the result of applying the same program $\mathcal{P}$ on each of the trees in the forest. If T is an empty forest, then $\mathcal{P}(T)$ is also an empty forest.
3. Otherwise, if $T = a(T_1, \ldots, T_n)$ is a tree with the root element labeled a and with a forest of immediate subtrees $(T_1, \ldots, T_n)$, and if $\mathcal{R}_i$ is the first rule in $\mathcal{P}$ that matches
$$a(\mathcal{P}(T_1, \ldots, T_n)) \ , \tag{1}$$
then
$$\mathcal{P}(T) = \mathcal{R}_i(a(\mathcal{P}(T_1, \ldots, T_n))) \ . \tag{2}$$

Equations (1) and (2) mean that the current subtree is transformed *after* its subtrees have been transformed, i.e., the evaluation proceeds bottom-up. The rules are chosen in the order they appear. We refer to the remaining sections for practical examples of this evaluation order. We want to stress that there is no evaluation order defined between nodes at the same level in the tree. For example, the leaves (i.e., data) could be visited in an arbitrary order or even in parallel.

4 Transformation rules

In this section, we present the basic components of the TranSID language through small examples. In the following sections, we will present more TranSID operators and study some more advanced transformation examples. By a TranSID transformation we denote the process described in the previous section consisting of parsing, transforming and linearizing one or several input SGML documents.

A transformation program consists of transformation rules. A transformation rule has the following format.

Node type	Node name or *
`WHEN`	condition
`BECOMES`	set of new subtrees ;

Any node in the source tree, such as an element or an attribute is first recognized by a node clause and further tested for a condition. The type of the node can be qualified by the reserved words ELEMENT, ENTITY, PI, ATTRIBUTE, DATA, and NODE. Here, ELEMENT locates elements, ENTITY entities, etc. NODE accepts nodes of any type. All these reserved words must be succeeded by a node name or the asterisk * which stands for any name. The node clause and an optional condition (WHEN ...) constitute the source clause. If the condition holds, the node is replaced in the result tree by a forest of trees (actually a list of nodes) specified in the target clause beginning with BECOMES.

For example, the following TranSID program consisting of two rules prunes an SGML document retaining only those sections that in their title contain the string TranSID.

```
transformation begin
ELEMENT "SECTION"
WHEN current.children.having(this.name == "TITLE").children.find("TranSID")
BECOMES <"TRANSID_SEC">{current.children} ;

ELEMENT "SECTION"
WHEN not current.children.having(this.name == "TITLE").children.find("TranSID")
BECOMES null ;
end
```

The source clause of the first rule locates SECTION elements but only when its condition holds. The condition is stated as an *orientation expression* which consists of *locators* separated by dots ('.'). In the above expression, the locator current points to the node that is being transformed and the relative locator children locates the subelements of the current node. The evaluation of the expression flows from left to right. Every locator produces a list of nodes that is used as input for the next locator in the expression. The relative locator having selects the nodes that satisfy the condition expressed as a parameter of the having locator. In this case, its formal parameter this refers to each child of the current node at a time. The property operator name locates the name of the elements and the entire condition checks whether the found name equals the constant string TITLE. Only the nodes that satisfy this condition will be passed for the next locator, which locates the children of the TITLE elements. In this case, we assume them to be text strings. If the string matching operator find succeeds in locating text elements that contain the string TranSID, the entire expression evaluates to true.

The source clause of the rule above will match sections like

```
<SECTION>
        <TITLE>TranSID transformations</TITLE>
        ...
<SECTION>
```

So the first rule matches SECTION elements that contain the string TranSID in their TITLE element. The second rule will do exactly the opposite because its source clause contains the same condition negated.

The target clause of the first rule constructs new elements named TRANSID_SEC. The name of the new element is stated in SGML style between angle brackets < and >. The contents of the new element is stated in a list between braces. The contents is deduced by the orientation expression `current.children` that locates and copies all the subelements of the current node as the contents for the new `TRANSID_SEC` element. The second rule will remove all `SECTION` elements that do not satisfy the condition of the first rule. This is achieved by replacing those elements by the empty list `null`.

All TranSID examples including the example above are based on the DTD and the SGML document shown in Figure 3 in Section 6.

The transformation may not only modify elements but also their attributes. The following rule shows an example of removing an attribute and inserting its value as part of the text string of an element.

```
ELEMENT "BIBITEM"
BECOMES <"BIBITEM">{"[", current.attribute("BID").value.
              match_replace("-" -> "+";
                            ".$" -> this.tolower), "] ",
              current.children };
```

The rule locates `BIBITEM` elements and replaces them with corresponding elements where the value of the `BID` attribute (bibitem identifier) is written slightly modified inside brackets at the beginning of the actual bibliography item. The rule will replace the element

```
<BIBITEM BID="AHH-96A">Helena Ahonen, ...</BIBITEM>
```

with the corresponding element

```
<BIBITEM>[AHH+96a] Helena Ahonen, ...</BIBITEM>
```

The rule replaces the `BIBITEM` elements with new `BIBITEM` elements without attributes, but where the contents have been changed compared to the old element. The new contents is a string beginning with `[`, and followed by the `BID` attribute value where all `-` characters have been changed to `+` characters and where the last character of the attribute value has been changed to lower case if it is a letter. After the modified attribute value follows a `]` and then the original contents of the element.

5 TranSID operators

TranSID manipulates all data in the form of polymorphic lists. The only data type of the TranSID language is a *list* of nodes. A list can also be empty. A node can be an SGML element, an entity, a processing instruction, an attribute, or a data content node (i.e., a string). A node is equivalent to a singleton list. An element node can have both attributes and children. The attributes of an element have the element node as their parent, but no ordering between them is defined. Strings, integers and Boolean values are special cases of lists. In a conditional expression, an empty list is interpreted as false, and a non-empty list is interpreted as true.

An orientation expression has to start from a constant list or a variable. Variables are either built-in or user-defined. A built-in variable has a fixed meaning in a given context. For example, the built-in variable `source` locates the root of all the source trees, and `current` the node that is being transformed. User-defined variables are initiated with the operator `set(`*Variable*`)`, and they are local to a transformation rule. An orientation expression can contain additional locators and modifiers as discussed below. There is also a special list expression `null` which evaluates to an empty list.

TranSID programs may use a variety of tree transformation operators, string operators, regular expressions, etc. Our design objective was to have a declarative and complete set of tree transformation operators which are appropriate for modifying document trees.

Each TranSID expression returns a list. A comma is used simultaneously as a separator of expressions and as a list catenation operator. (We used them already in the previous example to build the content of the `BIBITEM` elements.) For example, `("word", 2, 3+4)` is a TranSID list consisting of a string and two integers.

Relative locators locate a new set of nodes from a node list. There are *positional locators* like `children` and `attributes` that locate the various subcomponents of an element. The positional locator `children` will locate the content (elements, entities, data nodes and processing instructions) of elements, whereas `descendants` locates all nodes in the subtree rooted at element nodes of its input list. Locator `parent` returns a list consisting of the parents of its input nodes, and `ancestors` returns a list of all the ancestors of the input nodes up to the root of the tree. Other positional locators are `left`, `right`, and `siblings`, which locate the left, the right, or all the siblings of nodes.

Relative locators also include *filtering locators* which select some of the nodes in their input list. Examples of these are `first`, `first(`*Integer*`)`, `having(`*Condition*`)`, `last`, `last(`*Integer*`)`, and `sublist(`*Integer*;*Integer*`)`. The locators `first` and `first(n)` locate the first one or the first `n` nodes of a list; `last` and `last(n)` the last one or the last `n` of them. The locator `having(`*Condition*`)` will test nodes in a list for a condition and return only those that satisfy the condition. The condition is a Boolean-valued expression which may use the formal variable `this` to refer to each of the tested nodes at a time. The locator `sublist(`$n;m$`)` returns a specified subset of nodes from a node list. The parameters of `sublist` are interpreted similarly to the dimension specifications in the HyTime standard [ISO92], which allows nodes to be located relative to either end of the list.

TranSID includes also powerful *list modification operators*. The operator `map(`*Condition*; *Replacement*`)` replaces each node that satisfies expression *Condition* by the value of the expression *Replacement*. The `map` operator may use the formal variable similarly to the `having` locator. (For an example see the example in next section.) The operator `glue(`*Condition*; *Condition*; *Replacement*`)` is a generalization of `map`, which is especially suitable for manipulating sub-sequences of lists as groups. It gathers consecutive nodes together if the nodes satisfy the first condition but not the second, and replaces them by the value of the expression *Replacement*. The list of located nodes may be referenced by the formal variable `these`.

For example, if we want to modify an SGML document by wrapping all consecutive `AUTHOR` elements in a single `AUTHORS` element we may use the `glue` operator to do this. The following rule

```
ELEMENT "HEAD"
BECOMES
    <"HEAD">{ current.children.glue(this.name == "AUTHOR";
                                    this.name != "AUTHOR";
                                    <"AUTHORS">{these})
             } ;
```

produces on the document in Figure 3 the output

```
  <HEAD>
    <TITLE LABEL="DOC">TranSID: an SGML tree transformation language</TITLE>
    <AUTHORS>
        <AUTHOR>Jani Jaakkola</AUTHOR>
        <AUTHOR>Pekka Kilpeläinen</AUTHOR>
        <AUTHOR>Greger Lindén</AUTHOR>
    </AUTHORS>
    <AFFILIAT>Department of Computer Science, P.O.Box 26, ...</AFFILIAT>
    <ABSTRACT>We present a powerful ...</ABSTRACT>
  </HEAD>
```

There are also operators for accessing *properties* of nodes. The operator `name` returns the name of an element, attribute or entity, while `attribute(`*Attribute name*`)` locates a certain attribute of an element. The operator `siblingnum` returns the ordinal number of the node among its siblings, and `samenum` returns the ordinal number of the node among siblings with the same name. The operator `count` returns the length of a list.

Several other operations have been included into the TranSID language. *String operations* and *regular expressions* were implemented as a student project work [MPP+97]. These operations include ordinary string operations such as comparison, catenation and search, as well as more sophisticated operations for string matching and replacement based on regular expressions. As an example consider the following rule.

```
    DATA *
    WHERE current.matches(" defini[a-z]+")
    BECOMES matches_replace("%a=(S[A-Z][A-Z]L)" -> "the standard ", %a) ;
```

The rule modifies data elements containing a word beginning with `defini`, like `definition` and `defining`. The rule will replace four-letter upper-case strings beginning with the letter `S` and ending with `L` by the string preceded by `the standard`, e.g., the strings `SGML` and `SMDL` will be replaced with `the standard SGML` and `the standard SMDL`, respectively. Regular expressions use local variables of the form *%name*. In this case, the variable `%a` is set to the matched string and later used in the replacing expression.

TranSID uses dynamical type conversions. Every operator expects lists of some specific type for their parameters. The required type depends on the operator: arithmetic operators expect integers as their operands, find(String) requires string type, having(Condition) requires an expression returning a Boolean type. Operators initiate automatic type conversion, which transforms the parameters into the required type. If the type conversion fails, a warning is issued.

6 An example transformation

In this section we show a more advanced transformation. The SGML document we use is a simplified version of this paper (Figure 3). It consists of a DTD followed by a document instance. Both the DTD and the document instance follow common SGML practice. The DTD shows that a `DOC` element contains a `HEAD` and a `BODY` element. The `HEAD` element contains a title, one or several authors, affiliation, and an abstract as well as zero or more keywords. An asterisk `*` stands in the DTD for zero or more repetitions, while the plus sign `+` stands for at least one repetition. Consecutive elements are connected with a comma while alternative components are connected with a bar. Text content is denoted by the keyword `#PCDATA`.

Some elements also have attributes. The element `TITLE` has a required attribute named `LABEL` of domain `ID`. The empty element `CITEREF` has a required attribute `RID` of domain `IDREF`. An empty element may have no contents and the end tag must be omitted, denoted here by the letter `O`. All other elements, when marked in the instance must have both a start tag and an end tag, which is implied by the string `- -`. The `BID` attribute is an optional attribute denoted by `#IMPLIED` and may be omitted in the SGML instance. The idea of these attributes is to provide a possibility to refer to references in the article text. `ID` and `IDREF` are two attribute domain types that are used for identifiers and identifier references, respectively.

In our example transformation, we show a way of producing HTML from an SGML document. The transformation constructs a table of contents containing links to the corresponding sections of the article (Figure 4). The program constructs anchors of the sections locally when processing the `TITLE` elements, while the table of contents is constructed when processing the root of the instance.

The transformation is specified by the following TranSID program.

```
ELEMENT "HEAD"
BECOMES <"H1">{current.origin.children.first.children} ;

ELEMENT "TITLE"
WHEN current.parent.name == "SECTION"
BECOMES
 <("H", current.ancestors.having(this.name == "SECTION").count+1)>{
 <"A" "NAME" = current.attribute("LABEL").value.set(v) "HREF" = ("#TOC_",v)>{
 current.children}
 } ;

ELEMENT "DOC"
BECOMES
 <"HTML">{
```

```
<!DOCTYPE DOC [
<!ELEMENT DOC          - - (HEAD, BODY)>
<!ELEMENT HEAD         - - (TITLE, AUTHOR+, AFFILIAT, ABSTRACT, KEYWORD*)>
<!ELEMENT (TITLE | AUTHOR | AFFILIAT | ABSTRACT | KEYWORD | P | BIBITEM)
                       - - (#PCDATA)>
<!ATTLIST TITLE        LABEL  ID      #REQUIRED>
<!ELEMENT BODY         - - (SECTION+, BIBLIO)>
<!ELEMENT SECTION      - - (TITLE, (P|SECTION|CITEREF)*)>
<!ELEMENT CITEREF      - O EMPTY>
<!ATTLIST CITEREF      RID  IDREF      #REQUIRED>

<!ELEMENT BIBLIO       - - (BIBITEM+)>
<!ATTLIST BIBITEM      BID  ID      #IMPLIED>
]>
<DOC>
  <HEAD>
    <TITLE LABEL="DOC">TranSID: an SGML tree transformation language</TITLE>
    <AUTHOR>Jani Jaakkola</AUTHOR>
    <AUTHOR>Pekka Kilpeläinen</AUTHOR>
    <AUTHOR>Greger Lindén</AUTHOR>
    <AFFILIAT>Department of Computer Science, P.O.Box 26, ...</AFFILIAT>
    <ABSTRACT>We present a powerful ...</ABSTRACT>
  </HEAD>
  <BODY>
    <SECTION><TITLE LABEL="INTRO">Introduction</TITLE>
    <P>In the world of ...</P>
    <P>On the other hand ...</P>
    <CITEREF RID="AHH-96A">
    </SECTION>
    <SECTION><TITLE LABEL="MODEL">Overall control and data model</TITLE>
      <P>A transformation engine ...</P>
    </SECTION>
    <SECTION><TITLE LABEL="SEMANTIC">Semi-formal semantics</TITLE>
      <P>...</P>
    </SECTION>
    <BIBLIO>
      <BIBITEM BID="AHH-96A">Helena Ahonen, ...</BIBITEM>
    </BIBLIO>
  </BODY>
</DOC>
```

Figure 3: An SGML document.

```
<HTML><BODY>
<UL>
<LI> <A NAME="TOC_INTRO" HREF="#INTRO"> Introduction</A> </LI>
<LI> <A NAME="TOC_MODEL" HREF="#MODEL"> Overall control and data model</A></LI>
<LI> <A NAME="TOC_SEMANTIC" HREF="#SEMANTIC"> Semi-formal semantics</A> </LI>
</UL>
<H1>TranSID: an SGML tree transformation language</H1>
<P><P><H2><A NAME="INTRO" HREF="#TOC_INTRO"> Introduction</A> </H2>
<P>In the world of ...
<P>On the other hand ...

<P><H2><A NAME="MODEL" HREF="#TOC_MODEL">Overall control and data model</A></H2>
<P>A transformation engine ...

<P><H2><A NAME="SEMANTIC" HREF="#TOC_SEMANTIC">Semi-formal semantics</A></H2>
<P>...

<P><P>Helena Ahonen, ...
</BODY></HTML>
```

Figure 4: The result of the example transformation.

```
   <"BODY">{
     <"UL">{
       current.descendants.having(this.name == "A").map(1;
         <"LI">{<"A" "NAME" = ("TOC_", this.attribute("NAME").value.set(v))
                     "HREF" = ("#", v)>{this.children}
               } )
    },                  // end of UL
    current.children}  // end of BODY
 } ;

ELEMENT *
BECOMES "<P>", current.children, "\n" ;
```

The first rule substitutes the `HEAD` element of the input document for an HTML `H1` element. The expression `current.origin.children.first.children` retrieves the contents of the first child of the original `HEAD` element in the source tree, which effectively inserts the contents of the original document `TITLE` element as the contents of the new `H1` title.

The second rule transforms section titles into Hn elements, where n stands for the level or depth of the title element. Sections will be denoted by `H2` elements, subsections by `H3` elements, etc. The new tag is computed by catenating after `H` the value of a TranSID expression which computes the nesting depth of the title in `SECTION` elements.

The content of the new Hn elements consists of an `A` element which contains the attributes `NAME` and `HREF`. The attribute `NAME` is assigned the value of the attribute `LABEL` in the source document, which is also stored in a variable `v`. The attribute `HREF` is set to the string `#TOC_` succeeded by the value of the variable. This attribute will serve as a back reference to the corresponding item in the table of contents. The content of the element `A` consists of the title text.

The third rule constructs the table of contents from the new `A` elements. As the transformation is executed bottom-up, the `TITLE` elements are processed before the `DOC` element is modified. The last rule creates an `HTML` element containing a `BODY` element which begins with the table of contents represented as an unordered list `UL`. Finally, the each list item consists of an `A` element that is computed in the following way. The orientation expression `current.descendants.having(this.name == "A")` locates all `A` elements in the modified document. The list of elements is transformed into a list of anchors (`A`) in list item (`LI`) using the `map` operator. Because the condition of the operator is always true, `map` will modify all elements in the list.

The last rule specifies that all other elements are formatted simply as their contents, preceded by an HTML paragraph tag `P` and followed by a line feed.

When the result is shown in an HTML browser the user can jump to the sections by clicking the titles in the table of contents. He can also jump back to the table of contents by clicking the titles in the sections.

7 Implementation

The TranSID evaluation environment has been implemented in C and C++ and it has been tested to run in the Linux, Solaris, and AIX environments. The environment consists of the SP SGML parser [Cla96b], a TranSID parser implemented with `yacc` and `lex`, and an evaluator and a linearizer both implemented in C.

TranSID uses lazy type conversions. Type conversions are performed implicitly when a certain type is needed. TranSID maintains an internal tree database for managing SGML trees. Unused nodes are automatically reclaimed using reference counters. The data structures in the database cannot contain cycles. This guarantees that the process cannot construct infinite trees with TranSID- expressions and that reference counters can be used for memory management.

Evaluation of expressions is implemented using lighter data structures than the tree database. Variables and lists are implemented using these data structures (thus variables and lists contain pointers to the real nodes, not trees of their own).

Memory management seems to be the bottle neck of the current implementation. The source and parse trees are constructed and maintained in main memory until the transformation is done. Therefore, memory usage may be high. Almost all transformations build their target trees from the source trees. The internal tree database is based on data structures that try to utilize this by sharing structures, i.e., by representing nodes of the target tree originating from the source tree as references to the nodes instead of copying them. This solution seems to be especially efficient in the execution model of TranSID.

8 Conclusion and future work

We have presented TranSID, a powerful tree-based transformation language, especially targeted at SGML applications. TranSID is a declarative language that lets the user specify SGML tree transformations in an easy and declarative manner. The tree-based strategy is a powerful but somewhat inefficient way of handling transformation. We therefore look for ways to minimize the tree structure kept in main memory during the transformation. We have also designed an event-based top-down strategy for performing simpler conversion tasks that may be included in the linearization phase. Additionally, we have designed and implemented a variation of the evaluation semantics where multiple rules can be applied to a single node, which simplifies the specifying of some complex transformations.

TranSID has been successfully used for transforming its own documentation from an SGML form into LaTeX and HTML. We are developing and experimenting TranSID further in a project dealing with intelligent document assembly [AHH+96a, AHH+96b]. In document assembly, new documents are constructed from a pool of documents. For this purpose, we have developed a server version of TranSID, which is able to respond to queries on a collection of structured documents. Used in this way TranSID will allow a user to assemble new valid SGML documents by locating, modifying and streamlining document fragments.

References

[AHH+96a] Helena Ahonen, Barbara Heikkinen, Oskari Heinonen, Jani Jaakkola, Pekka Kilpeläinen, Greger Lindén, and Heikki Mannila. Intelligent assembly of structured documents. Technical Report C-1996-40, Department of Computer Science, University of Helsinki, June 1996.

[AHH+96b] Helena Ahonen, Barbara Heikkinen, Oskari Heinonen, Jani Jaakkola, Pekka Kilpeläinen, Greger Lindén, and Heikki Mannila. Constructing tailored SGML documents. In Janne Saarela, editor, *Proceedings of SGML Finland 1996*, pages 106–116, October 1996.

[AIS96] AIS Berger-Levrault. *Balise Reference Manual, Release 3*, 1996.

[Cla96a] James Clark. Jade — James' DSSSL engine, November 1996. http://www.jclark.com/jade/.

[Cla96b] James Clark. *An SGML System confining to International Standard ISO 8879 — Standard Generalized Markup Language*, 1996. url: http//www.jclark.com/sp/.

[Exo93] Exoterica Corporation. *OmniMark Programmer's Guide*, 1993.

[Gol90] Charles F. Goldfarb. *The SGML Handbook*. Oxford University Press, 1990.

[Har93] Klaus Harbo. CoST version 0.2 — Copenhagen SGML Tool. Technical report, Department of Computer Science & Euromath Center, University of Copenhagen, 1993.

[ISO86] ISO. *Information Processing — Text and Office Systems — Standard Generalized Markup Language (SGML), ISO 8879*, 1986.

[ISO92] ISO and IEC. *Information technology — Hypermedia — Time-based Structuring Language (HyTime), ISO/IEC 10744*, 1992.

[ISO96] ISO and IEC. *Information technology — Processing Languages — Document Style Semantics and Specification Language (DSSSL) ISO/IEC 10179*, 1996.

[JKL96a] Jani Jaakkola, Pekka Kilpeläinen, and Greger Lindén. TranSID: A language for transforming SGML documents. Technical report, Department of Computer Science, University of Helsinki, June 1996.

[JKL96b] Jani Jaakkola, Pekka Kilpeläinen, and Greger Lindén. TranSID reference manual. Technical report, Department of Computer Science, University of Helsinki, September 1996.

[KPPM84] S. E. Keller, John A. Perkins, Teri F. Payton, and Susan P. Mardinly. Tree transformation techniques and experiences. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction, SIGPLAN Notices* 19(6), *Montreal, Canada*, pages 190–201, New York, June 1984. ACM, ACM.

[MID95] MID/Information Logistics Group GmbH. *MetaMorphosis Refence Manual*, 1995.

[MPP+97] Olli-Pekka Mahlamäki, Kimmo Paasiala, Santeri Pienimäki, Tomi Sarajisto, and Juha Sievänen. Implementation of an SGML transformation language (in Finnish). Project work report, Department of Computer Science, University of Helsinki, February 1997.