Department of Computer Science
Series of Publications A
Report A-1997-2

Structured Document Transformations

Greger Lindén

University of Helsinki
Finland

Department of Computer Science
Series of Publications A
Report A-1997-2

Structured Document Transformations

Greger Lindén

To be presented, with the permission of the Faculty of Science of the University of Helsinki, for public criticism in the Auditorium at the Department of Computer Science, Teollisuuskatu 23, Helsinki, on June 18th, 1997, at 10 o'clock.

University of Helsinki
Finland

Contact information

Postal address:
Department of Computer Science
P.O. Box 26 (Teollisuuskatu 23)
FIN-00014 University of Helsinki
Finland

Email address: postmaster@cs.Helsinki.FI (Internet)

URL: http://www.cs.Helsinki.FI/

Telephone: +358 9 708 51

Telefax: +358 9 708 44441

ISSN 1238-8645
ISBN 951-45-7766-3
Computing Reviews (1991) Classification: I.7.2, D.3.4, F.4.2
Helsinki 1997
Helsinki University Printing House

Structured Document Transformations

Greger Lindén

Department of Computer Science
P.O. Box 26, FIN-00014 University of Helsinki, Finland
Greger.Linden@cs.helsinki.fi, http://www.cs.helsinki.fi/~linden/

PhD Thesis, Series of Publications A, Report A-1997-2
Helsinki, June 1997, 122 pages
ISSN 1238-8645, ISBN 951-45-7766-3

Abstract

We present two techniques for transforming structured documents. The first technique, called TT-grammars, is based on earlier work by Keller et al., and has been extended to fit structured documents. TT-grammars assure that the constructed transformation will produce only syntactically correct output even if the source and target representations may be specified with two unrelated context-free grammars. We present a transformation generator called ALCHEMIST which is based on TT-grammars. ALCHEMIST has been extended with semantic actions in order to make it possible to build full scale transformations. ALCHEMIST has been extensively used in a large software project for building a bridge between two development environments. The second technique is a tree transformation method especially targeted at SGML documents. The technique employs a transformation language called TranSID, which is a declarative, high-level tree transformation language. TranSID does not require the user to specify a grammar for the target representation but instead gives full programming power for arbitrary tree modifications. Both ALCHEMIST and TranSID are fully operational on UNIX platforms.

Computing Reviews (1991) Categories and Subject Descriptors:
I.7.2 Text processing: Document preparation
D.3.4 Programming languages: Processors
F.4.2 Mathematical Logic and Formal Languages: Grammars and Other Rewriting Systems

General Terms:
Algorithms, Design

Additional Key Words and Phrases:
structured documents, tree transformation, SGML transformation

Acknowledgements

I am most grateful to my advisor professor Heikki Mannila, for his support and encouragement in my research. He introduced me to the research area of structured documents and text databases and has provided me with valuable guidance and insightful comments. I am very glad that he never gave up on me during this long-lasting work.

The Department of Computer Science, headed by professor Martti Tienari, has provided me with excellent working conditions. It has been an inspiring and intriguing research environment for which I thank all my colleagues.

My special thanks to Heikki Mannila, Erja Nikunen and Pekka Kilpeläinen, who as members of the RATI project in the late eighties, awoke my interest for structured documents. The knowledge achieved was put to good use in later projects. Thanks also to Henry Tirri and Inkeri Verkamo as well as to other members of the VITAL project. Together with Henry and Inkeri we designed and implemented the ALCHEMIST system. Thanks to Jukka-Pekka Vainio and Tomi Silander who helped programming the system. Thanks to Jani Jaakkola and Pekka Kilpeläinen of the SID project where we designed the TranSID system of which Jani made an excellent implementation. Thanks also to the other members of the SID project, Helena Ahonen, Barbara Heikkinen and Oskari Heinonen, for inspiring discussions.

I would also like to thank my friends, in particular Lars-Åke Olsson, and my relatives for their support and encouragement. Where would I be today without you?

The financial support of the Academy of Finland, the Technology Development Center (TEKES), and the Helsinki Graduate School of Computer Science and Engineering is gratefully acknowledged.

Contents

Chapter 1

Introduction

Data transformations are important in many computer applications. Despite the efforts for standardization in many areas, the end user is often at loss when it comes to the cooperation of commercial and custom-made application tools. The user usually ends up with a large set of diverse tools, such as editors, database tools, and formatters that are not compatible, or if one subset of tools work together, the set of programs will often not work with another set.

A commonly used solution is to employ another large set of *ad hoc transformation programs*, *conversion tools*, *source-to-source translators*, *specifically tailored transformation generators* and the like. Often the user is forced to finish off a transformation from the format of one tool to another manually without any other help than common sense. This usually applies only to the expert user, as the end user long ago has rejected most of his uncompatible tools and tries to get along with as few tools as possible.

A typical application area for data transformations is *document preparation*. Document preparation consists of the production and printing of documents. Most document preparation systems contain a plethora of different tools, such as editors, spell checkers, formatters, etc. for producing output on different media such as paper, screen, or CD-ROM. Commercial document preparation systems try to be as independent as possible. They contain features for formatting and printing within the preparation system. Should the user, however, want to move a part of the documents to another system, he/she soon realizes that font types, margin settings, or even logical markup such as sections and bibliography list markup have been be lost in the transformation.

Transformations are required not only in formatting. With the emergence of the World Wide Web and Hypertext Markup Language (HTML), we have seen a large need for transformations to and from HTML. HTML requires the user and HTML programmer to be one step more abstract in preparing documents. The user has to mark up the document with appropriate tags. In this way an HTML document can well be considered to contain *declarative markup*, i.e., the user tells *what* he/she wants to see in the browser (a section title or a list of items) instead of *procedural markup* that specifies *how* they should be presented. The document conforming to the HTML standard can be presented on different platforms and with different browsers as long as the browsers understand the standard.

1.1 Structured documents and transformations

HTML documents are one example of *structured documents* [AFQ89b]. The structured document model [AFQ89a] decomposes the document into logical parts. Examples of structured documents are manuals, telephone directories, dictionaries, and computer programs. A structured document may in fact be any document where there is more information than just the text itself. This additional information is called the structure.[1] Computer programs and structured documents follow a well-specified standard of what is correct programming and what is not. Any structured document can be specified with the SGML (Standard Generalized Markup Language) [ISO86, Gol90] or the ODA (Open Document Architecture) [ISO89] standards ensuring that the documents can be used on different platforms and in different applications in a standard way.

Example 1.1 In Figure 1.1 we see an example of a structured document. The document has been marked up with SGML and it also contains the document type definition (DTD). The DTD is loosely based on the dictionary description presented in [BBT92]. The document contents (the dictionary entry for *spaz*) is based on an entry in the Oxford English Dictionary [Oxf96] but has been slightly modified and shortened. We base many of the following examples on this first one and shall therefore explain the example in detail.

Our example dictionary contains entries of words where each entry contains the headword, its pronunciation, part of speech, and etymology as well as its separate senses. A sense contains a definition and possibly several quotations each consisting of a date, author, work, and quotation text.

A structural component of a document is in SGML called an element. The *document type definition* (DTD) that describes this document tells us

[1] A related view is that all documents have (implicit) structure, but in some documents it has been marked explicitly [Möl94].

```
<!DOCTYPE Dictionary [
<!ELEMENT Dictionary     - -    (Entry)+>
<!ELEMENT Entry          - -    (HWGroup,Etymology?,
                                Sense+)>
<!ELEMENT HWGroup        - -    (Headword,Pronunciation,
                                PartofSpeech)>
<!ELEMENT Headword       - -    (#PCDATA)>
<!ELEMENT Pronunciation  - -    (#PCDATA)>
<!ELEMENT PartofSpeech   - -    (#PCDATA)>
<!ELEMENT Etymology      - -    (#PCDATA)>
<!ELEMENT Sense          - -    (Definition,Quotation+)>
<!ELEMENT Definition     - -    (#PCDATA)>
<!ELEMENT Quotation      - -    (Date,Author?,Work,Text)>
<!ELEMENT Date           - -    (#PCDATA)>
<!ELEMENT Author         - -    (#PCDATA)>
<!ELEMENT Work           - -    (#PCDATA)>
<!ELEMENT Text           - -    (#PCDATA)>
]>
<Dictionary>
  <Entry>
    <HWGroup>
      <Headword>spaz</Headword>
      <Pronunciation>spæz</Pronunciation>
      <PartofSpeech>n</PartofSpeech>
    </HWGroup>
    <Etymology>Abbreviation of spastic n.</Etymology>
    <Sense>
      <Definition>= spastic</Definition>
      <Quotation>
        <Date>1965</Date><Author>P. Kael</Author>
        <Work>I lost it at the movies III. 259</Work>
        <Text>The term that American teen-agers now use as
              the opposite of ‘tough’ is ‘spaz’.</Text>
      </Quotation>
      <Quotation>
        <Date>1975</Date><Author>M. Amis</Author>
        <Work>Dead babies viii. 47</Work>
        <Text>I know how long, you little spaz.</Text>
      </Quotation>
    </Sense>
  </Entry>
</Dictionary>
```

Figure 1.1: An example of a structured document marked up with SGML.

that an SGML document called `Dictionary` consists of one or more `Entry` elements. Each `Entry` element consists of a `HWGroup` element followed by one or zero `Etymology` elements, followed by one or more `Sense` elements. The plus symbol (`+`) in this description stands for one or more elements, the question mark (`?`) for one or zero elements, and the comma (`,`) catenates the elements. The `#PCDATA` element denotes the contents of elements that do not (in this description) contain other elements but text only. We describe SGML in more detail in Section 2.4.

The DTD is followed by the *document instance* where the text has been *marked* with element names described in the DTD. Such marks are also called *tags*. This particular instance contains one entry that consists of one headword group, an etymology, and a sense. The sense contains one definition and two quotations, etc. □

A structured document is not an end *per se*. For displaying, modifying, extracting information, or printing we need to transform the document into another representation. Even if the document has been marked up with SGML, the standard only specifies the syntax of the markup, not the semantics of the document. As a matter of fact, this is the purpose of SGML: to provide a standard way of marking a structured document in declarative markup, not procedural. It is up to the document application and the user to interpret the markup when transforming the document into an appropriate format. This format may well be the proprietary format of a commercial document preparation system, or it may be HTML for presenting the document in an HTML browser.

We call this the paradigm of a *logical document vs. document views*. Just as in database applications, the user may keep a logical document corresponding to the database contents. When modifying or printing the document, the user may choose between several views of the document contents. He/She may choose to use different formatting commands for printing on paper and on screen, respectively. The user may also filter out certain information, e.g., choosing to print only section titles or bibliographic references. He/She may even choose to add information to the document by allowing updatable views of the document. Many document preparation systems provide multiple views of a document [QV86, FS88, CH88, KLMN90, Bro91]. Especially, these systems often show a *textual view* and a *formatted view* of a document simultaneously. The paradigm also usually requires inverse transformations. If the views are updatable, transformations must be specified in both directions, from the logical document to the document views and vice versa.

```
{\bf spaz} (spæz) {\em n.} [Abbreviation of spastic n.]
= spastic
\newline
{\bf 1965} {\sc P. Kael}
{\em I lost it at the movies III. 259}
The term that American teen-agers now use as the
opposite of ‘tough’ is ‘spaz’.
\newline
{\bf 1975} {\sc M. Amis} {\em Dead babies viii. 47}
I know how long, you little spaz.
```

Figure 1.2: An example of a textual view.

Example 1.2 In Figures 1.2 and 1.3, we see two views of the document in Example 1.1. Figure 1.2 shows a view of the document where the SGML tags have been removed and LaTeX [Lam86] formatting commands have been inserted in the text. Figure 1.3 shows a formatted view produced by LaTeX.

> **spaz** (spæz) *n.* [Abbreviation of spastic n.] = spastic
> **1965** P. KAEL *I lost it at the movies III. 259* The term that American teen-agers now use as the opposite of ‘tough’ is ‘spaz’.
> **1975** M. AMIS *Dead babies viii. 47* I know how long, you little spaz.

Figure 1.3: An example of a formatted view.

□

Sometimes there is need for processing more than one document at a time. The user may choose to combine several structured documents into one, by transforming them to use the same structure. Such transformations are needed especially in document assembly [AHH+96a, AHH+96b], where the user needs to combine document fragments from different documents. The assembled document is modified to conform to a certain structure and can then be presented consistently on different media.

1.2 Transformation solutions

As mentioned earlier, we are neither short of solutions in the general case (any data transformation) nor in the specific document transformation case. Many *ad hoc* transformations have been defined and implemented for some of the problematic transformations mentioned above. For example, the `LaTeX2HTML` and `HTML2LaTeX` [Dra96] set of transformation programs transform between the LaTeX document preparation system [Lam86] and the HTML format. Many commercial document preparation systems have similar extensions that allow the user to produce at least some non-proprietary formats such as HTML and SGML. On the other hand, if no transformation module is available for the needed transformation, the user may have to build the transformation from scratch. This task is, however, often error-prone. It may also be tedious as the final transformation module often contains similar but complicated parts that the user have specified in other transformations.

A convenient way to avoid these drawbacks is to use *transformation generators*. A transformation generator lets the user specify a transformation, and the generator then produces program code for the corresponding transformation module. Instead of building a transformation from scratch, the user is able to produce a transformation of some specified representations. Examples of typical parser generators are `yacc` [Joh75] and PCCTS [PDC92].

These parser generators, also called *compiler-compilers*, tend to concentrate on the *front-end* of the transformation process. The user may specify the input representation in great detail, ensuring that no incorrect document is accepted by the transformation module. The input representation is specified, e.g., using a *context-free grammar* (see Section 2.1) that restricts the input documents over this certain grammar. On the other hand, the output side or the *back-end* of the transformation may require much less specification. It can be argued that this is the strong side of parser generators. The user may use any available instructions from the underlying programming language when he/she defines the output of the transformation. This openness puts some special stress on the user to write correct output instructions so that the transformation output actually follows the expected syntax.

We will see that by using techniques from compiler-compiler theory it is possible to build transformations for structured documents easily and with less effort. Available parser generators require an *input grammar* of the input document of a transformation. In most cases, structured documents follow such grammars. SGML documents even require such a *document type definition* (DTD) to exist. Also several other document preparation systems use the grammar concept for defining the structure of the documents. Examples include publicly or commercially available systems such as LaTeX [Lam86] and Grif [QV86] and many research prototypes such as HST [KLMN90] and Syndoc [KP91]. Applying then a strict front-end to document transformation is not a problem. The input documents follow a certain syntax and can be checked with a parser. All of these systems have a more or less open back-end. For example, in the case of LaTeX the back-end has been specified before, while in the case of HST the user has full control in its specification.

Many SGML transformation languages are based on the idea of a parser with a well-specified front-end and an open back-end. In *event-based languages* such as OmniMark [Exo93] and CoST [Har93], the user may specify output actions to be performed for each *event* the parser encounters in the input stream. In both cases, general SGML parsers are used at the front end, producing a stream of events such as *start section element* and *end list element*. This stream of events is called the element information structure set (ESIS) [Gol90]. At each event, the user may specify how to process the current document fragment, but already processed parts or yet unseen parts are not accessible. Above all, there is no restriction on the output the user may produce, and therefore also the transformations do not guarantee that the output representation is correct.

Example 1.3 Figure 1.4 shows a small example of an event-based SGML transformation language CoST [Har93]. Each SGML element has its own rule, which states the actions to be taken when either a start or end tag, or the contents of an element, is encountered in the input stream. This transformation surrounds the headwords in the dictionary with the strings `{\bf` and `}`. In CoST, the instruction `puts` automatically prints a newline character where it is not explicitly prohibited by the `nonewline` instruction. □

1.3 ALCHEMIST — a powerful transformation generator

In this thesis we present a simple and powerful tool ALCHEMIST [TL94a, LTV96] for specifying and generating transformations of structured documents. ALCHEMIST requires the user to specify both the input and output document representations. The tool then generates transformation modules that accept only correct input documents and produce only correct output documents according to the specifications. Like many other doc-

```
element Entry {
  start { puts stdout "" }
  end { }
}

element HWGroup {
  start { puts stdout "" }
  end { }
}

element Headword {
  start { puts stdout "{\bf " nonewline }
  end { puts stdout "}"}
}

TEXT { puts stdout $data nonewline }
```

Figure 1.4: An example of the event-based SGML transformation language CoST.

ument transformation systems (e.g., HST [KLMN90], ICA [MBO93], and Syndoc [KP93, Kui96]), ALCHEMIST relies on context-free grammars for representation specification. Unlike many systems, however, the representation grammars may be totally *unrelated*. The actual transformation is specified based on these grammars.

ALCHEMIST adds some other properties to transformation generation as well. Many transformations cannot be completely specified before-hand, but require user interaction. With ALCHEMIST the user may specify where and when and how he/she wants to interact in the generated transformation. ALCHEMIST is used mainly for producing transformations between two different structured document representations *without* an explicit intermediate format between the representations. In a set of different representations we have to specify a quadratic number of transformations to be able to provide transformation between any arbitrary chosen representations. That is, if we have n different representations, we need to build $n(n-1)$ or $O(n^2)$ different transformations to be able to transform any representation into any other representation in the set. On the other hand, if a canonical representation for the whole set is found, we will manage with only $2n$ transformations. In that case there are two transformations for each representation, one transforming to the canonical representation and one transforming from the canonical representation to the specific representation. In the case of ALCHEMIST, we do not require explicit canonical representations. Still there is no disadvantage because we have the choice of defining one if it helps.

Example 1.4 Figure 1.5 shows an example of an ALCHEMIST transformation specification where both input and output specifications are required. The specification specifies the same transformation as in Example 1.3, i.e., a transformation that surrounds the headwords in the dictionary with the strings `{\bf` and `}` and ends each entry and each headword group with a newline character. The input representation is specified on the left, the corresponding output representation on the right. The figure gives the flavor of an ALCHEMIST mapping. We give a more detailed description of the notation in Chapter 3.

```
Entry ->                    Entry.Entry -> "\n"
    HWGroup                     HWGroup.HWGroup
    Etymology                   Etymology.Etymology
    Senses                      Senses.Senses

HWGroup ->                  HWGroup.HWGroup ->
    Headword                    "{\\bf " Headword.Headword
    Pronunciation               "} "
    PartofSpeech                Pronunciation.Pronunciation
                                PartofSpeech.PartofSpeech

Headword -> IDENTIFIER      Headword.Headword ->
                                IDENTIFIER.IDENTIFIER
```

Figure 1.5: An example of an ALCHEMIST transformation specification.

□

ALCHEMIST is now fully operational in UNIX environments with both a graphical and a textual interface.

1.4 TranSID — an SGML transformer

In addition to ALCHEMIST, we also present an SGML transformation language called TranSID [JKL96a, JKL96b, JKL97], which is based on tree transformations. TranSID requires a specification of the input representation, in the way that the input SGML document must have a DTD, but no output DTD is used. TranSID contains normal tree transformation operations [JKL96b] and it has been extended with powerful string operations and regular expressions [MPP+97].

Design goals of the TranSID language included declarativeness, simpleness and implementability with reasonable effort. Special features include a *bottom-up evaluation* process and a possibility to restrain the transformation to the event-based strategy. The event-based or top-down strategy is sufficient for simple formatting of the SGML document. Bottom-up evaluation is a declarative way of defining some transformations that would be awkward to define in a top-down manner. The TranSID language also includes high level declarative commands that frees 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].

Example 1.5 Figure 1.6 shows an example of a TranSID program that performs the same transformation as in Example 1.3. The two first rules modify the document as in the earlier example. The last rule removes the SGML tags from all other elements. □

```
transformation begin

ELEMENT "Entry"
BECOMES "\n", current.children ;

ELEMENT "HWGroup"
BECOMES "{\\bf ",
        current.children.having(this.name == "Headword"),
        "} ",
        current.children.having(this.name != "Headword");

ELEMENT *
BECOMES current.children ;

end
```

Figure 1.6: An example of a TranSID transformation specification.

Even if the two-grammar transformations implemented by ALCHEMIST are a sort of ideal solution to the document transformation problem, there are many transformations where the ALCHEMIST solution is quite complicated. TranSID requires the user to specify only those parts of the document that are modified. The part of the document that remains intact does not have to be specified. Both systems require the user to specify the source representation in a source grammar and DTD, respectively, but only ALCHEMIST requires a target grammar as well. Therefore, only ALCHEMIST can guarantee the correctness of the target. TranSID also lets the user reference the complete input document during the transformation.

1.5 Aim and organization of the thesis

In this thesis, we aim to show the benefits of using TT-grammars in some transformations, and the syntax-directed approach in others. We present both ALCHEMIST and TranSID environments with examples. Our empirical experience shows that they complement each other, thereby covering most transformation problems that arise in structured document management.

The rest of this thesis is organized as follows. In Section 2, we define some general concepts from compiler-compiler theory which form a basis for our transformation systems. We also take a closer look at SGML and its background. We give a general presentation of structured transformations in Section 3 and present syntax-directed translations suitable for document transformations as well as tree transformation grammars on which ALCHEMIST transformations are based. We also briefly discuss event-driven transformations which are common in the SGML case. In Section 4 we see how the tree transformation grammar technique was implemented in ALCHEMIST, and in Section 5 we discuss the tree transformations of TranSID. We report our experience and evaluation of ALCHEMIST and TranSID in Section 6. Section 7 we give an overview of related work in the area of structured document transformations and transformation generators. Finally, Section 8 is a short conclusion.

Chapter 2

Preliminaries

The possible structure of a set of structured documents can be defined using *grammars*. As we saw in the last chapter, SGML documents are described through document type definitions which are one kind of grammars. The most important part in a grammar is a set of rules that describes very exactly how to construct documents (or strings) that are allowed according to the definition.

If we have an arbitrary document, we can then use *parsing* to compare the document with a given grammar to check if it is *consistent* with the grammar. Before parsing the document, we use *lexical analysis* to recognize words, reserved words, formatting commands, etc., in the document. Parsing constructs a *parse tree* from the document, a data structure, representing the document.

Once the document has been parsed, we can begin its transformation. A parsed document gives us more opportunities for transformation and tells us more details about the document itself. For example, we see how different sections of the document are related to each other structurally. We can even use several different grammars for one single document type, thereby achieving different structural views of one document.

Context-free grammars [Cho56] are frequently used in document management systems [BR84, CIV86, GT87, FQA88, QV86, KLMN90, KP93, Kui96]. Context-free grammars constitute a restricted set of all possible grammars and are easier to process than general grammars [ASU86]. In this chapter, we define context-free grammars and we also briefly review different parsing techniques for context-free grammars. Parsing is often used as a front-end in a transformation. Such a transformation is called a *syntax-directed translation* because the reading and checking of the document is directed by its syntax defined in a grammar. We shall look at several syntax-directed translation methods that are suitable for document transformation.

2.1 Context-free grammars

With the help of *context-free grammars* [Cho56] we can precisely define the structure of the strings in a document, or the documents in a class. Formally, a context-free grammar for a class of documents is a quadruple $G = (N, \Sigma, P, S)$, where N is a finite set of *nonterminals*, Σ is a finite set of *terminals*, P is a set of *productions*, and S is a special nonterminal called the *start symbol*. The grammar G defines the language $L(G)$ that contains all possible sequences of terminals that are correct according to the grammar G.

The set of terminals contains all the symbols or *tokens* that may occur in the document according to the grammar. As a matter of fact, any document over a given grammar consists of a sequence of terminals of that grammar. Also the sequence of zero terminals is a string called the *empty string* and denoted ϵ. The set of all possible terminal strings including the empty string in Σ is denoted Σ^*. Thereby we can say that any document over the grammar G is in Σ^*.

On the other hand, the nonterminals in N do not occur in the documents. They can be thought of as representing sets of sequences of terminals in the language $L(G)$. Especially, the start symbol S represents all possible sequences of terminals in the language. The sets of terminals and nonterminals together are called the *vocabulary* of the language. A sequence of nonterminals and terminals is called a *string*. We denote the vocabulary $N \cup \Sigma$ by V and all possible strings over the grammar are denoted V^*.

The productions of the grammar tell us how to construct correct sentences in the language $L(G)$. A production is denoted $A \rightarrow \alpha$, where A is a nonterminal and α is a string consisting of terminals and nonterminals, i.e., $A \in N$ and $\alpha \in V^*$. We call A the *left hand side* of the production and α the *right hand side* of the production. The symbol $\rightarrow$ is the *rewrite symbol* of the production.

The productions are used in the following way to construct sentences in the language $L(G)$. We start with the start symbol S and choose any production $S \rightarrow \alpha$ with the symbol S on its left hand side. We then replace or *rewrite* the nonterminal S with the right hand side α of the production. We continue by choosing any nonterminal A in α and replacing it with the right hand side β of a corresponding production $A \rightarrow \beta$. We continue rewriting nonterminals until we are left with only terminals. These cannot be rewritten because no terminal can occur on the left hand side of a production. Every such rewrite step is called a *derivation step* and is denoted by $\Rightarrow$. The entire process from the start symbol S to the final terminal string is called a *derivation*.

Formally, we have that the relation $\alpha \Rightarrow \beta$ holds if and only if there are strings α_1, α_2, A, and γ, such that $\alpha = \alpha_1 A \alpha_2$ and $\beta = \alpha_1 \gamma \alpha_2$ and $A \rightarrow \gamma$ is a production in the system. We also say that α *derives* β. The *reflexive transitive closure* of the relation $\Rightarrow$ is denoted by $\Rightarrow^*$. Thus $\alpha \Rightarrow^* \beta$ holds if and only if there are $n \geq 1$ strings $\gamma_1, \ldots, \gamma_n$ such that $\alpha = \gamma_1$, $\beta = \gamma_n$ and $\gamma_i \Rightarrow \gamma_{i+1}$ holds for every $i = 1, \ldots, n-1$. Correspondingly, the *transitive closure* of the relation $\Rightarrow$ is denoted by $\Rightarrow^+$, and $\alpha \Rightarrow^+ \beta$ holds if and only if there are $n \geq 2$ strings $\gamma_1, \ldots, \gamma_n$ such that $\alpha = \gamma_1$, $\beta = \gamma_n$ and $\gamma_i \Rightarrow \gamma_{i+1}$ holds for every $i = 1, \ldots, n-1$. We say that α *derives* β *in zero or more steps* if $\alpha \Rightarrow^* \beta$ and that α *derives* β *in one or more steps* if $\alpha \Rightarrow^+ \beta$.

In the following, we use italicized capital letters from the beginning of the English alphabet and strings beginning with capital letters for nonterminals, e.g., A, B, *Journal*. Terminals are denoted by boldface strings, e.g., `a`, `begin`. For readability, we may surround terminals by quotation marks, e.g., '`.`' or "`a`". Capital italicized letters from the end of the alphabet stand for terminals or nonterminals, e.g., X, Y. The Greek letters α, β, ... denote any string over V.

In the following example, we use a special kind of *text* terminals, denoted by the terminal `TEXT`. The `TEXT` terminal is in this case a string consisting of any characters in the language. This is a useful abstraction. When defining our context-free grammar we do not have to write out every single possible string that may appear in our language. This simplifies our grammar considerably.

Example 2.1 In Figure 2.1 we see an example of a context-free grammar defining the dictionary document in Example 1.1. The context-free grammar is very similar to the DTD in Example 1.1, but we have used recursive productions instead of iteration symbols `*` and `+`. (It may be noted that an SGML DTD is generally not a context-free grammar.)

A derivation that yields a sentence in the language of the context-free grammar is shown in Figure 2.2. Each derivation step replaces a nonterminal with the right hand side of a production that has the nonterminal as its left hand symbol. The derivation here has been made left to right, where always the left-most nonterminal is rewritten.

□

Dictionary	$\rightarrow$	*Entries*	*Senses*	$\rightarrow$	*Senses* Sense
Entries	$\rightarrow$	*Entries Entry*	*Senses*	$\rightarrow$	*Sense*
Entries	$\rightarrow$	*Entry*	*Sense*	$\rightarrow$	*Definition*
Entry	$\rightarrow$	*HWGroup*			*Quotations*
		Senses	*Quotations*	$\rightarrow$	*Quotations*
Entry	$\rightarrow$	*HWGroup*			*Quotation*
		Etymology	*Quotations*	$\rightarrow$	*Quotation*
		Senses	*Definition*	$\rightarrow$	`TEXT`
HWGroup	$\rightarrow$	*Headword*	*Quotation*	$\rightarrow$	*Date Work Text*
		Pronunciation	*Quotation*	$\rightarrow$	*Date Author*
		PartofSpeech			*Work Text*
Headword	$\rightarrow$	`TEXT`	*Date*	$\rightarrow$	`TEXT`
Pronunciation	$\rightarrow$	`TEXT`	*Author*	$\rightarrow$	`TEXT`
PartofSpeech	$\rightarrow$	$\texttt{n.} \mid \texttt{v.} \mid \texttt{a.}$	*Work*	$\rightarrow$	`TEXT`
Etymology	$\rightarrow$	`TEXT`	*Text*	$\rightarrow$	`TEXT`

Figure 2.1: A context-free grammar.

2.2 Parse trees

One way to show how a sentence in a derivation was produced is to write out every single step in the derivation. A more graphical way is to draw a *parse tree*, showing how the nonterminals at each derivation step are replaced by new strings. A parse tree is a special case of a *tree*. A tree consists of a distinguished *node* n called the *root* of the tree and an ordered sequence of k *subtrees* $t_1, t_2, \ldots, t_k$, where $k \geq 0$. The subtrees are also trees. Denote the root of tree t_i by n_i; then n is the *parent* of n_i and $n_1, n_2, \ldots, n_k$ the children of n. The node n, the roots of $t_1, t_2, \ldots, t_k$, the roots of their subtrees, etc., are called the *nodes* of the tree. Those nodes that have no children are called *leaves*. All other nodes are *interior* nodes.

We may associate a *label* with each node in the tree. Given a context-free grammar $G = (N, \Sigma, P, S)$, we have that a *parse tree* is a tree, where [ASU86]:

1. The root is labeled by the start symbol.
2. Each leaf is labeled by a terminal or by ϵ.
3. Each interior node is labeled by a nonterminal.

Dictionary
$\Rightarrow$ *Entries*
$\Rightarrow$ *Entry*
$\Rightarrow$ *HWGroup Etymology Senses*
$\Rightarrow$ *Headword Pronunciation PartofSpeech Etymology Senses*
$\Rightarrow$ `spaz` *Pronunciation PartofSpeech Etymology Senses*
$\Rightarrow$ `spaz spæz` *PartofSpeech Etymology Senses*
$\Rightarrow$ `spaz spæz n.` *Etymology Senses*
$\Rightarrow$ `spaz spæz n. Abbreviation of spastic n.` *Senses*
$\Rightarrow$ `spaz spæz n. Abbreviation of spastic n.` *Sense*
$\Rightarrow$ `spaz spæz n. Abbreviation of spastic n.` *Definition Quotations*
$\Rightarrow$ `spaz spæz n. Abbreviation of spastic n. = spastic` *Quotations*
$\Rightarrow$
$\vdots$
$\Rightarrow$ `spaz spæz n. Abbreviation of spastic n. = spastic 1965 P. Kael I lost it at the movies III. 259 The term that American teen-agers now use as the opposite of 'tough' is 'spaz'. 1975 M. Amis Dead babies viii. 47` *Text*
$\Rightarrow$ `spaz spæz n. Abbreviation of spastic n. = spastic 1965 P. Kael I lost it at the movies III. 259 The term that American teen-agers now use as the opposite of 'tough' is 'spaz'. 1975 M. Amis Dead babies viii. 47 I know how long, you little spaz.`

Figure 2.2: Part of a derivation.

4. If A is the nonterminal labeling some interior node m and X_1, X_2, ..., X_n are the labels of the children of that node from left to right, then $A \rightarrow X_1X_2\cdots X_n$ is a production of the grammar G. Here $X_1, X_2, \ldots, X_n$ stand for symbols that are either terminals or nonterminals. As a special case, if $A \rightarrow \epsilon$ is a production in P, then a node labeled A may have a single child labeled ϵ.

We say that the production $A \rightarrow X_1X_2\cdots X_n$ has been *expanded* or *applied* at node m. The leaves of the parse tree read from left to right form the *yield* or the *frontier* of the tree that is the string *generated* or *derived* from the nonterminal at the root of the parse tree. The grammar G is *unambiguous* if for each terminal string there is at most one possible parse tree.

Example 2.2 Figure 2.3 shows the parse tree of the derivation in Example 2.1. The root is at the top and the leaves at the bottom of the picture.

The `TEXT` terminals have been replaced with the text from the derivation.
□

2.3 Parsing

Context-free grammars can be used to describe structured documents. We can check that a document follows a certain grammar by constructing a parse tree for the document. What we now need is an automatic way of matching a document against a grammar. Such a method is called *parsing*.

Most parsing methods fall into two classes, called *top-down* and *bottom-up methods*. These terms refer to the order in which the nodes in the trees are constructed. In top-down methods we start with the start symbol and build the tree downwards towards the leaves. In bottom-up methods we start with the leaves and construct the internal nodes before reaching the root.

Top-down parsing is a simpler method where the parser may be constructed manually. For example, a top-down method called *recursive-descent parsing* executes recursive procedures to process the input. A form of recursive-descent parsing is *predictive parsing* which uses the next input symbol (the lookahead symbol) to choose which production is used next. Every nonterminal has an associated procedure which is called when the nonterminal is processed. We start with the start symbol and call its associated procedure. First, the procedure chooses the production to apply at this stage of the input by reading the next input symbol. If there are two such possible productions in the grammar, this method cannot be used. Then the control is shifted to the procedures associated with the right hand side symbols of the chosen production. If a nonterminal symbol is processed, the corresponding procedure is called. If, on the other hand a terminal is being processed, the next symbol is read from the input. If the next input symbol does not match the symbol in the production an error is reported.

Recursive descent parsing is considered simple and easy to understand, but it can only be used for context-free grammars of a fairly restricted type [ASU86]. Bottom-up parsers can handle a larger class of grammars. One such parsing method is *shift-reduce parsing*. Shift-reduce parsing constructs the parse tree beginning with the leaves. The process tries to *reduce* the input string to the start symbol. At each step a substring matching the right hand side of a production is replaced by the left hand side symbol of the production.

One general method of shift-reduce parsing is called LR(k) parsing

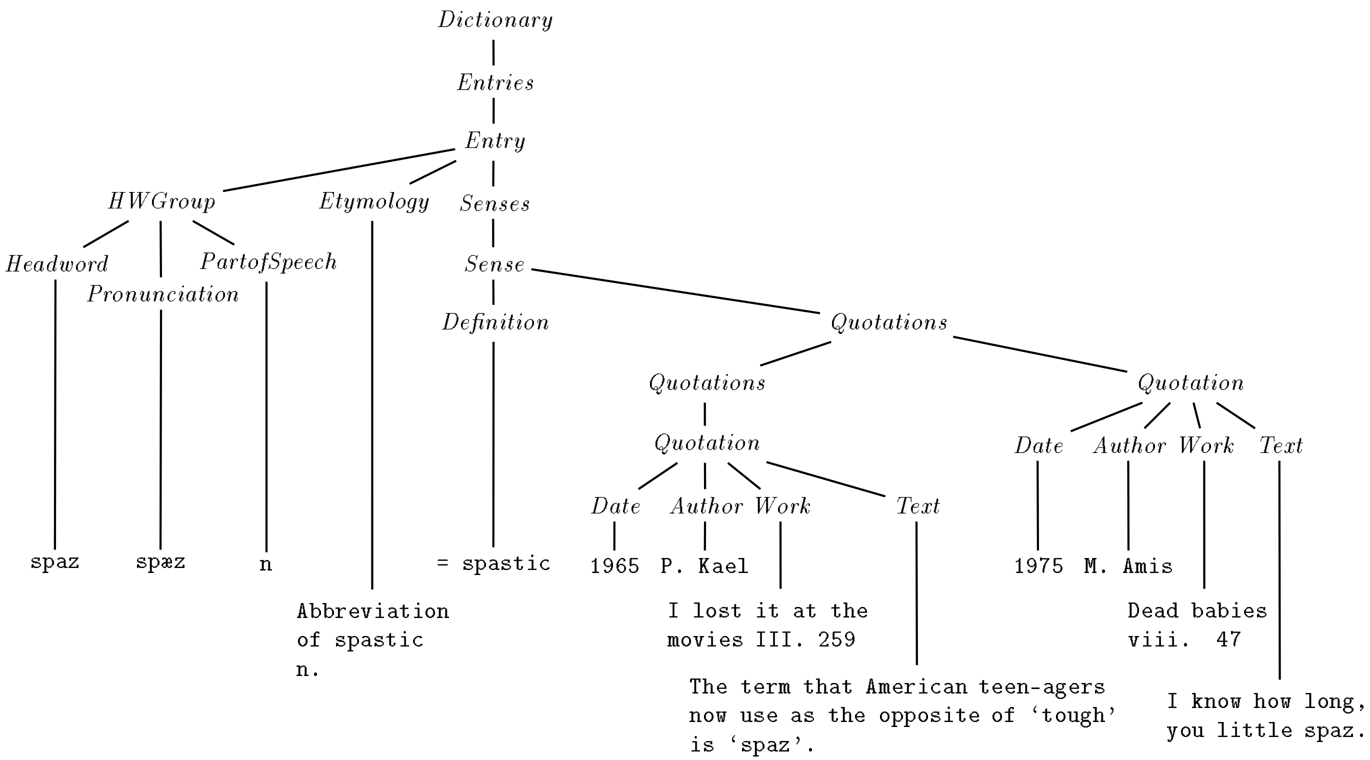


Figure 2.3: The parse tree of the derivation in Figure 2.2.

[Knu65]. LR(k) parsing needs to look ahead only k symbols in the input to be able to perform the parsing process. LR parsers can be constructed to recognize almost all programming language constructs for which context-free grammars can be written [ASU86]. It is, however, difficult to construct an LR parser by hand. Fortunately there are several LR parser generators available.

Again in both top-down and bottom-up parsing methods we define special terminals, such as texts, identifiers and numbers. The parser treats them as terminals. The lexical analysis, preceding the parsing and performed by a *scanner*, recognizes the form of the strings in the input and passes single tokens, e.g., `IDENTIFIER`, `TEXT`, and `NUMBER`, to the parser.

2.4 The Standard Generalized Markup Language

All documents have structure. By marking the structure explicitly in the document, applications may take advantage of the structure as well as the contents. By using a standard markup, the document becomes more portable and there is a greater possibility to find tools for updating and formatting the document [Möl94] .

The Standard Generalized Markup Language (SGML, [ISO86]) is a meta-language standard for defining markup languages. The markup language defines a set of markup conventions used together for encoding texts. It specifies what markup is allowed, what markup is required, how markup is distinguished from text, and what the markup means [SMB93].

SGML enforces *descriptive markup* which provides names to categorize parts of a document. By contrast a *procedural markup system* defines what processing is to be carried out at particular points in a document. With descriptive markup the document can be processed by many different types of software, each of which may modify or present the document in its own way.

We saw an example of an SGML document in the previous section (Figure 1.1 on page 3). We here only briefly define the most central features present in SGML. A structural component of a document is in SGML called an *element*. Elements are explicitly marked in a *document instance* by a *start tag* and an *end tag* containing the name of the element. For example, in the previous document, the headword is marked as an element by the start tag `<HeadWord>` and `</HeadWord>`. The text between the tags, i.e., in this case `spaz`, is the *content* of the element. Elements may have properties assigned to them in the form of *attributes*. Attributes are specified in the start tag of an element, e.g., the notation `<HeadWord Language=English>`

states that the `HeadWord` element has an attribute `Language` with value `English`. Documents may contain *entities* which reference external files, e.g., pictures, or they may also be used as macros or abbreviations for constant strings. Documents may also contain processing instructions that hold application-dependent information.

A document instance must have a *document type definition* (DTD), i.e., a grammar describing the structure of the document. In the DTD, every element is defined by its *content model* (or production) that shows which other elements or data it may contain. Attributes and entities must also be defined in the DTD. The DTD may be included in the same file as the document instance, as is the case in Figure 1.1. The document instance and the DTD may also reside in separate files. In the latter case the document instance must contain a reference to the DTD.

A typical content model may be found in the example DTD in Figure 1.1:

```
<!ELEMENT Entry - - (HeadWordGroup, Etymology?, Sense+)>
```

Here we have an element `Entry` which contains one `HeadWordGroup` element, a possible `Etymology` element and one or more `Sense` elements. Tags may be minimized, i.e., omitted, if the content model says so. This content model states that both the start tag `<Entry>` and the end tag `</Entry>` are required because the content model contains the characters "`- -`". Either the start tag or the end tag may be omitted if we specify "`O -`" or "`- O`", respectively and both tags may be omitted if we have "`O O`". Minimization is mostly a technical detail to reduce the work of the author and the size of the document instance. In the content model above, the group connector comma `,` stands for consecutive components. There are two other group connectors `|` and `&` (not present in this content model). Only one of the components connected with `|` may occur. Components connected with `&` may occur in any order. The occurrence indicators `?`, `+` and `*` stand for an optional occurrence, one or more occurrences, or zero or more occurrences of the component, respectively.

A complete SGML document contains an SGML *prolog* and a document instance. The prolog may contain an SGML declaration describing basic facts about the dialect of SGML being used, such as the character set, and the length of identifiers. The prolog must contain a document type definition (DTD).

An SGML *parser* validates the document instance, i.e., checks if the instance conforms to its DTD. An SGML parser takes an SGML document as input and parses the instance according to the DTD. In the simple case it only outputs whether the instance is correct. Usually, however, it splits up the instance in an element information structure set (ESIS, [Gol90,

Appendix B, Annex G]) which is a list of all the components of the instance, e.g., tags, elements, attributes, etc., in the order they appear. The ESIS output can more easily be processed by other applications.

Example 2.3 Figure 2.4 shows an example of a part of the ESIS output of the SGML parser `nsgmls` [Cla96] when it parses the document in Figure 1.1. The `nsgmls` parser states that the document is valid by printing the letter `C` at the end of the ESIS output. In the ESIS output, the start and end of elements is denoted by parentheses and the name of the element tag, e.g., `(HEADWORD` and `)HEADWORD`, respectively. The #PCDATA is preceded by a dash, e.g., `-spaz`. □

```
(DICTIONARY
(ENTRY
(HEADWORDGROUP
(HEADWORD
-spaz
)HEADWORD
(PRONUNCIATION
-spæz
)PRONUNCIATION
(PARTOFSPEECH
-n
)PARTOFSPEECH
)HEADWORDGROUP
(ETYMOLOGY
-Abbreviation of spastic n.
)ETYMOLOGY
```

```
(SENSE
(DEFINITION
-= spastic
)DEFINITION
(QUOTATION
(DATE
)DATE
(AUTHOR
-P. Kael
)AUTHOR
(WORK
-I lost it at the
  movies III. 259
)WORK
.
.
C
```

Figure 2.4: Part of the ESIS output of an SGML parser.

An SGML *transformer* or *converter* reads the output of an SGML parser and makes specified modifications to the document instance. An *event-driven* SGML *transformer* reads the ESIS and processes the events as they are entered. A *tree-based* SGML *transformer* may read the ESIS for constructing an internal parse tree for further processing.

There is now a new standard for specifying the semantics of SGML documents, called the Document Style Semantics and Specification Language (DSSSL) [ISO96]. This standard lets the user make exact specifications of how certain structured SGML documents should be processed. The standard contains separate languages for specifying document transformations and formatting, and for structured searches in the documents. The first prototypes based on the standard are beginning to emerge [Cla97]. A transformer/formatter based on DSSSL may produce any output format. The only requirement is that it reads SGML documents and DSSSL specifications and that it interprets the DSSSL specifications correctly.

Unfortunately, SGML is a very complex standard. This is perhaps the reason why it has not been more widely accepted as a document standard. For example, the possible complexity of SGML DTDs makes it very hard to build efficient SGML parsers. Recently, an international expert group has designed a subset of SGML called XML that lacks the drawbacks of full SGML [BSM96]. There is hope that XML will become the *de facto* document markup standard.

Chapter 3

Transformation of structured documents

In the following we attribute the name *source* to the input side of our transformations. We start with a *source document* described by a context-free grammar, a *source grammar* (also called an input grammar). The output of the transformation is a *target document.* Sometimes we also describe the target document with another context-free grammar, a *target grammar* (or output grammar). When parsing the source document, the parser constructs a *source parse tree.* Sometimes we also build a *target parse tree* before writing out the target document. As we have seen in the previous chapter, we can use context-free grammars and parsers to define, check, and modify structured documents. Before we go into these *syntax-directed translations*, we shall take a brief look on other techniques that could be used in transforming structured documents.

The simplest transformation technique is *string matching* with *string replacing*, where we define a transformation consisting of string patterns, exact or approximate, and corresponding replacements. Exact matching with replacement is available in most editors. Approximate matching with replacement is not very common, but, e.g., in most UNIX operating system we may use regular expressions for string matching and simple string replacement. String matching with replacement accounts for a large group of text transformations, but when it comes to more complicated structural modifications, e.g., swapping the order of document sections, this technique is not sufficient.

By parsing we recognize not only the contents but also the structure of the document. *Syntax-directed translation techniques* are based on this fact. Usually, the output document is written at the time of parsing without constructing an additional parse tree representing the output. Most parser generators are used to produce such transformation modules. The user specifies the source document with a context-free grammar and the output at each recognized substructure of the document. When the source document is parsed, the target document is written at the same time. With the help of attribute grammars [Knu68] we can perform syntax-directed translation of this kind.

A more general transformation technique is based on *subtree matching and replacement*. This technique is similar to the string-based one, but before performing the transformation we need to parse the input to obtain a parse tree which can be modified. The transformation is based on input and output tree templates. When an input template is matched in the parse tree, the matched nodes are replaced with the structure described in the output pattern. Tree template matching can be performed rather efficiently, especially when the underlying context-free grammar is known [Kil92, KM95].

Some syntax-directed techniques require that we also define a target grammar. This guarantees that the constructed target parse tree and the target document are syntactically correct over the grammar. Examples of these techniques are *syntax-directed translation schemas* [Iro61] and *tree transformation grammars* [KPPM84].

3.1 Syntax-directed translation and attribute grammars

Syntax-directed translation is based on first recognizing the structure of the document before performing the transformation. In the general case, it does not mean that we have to build the source parse tree. The target document is output at parsing time which restricts the applicability of the transformation. Still, this technique is used quite extensively in commercial transformation languages. Often, though, the languages have been extended with attributes, whereby it is more appropriate to say that they are based on attribute grammars.

Example 3.1 Figure 3.1 shows an example of a syntax-directed translation defined with yacc. The specification contains three rules that surround headwords within the strings `{\bf` and `}`, respectively, and print a new line after each entry and headword group in the dictionary. □

All SGML transformation languages are based on syntax-directed translation. An SGML parser reads the SGML instance and returns, e.g., an output

```
Entry:              HWGroup Etymology Senses
                      { printf("\n") } ;

HWGroup:            Headword Pronunciation PartofSpeech
                      { printf("\n") } ;

Headword:           TEXT
                      { printf("{\bf %s} ", yytext) } ;
```

Figure 3.1: An example of a part of a `yacc` specification.

in ESIS form that is the base for further transformation. Both event-driven transformations and tree-based transformations use SGML parsers. In a tree-based transformation an internal structure is built based upon the output of the parser. In some cases also event-based SGML transformations may benefit from temporary constructs comparable to attribute grammars.

An *attribute grammar* (AG) [Knu68, LRS74] is a context-free grammar where each production has associated with it a set of semantic rules of the form $b := f(c_1, \ldots, c_n)$, where b and c_i are variables or *attributes* and f is an n-ary function. Translations are performed by parsing the source according to the grammar, then evaluating the attributes at each node in the parse tree giving a *decorated tree*. In a decorated tree, all attributes have values that are consistent with their definition. Attributes can be evaluated top-down or bottom-up, or through *several passes* [DJL88, Yel88]. Assuming that we do allow functions with side effects, we may include output actions or even tree construction operators and thereby obtain an *attributed translation*, typically used in program compiling. Attribute grammars are often used as an underlying strategy when implementing higher level transformation techniques such as tree transformation grammars [KPPM84]. AGs are somewhat tedious for using in *ad hoc* transformations, because it is again up to the user to control that the produced output follows the intended target grammar.

Example 3.2 Figure 3.2 shows an example of a syntax-directed translation with attributes. The specification is written in `yacc`. Every nonterminal in the `yacc` specification is allowed one attribute, denoted `$n`, where `n` = `$`, `1`, `2`, ... The symbol `$$` denotes the attribute of the left hand side nonterminal of the production, and `$n` the attribute of the nth symbol on the right hand side. This short specification moves the headword terminal

```
Entry:              HWGroup Etymology Senses
                      { printf("\n") ;
                        $$ = $1 } ;

HWGroup:            Headword Pronunciation PartofSpeech
                      { printf("\n") ;
                        $$ = $1 } ;

Headword:           TEXT
                      { printf("{\bf %s} ", yytext) ;
                        $$ = yytext } ;
```

Figure 3.2: An example of a yacc specification with attributes.

up to the *Entry* nonterminal, where it can be used, e.g., when traversing the rest of the dictionary entry. □

3.2 Syntax-directed translation schemas

Neither syntax-directed translation nor attribute grammars require a target grammar. This means that there is a great stress on the user to produce the correct output instructions in the specifications. A syntax-directed translation schema (SDTS) [Iro61, BF61, LS68], however, requires both a source grammar and a target grammar, even though the grammars must be very similar for the schema to work. Syntax-directed translation schemas (SDTS) have been used in several document transformation systems [KLMN90, KP91, Kui96].

Formally, a *syntax-directed translation schema* (SDTS) is a quintuple $T = (N, \Sigma, \Delta, R, S)$, where N is a finite set of nonterminal symbols, Σ is a finite input alphabet, Δ is a finite output alphabet, R is a finite set of rules of the form $A \to \alpha, \beta$, where $\alpha \in (N \cup \Sigma)^*$ and $\beta \in (N \cup \Delta)^*$ and the nonterminals in β are a permutation of the nonterminals in α, and S is a distinguished nonterminal called the start symbol [AU72]. In each rule $A \to \alpha, \beta$ we associate occurrences of the same nonterminals in α and β. If a nonterminal appears only once in α and β, respectively, the association is obvious. Otherwise the nonterminals may be indexed to denote the associations. Terminals occurring only in α or only in β are not associated.

For the translation we define a translation step with the help of a *trans-*

lation form. Firstly, the pair (S, S), where S is the start symbol, is a translation form. Secondly, if $(\alpha A\beta, \alpha' A\beta')$ is a translation form where the two nonterminals A are associated, and if $A \rightarrow \gamma, \gamma'$ is a rule in R, then also $(\alpha\gamma\beta, \alpha'\gamma'\beta')$ is a translation form. The nonterminals in γ and γ' are associated as they are in the rule, and nonterminals in α, α', β, and β' are associated as they were in the previous translation form.

The process of computing such a translation form from another form we call a *translation step.* A translation step is denoted by the derivation symbol $\Rightarrow_T$. We also use $\Rightarrow_T^*$ to denote the reflexive transitive closure, a translation that consists of zero or more translation steps, and $\Rightarrow_T^+$ to denote the transitive closure, a translation that consists of one or more translation steps. The *translation* defined by T, denoted $\tau(T)$, is the set of pairs [AU72]

$$\{(x, y) | (S, S) \Rightarrow_T^* (x, y), \text{ where } x \in \Sigma^* \text{ and } y \in \Delta^*\}.$$

Informally, we say that y is the translation of x under SDTS T or that x is the translation of y.

Aho and Ullman give Algorithm 3.1 for performing a syntax-directed translation according to an SDTS [AU72].

Algorithm 3.1 (Tree transformations via an SDTS)

Input. An SDTS $T = (N, \Sigma, \Delta, R, S)$, with input grammar $G_s = (N, \Sigma, P, S)$, output grammar $G_t = (N, \Delta, P', S)$, and a derivation tree D in G_s, with frontier in Σ^*.

Output. A derivation tree D' in G_t such that if x and y are the frontiers of D and D', respectively, then $(x, y) \in \tau(T)$.

Method.

1. Apply step 2, recursively, starting with the root of D.
2. Let this step be applied to node n. It will be the case that n is an interior node of D. Let n have the children $n_1, \ldots, n_k$.
   (a) Delete those of $n_1, \ldots, n_k$ that are leaves (i.e., have terminal or ϵ-labels, but that are not text terminals)
   (b) Let the production of G_s represented by n and its direct descendants be $A \rightarrow \alpha$. That is, A is the label of n and α is formed by concatenating the labels of $n_1, \ldots, n_k$. Choose some rule of the form $A \rightarrow \alpha, \beta$ in R. Permute the remaining direct descendants of n, if any, in accordance with the association between the nonterminals of α and β.

(c) Insert direct descendant leaves of n so that the labels of its direct descendants form β.

(d) Apply step 2 to the direct descendants of n that are not leaves, from left to right.

3. The resulting tree is D'.

□

Aho and Ullman also prove that if x and y are the frontiers of D and D', respectively, in Algorithm 3.1, then (x, y) is in $\tau(T)$ [AU72] .

Example 3.3 Figure 3.3 shows an example of a syntax-directed translation schema. The schema defines a translation that *removes* the senses of a dictionary entry, *reorders* the pronunciation and part of speech information of a headword and *inserts* some constant strings for a LaTeX document. An SDTS can also introduce new nonterminals, the contents of which will remain empty as there is no way of specifying their structure.

Note that we treat text terminals as nonterminals. Text terminals may be associated in the same way as nonterminals are. In this way, we assure that the contents of the source document can be copied to the target document. This is not strictly according to Algorithm 3.1 which always deletes source terminals.

Entry	→	*HWGroup Etymology Senses*,
		HWGroup Etymology
HWGroup	→	*Headword Pronunciation PartofSpeech*,
		`{\bf` *Headword* `}` *PartofSpeech Pronunciation*
Headword	→	`TEXT`, `TEXT`

Figure 3.3: An example of an SDTS.

□

Several restrictions and extensions have been defined for syntax-directed translation schemas. By requiring that all associated nonterminals for every rule $A \rightarrow \alpha, \beta$ in R occur in the same order in α and β, we obtain a *simple* SDTS [AU72]. With a simple SDTS, we cannot change the order of the document parts, we can only remove and insert terminals. Some extensions allow that α and β contain different nonterminals that are not associated. If α contains a nonterminal that is not present in β, the corresponding document part is removed. If β contains a nonterminal that is not present in α, this part is added to the document (possibly with empty contents) [KP93].

SDTSs have been extended with semantic rules [AU71, Bak78], predicates that select a target production [PB78] or even small programs attached to the rules [Shi84], but these extensions do not support the correctness of the output and we thereby lose the main advantage of using SDTSs.

3.3 TT-grammars

Using an SDTS achieves our main goal for a transformation technique for structured documents. The definition of a target grammar and its use in the SDTS guarantees that the transformation produces only correct target documents. On the other hand, an SDTS is quite restricted. Firstly, the source and target grammars must be very similar. They must contain nonterminals with the same names, and there must a corresponding target production for each source production. In the case where we start with two document representations that have been defined with two arbitrary grammars, we must first redefine one of the grammars to be able to define an SDTS.

Secondly, an SDTS cannot add or remove levels of structure in the parse trees. The transformation always works on one level in the source parse tree, removing or adding terminals, and reordering the nonterminals. Sometimes we need to remove or add levels of nodes when we want to introduce more internal structure in our document or remove some structure. This is especially useful when the target document itself becomes a source document of another transformation.

To solve these problems, we extend the SDTSs and introduce *tree transformation grammars* or TT-grammars [KPPM84]. A TT-grammar is like an SDTS without the implicit associations between nonterminals in the rules. On the contrary, the user must explicitly define these associations. This means also that he/she can associate nonterminals with different names. Additionally, TT-grammars work with as many node levels in the parse trees as wanted.

A TT-grammar describes a relationship between a syntax tree over a grammar G_1 and a syntax tree over a grammar G_2. Transformations can be specified both ways, from trees over grammar G_1 to trees over grammar G_2 or vice versa, thus being especially suitable for purposes where two-way transformations are common. Here we concentrate on one-way transformations from trees over a *source grammar* G_s to trees over a *target grammar*

G_t. The relationship is described by associating groups of productions in G_s with groups of productions in G_t. In addition one needs to associate symbol occurrences in G_s with symbol occurrences in G_t.

Formally, a TT-grammar is a sextuplet (G_s, G_t, S_s, S_t, PA, SA), where G_s and G_t are the source and target grammars, respectively, S_s and S_t are sets of *source* and *target subgrammars*, respectively, PA is a set of *production group associations*, and SA a set of *symbol associations*. The source and target grammars are context-free grammars. The source and target subgrammars consist of subsets of the source and target grammars, respectively. A production group association is a pair consisting of a source subgrammar in S_s and a target subgrammar in S_t. A symbol association relates a symbol in a source subgrammar to a symbol in a target subgrammar (within a certain production group association, or in separate production group associations).

A source subgrammar must satisfy the following restrictions. First, there must be a single start symbol. Second, every other symbol in the subgrammar must be derivable from this start symbol. Source subgrammars specify subtree patterns to be matched against in the source tree; the target subgrammars specify the subtrees that are to be constructed as part of the target parse tree. A target subgrammar is *not* required to have a single start symbol; it can have several, resulting in a forest of target subtrees.

A TT-grammar may be viewed as generating subtrees in G_t from subtrees in G_s as follows. Let (pg_s, pg_t) and (pg'_s, pg'_t) be production group associations, where pg_s and pg'_s are source subgrammars and pg_t and pg'_t target subgrammars. The productions in pg_t (pg'_t) are used to construct target subtrees every time the productions in pg_s (pg'_s) have been applied (all of them) in the source tree. In Figure 3.4a, the application of the source subgrammars has been denoted by two dashed triangles in the source tree. Let two of the produced target subtrees correspond to the productions $A \to \alpha B \beta$ in pg_t and $B \to \gamma$ in pg'_t (Figure 3.4b). Assume also that both target nodes labeled B are associated with the same source node p (denoted by dotted lines between the source node p and the two nodes labeled B in Figure 3.4). Then the two target subtrees are linked through B to form a single target subtree (Figure 3.4c). Note that this technique is more general than using input and output tree templates. In a subgrammar we may use recursive productions and thereby describe more complicated tree patterns than are possible with tree templates.

The algorithm for applying a TT-grammar transformation is given as Algorithm 3.2.

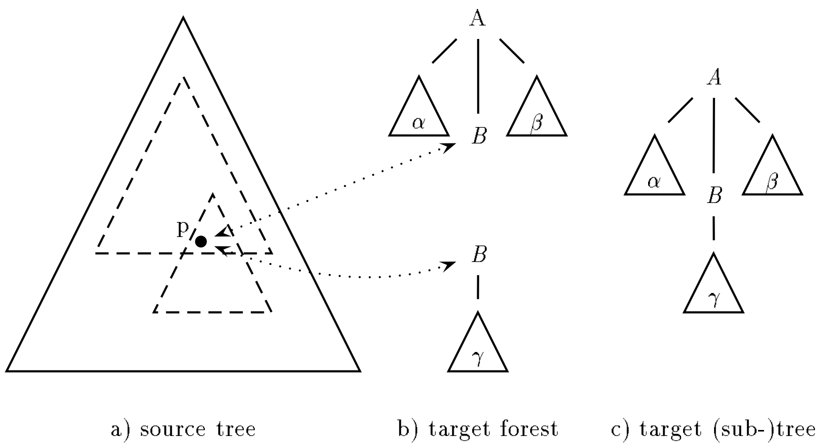


Figure 3.4: Three phases in TT-grammar application..

Algorithm 3.2 (Tree transformations via a TT-grammar)

Input. A TT-grammar $TT = (G_s, G_t, S_s, S_t, PA, SA)$, with source grammar $G_s = (N, \Sigma, P, S)$, target grammar $G_t = (N, \Delta, P', S')$, and a derivation tree D in G_s, with frontier in Σ^*.

Output. A derivation tree D' in G_t such that if x and y are the frontiers of D and D', respectively, and y contains terminals only, then y is the TT-grammar translation of x. (But depending on how the transformation is specified, we may also have a forest of trees, all over G_t.)

Method.

1. Apply step 2 to all nodes in tree D, starting with *any* nonterminal node of D. When all nodes have been matched against source subgrammars goto step 3.

2. Let this step be applied to node n with label A. It will be the case that n is an interior node of D.

   (a) Choose a production group association $pga = (pg_s, pg_t)$ where the source subgrammar pg_s has start symbol A and where the tree structure denoted by the subgrammar matches the subtree at node n. If there are no such production group association, return to step 1.

(b) For every production $B_j \rightarrow X_{j_1} \cdots X_{j_k}$ in pg_t construct a separate target subtree with B_j as its root and $X_{j_1}, \ldots, X_{j_k}$ as its children.

(c) Let the symbol association set of the production group association pga be sa. For every symbol association (s_s, s_t) in sa, associate the symbols s_s and s_t, i.e., make an association between the instance of the symbol s_s in the derivation tree D and the instance of the symbol s_t in some target subtree over the production $B_j \rightarrow X_{j_1} \ldots s_t \ldots X_{j_k}$

3. Apply step 4 to all root nodes of separate target subtrees created in step 2. When no more subtrees can be linked go to step 5.

4. Let this step be applied to the root m of the target subtree st_m. Let m have the label B and a symbol association to source node p.

   (a) Find a leaf node n in any of the other target subtrees with label B and an association to the same source node p. Let this subtree be st_n. Merge the subtrees st_m and st_n at the node n, i.e., replace the leaf node n in subtree st_n with the subtree st_m

5. If the result is a connected tree and all the leaves are terminals, it is D'.

□

We observe that Step 2 in the algorithm only constructs subtrees over a subgrammar of G_t. Thereby, if the result is one connected tree, the tree must be over the target grammar G_t.

Example 3.4 We shall define a TT-grammar for our dictionary document. Our example transformation makes several transformations to the dictionary that are not possible to define with an SDTS. We shall print out only headwords, their etymology, and their examples: additionally, parts have been reordered and the text enhanced with constant strings. A possible output of this word list view is the LaTeX output in Figure 3.5.

To achieve this view we must transform the dictionary into the LaTeX declarations of Figure 3.6

We see that we have discarded information about part of speech and sense definitions, and reordered some other information. We also insert

spaz (Abbreviation of spastic n.)

- I know how long, you little spaz. (M. Amis, Dead babies viii. 47, 1975)
- The term that American teen-agers now use as the opposite of 'tough' is 'spaz'. (P. Kael, I lost it at the movies III. 259, 1965)

Figure 3.5: A LaTeX view of the target document.

```
{\bf spaz} (Abbreviation of spastic n.)
\begin{itemize}
\item I know how long, you little spaz.
     (M. Amis, Dead babies viii. 47, 1975)
\item The term that American teen-agers now use as
     the opposite of 'tough' is 'spaz'.
     (P. Kael, I lost it at the movies III. 259, 1965)
\end{itemize}
```

Figure 3.6: An example target document.

new constant strings in the text. To do this, we modify the source parse tree extensively. We specify this transformation by first describing the source grammar of the dictionary and the target grammar of the word list, and then by giving the rules of the mapping, rules that are based on both grammars. The source grammar has already been given in Figure 2.1. The target grammar is shown in Figure 3.7. Note especially that we have used both source grammar nonterminals and new nonterminals. The mapping does not restrict the use of nonterminals in the grammars as is the case in an SDTS.

When parsing our source document, the dictionary, we achieve the parse tree depicted in Figure 2.3 on page 19. The complete set of mapping rules for this transformation is given in Figure 3.8 on page 40.

The first rule applies to the root of the source tree and states that whenever we find a subtree as specified in the source subgrammar of the rule in the source tree, we construct the subtrees defined in the target

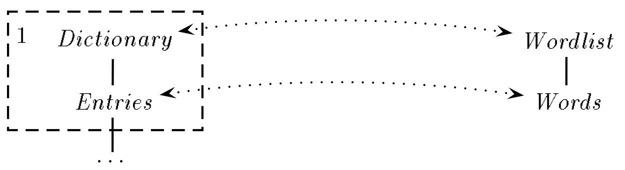
Wordlist	→	*Words*	*Examples*	→	*Example*
Words	→	*Words Word*	*Example*	→	*Quotations*
Words	→	*Word*	*Quotations*	→	*Quotation*
Word	→	*Headword*			*Quotations*
		Etymology	*Quotations*	→	*Quotation*
		`\begin{itemize}`	*Quotation*	→	`\item` *Text* (
		Examples			*Author Work*
		`\end{itemize}`			*Date*)
Headword	→	`{\bf TEXT }`	*Date*	→	TEXT
Etymology	→	(TEXT)	*Author*	→	TEXT
Examples	→	*Examples*	*Work*	→	TEXT
		Example	*Text*	→	TEXT

Figure 3.7: An example target grammar.

subgrammar.

1 *Document* → *Entries* *Document.Wordlist* → *Entries.Words*

Symbol associations are denoted in the target subgrammars by *source_symbol.target_symbol*. In this case the rule matches a subtree at the root of the source tree and constructs a target subtree as follows.

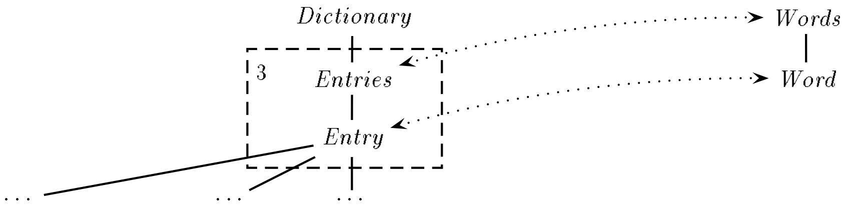


The matched subtree is denoted by the dashed box, and the rule number is stated in the top left corner of the box. The transformation constructs one target subtree. Symbol associations are denoted in the figure by dotted curves. Symbol associations between nonterminals, or nonterminals and terminals are used for connecting separate target subtrees.

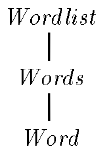
The following two rules map an iteration in the source on a similar iteration in the target.

2	*Entries*	→	*Entries* *Entry*	*Entries(1).Words*	→	*Entries(2).Words* *Entry.Word*
3	*Entries*	→	*Entry*	*Entries.Words*	→	*Entry.Word*

Here several occurrences of the same nonterminal are distinguished by indexing the occurrences. Rule 2 is never used in our example because such a source subtree cannot be found in our source tree. However, rule 3 is used, matching at the source subtree starting at node *Entries*.



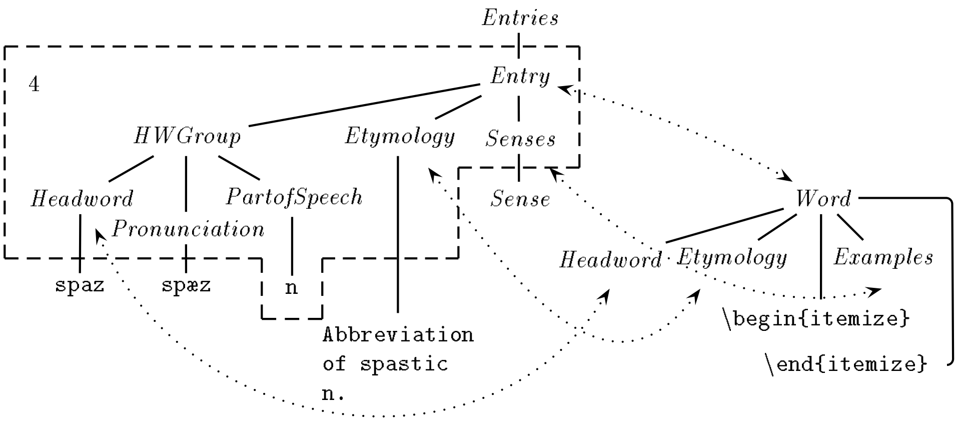
As a result, we now have two separate target subtrees. In the subtrees there are two target nodes labeled *Words* associated with the same instance of a source node *Entries*, and the former and the new target subtrees are linked into one subtree.



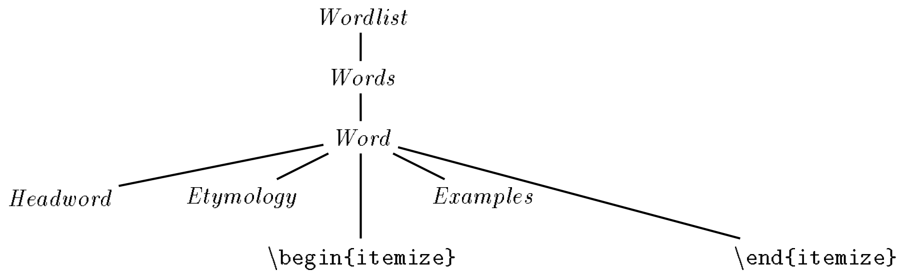
Rule number 4 contains a source subgrammar with several productions.

4	*Entry*	→	*HWGroup* *Etymology* *Senses*	*Entry.Word*	→	*Headword.Headword* *Etymology.Etymology* `\begin{itemize\` *Senses.Examples* `\end{itemize}`
	HWGroup	→	*Headword* *Pronunciation* *PartofSpeech*			
	PartofSpeech	→	`n.`			

The transformation constructs a tree pattern from these productions with the first nonterminal (*Entry*) as a root. Applying such a rule in an SDTS is not possible as an SDTS only associates productions not subgrammars. Using the rule constructs a target tree where some of the information, such as the pronunciation and part of speech categorization, has been discarded. Also, our transformation includes only entries where the `PartofSpeech` element has been specified as a noun (`n.`). If our small dictionary contained also verbs and adjectives, they would not be included in the target document.



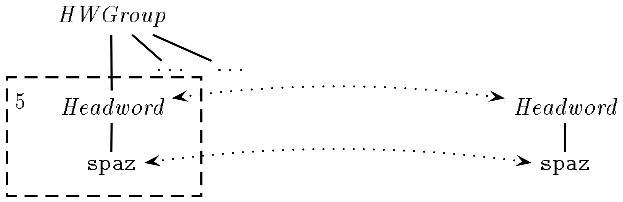
Again, we have two nodes in the target subtrees labeled *Word* that are associated with the same instance of a source node labeled *Entry*. The two target subtrees are connected to form a new tree as follows.



As an example of text terminal mapping, we have rule 5

5 *Headword* → `TEXT` *Headword.Headword* → `{\bf TEXT.TEXT }`

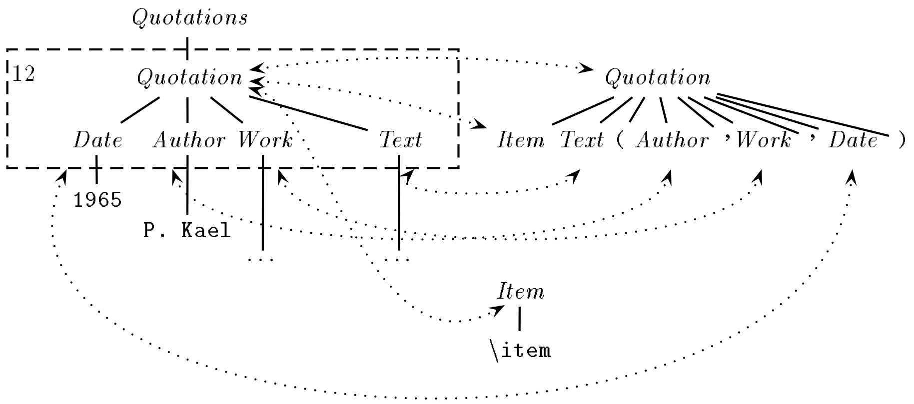
which matches the source tree at node *Headword*. The transformation produces a similar target subtree where the identifier has been copied. Here, we have the second reason for symbol associations demonstrated. Symbol associations between terminals are used for copying the contents of the terminal to the target side.



As a final example of rule application, we take a look at rule 12.

12	*Quotation*	→	*Date*	*Quotation.Quotation*	→	*Quotation.Item*
			Author			*Text.Text*
			Work			(*Author.Author* ,
			Text			*Work.Work* ,
						Date.Date)
				Quotation.Item	→	`\item`

This rule demonstrates the reordering power of the mapping. Logical parts of the document are reordered at the same time as terminals and nonterminals are inserted in the target tree. We also see how one rule produces several target subtrees to be linked later. This is not possible with an SDTS.



We have started the simulation of the transformation by traversing the source tree in preorder. Note, however, that there is no restriction on the order in which the rules are applied, as long as all rules that match in the tree are used. The matching phase is performed first after which all the produced target subtrees can be linked to form the target subtree.

The rest of the transformation is performed in a similar way. The complete set of mapping rules is given in Figure 3.8. The figure reads

pga	*source subgrammars*			*target subgrammars*		
1	*Document*	→	*Entries*	*Document.Wordlist*	→	*Entries.Words*
2	*Entries*	→	*Entries Entry*	*Entries(1).Words*	→	*Entries(2).Words* *Entry.Word*
3	*Entries*	→	*Entry*	*Entries.Words*	→	*Entry.Word*

4	*Entry*	→	*HWGroup*
			Etymology
			Senses
				Entry.Word	→	*Headword.Headword*
						Etymology.Etymology
						`\begin{itemize}`
						Senses.Examples
						`\end{itemize}`

	HWGroup	→	*Headword* *Pronunciation* *PartofSpeech*			
	PartofSpeech	→	`n.`			
5	*Headword*	→	`TEXT`	*Headword.Headword*	→	`{\bf TEXT.TEXT }`
6	*Etymology*	→	`TEXT`	*Etymology.Etymology*	→	`TEXT.TEXT`
7	*Senses*	→	*Senses Sense*	*Senses.Examples*	→	*Senses.Examples* *Sense.Example*
8	*Senses*	→	*Sense*	*Senses.Examples*	→	*Sense.Example*
9	*Sense*	→	*Definition* *Quotations*	*Sense.Example*	→	*Quotations.Quotations*
10	*Quotations*	→	*Quotations* *Quotation*	*Quotations(1).Quotations*	→	*Quotation.Quotation* *Quotations(2).Quotations*
11	*Quotations*	→	*Quotation*	*Quotations.Quotations*	→	*Quotation.Quotation*
12	*Quotation*	→	*Date* *Author* *Work* *Text*	*Quotation.Quotation*	→	*Quotation.Item* *Text.Text* (*Author.Author* , *Work.Work* , *Date.Date*)
				Quotation.Item	→	`\item`
13	*Date*	→	`TEXT`	*Date.Date*	→	`TEXT.TEXT`
14	*Author*	→	`TEXT`	*Author.Author*	→	`TEXT.TEXT`
15	*Work*	→	`TEXT`	*Work.Work*	→	`TEXT.TEXT`
16	*Text*	→	`TEXT`	*Text.Text*	→	`TEXT.TEXT`

Figure 3.8: A mapping between the grammars of Figures 2.1 and 3.7.

as follows. Product group associations are numbered from 1 to 16. The source subgrammar of a pga is given to the left and the corresponding target subgrammar to the right. Symbol associations are shown in the target subgrammars by connecting associated symbols with a period. All together, the transformation produces 17 target subtrees that are linked together through the symbol association to form the complete target tree in Figure 3.9.

□

Above we have seen examples of how a parts of structured document can be renamed, reordered, removed, or inserted, how levels of the document tree can be suppressed or inserted, and how terminals are copied to the new document. Unfortunately, there are transformations that require more complex actions. This is also known in attribute grammars, where semantic rules go beyond the syntactical transformations. We have extended TT-grammars with semantic actions also in the ALCHEMIST system. The next chapter considers this system.

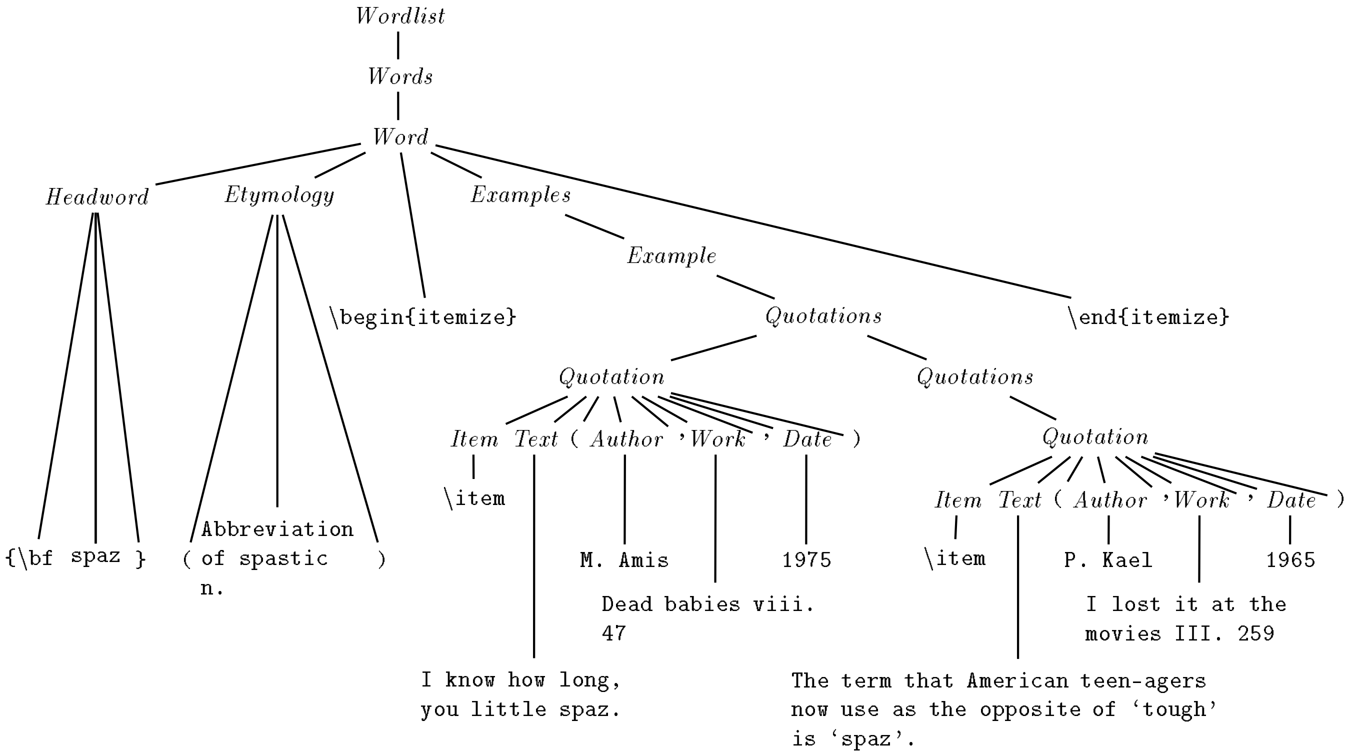


Figure 3.9: A target parse tree.

Chapter 4

The transformation generator ALCHEMIST

As we have sen in the previous chapter, the notion of TT-grammars can well be used in the implementation of a transformation generator for structured documents. One such example is the SSAGS transformation generator [KPPM84], but it only implements a subset of the TT-grammar technique. In this chapter we present a transformation generator called ALCHEMIST [TL94a, LTV96] which is based on the TT-grammar technique. ALCHEMIST also provides a graphical interface for specifying transformations. Additionally, ALCHEMIST automatically generates the transformation code and calls a compiler that links and compiles the code into an executable transformation.

ALCHEMIST is similar to many other transformation generators such as ICA [MBO93]. The motivation behind ALCHEMIST lies in the fact that we wanted to provide a general transformation generator where the user does not have to bother about the similarity of the source and target grammars. Also, we wanted a generator that is equally suitable for producing transformations between structured documents as well as between other structured representations such as computer programs and persistent representations of data in computer applications. In this chapter we describe the structure and use of ALCHEMIST and also present its implementation. We describe the experience in its use and evaluation of the system in Chapter 6. First, though, we take a look at the small differences in the ALCHEMIST TT-grammars from those defined in Section 3.3.

4.1 ALCHEMIST TT-grammars

ALCHEMIST implements a subset of TT-grammars where multiple source productions are allowed in a source subgrammar. This allows the user to define more complex source subtree templates to be matched against in the source parse tree. Instead of only removing and adding terminals and reordering nonterminals in the source parse tree, the user can also add or remove levels of internal nodes in the tree.

The general idea of a source subgrammar *as a grammar* has not though been implemented in ALCHEMIST. This idea would allow the use and interpretation of recursive productions in the source subgrammar to describe arbitrarily big recursive subtrees to be matched against. This would definitely add to the transformational power of ALCHEMIST, but it seems that this additional power is not worth the trouble. In the current version of ALCHEMIST the source subgrammar corresponds to a connected tree substructure to be matched against in the source parse tree. Also the indented but semantically unclear symbol associations between symbols in *different* production group associations have not been implemented in ALCHEMIST. These associations might be used perhaps as a short cut in linking the target subtrees together at the end of the mapping but it is unclear to the author how they are used in [KPPM84]. These minor restrictions to the TT-grammar technique in the implementation of ALCHEMIST do not much decrease the power of the transformation generator.

In addition, several extensions have been made to the TT-grammars in ALCHEMIST. Firstly, identifier copying from the source to the target has been added. The user associates identifiers on both sides of the transformation (source and target) and assures that identifiers are correctly “brought over” to the target side. Secondly, semantic actions have been added for processing on the target side. The user can add semantic actions to check for identifiers, perform computing, etc., in the transformation specification. This makes the transformation as powerful as any programming language. As a matter of fact, the language used in semantic action *is* a programming language (C++).

4.2 ALCHEMIST structure

ALCHEMIST divides *transformation construction* into three phases: *specification*, *generation*, and *compilation*. Specification includes defining both the source and target grammars as well as a mapping between the grammars based on the TT-grammar technique. Generation produces programming code from the specifications and compilation the code of the executable transformation module. ALCHEMIST contains interacting software modules for all these phases (Figure 4.1). These modules have all been named with concepts from the alchemy domain.

SPELLBOUND is the specification interface and contains only one tool, the MAPPERTOOL, for specifying mappings. Grammars are given in text files. The output of the specification phase consists of a source and a target grammar, and a mapping between these two grammars.

SPELLTOOL comprises tools for generating the transformation code and compiling it into an executable transformation module. It contains the following modules:

SEER generates a parser from the source grammar. The source parser reads source documents and builds the corresponding source parse tree.

STONE generates a mapper from the mapping specification. The source-to-target mapper transforms the source parse tree into a target parse tree.

SWINDLER generates a target parser or unparser from the target grammar. The target parser traverses the target parse tree and writes its frontier, the target document, to a file.

COMPILER compiles and links the generated code into an executable transformation.

SUBSTANCE is the interface for defining the internal parse tree structures. Depending on the transformation applications, the user needs to use parse trees of different complexity.

GLASS is the interface for defining user interface for user interaction when the user needs to interact in some semi-automatic transformations.

GLASS and SUBSTANCE GENERATORS. There are two more modules generating transformation code: one for generating code for the internal parse trees and one for generating the user interaction interface.

POT handles the storage of reusable components, such as grammars and mapping specifications.

In Figure 4.1 we see a schema of the ALCHEMIST process and its intermediate and end results. All intermediate results, such as grammars, code, etc, are stored persistently in files. Figure 4.1 shows implemented

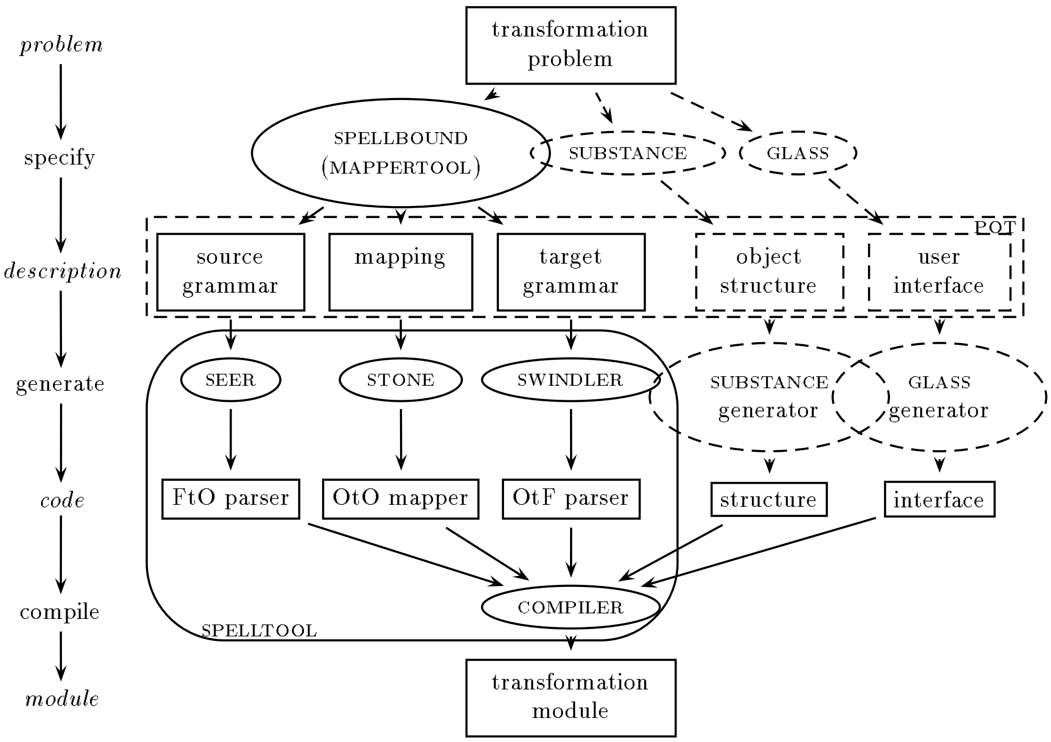


Figure 4.1: The ALCHEMIST environment.

components with unbroken lines and unfinished components with dashed lines.

We want to make a clear distinction between the transformation construction level (specification, generation, and compilation) and the execution level (of using the transformation). Therefore we name all the meta level components of the construction level with capital letters: ALCHEMIST, SPELLBOUND, etc. At the execution level, ALCHEMIST transformations are called *spells*, and all components linked to the execution level (source documents, spells, internal implementation components) are written with small letters.

The spell process, i.e., the data flow of an ALCHEMIST transformation, is shown in Figure 4.2. A spell consists of three modules, *a parser*, *a mapper*, and *an unparser*. The parser reads the source document and builds the corresponding parse tree. The mapper transforms the parse tree into a target parse tree, and the unparser writes the frontier of the target parse tree into the target document. Again, the intermediate representations of the document, the parse tree, may be saved persistently through *spellpot*. In this way we are less dependent on memory size if the documents are large.

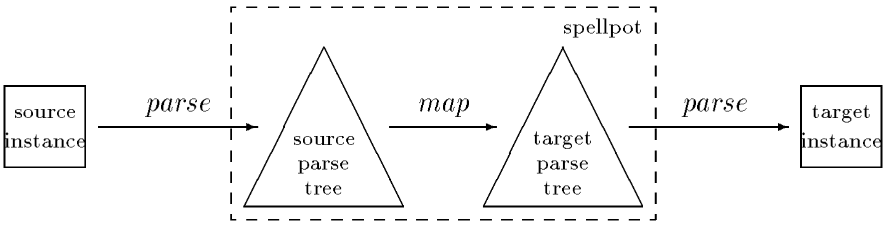


Figure 4.2: Data flow of a spell.

Between the construction level and the execution level lies APPRENTICE, an interface for executing a set of spells. APPRENTICE provides a convenient interface for selecting and starting the appropriate spell. The ALCHEMIST environment is shown in Figure 4.3.

4.3 ALCHEMIST use

As mentioned earlier, ALCHEMIST divides the transformation or spell construction into several phases. In this section we take a closer look at these phases together with examples and see how ALCHEMIST implements the TT-grammar technique (see also [LT95, LTV95a]). The phases on the construction level supported by ALCHEMIST are

1. the *specification phase* where the user specifies the transformation with SPELLBOUND giving as results a source and a target grammar, and a mapping between the grammars,

2. the *generation phase* where the user generates transformation code with SPELLTOOL, and

3. the *compilation phase* where the user with the help of SPELLTOOL calls the appropriate compiler for producing an executable spell.

Additionally, on the execution level, supported by ALCHEMIST we have

4. The *execution phase* where the user starts a spell with APPRENTICE.

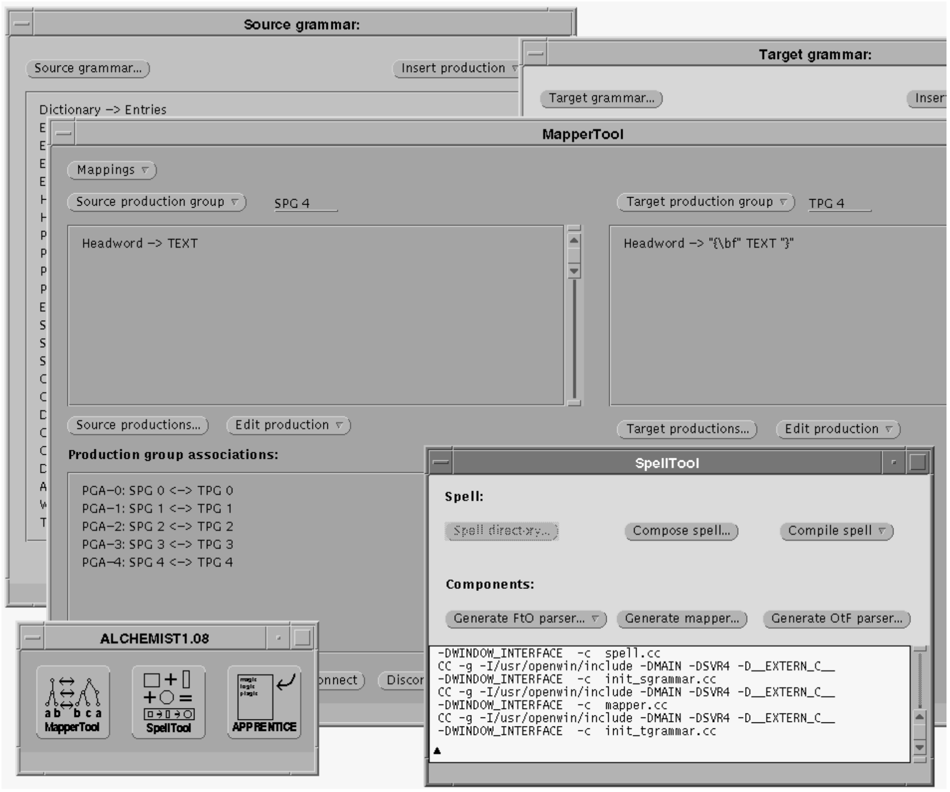


Figure 4.3: The ALCHEMIST environment.

4.3.1 Spell specification

The first construction phase includes grammar and mapping specification. The user specifies both a source and a target grammar for the spell. A mapping between the grammars is constructed based on the TT-grammar technique.

Grammar specification

The source and target grammars are context-free grammars. The grammars follow a very simple syntax. Nonterminals are any identifiers beginning with a letter and followed by letters and/or numbers. A terminal is either surrounded by double quotes or consists of a special terminal. A quotation mark in a terminal is in itself surrounded by double quotes. A special terminal is a source terminal or a target terminal. Source terminals are

`IDENTIFIER`	An identifier, a letter followed by letters or numbers.
`TEXT`	A string of text, application dependent.
`NUMBER`	A number, a digit followed by digits (integers only).
`STRING`	A string, any sequence of characters surrounded by double quotes.

Target terminals are used for producing certain strings in the target document, e.g., dates or binary numbers. The target terminals are

`IDENTIFIER`	An identifier, a letter followed by letters or numbers.
`NUMBER`	A number, a digit followed by digits (integers only).
`BINARY_NUMBER`	A binary number, for producing a binary number.
`IDENTIFIER[n]`	An identifer of length `n`.
`NUMBER[n]`	A number of length `n`.
`BINARY_NUMBER[n]`	A binary number of length `n`.
`STRING`	A string surrounded by double quotes.
`CHAR[n]`	The ASCII character with number `n`.
`NULL[m]`	Produces `m` NULL characters.
`CURRENT_DATE`	The current date in the format YYMMDD.
`CURRENT_TIME`	The current time in the format HHMMSS.
`INCLUDE_FILE`	Inserts the contents of the file which is given as the next (terminal) symbol after `INCLUDE_FILE`.

The production rewrite symbol is denoted by `->`. Iterations are specified with recursive productions. Disjunctions are specified by giving several productions with the same nonterminal on the left hand side. The grammar syntax is very simple and specification can be made with an ordinary text editor. Figure 4.4 shows an example of an ALCHEMIST grammar corresponding to the DTD in Example 1.1 on page 2.

When actually used in spell specification, the source and target grammars are loaded into the grammar windows of the MAPPERTOOL (Figure 4.5).

Mapping specification

The mapping connects the source and target grammars together based on the TT-grammar technique. As expected this is the most complicated part of a spell specification. The mapping is specified through MAPPERTOOL (Figure 4.6).

Production group associations are specified in the main window of MAPPERTOOL (Figure 4.6). The user opens up the source and target grammars in separate windows and selects the appropriate production into

```
Dictionary      ->      "<Dictionary>"  Entries "</Dictionary>";
Entries         ->      Entries Entry ;
Entries         ->      Entry ;
Entry           ->      "<Entry>" HWGroup Etymology Senses
                        "</Entry>" ;
HWGroup         ->      "<HWGroup>" Headword Pronunciation
                        PartofSpeech "</HWGroup>" ;
Headword        ->      "<Headword>" TEXT "</Headword>" ;
Pronunciation   ->      "<Pronunciation>" TEXT "</Pronunciation>" ;
PartofSpeech    ->      "<PartofSpeech>" "n." "</PartofSpeech>" ;
PartofSpeech    ->      "<PartofSpeech>" "v." "</PartofSpeech>" ;
PartofSpeech    ->      "<PartofSpeech>" "a." "</PartofSpeech>" ;
Etymology       ->      "<Etymology>" TEXT "</Etymology>" ;
Senses          ->      Senses Sense ;
Senses          ->      Sense ;
Sense           ->      "<Sense>" Definition Quotations "</Sense>";
Quotations      ->      Quotations Quotation ;
Quotations      ->      Quotation ;
Definition      ->      "<Definition>" TEXT "</Definition>" ;
Quotation       ->      "<Quotation>" Date Author Work Text
                        "</Quotation>" ;
Date            ->      "<Date>" TEXT "</Date>" ;
Author          ->      "<Author>" TEXT "</Author>" ;
Work            ->      "<Work>" TEXT "</Work>" ;
Text            ->      "<Text>" TEXT "</Text>" ;
```

Figure 4.4: An example of an ALCHEMIST grammar.

source and target subgrammars. The source subgrammar must contain one single start symbol from which all other symbols are derivable. By default, the left hand side of the first production in this group is considered as the start symbol of the source subgrammar. When the user has specified the subgrammars, he/she connects them forming a production group association. A subgrammar may be used in several group associations.

Symbol associations are specified in the symbol association window (Figure 4.7). The window shows all the nonterminal symbols and special terminals corresponding to the current production group association in the MAPPERTOOL main window. The user makes symbol associations by selecting source and target symbols. Several symbol may be chosen at a time.

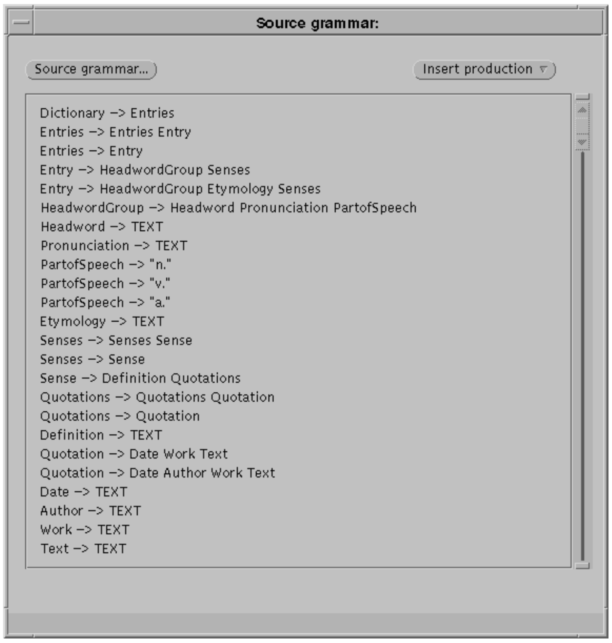


Figure 4.5: An ALCHEMIST grammar in the source grammar window of MAPPERTOOL.

Semantic actions are specified by selecting a target symbol and specifying the action in the semantic actions window (Figure 4.8). An action can be performed before or after a target symbol is processed by the mapper of the spell. An action before processing the symbol may make modifications to the symbol, like capitalizing, etc. An action after processing may insert the symbol in a symbol table.

4.3.2 Spell generation

The second phase of spell construction includes generating the spell code from the spell specification. In principle, code generation is independent of the programming language but in this special case ALCHEMIST generates C++ code as the semantic actions are written in this language. Code generation is performed with the help of SPELLTOOL (Figure 4.9).

For each subspecification, the user needs to generate a spell module. SEER and generates a parser from the source grammar, STONE generates a mapper from the mapping specification, and SWINDLER generates an unparser from the target grammar. Following the ALCHEMIST design prin-

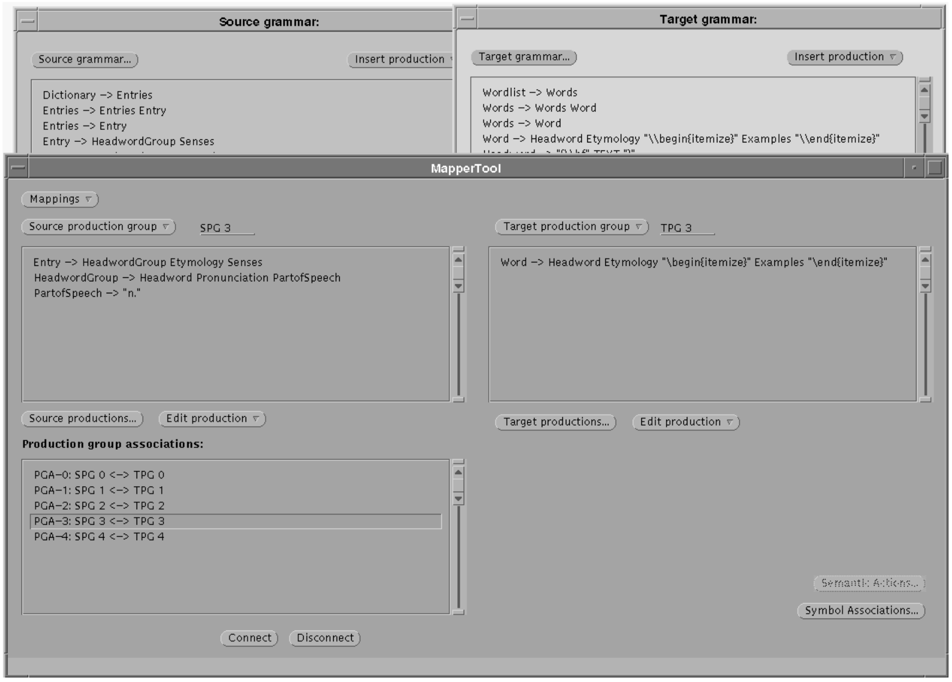


Figure 4.6: MAPPERTOOL for specifying production group associations.

ciples of storing any intermediate result persistently, the code may be inspected and even modified (but on the user's own risk). In this way, the user may tailor the transformation to very specific needs not describable with SPELLBOUND.

4.3.3 Spell compilation

The third phase of spell construction includes compiling and linking the spell code into an executable module. This phase includes default components like a spell user interface and connections to object management. Compiling is also performed with the help of SPELLTOOL (Figure 4.10).

The user has several options. He/She can choose to compile a spell with a graphical user interface or a textual interface. He/She can also limit the compilation to the source parser which is very convenient for testing purposes. In this way the user can convince himself that the grammar is correct before continuing with the rest of the spell specification. By using the target grammar as a source grammar, the user can also produce a parser

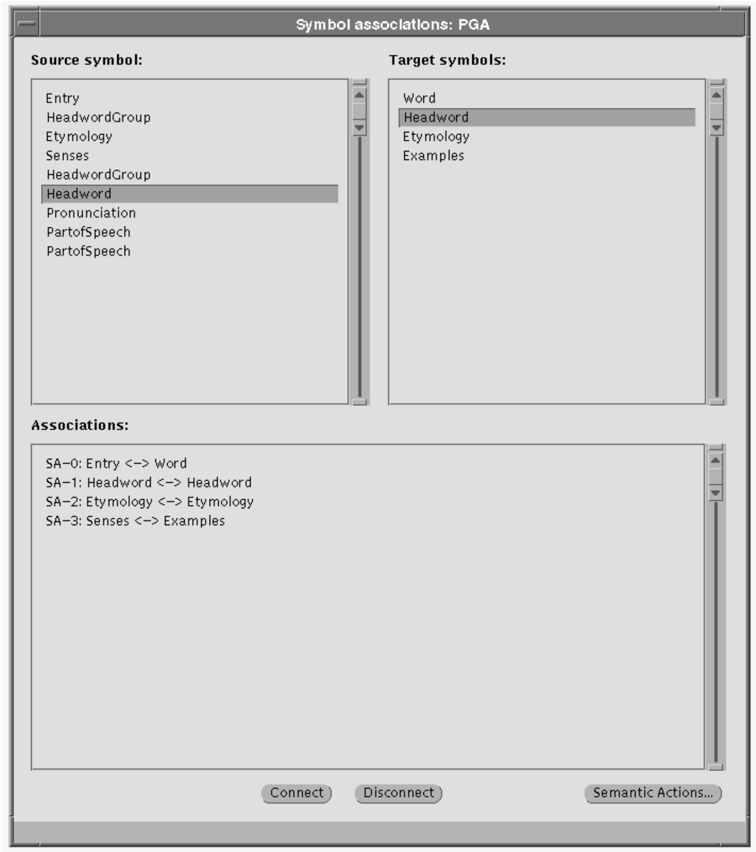


Figure 4.7: The MAPPERTOOL window for specifying symbol associations.

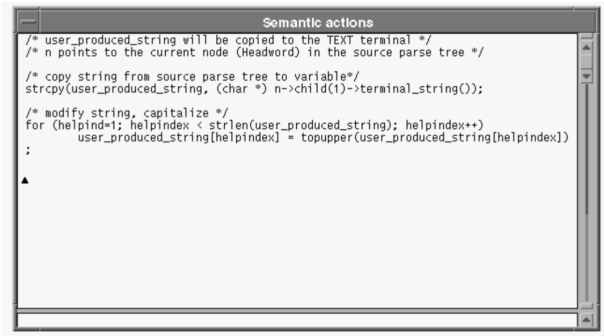


Figure 4.8: The MAPPERTOOL window for specifying semantic actions.

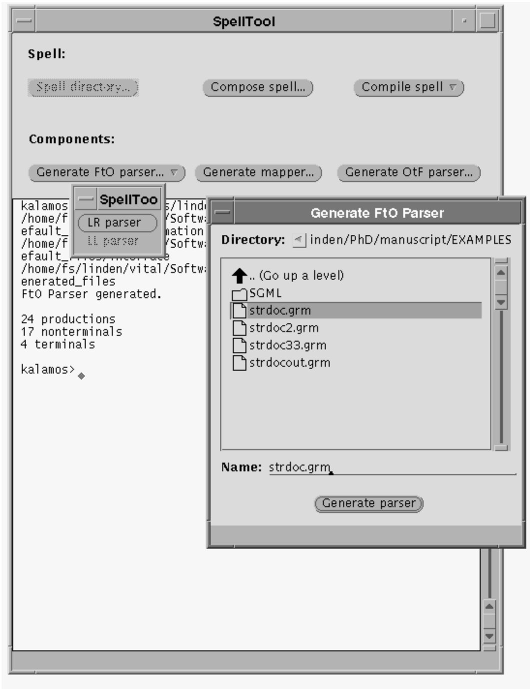


Figure 4.9: SPELLTOOL for generating spell code.

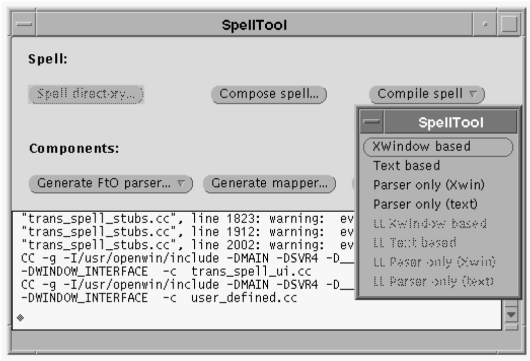


Figure 4.10: SPELLTOOL for compiling a spell.

for this grammar.

4.3.4 Spell execution

When all of the phases in spell construction have been performed, the spell is ready to be executed. If the user has constructed a spell with a graphical interface (Figure 4.11), he/she can either start the spell through APPRENTICE (Figure 4.12) or in a command window. The user writes the file names of the source and target documents in the graphical spell interface. He/She may open the source document to check that the correct document has been chosen. Also the target document may be opened after the transformation. The spell interface shows the percentage of the completed transformation process. For debugging and tracing reasons, the user can select the amount of debugging messages. A higher trace level gives the user a better chance to follow the distinct phases of the transformation.

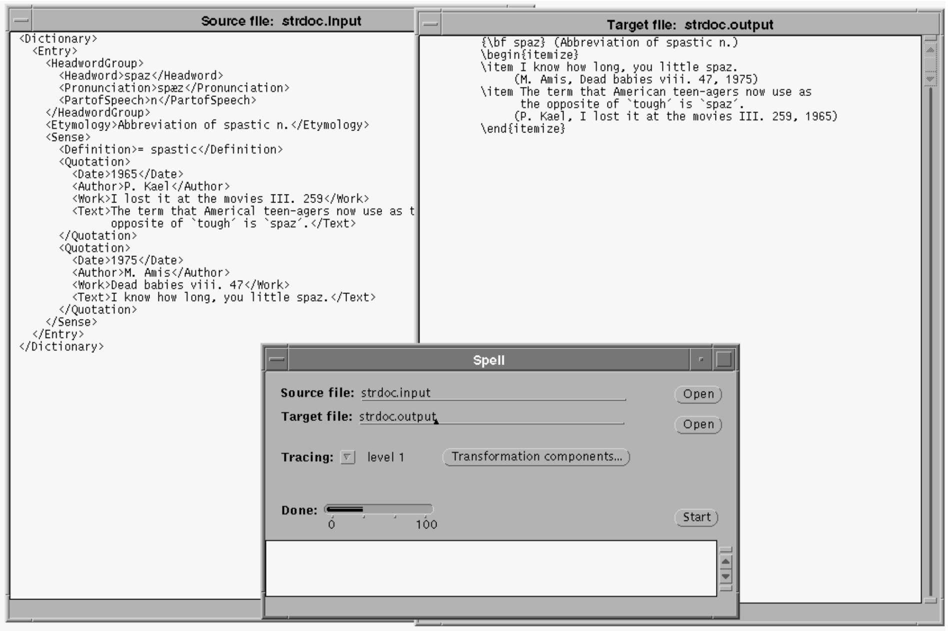


Figure 4.11: An example of a spell interface.

In addition to the above mentioned spell components, the user may also include pre- and postprocessing commands in a spell. A preprocessing command is performed on the source document before it is parsed, while a postprocessing command is performed on the target document before it

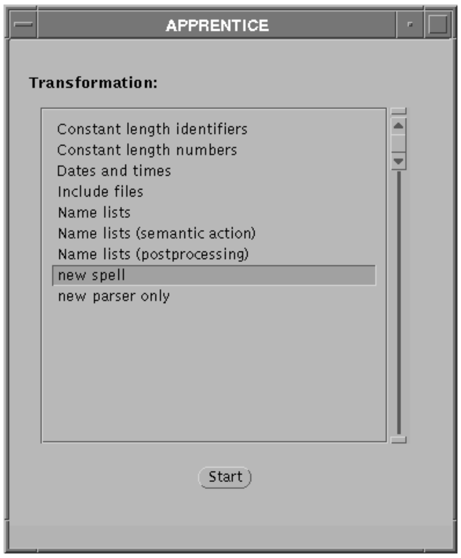


Figure 4.12: APPRENTICE provides a start up interface to a set of spells.

is written to a file. Pre- and postprocessing commands can contain available UNIX level commands and applications. Preprocessing commands are especially useful for simplifying the source document by removing parts unnecessary for the transformation and perhaps streamlining similar parts. Preprocessing may simplify the source grammar extensively and thereby also the mapping specification. Postprocessing, on the other hand, is suitable for making simple conversions like translating the target document into the appropriate platform format (e.g., PC or UNIX). Pre- and postprocessing commands can also be used for creating macro spells by linking several spells together. Pre- and postprocessing commands are defined in the Transformation components window of the spell (Figure 4.13).

The complete spell (Figure 4.14) then can consist of

- *preprocessing commands* to be performed on the source document before it is parsed,
- a *source parser* that reads the source document and builds the corresponding source tree,
- a *source-to-target mapper* that traverses the source parse tree and constructs the corresponding target parse tree according to the TT-grammar technique,

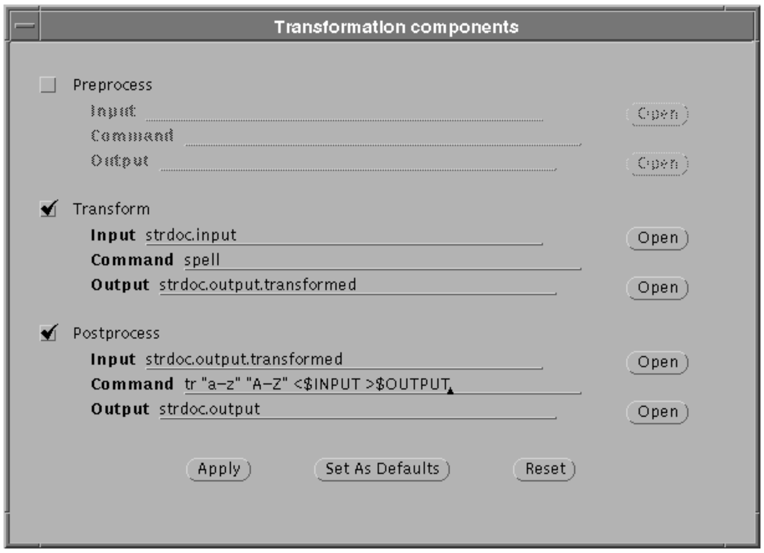


Figure 4.13: Spell components to be included in the spell.

- a *target parser* or unparser that writes out the frontier of the target parse tree, and finally
- *postprocessing commands* performed before the target document is written to a file.

Pre- and postprocessing are optional as is the mapping phase. If the mapping phase is missing, the source tree is copied directly to the target tree.

4.4 ALCHEMIST implementation

The architecture of ALCHEMIST has been kept as open as possible. All subcomponents of ALCHEMIST run also stand-alone without the unifying framework. For example, the user may want to use only MAPPERTOOL without the other ALCHEMIST tools, or he/she may want to produce a stand-alone parser with SEER and may do so without the help of the SPELLTOOL. On the execution level, all spells run stand-alone without the need of APPRENTICE.

The implementation has been done in C++ through object-oriented programming. `yacc` and `lex` are used in SEER to generate the source parser, but all other ALCHEMIST components including the mapper generator SWINDLER has been implemented from scratch. ALCHEMIST with

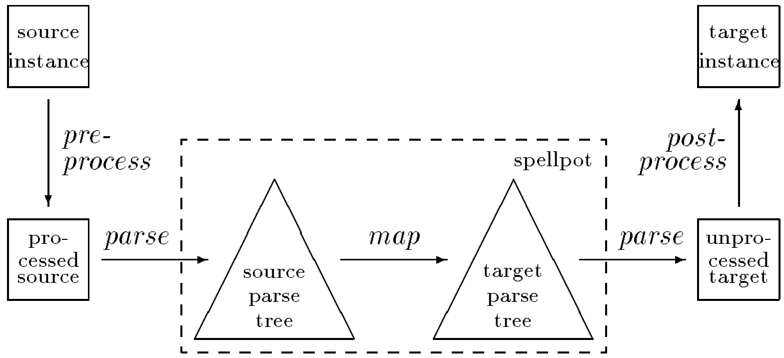


Figure 4.14: Data flow of a spell.

components consist of about 15 000 lines of C++ code. Typical spells contain about 9000 lines of C++ or more depending on the size of the grammars and the mapping. The majority of these 9000 lines are default code lines for interface and predefined semantic actions like symbol table checking. ALCHEMIST and its spells run under the Solaris 2.x and CDE operating system on Sparc machines.

Spells can be compiled with several C++ compilers. We have used both the AT&T C++ compiler and the Gnu `g++` compiler. Spells can easily be extended with the C++ programming language without changing the file structure of the generated code. A C++ file for user defined procedures has been included.

ALCHEMIST is fully operational in the way as has been explained in this chapter. ALCHEMIST has been extensively used for building transformations, especially for providing an interface between two development environments [LV95].

Chapter 5

The SGML transformation language TranSID

The TranSID language is a tree-based transformation language [JKL96a, JKL96b, JKL97]. 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 SGML document. Design goals of the language included declarativeness, simpleness and implementability with reasonable effort. Special features include a *bottom-up evaluation* process and the possibility to restrain the transformation to the event-based strategy. The event-based top-down strategy is sufficient for simple formatting of the SGML document. Bottom-up evaluation is a declarative way of defining some transformations that would be awkward to define in a top-down manner. The TranSID language also includes high level declarative commands that frees 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].

The Document Style Semantics and Specification Language Standard (DSSSL, [ISO96]) defines a related transformation language. DSSSL is, however, 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. Above all, we have strived to make TranSID a simple, declarative language that is easy to use. Simple transformations should be easy to specify!

In this chapter we present the TranSID language and its implementation. We start by giving a short explanation of the data model and the evaluation strategy of TranSID and then give some examples of its use. We present the language through examples. We conclude by giving an overview of the implementation.

5.1 Overall control and data model

The transformation process in the TranSID language is similar to the grove transformation process of the DSSSL standard [ISO96] and also to the spell process of ALCHEMIST. The basic environment consists of an SGML parser, a TranSID parser, a transformer and a linearizer (Figure 5.1).

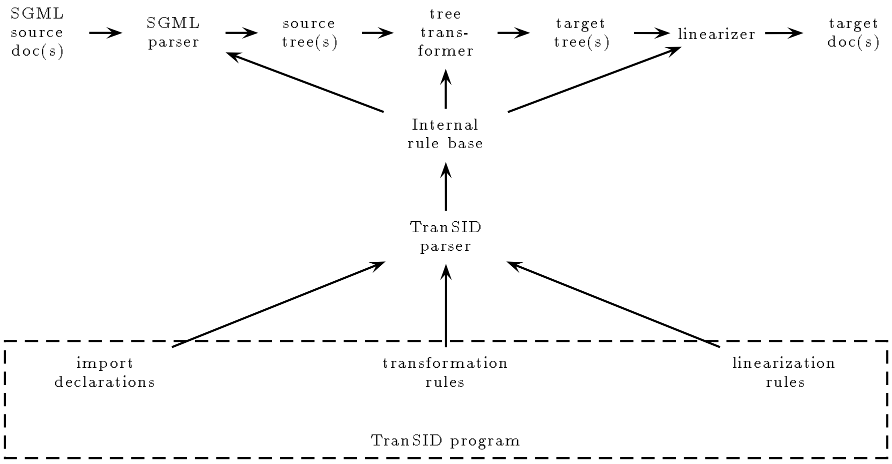


Figure 5.1: The TranSID transformation process

A TranSID transformation starts by parsing an SGML document instance and constructing an internal document tree. We use the SP parser [Cla96] 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. The transformation is performed by the tree transformer which traverses the constructed parse tree and applies the transformation rules to build a corresponding target representation 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.

5.2 Semi-formal semantics

We present a semi-formal syntax and 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). The following description is adapted from [JKL97].

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 5.2).

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 include insertions of new structures and selection and combination of tree components relative to the root of T.

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. 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 $\mathcal{R}_i$ is the first rule in $\mathcal{P}$ that matches

$$a(\mathcal{P}(T_1, \ldots, T_n)) \ , \tag{5.1}$$

then

$$\mathcal{P}(T) = \mathcal{R}_i(a(\mathcal{P}(T_1, \ldots, T_n))) \ . \tag{5.2}$$

3. 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 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.

Equations (5.1) and (5.2) 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 want to stress that there is no evaluation order defined between nodes at the same level in the tree. For example, leaf level nodes (data) may be evaluated in an arbitrary order or even in parallel.

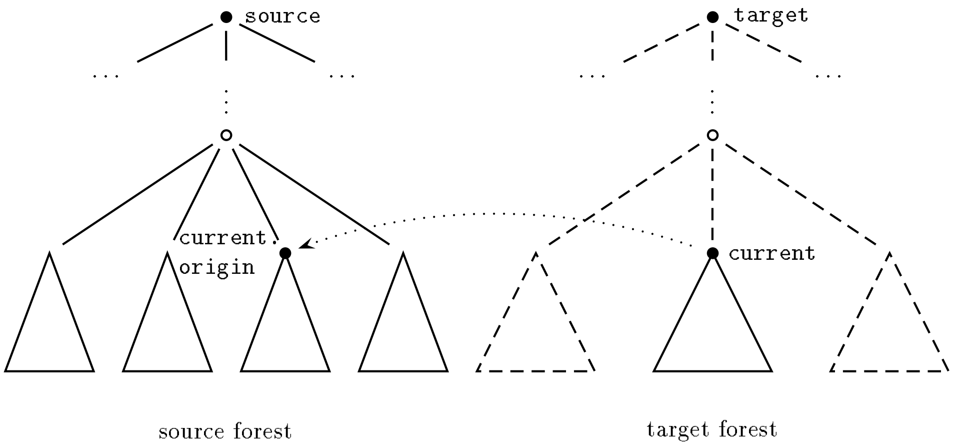


Figure 5.2: Source and target forests of a transformation process. Reachable structures from the `current` node are marked with solid lines, unreachable or yet uncreated ones with dashed lines.

5.3 TranSID transformations

We present the basic components of the TranSID language through small 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 document instances.

A transformation program consists of *transformation rules*. A transformation rule consists of a *source clause* and a *target clause*. A source clause locates a node in the source tree. The node can be located by name and/or additional conditions that refer to any part of the source tree. During the transformation, the source tree is traversed in a bottom-up way. For each node, the rule base is checked for a rule with a matching source clause. When a rule is found, the actions specified in the target clause are performed. A target clause describes how the located node is replaced by a target forest.

A transformation rule then has the following format.

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

Any node type in the tree, such as an element or an attribute is first recognized by a node clause and further tested for a condition. These two lines 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 two rules

```
ELEMENT "Entry"
WHEN    current.descendants.having(this.name ==
          "PartofSpeech").children.find("n.")
BECOMES <"Noun_Entry">{current.children} ;

ELEMENT "Entry"
WHEN    not current.descendants.having(this.name ==
          "PartofSpeech").children.find("n.")
BECOMES null ;
```

prunes an SGML document and includes only entries that in their `PartofSpeech` element contain the string "`n.`". The node clause of the first rule

```
ELEMENT "Entry"
```

locates `Entry` elements but only when the condition

```
current.descendants.having(this.name ==
        "PartofSpeech").children.find("n.")
```

holds. The condition is stated as an *orientation expression*. An orientation expression consists of *locators* separated by dots ("."). The first locator must always be *absolute*, i.e., point to a certain node in the tree. In the above expression, the locator `current` is absolute and points to the node that is being transformed. All other locators in an orientation expression must be relative, i.e., relative to the node or nodes indicated by the absolute locator. In the above expression, the relative locator `descendants` locates the subelements of the `current` node.

The evaluation of the expression proceeds from left to right. Every locator returns a list of nodes that are used as input for the next locator in the expression. In this sense TranSID expressions resemble expressions in the MetaMorphosis transformation language [MID95], which was an important source of inspiration for the design of TranSID. The relative locator `having` selects the nodes that satisfy the condition expressed as a parameter of the `having` locator. In this case, the `having` condition contains an orientation expression and a constant string. The locator `this` refers here to the descendants of the current node, one at a time. The property operator `name` locates the name of the descendant elements and the entire condition checks whether the found name equals the constant string `PartofSpeech`. Only elements that satisfy this condition are used as input for the next locator which locates the children of the `PartofSpeech` elements. In this case, we assume them to be text strings #PCDATA in SGML. The string operator `find` locates only the text elements that contain the string "`n.`".

The source clause of the rule above matches sections like

```
...
<Entry>
        <HWGroup>
                <Headword>spaz</Headword>
                <Pronunciation>spæz</Pronunciation>
                <PartofSpeech>n.</PartofSpeech>
        </HWGroup>
        ...
</Entry>
...
```

but it does not match entries that do not contain the string "`n.`" in the `PartofSpeech` element. Those entries are matched by the second rule because it contains the same condition negated by the Boolean operator `not`.

The target clauses of the rules are different as well. The target clause of the first rule constructs new elements named `Noun_Entry`. The name of the new element is stated between angle brackets. The contents of the new element is stated as a list between braces. The contents is deduced by the orientation expression that locates and copies all the subelements of the current node as new contents for the new `Noun_Entry` element. Intuitively, the meaning of the first rule is then to locate `Entry` elements with the string "`n.`" in their `PartofSpeech` subelement and to replace these `Entry` elements with `Noun_Entry` elements that contain the same subelements as the original `Entry` elements. The second rule removes all `Entry` elements that do not satisfy the condition of the first rule (and thereby satisfy the condition of the second rule). Replacement of the elements by the empty list `null` effectively removes them from the result.

When the above two rules are applied to the above entry using TranSID, the result is

```
<Noun_Entry>
        <HWGroup>
                <Headword>spaz</Headword>
                <Pronunciation>spæz</Pronunciation>
                <PartofSpeech>n.</PartofSpeech>
        </HWGroup>
        ...
</Noun_Entry>
```

with entries that are not nouns removed.

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

```
ELEMENT "Entry"
BECOMES <"Entry" PoS = current.descendants.
            having(this.name == "PartofSpeech").children>{
                <"HWGroup">{
                  current.children.having(this.name !=
                    "PartofSpeech")
                },
                current.children.having(this.name !=
                  "HWGroup")
            } ;
```

The rule locates `Entry` elements and replaces them with corresponding elements where the contents of their `PartofSpeech` element has been added as the value of the attribute `PoS`. In the example above we get the following result.

```
<Noun_Entry PoS ="n.">
        <HWGroup>
                <Headword>spaz</Headword>
                <Pronunciation>spæz</Pronunciation>
        </HWGroup>
        ...
</Noun_Entry>
```

Finally, we have the transformation described in Section 1.1, which turns the dictionary entry into a LaTeX formatted version in Example 1.2. Specifying this transformation with the TranSID language, we get the following program.

```
transformation begin

ELEMENT "Headword"
BECOMES "{\\bf ", current.children, "} " ;

ELEMENT "Pronounciation"
BECOMES "(", current.children, ") " ;

ELEMENT "PartofSpeech"
BECOMES "{\\em ", current.children, "} " ;

ELEMENT "Etymology"
BECOMES "{\\em ", current.children, "} " ;

ELEMENT "Definition"
BECOMES current.children, "\n", "\\newline", "\n" ;

ELEMENT "Quotation"
BECOMES current.children, "\n", "\\newline", "\n" ;

ELEMENT "Year"
BECOMES "{\\bf ", current.children, "} " ;

ELEMENT "Author"
BECOMES "{\\sc ", current.children, "} " ;

ELEMENT "Work"
BECOMES "{\\em ", current.children, "} " ;

ELEMENT *
BECOMES current.children

end
```

Here we can use extensively the default rules of TranSID. If there is no rule given for a certain element, the element is just copied to the target. However, in this program we have the asterisk rule that matches all elements but only if the previous rules do not match. The asterisk rule (the last rule) removes the SGML tags and copies only the contents to the target. All other rules remove tags in addition to inserting constant strings around the element contents. The result of the transformation performed on the example SGML document can be seen in Figure 1.2 on page 5.

5.4 TranSID operators

The only data type of the TranSID language is a *list* of nodes. A list can also be empty. TranSID uses the concept of polymorphic lists. A node can be an SGML element, a #PCDATA element, a processing instruction, an attribute, or 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.

Therefore all TranSID operators operate on lists. TranSID programs may use a variety of tree transformation operators, string operators, regular expressions, etc. The idea is to have a declarative, quite complete set of tree transformation operators that may be used in a transformation modifying the SGML trees. Nodes in the SGML tree may be located by the reserved words of the node clause, like `ELEMENT`, `ENTITY`, `PI`, `ATTRIBUTE`, `DATA`, `NODE`. Here, `ELEMENT` locates elements, `ENTITY` entities, `PI` processing instructions, `ATTRIBUTE` attributes, and `DATA` #PCDATA (and also other data). `NODE` locates any type of nodes. All these reserved words must be succeeded by a node name or the asterisk `*` which stands for any name.

Absolute locators are `null`, `source`, `current`, `these`, and `this`. The locator `null` is used for removing nodes as in the example above, while `source` locates the root of all the source trees, and `current` the node that is being transformed. The locators `these` and `this` refer to nodes that are being processed all or one at a time in conditions, or in the `map` and `glue` operators described below.

Relative locators produce a new set of nodes from a node list. There are *positional locators* like `elements`, `entities`, `attributes`, and `pi` that locate the various subcomponents of an element. The locator `children` locates all the children (elements, entities, attributes, and processing instructions) of its input nodes, whereas `descendants` locates all of their descendants. Consequently, `ancestors` locate 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 the input nodes. On the other hand, locators `previous` and `next` returns the previous or next nodes in postorder, respectively, and `predecessors` and `successors` locates all previous or succeeding nodes in postorder, respectively. The locator `parent` locates the parent of the input nodes while `data` returns the data, only.

Quite related locators are the *filtering locators* `first`, `first(n)`, `having(`*Condition*`)`, `last`, `last(n)`, and `sublist(`$n; m$`)`. The locators `first` and `first(n)` locate the first or first `n` nodes of the input nodes; `last` and `last(n)` the last or last `n` nodes. The expression `having(`*Condition*`)` tests the input nodes for a condition and locates only those that satisfy the condition. The condition may be an arbitrary orientation expression that references any part of the SGML trees. The locator `sublist(`$n; m$`)` locates a specified subset of the input nodes. 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. Assume that `m` and `n` are positive integer values. Then the `sublist` operator locates nodes in a list as follows.

`sublist(m, n)`	Select `n` elements starting at element `m` from the beginning of the list
`sublist(-m,-n)`	Select `m` elements starting at element `m+n-1` from the end of the list
`sublist(m,-n)`	Select middle elements starting at element `m` from the beginning of the list and ending at the element `n` from the end of the list
`sublist(-m,n)`	Select `n` elements starting at element `m` from the end of the list

The *application operators* `glue`, and `map` are two very strong operators. The operator `map(`*Condition*; *Construction of target subtrees*`)` performs the actions for every single node that satisfy the condition. The operator `glue(`*Condition*; *Condition*; *Construction of target subtrees*`)` groups nodes together if the nodes satisfy the first condition but not the second and performs the action specified in the third parameter. The located nodes may be referenced by the absolute locator `these`.

Nodes may be tested for *properties*. The operator `name` locates the name of an element, attribute or entity, while `attribute(`*Attribute name*`)` locates a certain attribute of an element. The locator `siblingnum` returns the order number of the node among its siblings, and `samenum` the order number of the node amongs siblings with the same name. The locator `count` counts the number of the nodes in a node list.

Several other operations have been included into the TranSID language. *String operations* and *regular expressions* include ordinary string operations such as comparison, catenation and search, as well as more sophisticated operations based on regular expressions for string matching and replacement. As an example consider the following rule.

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

This rule replaces four-letter upper-case strings beginning with the letter `S` and ending with `L` by the string `the standard` followed by the located string. For example, the strings `SGML` and `SMDL` are replaced with `the standard SGML` and `the standard SMDL`, respectively, but only if the #PCDATA element contains a word beginning with `defini`, like `definition` or `defining`. There are also operations for searching and matching strings, for simple testing if a string contains only letters or digits or both, for converting capital to small letters and vice versa and for extracting file names and url components from strings.

5.5 TranSID implementation

The TranSID evaluation environment has been implemented in C and C++ and has been tested to run in the Linux, Solaris, and AIX environments. The environment consists of the SP SGML parser [Cla96], a TranSID parser implemented with `yacc` and `lex`, and an evaluator and a linearizer both implemented in C. All modules are independent and may call each other recursively. The source code of the current version contains about 15 000 lines of code.

The implementation is fairly straight-forward. TranSID maintains an internal tree database for managing the SGML trees. Memory usage might therefore be high. This bottleneck is solved by using sharing structures, i.e., using references instead of copies of tree nodes.

Chapter 6

Experience and evaluation

In this chapter we describe the experience we have gained in using ALCHEMIST and TranSID. We start by describing different applications built by ALCHEMIST, and then move on to applications of TranSID. We finish by making comparisons between the two systems.

ALCHEMIST has been developed during several years as a part of a project called VITAL [SMR93]. The VITAL project defined and implemented a methodology for building knowledge-based software systems. We have had numerous possibilities of testing and evaluating ALCHEMIST in this project. Especially, we have received feedback from other project partners, which we have been able to take into account in developing ALCHEMIST further.

The main ALCHEMIST application until now is the VITAL *bridge* [LTV95b] we built between the VITAL workbench [DMW93] and a commercial computer-aided software engineering (CASE) tool FOUNDATION by Andersen Consulting [And93a]. The VITAL workbench consists of several knowledge-based software development tools, such as knowledge acquisition and conceptual modelling tools. The CASE tool has similar components for defining concepts such as entity relationship modelling and data flow diagrammers. In an ideal KBS development environment the user should be able to use KBS tools for building KBS specific parts of a software system, and CASE tools for building traditional parts such as user interfaces. For achieving this we built bridges, i.e., spells, between tool representations, between a conceptual modelling tool in the VITAL workbench and several corresponding tools in the CASE environment. The user may freely change environments depending on the development task. Here we should note that all underlying persistent representations are considered as structured documents. Therefore, ALCHEMIST is highly suitable in building transformations between the representations.

ALCHEMIST has also been used in another bridge from the VITAL workbench. We have also built a spell from a hierarchy laddering tool called ALTO [MR90] to C++. The user specifies a graphical hierarchy of concepts with attributes in ALTO and can automatically transform the hierarchy into C++ definitions, which provides an easy and fast way of producing a consistent set of C++ class and object descriptions.

Additionally, we have experimented with some smaller applications. As our first case we built a spell between a simplified version of an entity-relationship model representation and a relational database system. In the beginning this spell was purely syntactical but we soon learned that we needed also semantic actions to be able to complete the spell specification.

Our experience gained from TranSID is not so extensive as TranSID mainly has been designed and developed during the last two years. We have used TranSID in usual SGML transformations, such as transformations between SGML and LaTeX and SGML and HTML. The experience shows that, especially, when the transformations are simple, also the transformation specifications are very simple. Also, the relative declarativeness of TranSID helps the user in writing simple and understandable programs.

In this chapter we present some ALCHEMIST and TranSID applications, sometimes with the help of examples and see what lessons we learned from their implementation. Often we could use feedback from one transformation when building the next. In some occasions, we had to introduce new features into the systems to be able to solve more complicated problems. We start with ALCHEMIST and its the main application, the interface between the two development environments, and then to the smaller applications. We also briefly note some lessons we learned in spell generation and give some evaluation of ALCHEMIST and its appropriateness for document transformations. Thereafter we take a look at TranSID applications and compare the usefulness of TranSID with ALCHEMIST.

6.1 An ALCHEMIST interface between two development environments

The VITAL workbench [DMW93] contains a set of tools for constructing a knowledge-based software system. The workbench is based on the VITAL methodology for building such systems [SMR93]. The workbench contains tools for knowledge acquisition and modeling as well as system design and visualization. On the other hand, the FOUNDATION CASE tools contains tools for software design such as different conceptual modeling tools and tools for defining user interfaces [And93b].

There are several reasons for the bridge between the two development environments [Ver94]. Firstly, one of the design principles of the VITAL workbench was openness. The user of the workbench should be able to import as well as export design schemas and models to and from the workbench. Especially, the user may want to use a special tool for providing a certain component of the system he/she is building. We do not, of course, provide spells from and to *all* external tools, but the user can use ALCHEMIST to build a new interface for some additional tool. Secondly, the VITAL workbench is a specialized environment for building knowledge-based systems and thereby concentrates on KBS properties. On the other hand, general CASE environments usually contain quite sophisticated tools and techniques for typical software components such as user interfaces, data structure planning, etc. The VITAL methodology expects and depends on the user to use other external tools as well for building KBS systems.

For the bridge between the VITAL workbench and the FOUNDATION CASE tool, we finally chose some very specific tools to interface [LV95]. In the VITAL workbench we chose the Operationalizable Conceptual Modelling Language (OCML) editor and in FOUNDATION we chose several diagram tools, such as an entity-relationship diagram tool, a data flow diagram tool, a procedure diagram tool and a data objects specifier tool. Interfacing on the conceptual modeling level seems most appropriate, as this level contains well-defined specifications without going into implementation details. The OCML editor lets the user specify a knowledge-based system with the help of domain, task and model diagrams. A domain diagram describes concepts, their instances, attributes and relationships, as well as relations between the concepts. A task diagram describes a certain problem solution containing processes and data elements. The diagram may contain sequential and choice tasks, and tasks may be recursive. In overall, a task diagram gives a graphical view of a knowledge-based program, and the diagram also works as a visualization of the program. The model diagram collects certain domain and task diagrams that together describe a certain problem and its solution.

The FOUNDATION Design environment contains, among other tools, an entity-relationship diagrammer for drawing entity relationship diagrams, a data flow diagrammer for data flows and a procedure diagrammer for describing the solution process of a problem (Figures 6.1 and 6.2). The data objects specifier lets the user describe data objects, their types and relations in table format. The entity relationship diagram contains entity and relationships types. Entity types may also have attributes. A data flow diagram contains data collections and processes and their connections. A procedure diagram shows in what order the processes and subprocesses of the data flow diagram are executed. It may contain iterative processes and conditional clauses [And93a].

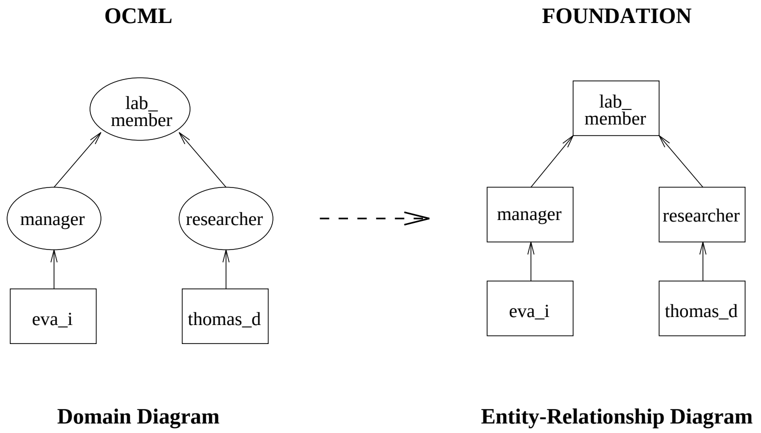


Figure 6.1: An example transformations from an OCML domain diagram to an FOUNDATION ER diagram.

In our CASE bridge we chose to interface the OCML domain diagrams with entity relationship diagrams because of their close resemblance. Both describe concepts and their relations and properties. The concepts are also transformed into a data object table maintained by FOUNDATION. We can say that there is a one-to-one connection in both cases as the OCML domain diagrams are completely represented in both entity relationship diagrams and data object tables. There was however, no close relative of the OCML task diagrams in the FOUNDATION environment. Therefore, we chose to interface task diagrams with *two* FOUNDATION diagrams types, the data flow diagrams and the procedure diagrams. The data flow of the task diagrams are transformed into FOUNDATION data flow diagrams and the process is transformed into procedure diagrams.

The bridge between the environments was implemented only in one direction, from the VITAL workbench to the FOUNDATION environment. It was obvious that the need in this direction was greater. The user could first start by specifying and building knowledge-based parts of the system and then completely change environment and finish the system in the FOUNDATION environment [VL94].

The spells included in the VITAL bridge between the workbench and the FOUNDATION CASE tool are [LMQ+95, LQV95]

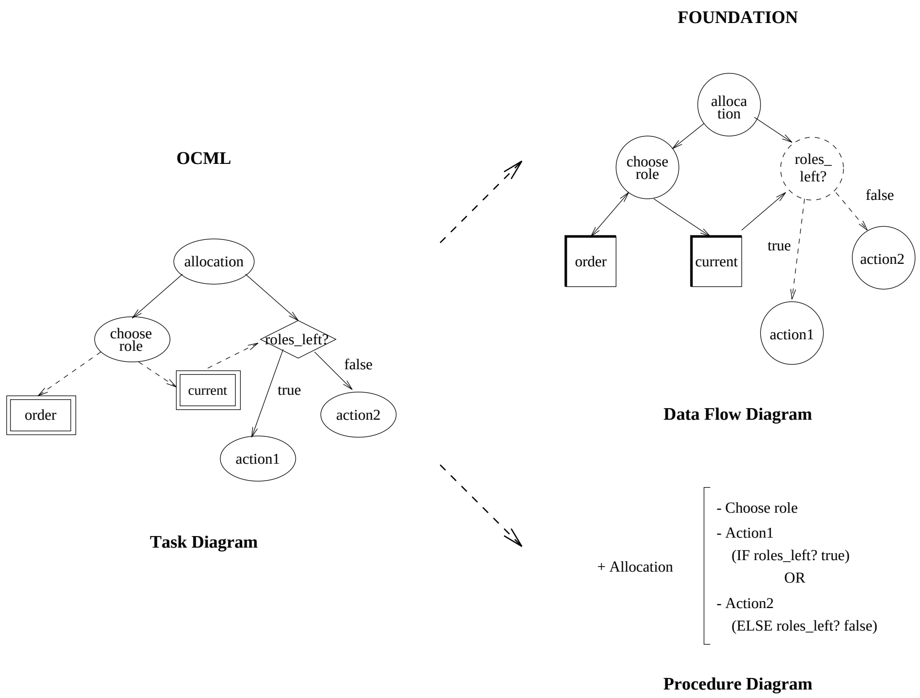


Figure 6.2: Transformations from OCML task diagrams to FOUNDATION.

- DOM2ERD, a spell for transforming OCML domain diagrams into FOUNDATION Design entity-relationship diagrams,
- DOM2OBJS, a spell for transforming OCML domain diagrams into FOUNDATION tables of data objects,
- TASK2DFD, a spell for transforming OCML task diagrams into FOUNDATION Design data flow diagrams, and
- TASK2PD, a spell for transforming OCML task diagrams into FOUNDATION Design procedure diagrams.

In Figures 6.1 and 6.2 (from [LTV96]), we see examples of the graphical representations of the transformations performed by the spells DOM2ERD, TASK2DFD, and TASK2PD. The OCML digrams describe a simple room allocation problem called Sisyphus [Lin92] where a set of laboratory workers should be assigned rooms in an office building. There are, however, several restrictions that say, e.g., that secretaries should be placed near the manager, and that a manager should get the biggest office. These restrictions are solved in the OCML editor in the VITAL workbench. The Figures 6.1 and 6.2, show how simplified diagrams are transformed into diagrams of the FOUNDATION CASE tool. The DOM2OBJS spell is straightforward and not shown here.

All the diagrams and tables involved in the spells have a persistent representation based on either text files or binary number files. The OCML persistent representation is written in Lisp, where some additional features have been included for representing coordinates of the graphical figures of the diagrams (Figure 6.3). The FOUNDATION representations are more complex. Information may only be imported into the CASE tool through special import files and their graphical representations. Data flow and procedure diagrams are therefore represented by two different files each, one text file for the logical objects and one binary number file for the graphical outlook of the diagram (Figure 6.3). These two files must, naturally be consistent with each other, i.e., contain the same objects in the same order. However, both graphical and logical information is represented in the same import file for entity-relationship diagrams. Also data objects are specified in one import file

These restrictions put some more strain on the bridge implementation. The DOM2ERD and DOM2OBJS spells only produce one file each for every transformation. The TASK2DFD and TASK2PD spells, though, must produce both a text file and a binary number file for each transformation. This problem was solved by producing a combined file as a result that was divided with a postprocessing command in the ALCHEMIST spell.

Figure 6.4 shows an example of a complex mapping rule in the specification of the TASK2DFD spells. The source subgrammar identifies a task diagram task and its name. The target subgrammar contains the definition of a data flow process where the task name is copied several times into the constructed process.

The problem list that had to be solved in the spells is quite extensive [VL95], ranging from simple identifier modifications and checking to global computation of object order and numbers. We saw that ALCHEMIST was very suitable for handling local transformations, where one source object corresponds to one or several task objects. All identifier requirements could be met through semantic actions. For example, identifiers in FOUNDATION had to be unique, while in OCML a name could appear several times in different contexts. More difficult problems to solve where the global computation needed for the binary number files (the graphical files). These files had to contain information about how many symbols the diagrams contained. Each object also had to have its unique order number, the only

OCML task diagram:

```
;;; -*- Mode: Lisp, Design Task Layer; Package: DL -*-

(dale:defgraphics dale:coordinates
 dale::subtask (choose-role (130 130 210 180)
                action1 (305 225 385 275)
                allocation (248 16 328 66)...)
 dale::choice (roles_left? (502 117 582 167)...)

(def-task choose-role
 ((|:::| allocation dale::kldesign-data-link dale::role-alias))
 ((:rule-iteration-type . :try-once)
  (:represented-as :rules choose-role1 choose-role2 choose-role3)
  (:inter-diag-alias)))

(def-choice-task roles_left?
 ((false action2 dale::kldesign-f-control-link dale::subtask)
  (true action1 dale::kldesign-t-control-link dale::subtask)) NIL)

(def-role order
 ((|:::| choose-role dale::kldesign-data-link dale::subtask))
 ((:represented-as :relation role-order)
  (:inter-diag-alias)))
...
```

FND data flow diagram, logical file: FND data flow diagram, graphical file:

```
HEADER              ...            F   F   O   R   M   X   X   X
0                   ...            U 050 049 052 032 032 032 032
2DEDFDIAG        1  ...            J   B   B 032 032 032 032 032
2DEDFDIAG        2sisyphus       032 032 032 032 032 032 032 032
2DEPROCSS        1  ...          ...
2DEPROCSS        2CHOOSE-ROLE      D   A   T   A 032   F   L   O
...                                W 032   D   I   A   G   R   A
2DEEXTENT        1  ...            M 032 045 064
2DEEXTENT        2ORDER          ...
...                              001 000   D   F   S   Y   M 032
3DEDFDIAGDEPROCSS3  ...          032 032 046   D   W   B 028 000
3DEDFDIAGDEPROCSS4sisyphus       ...
 ... 0000100000 ACR              001 032 001 020 000 003 000 000
CHOOSE-ROLE                      000 000 000 000   D   R   D   O
...                                C   D   O   C 126 052 045 059
3DEDFDIAGDEEXTENT3  ...          071 034 103 059 000 000 000 000
3DEDFDIAGDEEXTENT4sisyphus       000 000 000 000 000 000 000 000
 ... 0000100000 ACR  ORDER       000 000 000 000 000 000 000 000
...                              000 000 000 000 018 000   A   L
                                   L   O   C   A   T   I   O   N
                                 013
                                 ...
```

Figure 6.3: Different representations formats in the VITAL bridge.

OCML task diagram source production group:

```
Task -> "(" "def-subtask"  Task_Name  Links  Attributes ")" ;
Task_Name ->
    OCML_IDENTIFIER ;
```

FOUNDATION data flow diagram target production group:

```
Task.Entity_procss_data_record ->
    "2" "DEPROCSS" "        " "2" Task_Name.Part_entity_id
    "0000100000" " " "ACR  " "        " "        " "        "
    "        " "          " "  " "00001" "        " "00225"
    "000100000000" "        " "ACRA" "+0001" "+0000"
    "DEPROCSS" Task_Name.ENTITY_ID "  " Task.PRC_TYPE
    "0000+000" "0" "           N" "VITAL   +" CURRENT_DATE
    "VITAL   +" CURRENT_DATE "+" CURRENT_TIME "        "
    "+000000" "+000000" "+0+0+0+0+0"
    Task_Name.ENTITY_SHORT_DESC "E"
    Task_Name.Short_description "\n" ;

Task.PRC_TYPE ->
        "2" ;

Task_Name.ENTITY_ID -> IDENTIFIER[32] ;
    /* semantic action */
    symbol_table.insert( OCML_IDENTIFIER ) ;
    task_name = symbol_table.get_FND_id( OCML_IDENTIFIER ):
    set_id(IDENTIFIER[32], task_name) ;
    /* end semantic action */
Task_Name.Part_entity_id -> IDENTIFIER[32] ;
    /* semantic action */
    set_id(IDENTIFIER[32], task_name) ;
    /* end semantic action */
Task_Name.ENTITY_SHORT_DESC -> IDENTIFIER[32] ;
    /* semantic action */
    set_id(IDENTIFIER[32], task_name) ;
    /* end semantic action */
Task_Name.Short_description -> IDENTIFIER[25] ;
    /* semantic action */
    set_id(IDENTIFIER[25], task_name) ;
    /* end semantic action */
```

Figure 6.4: A OCML source subgrammar and a FOUNDATION target subgrammar with appropriate semantic actions attached.

Spell	Source productions	Target productions		Mapping rules
		logical	graphical	
DOM2ERD	85	71		35
DOM2OBJS	85	81		10
TASK2DFD	80	49	61	37
TASK2PD	80	17	37	13

Table 6.1: Number of productions and rules in the spell specifications of the VITAL bridge.

label through which it could be referenced. These problems were solved by using semantic actions directly implemented in the underlying programming language.

The sizes of the spell specifications are shown in Table 6.1. In this bridge implementation, it was especially convenient to use ALCHEMIST as we could reuse grammars and mappings already defined, saving a lot of work. In all cases, the source grammars were quite large, 85 and 80 productions, respectively. The number of productions were reduced from about 120 with the help of a suitable preprocessing command that simplified the source representation. The task grammars contained from 50 to 80 productions; the logical and graphical representations were usually quite distinct and required two separate grammars. From the number of mapping rules, we can also conclude that two of the mappings were more extensive than the other two. On the other hand, especially the TASK2PD spell required quite a lot of global computation which does not show in the number of mapping rules.

6.2 Other ALCHEMIST transformation applications

Even if the CASE bridge was the most extensive generation of spells with ALCHEMIST in the VITAL project, we used ALCHEMIST also in some other spells within the project. Among others, we defined and implemented a spell ALTO2C++ from a laddering tool called ALTO [MR90] to C++ definitions [TL94b]. With ALTO, which is part of the VITAL workbench, the user may draw conceptual hierarchies of the problem domain. In a hierarchy, a concept may have subconcepts and there may be instances of a concept. A concept may also have attributes that are inherited by subconcepts. The hierarchy can be used as the object model of a C++ program. Concepts correspond to classes, attributes to class attributes, and instances to objects. The spell ALTO2C++ transformed these concept hierarchies to the corresponding C++ definitions. This spell was half implemented with ALCHEMIST, half by hand. We used the gained experience in developing ALCHEMIST further. Among other things we saw that we had to include semantic actions in the spell specification. For example, the ALTO tool did not make any difference between inherited attributes and local attributes. Therefore, we had to check with a semantic action if an attribute had been defined in a superclass every time it appeared in a class definition. If it had been defined, we did not have to define it again, otherwise it was defined for the first time in this class description.

Outside the VITAL project, we also defined and implemented a spell from a simplified entity-relationship model to a relational database language. The spell translated a textual representation of an ER model into SQL that declared the corresponding relational tables in the database language. In this spell as well we needed to use semantic actions to perform part of the transformation.

Several other toy examples were solved with ALCHEMIST. Especially, we learned the need for pre- and postprocessing of files. We implemented some simple SGML transformers, reading SGML documents and outputting formatted versions of the documents. In these cases we had to redefine the SGML DTDs into ALCHEMIST source grammars which is quite straightforward, e.g., the SGML content model corresponds to the right hand side of a production. Only iterations in the DTDs must be slightly modified and expressed through recursive productions.

6.3 ALCHEMIST observations

During the use and testing of ALCHEMIST in the VITAL project and outside the project, we obtained information about the usefulness of ALCHEMIST. Some of this experience also helped us in the development of ALCHEMIST suggesting further functionalities to be included in the system.

Above all, we were impressed with the generality of the system. Our spells ranged from simple syntactic ones to very complex, large and detailed ones. We were able to use ALCHEMIST in building them all. The high level of abstraction made it easier to make changes in the specifications, especially when the underlying representations changed. The implemented spells were fast enough, usually much faster than, e.g., reading the persistent representations into the tools that were interfaced. Of course, we also encountered a number of problems, all of which were solved during the development.

The possibility of specification reuse was well appreciated. In the bridge between the VITAL workbench and the FOUNDATION CASE tool, we built several spells that were based on the same representations. For example, we were able to reuse the grammar of the domain diagrams from the DOM2ERD when building the DOM2OBJS spell. Even if the target representations often differed, we were able to reuse certain functionalities and procedures previously defined in another spell.

By separating the specification from the implementation, we were able to update our spells easily and quickly when the underlying representations were changed. During spell development, the VITAL workbench was still being constructed, and we had to take into account the changes in the persistent representation of the OCML editor.

By using a strategy for developing some spells on the side of ALCHEMIST itself, we were able to use immediate feedback in improving ALCHEMIST. When building the ALTO2C++ spell, we did many things by hand. In the DOM2ERD spell, we only had to add some semantic actions manually. In the TASK2DFD spell, adding of the semantic actions was already included in the specification phase. Most of the user defined routines for semantic checking, etc., were developed alongside the spells and are now provided with the current version of ALCHEMIST.

Naturally, we also encountered some problems and shortcomings in the use of ALCHEMIST. Defining the grammars of the involved representations can be a difficult task. The user needs to know something about context-free grammars and their use. Our tool representations were so complicated that the first tries of defining grammars were rather unsuccessful. By including preprocessing of the source files, we were able to simplify the grammars so that they became unambiguous. At the beginning we even had problems with the capability of `yacc` and `lex`; the parser generator just could not handle our large grammars.

Building the grammars from scratch was a tedious task. We had only a few details about the representations, and often the descriptions were not quite correct. The first phase of the spell specifications usually went into a trial-and-error approach of testing the tools to see what kind of persistent representations they used. Only through a very concentrated detailed work were we able to describe the representations exactly. The descriptions of the four spells of the VITAL bridge take about 180 pages [LQV95]! On the other hand, when the representations had been elucidated, building the transformations with ALCHEMIST was a more straightforward thing. Instead of months we were soon down to weeks and days for implementing a particular spell.

As has been mentioned earlier, ALCHEMIST may not be the best tool for building transformations that contain a lot of global computing. ALCHEMIST seems to be very suitable for transformations where a source object transforms into one or several target objects. This was very much the case in the VITAL bridge. FOUNDATION representations were much more complex and often some declarations had to be repeated several times for a certain object. For example, drawing a relationship in the FOUNDATION entity-relationship diagrams required that the logical relationship as well as the graphical object had been defined. Additionally, the FOUNDATION persistent representation required both logical definitions about which entity types were connected to the relationship and graphical definitions about where and how the relationship was drawn in the diagram. Therefore a simple OCML relationship (an arrow between a concept and its subconcept) was translated into *six* different relationship definitions on the FOUNDATION side. With ALCHEMIST these multiple definitions cause no problems as the target side may be specified with a subgrammar that constructs several separate target subtrees.

On the other hand, the TASK2PD spell required that most of the target document was specified in the graphical file. The logical file only contained the name of the procedure diagram, while all other procedure names and structures as well as the object positions were specified in the graphical file. All graphical objects were indexed with a running number that was thereafter the only way to reference the objects in the file. The solution to this spell required some extra data structures for maintaining indices that were not available in ALCHEMIST. Still, as the user is allowed to define his or her own procedures as well as including procedure calls as semantic actions, the problem could be solved.

The performance of the spells was in all cases acceptable. Without direct comparison with hand-made transformations, we still believe that our ALCHEMIST spells work with satisfactory speed. In all our test cases, spell execution took less than two minutes to perform while importing the diagrams into the FOUNDATION environment could take as much as five minutes. The biggest diagrams we used contained about 50 graphical objects; more objects tended to obscure the diagram and would not be sensible.

In the performance we noticed that at least in the VITAL bridge the transformation time was linear to the size of the source documents (Figure 6.5). As the spells were mainly concerned with local transformations,

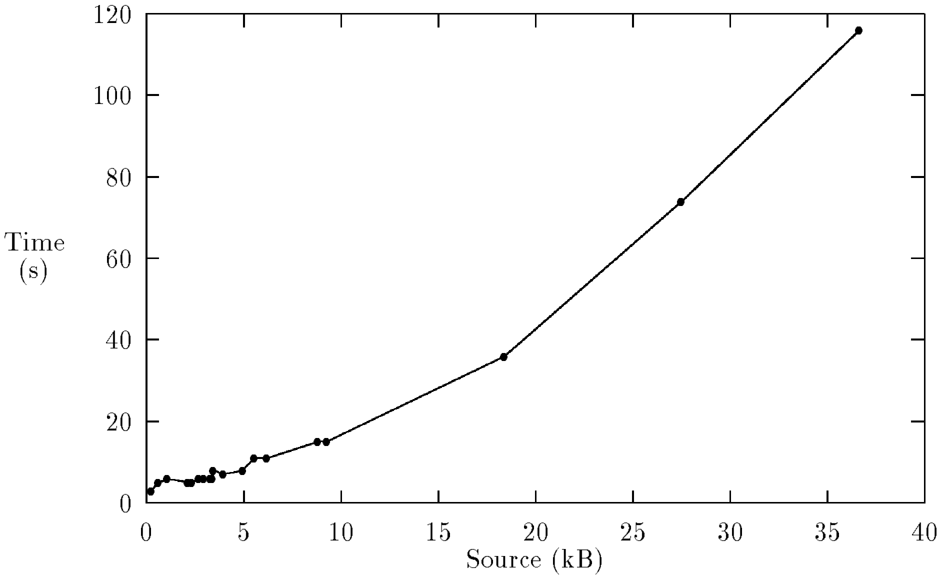


Figure 6.5: Effect of source file size to spell execution time, the TASK2DFD spell.

we seldom had to traverse the entire source document to produce a certain target object. This was not the case in all our test spells. Especially the ALTO2C++ spell execution time was proportional to the square of the source document size. This relation is due to a (too) simple mechanism for checking whether attributes have been defined before by traversing the entire source document for each attribute.

We run the test spells on a Sparc workstation. A typical source file of about 5 kB and about 20 graphical objects was transformed in about 10 seconds into a target file almost 10 times its original size. The biggest source files were about 10 kB containing about 50 graphical symbols. For testing reasons we also used some bigger source files of up to 40 kB but they contained so many objects that their usefulness was suspect (Figure 6.6).

6.4 TranSID applications

TranSID has been developed mainly during the past two years and we have not gained as much experience from its use as in ALCHEMIST's case. During its design and implementation TranSID has been tested in usual SGML

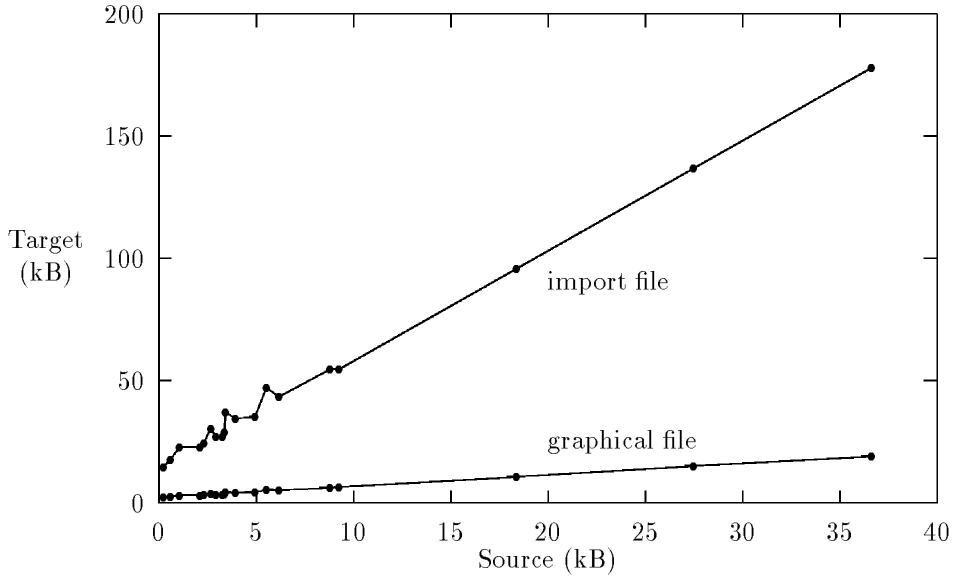


Figure 6.6: Effect of source file size to target file sizes, the TASK2DFD spell.

transformations such as the generation of LaTeX and HTML from SGML instances. We have also gained experience in the use of TranSID from a project implementing document assembly [AHH+96a, AHH+96b]. In document assembly, new documents are constructed from a pool of documents. TranSID is used to locate and streamline document fragments and to form a new SGML document.

A typical example is the TranSID reference manual, which was written in SGML and transformed both into HTML and LaTeX[1]. In Figure 6.7 we see the beginning of the reference manual in SGML. The DTD (not shown here) is very simple, containing only very basic elements corresponding fairly well to LaTeX commands. Also references, both backwards and forwards, have been coded in SGML.

In Figure 6.8 we see one of the rules in the TranSID program. It constructs a table of contents with links to the corresponding sections. This is quite a complicated rule which shows the power of the TranSID language. We shall give a short explanation of the rule. The main purpose of the rule is to construct the main page of the reference manual with a table of con-

[1]This transformation was designed and implemented by Jani Jaakkola in September 1996.

tents. Lines 6-7 of Figure 6.8 tell us that the element `TSDOC` is replaced with an `HTML` element. Lines 8-9 assign the document title to the local variable `maintitle` with the `set` operator, but the `null` operator at the end of the expressions prevents it from being copied to the result (yet). The variable is used later in the rule to include the document title in HTML `TITLE` and `H1` elements. Line 12 shows how TranSID may produce an element as a string, and line 13 how it produces an element as a structure node.

Further on in the rule, lines 20-22 make a list of `TITLE`s of `SECTION` elements. The list items function as links as well to the corresponding sections. Lines 23-28 perform the same transformation for subsection titles. Both section and subsection titles are preceded with their respective numbers computed by TranSID. Finally, after the table of contents, we have the main content of the document included by line 31.

The resulting HTML files when presented in Netscape is shown in Figure 6.9.

6.5 TranSID observations

Compared to other SGML transformers we have found TranSID both easy to use and efficient. We shall give some approximate numbers to help the reader understand how fast and efficient TranSID is. In the TranSID application presented in the previous section, the complete TranSID reference file in SGML is about 33 kB (1100 lines) and the corresponding DTD about 800 bytes (38 lines) (i.e., very small). The resulting set of HTML files is about 44 kB (1320 lines) and the TranSID script transforming the SGML instance is about 3.5 kB (175 lines). Part of the script is shown in Figure 6.8. The transformation takes about 3 seconds on a 133 MHz Pentium machine running Linux. The transformation uses about 14 400 nodes for the SGML trees and the peak memory use was about 1.1 MB.

The high use of memory is perhaps the main drawback of TranSID. Internal representations are constructed both for the source and the target, even though many nodes could be shared as they are not modified in the transformation.

6.6 Comparison between ALCHEMIST and TranSID

As we saw earlier, the strong points of ALCHEMIST were its generality, high level of abstraction, and spell reuse, as well as producing very maintainable transformations. TranSID is not in this sense as general as ALCHEMIST as it

```
<!DOCTYPE TSDOC SYSTEM "tsdoc.dtd"
-- TranSID reference manual, last update for V0.018 16.9.96 --
>
<TSDOC>
  <title>&tsid; reference manual</title>

  <SECTION>
    <title>General</TITLE>
    <ssect>
      <title>Running &tsid; programs </title>
      <para>&tsid; is invoked as follows</para>
      <code>
        Transid [options] [transformation program] [SGML files]
      </code>
      <para>The options include</para>
      <dlist>
        <d>-q</d><li>Do not output the target document</li>
        <d>-D [debug level]</d><li> Switch debugging level.
          Valid levels are 1-7 where level 7 produces globs
          of debugging output and level 1 produces output
          only when &tsid; panics.</li>
        <d>-L [debug section]</d><li> Debug a certain
          section of &tsid; program. Section may be a
          C-source file or a section marked with C-preprocessor
          macros.</li>
      </dlist>
    </ssect>

    <Ssect>
      <title>Syntax</title>
      <para>
      &tsid has C++ like comments. Comments include sections
      started with <B>/*</B> and ended with <B>*/</B> and
      sections started by <B>//</B> and ended with a newline.
      </para>
```

Figure 6.7: The beginning of the TranSID reference manual in SGML.

```
// One rule in a TranSID program that generates HTML frames
// for transid SGML documentation Version 0.018

transformation begin

element "TSDOC" becomes
  <"HTML"> {
    (current.origin.children.having(this.name=="TITLE")
      .children).set(maintitle).null,
    <"HEAD"> {<"TITLE"> { "TranSID documentation: ",
      maintitle }},
    "<BASE TARGET=\"kanveesi\">",
    <"BODY" "BGCOLOR"="bfdfbf"> {
      <"H1"> { maintitle }, <"H2"> { "Table of contents" },
      <"UL"> {
        current.origin.children
          .having(this.name=="SECTION").map(TRUE;
            (thisnum).set(num1).null,
            ("sectframe-",num1,".html#",num1).set(link).null,
            <"A" "HREF"=(link)> {<"LI"> {num1," ",
                 this.children.having(this.name=="TITLE").
                 children}},
            this.children.having(this.name=="SSECT").children
            .having(this.name=="TITLE").set(titles).null,
            <"UL"> {titles.map(TRUE;
                <"A" "HREF"=(link,".",thisnum)> {
                  <"LI"> {num1,".",thisnum," ",
                    this.children}})}
          )
      },
      current.children, <"HR">,
      <"I"> { "Automatically generated from ", "SGML source",
        " by ", <"A" "HREF"="./doc2frames.trs">
          {"doc2frames.trs script"}, "\n" }
    }
  };
end
```

Figure 6.8: One rule in the transcript converting SGML documentation into HTML with frames.

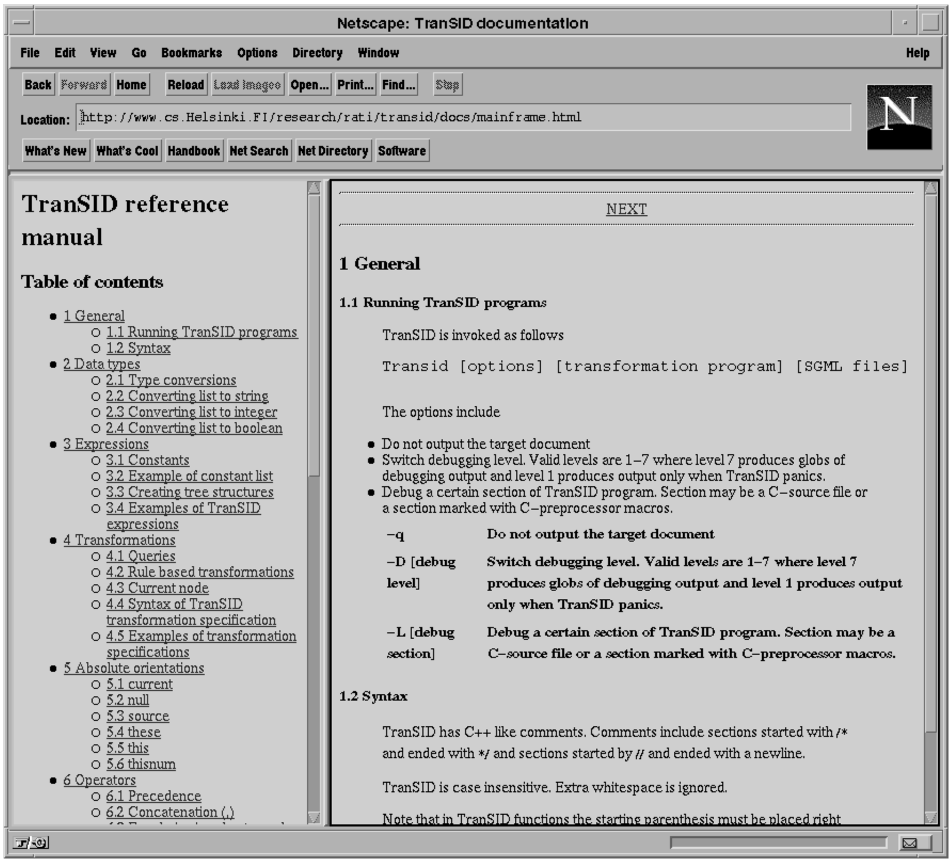


Figure 6.9: The TranSID reference manual transformed into HTML frames.

is mainly intended for SGML transformations. TranSID could be extended however to handle all kinds of structured documents (and their representation grammars) as the transformation mechanism in itself is based on tree transformation; the transformation is performed between the internal representations of the documents. Reading and writing SGML is just an additional feature of the system.

DTDs may be just as complicated or even more complicated than the ALCHEMIST grammars. However, often they are provided with the input and the user does not have to construct them himself. ALCHEMIST also requires the user to construct a target grammar. The ALCHEMIST transformation process itself guarantees that only targets that are syntactically correct are constructed. TranSID, however, does not require a target DTD.

Therefore the target may be syntactically incorrect compared to a DTD the user had in mind when he specified the transformation.

The transformation specification differs greatly in the systems. ALCHEMIST, based on TT-grammars, requires the user to explicitly specify which structures correspond to each other in the source and target representations. This leads to a somewhat tedious transformation specification in the cases when the modifications are minor; the user must also include structures that do not change. The default rule in ALCHEMIST is to remove all parts that are not included in the specification. In TranSID, the default rule is to copy all parts that are not included in the rules. Therefore, the user only specifies rules for document parts that are modified. This leads to simple programs for simple modifications, while complicated modifications can require complicated programs.

TranSID is also more suitable for global transformations where document parts may depend on any other part in the document. This is due to the fact that both the source tree and part of the target tree are accessible during the entire transformation. As we also saw in the example transformations of ALCHEMIST, TranSID is, of course, more suitable for SGML transformations, especially when more complicated features of the SGML standards are used.

Chapter 7

Related work

Document transformations have mostly been solved with tailored transformations for two particular representations. This has led to a huge amount of small transformation modules that solve one particular problem but that are unsuitable for other problems. Not very many transformation generators that could be used to solve general transformations have been built. In this chapter we take a look at some transformation generators and tree transformation systems that are suitable for building transformations between structured documents. For extensive, if somewhat outdated bibliographies on the manipulation of structured documents, we refer to [FSS82, And86, vVW86, Fur92] and [KN94].

We concentrate on systems based on two grammars, a source grammar and target grammar, where the user is actually required to define both the source and target representations.

Multiple view editors are typical applications for transformations of structured documents. A multiple-view editor is able to show at least two different views of a document. For example, it may show a textual version and formatted version. Depending on the system, the user may be allowed to modify only one particular view, or he/she may be allowed to make updates in any document view. A typical feature in these systems is that all other views are updated either automatically or on demand, when one view is modified.

We start by presenting more thoroughly a multiple view editor called HST based on syntax-directed translation schemas, and a transformation generator called ICA that is used especially for SGML document transformations. We continue with an overview of several other systems based on two grammars, both from the fields of structured documents and compiler generation. We also present some SGML transformation languages and other multiple-view editors. Finally, we give a summary of the different transformation systems.

7.1 A multiple-view editor

The Helsinki Structured Text Database System (HST) [KLMN90] is an environment for reading, writing, and querying structured documents. The system provides multiple views of a document in a graphical interface. The *logical document* is described through a context-free grammar. The user needs at least one *view* to be able to read and/or modify a logical document. A view is described through an *annotated grammar*, where the user may modify the logical grammar according to the rules of a syntax-directed translation schema: he/she may remove or add terminals, and reorder the nonterminals. The user may also remove or add nonterminals.

Some documents are easier to write and modify in a structured view, while others best benefit from a simple textual view. The HST system lets the user make modifications in any view; the modifications are automatically updated in the other open views. Updates are performed through syntax-directed translation from the modified view to the logical document, and from there to all other open views. Therefore, a view definition describes not only view computation from the logical document to a view, but also the *inverse transformation* [NM89, Nik90] of the view to the logical document.

The modification of a view leads to quite an extensive process of updates in the system (Figure 7.1). The modified view is first parsed. Then the view parse tree is inverted into the logical document, i.e., it is transformed via the view definition back to the logical document. Other opened views are updated from the logical document through their view definitions and the frontier of the new view trees are shown as the updated views. This process has also been incrementalized in HST. In such a process only modified parts of a view are parsed and translated [Lin93].

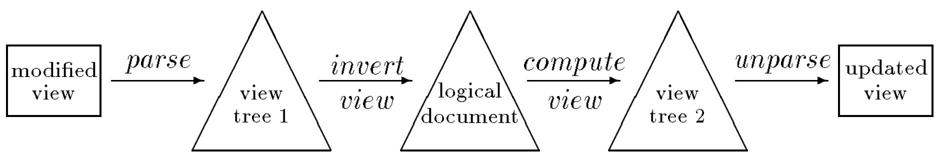


Figure 7.1: Modifications in a view lead to an update of the logical document and other open views in the HST system.

HST is a typical example of a multiple-view editor, where the user may modify any open view. HST is, however, able to show only textual views of a document, not a pretty-printed formatted version. The main strength of the system lies in the simpleness of the syntax-directed translation schemas. The user only needs to define one view, and the logical document is automatically transformed into the view and *vice versa*.

The main difference to ALCHEMIST is that HST is based on SDTSs while ALCHEMIST is based on TT-grammars. Therefore, the source and target grammars in HST are variations of the same grammar, with the same nonterminals, possibly in different order. ALCHEMIST, on the other hand, handles arbitrary different grammars. The main advantage with HST is, that it also defines the inverse transformations of a transformation. ALCHEMIST produces only one-way transformations, even if the inverse transformation may be defined by swapping the source and target grammars.

7.2 A structured document transformation generator

The Integrated Chameleon Architecture (ICA) [MKNS89, MBO93, MOB94] is a transformation generator that consists of several tools for building transformations. ICA relies on the definition of an intermediate representation that always lies between the source and target representations. The user defines only one grammar for the intermediate representation; the source and target representations are described by reordering the nonterminals in this grammar. Therefore, the transformation specification is very application dependent as all representations must be described by very similar grammars.

An ICA transformation is divided into several subtransformations (Figure 7.2). The user may also have to preprocess the source document. All internal representations in ICA are based on SGML. In order to be able to parse the source document, the user has to insert SGML tags, and sometimes replace other tags so that he/she achieves a fully braced document, i.e., all logical documents parts are marked with a start tag and an end tag. This process called retagging is supported by a special tagging tool. In this phase, however, it may well be that the user already solves several mapping problems. After the document has been retagged, it is translated into an intermediate representation according to the intermediate grammar, thereafter the intermediate representation is translated to an SGML document corresponding to the target, and finally the target SGML document is output as the target document with additional modifications to remove the SGML tags. Mapping the general intermediate document to the target document is automated so that the user never sees the target SGML document.

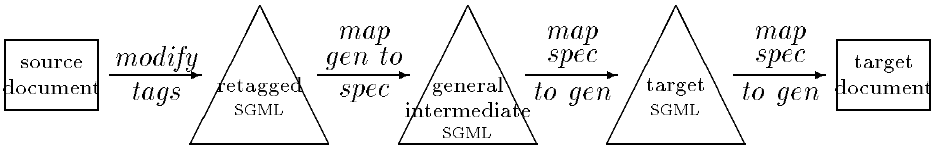


Figure 7.2: Transformation process of an ICA transformation.

As Figure 7.2 shows, the ICA transformation process is very similar to the one of HST. The intermediate document corresponds to the logical document in HST, while the specific SGML documents correspond to views. As a matter of fact, ICA may be considered to be based on SDTSs as well. By using an intermediate representation, ICA reduces the number of transformations needed to fully interface a set of representations. When including a new representation, the user only needs to define two transformations, one to the intermediate representation and one from it to the specific representation. Thereby he/she can transform from the new representation to any other representation in the set.

The main difference to ALCHEMIST is again due to the different underlying transformation techniques. ICA is based on SDTSs and requires the user to define an intermediate representation. ALCHEMIST allows arbitrary grammars. The user may define an intermediate representation with ALCHEMIST as well and thereby achieve the apparent advantage of ICA. ICA uses SGML for all internal representations of a document and saves them in temporary files during the transformation. ALCHEMIST relies on the parse trees which are kept in main memory only. ALCHEMIST could, however, be enhanced with the possibility of writing and reading the parse trees in SGML format.

7.3 Other two-grammar systems

We have seen examples of two-grammar systems above, systems that are either targeted at document preparation or document transformation. In this chapter we present some further systems that are based on a source and a grammar. Here, we do not, however, try to categorize the systems; many of them could well be both document preparation systems and document transformation systems. For another description of document transformation systems based on two grammars, see [KP95, Kui96].

The Syntax and Semantics Analysis and Generation System (SSAGS) [PKP+82, Pay88] is based on TT-grammars just as ALCHEMIST. SSAGS implements two subsets of TT-grammars. A *dual grammar translation scheme* (DGTS) [KPPM84] restricts source subgrammars to single productions, where left hand side symbols may be associated only with left hand side symbols in the target subgrammar within the same production group association. A DGTS corresponds to a syntax-directed translation scheme. Somewhat more general is the *single input production – explicitly qualified* (SIPEQ) TT-grammar [KPPM84]. The SIPEQ TT-grammar is also restricted to single production source subgrammars, but symbol associations may be established between any source and target symbols. A SIPEQ TT-grammar corresponds to ordered attribute grammars [Kas80]. The implementation of the SIPEQ TT-grammar also includes a simple case statement for choosing between target subgrammars, a copy instruction for multiplying target subtrees, and pseudoproductions for simplifying symbol associations. SSAGS has been used, among other things, in implementing an interface between the programming languages Ada and DIANA [PKPM83].

Chiba and Kyojima [CK95] use syntax-directed tree translation to perform structured document transformations. They encode trees into strings and then perform syntax-directed translation on the strings. This approach is more powerful than SDTSs because it permits suppression and insertion of tree levels, e.g., a new level of nodes may be introduced at an intermediate node level in the parse tree. The syntax-directed tree translation technique does not, however, support transformations dependent on the contents.

The Turing Extender Language (TXL) [CHP88b, Cor90] has been designed for providing extensions to existing programming languages. TXL transforms programs in a language into dialects of the language where new language features have been inserted or a different notation is used. A TXL transformation consists of three submodules. The parser is based on the base language grammar, but takes notion also of the differing target language features, the transformer transforms a parse tree according to some semantic rules, and the deparser writes out the target program. The transformation is done using a general purpose tree pattern matching algorithm. In short, the transformer generates a parse tree over the base language from the dialect language parse tree. In order to maintain the structural integrity of the parse tree, the replacement subtree is reparsed before being added to the main tree.

SIMON [FW93] is a system for restructuring documents that uses an intermediate representation in the transformation. SIMON requires a source grammar, a target grammar and a higher-order attribute grammar (HAG). SIMON uses an extra pair of trees for describing the source and target parse trees in canonical form. The HAG is used for describing transformations between the parse trees and these canonical trees called a basic tree and a consistent tree, respectively. The actual transformation is performed through attribute evaluation in the basic tree giving as a result an evaluated consistent tree. The transformation process is thereby augmented with two additional phases, transformation from the source tree to a basic tree, and transformation from the consistent tree to the result tree. The HAG is specified manually.

The Grif environment [QV86, QVB86a, QVB86b, FQA88] is an interactive system for editing structured documents. It is a structure-oriented editor which guides the user in accordance with the structure of the document. The user defines a structure schema that corresponds to the generic logical structure of a document. A view of the document is defined as a presentation schema, where the user describes the conversion rules that transform the document into a view. The transformation can both remove certain parts from a document and reorder elements in the document. A modification in a view propagates to other views. Grif recognizes the constraints between the modified part and other views, and updates can be done incrementally [QV87].

The Syndoc system [KP91] is based on SDTSs. The user may insert formatting details into a logical document. The user may add or delete terminals and reorder nonterminals. In an extended version of the system [KP93, Kui96], the user may also add or delete nonterminals as well as rename nonterminals through simple semantic actions.

The **pedtnt** system [Fur86, Fur87a, FQA88] is a testbed for the presentation and manipulation of structured documents. The system is based on context-free grammars and allows the user to define transformations between different presentations [FS88]. The documents are described through a generic logical grammar. The transformation method lets the user alter the grammar by defining a set of grammar modification rules to alter the productions. The system also requires the user to specify how the transformation between the documents of the two grammars are performed. In an extended version, the system has been augmented with attributes to give the logical grammar a flair of attribute grammars [Fur87b].

The Scrimshaw language [Arn93] lets the user define simple queries and transformations on a structured document. The rules consist of a matching part and a construction part. The transformation process matches some substructures in the parse tree, assigns some of the structure to variables, which then are used in the output rules that describe how the matched pattern is replaced. This language is more suited for simple transformations as the complete grammar of the structure is always given in one rule.

7.4 Other transformation systems

In the field of SGML, quite a few transformation languages have been designed. Many of these languages have been designed as back-ends to SGML parsers. An SGML parser parses an SGML document according to the corresponding document type definition (DTD). Often, the parser does not construct a parse tree, but returns only the ESIS output [Gol90, Appendix B, Annex G], a list of tokens in the source like the start and end tags, or data elements. Languages that are based on such parsers, like OmniMark [Exo93] and CoST [Har93], work as syntax-directed translators. The user may add actions to be taken at any token but he/she may usually not refer (without difficulty) to any other part in the source. Especially, it is difficult to make references to yet unparsed document parts. The Metamorphosis system [MID95] instead, builds the parse tree of the SGML document. The user specifies how each node in the parse tree should be modified and is allowed some more extensive references to the tree. Also Balise [Ber96] provides tree based transformations as an option. The user may choose between an event-driven or a tree-based approach. He/She must, however, explicitly state when he/she want the transformation to construct an internal parse (sub)tree of the source. None of these languages use a target grammar or support correct target syntax. If, however, the target is also an SGML document instance, the user may validate the instance against either the source instance DTD (if the changes have been minor), or an explicit target DTD that the user has constructed separately from the transformation.

Multiple-view editors provide several views of an underlying document. When the user modifies one view, the other open views are updated, often through some syntax-directed translation technique. Some systems also concentrate on *dynamic transformations*, where the target structure is not known before transformation time. This problem arises in syntax-directed editors when the user decides to move a document part to another place within the document. If the structure of the part is not allowed in the new place, the part must be transformed dynamically to fit in.

Janus [CKS$^+$81, CBG$^+$82] was one of the first two-view text processing system. It provides the user with two views of a document on *two different screens*. Other multiple-view editors are the VORTEX [CCH86, Che88, CH88, CHM88] document preparation system showing both textual and formatted versions of TEX documents [Knu87], and Lilac [Bro88, Bro91]. The Sam system [Tri81] was one of the first two-view editors for graphical pictures. It combines graphics and a layout language; the user can edit a picture in two views. Other two-view editors for graphical pictures are Juno [Nel85] and Tweedle [Ase87]. Quill [CHL+88, CHP88a, Cha88, Lun88, Cha90] supports full integration of various sorts of graphical editing together with text editing. Multiple views have also been implemented in program development environments, two of them being PECAN environment [Rei83] and the Synthesizer Generator [RT89].

Editing structured documents require dynamic translations of document parts. When the user moves or copies a document part to another place, the part must be transformed according to the structure of the target position. For example, Cole and Brown [CB90, CB92] have studied this problem and recognized several problems like validating the structured document during creation and editing, dealing with incomplete and temporarily incorrect structures, and identifying allowable structure edits. Akpotsui, Quint, and Roisin [AQ92, AQR93, Akp93] have developed some solutions to this problem by identifying the kind of transformations needed in structured editing. Note that the source and the target grammars are in this case the same grammar. Dynamic transformations are needed when a document part is transformed to satisfy a different subgrammar of the document grammar.

There are several other tree transformation systems based on single grammars (see, e.g., [Gra92, LMW88, LMW91, Hec88]). However powerful these systems are, they can, of course, not support the correct target syntax.

7.5 Summary of related systems

Some of the most important features of structured document transformation systems based on two grammars have been collected in Table 7.1. We have divided the table into three sections. The first section lists formal transformation techniques and ranges from the most simple one (or least powerful) simple SDTSs to more powerful ones like TT-grammars and attribute grammars. The second section lists particular transformation systems and the third section some SGML transformation systems. The systems are listed in alphabetical order, preceded by ALCHEMIST and TranSID, respectively.

Most listed techniques and systems require both a source grammar and a target grammar. Systems requiring a source grammar are marked with a bullet in the first column (SG). In some cases, the system does not require a target grammar, but the user may specify one and use it as a support when specifying the transformation. In the second column (TG) we mark those systems that require a target grammar with a bullet and those that can use an optional target grammar with a circle.

The third column (MAP) denotes the mapping formalism. The mapping is usually based on a formal technique such as syntax-directed translation (SDT), attribute grammars (AG), or TT-grammars (TT). In some cases, the mapping may rely on simple tree pattern matching and replacement (patt). In the case of SGML transformers, they are either mainly event-based (event) or tree-based (tree). See Section 3 for a presentation of the different techniques.

Even if the system requires the user to define a target grammar, the user may have to explicitly define what operations transform a source instance into a target instance over the target grammar. In this case, we do not consider the system to support correct target syntax as it is up to the user to specify the transformation steps. In the fourth column (TC) we mark those systems supporting correct target representations with a bullet.

Given a source grammar and a target grammar, some translation schemata may be constructed automatically (auto) or semiautomatically with some user interaction (semi). Others must be constructed manually (man). This feature is marked in the fifth column (Gen).

In the sixth column (D/S) we denote if the systems support dynamic or static transformations. If the target representation is chosen run-time, e.g., in a structured editor when one object is moved to a new spot, the transformation is considered dynamic (D). Then there can be an arbitrary number of target grammars. Transformations where there is a finite previously defined set of target grammars are considered static (S).

Possible modifications in the source parse trees are denoted in the columns labeled *Modifications: A/D, R, T*. Column 7 (A/D) describes systems that allow additions and deletions of nonterminals between the source and target grammars, column 8 (R) denotes reordering of nonterminals in the grammars, and column 9 (T) denotes addition and deletion of tree levels in the transformation. Adding or deleting nonterminals help the user form more simple or complicated views of a document, e.g., a stand-alone table of contents. Reordering nonterminals allows transformations where the document parts are reordered. Finally, addition and deletion of tree levels profoundly modifies the target parse tree and also lets the user specify more complicated tree patterns to be matched against in the source tree.

System	SG	TG	MAP	TC	GEN	D/S
Transformation techniques						
simp SDTS	•	•	SDT	•	semi	D/S
SDTS	•	•	SDT	•	semi	D/S
ESDTS	•	•	SDT	–	semi	D/S
pred SDTS	•	•	AG	–	?	D/S
SSDT	•	•	AG	–	semi	D/S
PSSDT	•	•	AG	–	semi	D/S
GSDT	•	•	AG	–	semi	D/S
AG	•	∘	AG	–	semi	D/S
ACG	•	∘	ACG	•?	auto	S
TT-grammar	•	•	TT	•	man	S
General transformers						
ALCHEMIST	•	•	TT	•	man	S
DGTS	•	•	SDT	•	semi	S
Grif	•	∘	AG?	–	auto	D/S
HST	•	•	SDT	•	semi	S
ICA	•	•	SDT	•	semi	S
pedtnt	•	•	?	–	man	S
Scrimshaw	•	∘	patt	–	man	S
SDTT	•	•	SDT	•	man	S
SIMON	•	•	AG	–	?	S
SIPEQ	•	•	TT*	•	semi	S
Syndoc	•	•	ESDTS	•	man	S
T-gen	•	–	patt	–	man	S
TXL	•	•	patt	•*		S
SGML transformers						
TranSID	•	–	tree	–	man	S
Balise	•	–	tree	–	man	S
CoST	•	–	event	–	man	S
MetaMorphosis	•	–	tree	–	man	S
OmniMark	•	–	event	–	man	S

• — yes, – — no, ? — unknown, * — restricted/simplified
∘ — optional

SG	Source grammar	GEN	Generation of transformation: automatic, manual, or semiautomatic
TG	Target grammar		
MAP	Mapping formalism		
TC	Target correctness	D/S	Dynamic vs. Static transformation

Table 7.1: Properties of some syntax-directed transformation systems and techniques.

System	Modifications A/D	R	T	ID	SA	Useful reference
Transformation techniques						
simp SDTS	–	–	–	•	–	[AU72]
SDTS	•*	•	–	•	–	[AU72]
ESDTS	•	•	–	•	•*	[KP93]
pred SDTS	•	•	•	•	•	[PB78]
SSDT	•	•	•	•	•	[Shi84]
PSSDT	•	•	•	•	•	[Shi84]
GSDT	•	•	•	•	•	[AU71]
AG	•	•	•	•	•	[DJL88]
ACG	•	•	?	•	–	[GG84]
TT-grammar	•	•	•	•	•	[KPPM84]
General transformers						
ALCHEMIST	•	•	•	•	•	[LTV96]
DGTS	•	•	–	•?	–	[KPPM84]
Grif	•	•	?	•	•	[QV86]
HST	•	•	–	•	–	[KLMN90]
ICA	•	•	–	•	–	[MBO93]
pedtnt	•	•	•	•	•	[FS88]
Scrimshaw	•	•	•	•	•	[Arn93]
SDTT	•	•	•	–	–	[CK95]
SIMON	•	•	•	•	•	[FW93]
SIPEQ	•	•	–	•?	–	[KPPM84]
Syndoc	•	•	–	•	•*	[KP93]
T-gen	•	•	–	•	•	[Gra91]
TXL	•	•	•	•	•	[CHP88b]
SGML transformers						
TranSID	•	•	•	•	•	[JKL96a]
Balise	•	•	•	•	•	[Ber96]
CoST	•	•	•	•	•	[Har93]
MetaMorphosis	•	•	•	•	•	[MID95]
OmniMark	•	•	•	•	•	[Exo93]

• — yes, – — no, ? — unknown, * — restricted/simplified

AD	Addition/deletion of nonterminals	T	Addition/deletion of tree levels
R	Reordering of nonterminals	ID	Identifier mapping
		SA	semantic actions

Table 7.1: Continued.

Most systems copy identifier-like tokens from the source side to the target side. This a natural feature of the transformation. The user does not only expect the parse trees to be modified, he/she also wants the frontier of the source tree to be copied, perhaps with modifications, to the target tree. Systems allowing this feature are denoted in column 10 (ID). Identifier copying can also be performed with semantic actions denoted in column 11 (SA). Semantic actions are also used for other transformation computing, symbol table checking, etc.

The last column of the table lists the main reference to the technique or the system. In the case of transformation techniques, we have usually listed a good introduction to the subject, not perhaps the first reference. For such references, we refer to Chapter 3. In the table, we have listed also some techniques that have not been further explained in Chapter 3. These include predicate syntax-directed translation schemas (pred SDTS), semantic syntax-directed translation (SSDT), programmed semantic syntax-directed translation (PSSDT), generalized syntax-directed translation (GSDT)), and attribute coupled grammars (ACG). They are all extended techniques of syntax-directed translation or attribute grammars. For other references to the transformation systems, we refer to Sections 7.1–7.4.

The transformation techniques based on syntax-directed translation schemas are all fairly similar. They differ mainly in the allowed modifications. The techniques based on attribute grammars are more powerful, but even if they allow the definition of a target grammar, they do not guarantee that the target syntax is correct as do the SDTSs and the TT-grammar technique.

The general transformation systems are a more heterogeneous group. Some of them require a target grammar that is based on the source grammar. ALCHEMIST is here the only exception allowing unrelated source and target grammars. They also differ in the allowed operations; in this sense we consider ALCHEMIST to be the strongest system, while the others only contain a part of the operations. Choosing a suitable system for a transformation is, though, highly dependent on the transformation. We hope that this part of the table will help the user in finding an appropriate system.

The SGML transformation systems are clearly divided into two groups: event-based systems and tree-based systems. Otherwise the systems are fairly similar. The tree-based systems TranSID and Balise differ in the transformation evaluation order. The tree-based transformation of Balise is performed top-down while TranSID performs it bottom-up.

Chapter 8

Conclusion

Transformations of structured documents can be implemented in several ways. We believe that requiring the user to specify both the input and the output supports the construction of correct transformation modules. We have presented different syntax-directed techniques that are suitable for document transformation. In the field of compiler generation, we find the theory and techniques that can be used in transformation implementation. Syntax-directed translation schemas are a simple technique that requires the user to describe his document representations through similar grammars. The power of the transformations is limited to adding and removing terminals, and reordering of the document parts. In an extended version of syntax-directed translation schemas we also have the possibility of removing document parts or including new document fragments. Attribute grammars can be used to obtain a more general transformation technique. The transformation still requires a source grammar to ensure that the source document is correct, but often the back-end of these transformations is open. The user may include any output instructions and the technique does not assure that the output follows a certain grammar.

A transformation technique based on TT-grammars is more general than the usual syntax-directed translation schemas. A TT-grammar requires the user to specify both a source grammar and target grammar, describing the source and target documents, respectively. The grammars, however, can be widely different. The user therefore explicitly describes in a mapping specification how the grammars relate to each other. Despite the manual effort, the technique gives a simple way of defining a transformation between two arbitrary representations. It also ensures that the produced output is syntactically correct with respect to the target grammar.

Our contribution has been to extend TT-grammars and, based on these extensions to implement a general transformation generator called AL-

CHEMIST. As an antipole to TT-grammars, we have also designed and implemented an SGML transformer called TranSID.

In the case of ALCHEMIST, we have described a transformation algorithm based on TT-grammars. Our extensions to the technique include the possibility of using semantic actions in the mapping specification, identifier copying from the source to the target side, and the possibility to let the user interact in the transformation. The transformation generator ALCHEMIST implements all these extensions. ALCHEMIST has a graphical user interface for specifying and generating transformation modules. Especially, the user may specify the TT-grammar mappings with a point and click user interface. ALCHEMIST generates transformation code on demand that is compiled into executable transformations called spells. All code is saved in files that the user may inspect and modify for specifically tailored transformations. Both ALCHEMIST and its spells are fully operational on a UNIX platform.

ALCHEMIST has been designed and implemented within a large software project. The generator has been extensively tested and used to build an interface between a KBS development environment and a commercial CASE tool. Experience has shown the usefulness of ALCHEMIST, especially the importance of its high level of abstractness. ALCHEMIST has been used also by other partners in the software project. Transformations are specified moderately fast if only the underlying representations have been described exactly. Performance of both ALCHEMIST and its spells have been found acceptable. In all implemented cases, spells have been found fast enough to satisfy the user's need.

We have experienced only some minor shortcomings. First, the TT-grammar technique requires a moderate understanding of grammars, parse trees, and parsing techniques. On the other hand, so do all other syntax-directed translation techniques. Second, building a grammar may be a very tedious task if an exact description of the representation is not available. Again this applies to all syntax-directed techniques. Third, ALCHEMIST is also not suitable for all kinds of transformations, especially transformations that require a lot of global computation, since they are not easily described with TT-grammars. Again, global computation may be included in semantic actions described by using programs written in C++.

We have reviewed several related transformation systems that are based on syntax-directed translation schemas. These systems guarantee that the output of the generated transformation is syntactically correct, but often the system has limited transformational power. On the other, more general systems based on attribute grammars, allow arbitrary output which, of course, cannot be guaranteed to be correct. In ALCHEMIST, we combine target correctness with a more general approach to specifying transformations. In this way, ALCHEMIST combines the best features of other techniques.

ALCHEMIST could be further enhanced in several ways. One extension would be to allow *semistructural transformations*. A semistructural transformation takes the document structure into account but does not require full specification of source and target representations. The user specifies only document parts that should be transformed; the rest of the document is copied to the target side as default. A semistructural technique is actually a tree pattern matching and replacement technique more than a syntax-directed technique. By specifying patterns with subgrammars (as in ALCHEMIST) we could achieve a more general technique.

Incremental transformations would be another useful extension. When a document is retransformed, the transformation would only need to make updates in an old target document corresponding to the modified parts in the source document. This would improve the performance of the transformation and reduce execution time. Incremental updates, however, require extensive data structures and procedures for maintaining data and pointers to modified and updated parts, and they have been omitted from the current version of ALCHEMIST. In order to fully take advantage of incrementality, all transformation phases in a spell process should be incrementalized. An incremental solution includes not only incremental parsing, translation, and unparsing, but also incremental pre- and postprocessing [Lin93].

The SGML interface of ALCHEMIST is currently rather limited. The user has to convert manually SGML DTDs into ALCHEMIST source grammars to be able to perform SGML transformations. In an extended version, ALCHEMIST could perform this conversion by itself. Another solution would be to have ALCHEMIST read SGML DTDs directly. Direct understanding of DTDs would require more substantial modifications to ALCHEMIST, as ALCHEMIST is based on LR parsing while DTDs are more suitable for top-down parsing [BK94]. On the other hand, it would be simple to extend ALCHEMIST to read and to write its internal documents as SGML documents, just as is done in the ICA transformation generator. The user would receive a handy intermediate representation, easily portable to any SGML system. At the same time, the user would be provided with persistent user readable representations of the internal structure. Of course, an ALCHEMIST spell may produce such a representation as its target document, as well.

ALCHEMIST is now based on LR parsing. A useful extension would include LL parsing as well with grammar extensions such as iterations instead of recursive productions. Many available representation definitions have been given in an LL grammar like fashion, and they would then be more easily adopted in an ALCHEMIST specification. Extending ALCHEMIST with LL parsing would, however, require extensive modifications to the TT-grammar technique.

Sometimes it would be useful to be able to *trace* target document parts to their corresponding source document parts. In a complex transformation, especially in tool representation transformations, it would be convenient to see the link between a target object and a source object. Including a tracing technique in the spell would also help in building and debugging the transformation. Another useful feature would be *two-way transformations*. ALCHEMIST generates only one-way transformations, but in some cases it would be possible to automatically generate the inverse transformation. We have made preliminary plans to include both tracing and inverse transformations in ALCHEMIST.

All suggested extensions are under consideration for future work on ALCHEMIST. ALCHEMIST is at the moment fully operational but still more of a prototype than a commercial product. ALCHEMIST would also need to be evaluated more thoroughly.

The TranSID language is mainly intended for SGML transformations. The language has been designed with the help of real world problems provided by commercial partners. Also TranSID has been implemented in a research and development project. Performance of TranSID is acceptable even if high main memory use is a significant drawback.

TranSID does not require as high expertise as ALCHEMIST in specifying the transformations. However, basic knowledge of SGML is required as well as some understanding of the evaluation order or TranSID transformations. Usually, and fortunately, DTDs are readily provided with SGML documents, which makes the task for the user a bit easier.

TranSID requires the user to specify only parts of the documents that are to be modified. This makes small transformations simple to specify. On the other hand, TranSID does not use a target grammar or DTD and therefore does not check that the target is syntactically correct. Validation must be performed with an external SGML parser and possibly a user-constructed new target DTD.

Extensions to TranSID in the future contain an optimization of main memory usage, and global indexing of sentences and words.

An ideal transformation system would combine the best features of both the ALCHEMIST and TranSID systems. For example, it should compute the mapping automatically based on only the source and target specifications. It is an open problem, though, to what extent a mapping can be computed from two context-free grammars (or DTDs). The ideal system would also let the user reference any part of the source documents in the transformation specification. Additionally such a system would keep the transformation specification work to a minimum; only the parts that change should require specification.

References

[ACM84] ACM. *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction, SIGPLAN Notices* 19(6)*, Montreal, Canada*, New York, 1984. ACM.

[ACM88] ACM. *Proceedings of the ACM Conference on Document Processing Systems, Santa Fe, New Mexico*, New York, 1988. ACM.

[AFQ89a] J. André, R. Furuta, and V. Quint. By way of an introduction. Structured documents: What and why? In André et al. [AFQ89b], pages 1–6.

[AFQ89b] J. André, R. Furuta, and V. Quint, editors. *Structured documents.* The Cambridge Series on Electronic Publishing. Cambridge University Press, Cambridge, 1989.

[AHH+96a] H. Ahonen, B. Heikkinen, O. Heinonen, J. Jaakkola, P. Kilpeläinen, G. Lindén, and H. Mannila. Intelligent assembly of structured documents. Report C–1996–40, Department of Computer Science, University of Helsinki, Finland, 1996.

[AHH+96b] H. Ahonen, B. Heikkinen, O. Heinonen, J. Jaakkola, P. Kilpeläinen, G. Lindén, and H. Mannila. Constructing tailored SGML documents. In J. Saarela, editor, *Proceedings of SGML Finland 1996*, pages 106–116, Helsinki, 1996. SGML Users' Group Finland.

[Akp93] E. K. A. Akpotsui. *Transformations de types dans les systémes d'édition de documents structurés.* PhD thesis, L'Institut National Polytechnique de Grenoble, France, 1993.

[And86] J. André. Manipulation de documents: bibliographie. *T.S.I. – Techniques et Sciences Informatiques*, 5(4):363–365, July – August 1986.

[And93a] Andersen Consulting. *FOUNDATION Application Development, Version 2.0*, 1993.

[And93b] Andersen Consulting. *FOUNDATION Design, Analyze Application Requirements, Version 2.0*, 1993.

[AQ92] E. K. A. Akpotsui and V. Quint. Type transformation in structured editing systems. In C. Vanoirbeek and G. Coray, editors, *EP92 — Proceedings of Electronic Publishing, '92, International Conference on Electronic Publishing, Document Manipulation, and Typography, Swiss Federal Institute of Technology, Lausanne, Switzerland*, The Cambridge Series on Electronic Publishing, pages 27–41, Cambridge, 1992. Cambridge University Press.

[AQR93] E. K. A. Akpotsui, V. Quint, and C. Roisin. Type modelling for document transformation in structured editing systems. Technical report, INRIA, France, 1993.

[Arn93] D. S. Arnon. Scrimshaw: A language for document queries and transformations. In Hüser et al. [HMQ93], pages 385–396.

[Ase87] P. J. Asente. *Editing Graphical Objects Using Procedural Representations.* PhD thesis, Technical report No. CSL-TR-87-343, Computer Systems Laboratory, Stanford University, USA, 1987.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers. Principles, Techniques and Tools.* Addison-Wesley, Reading, 1986.

[AU71] A. V. Aho and J. D. Ullman. Translations of context-free grammars. *Information and Control*, 19:439–475, 1971.

[AU72] A. V. Aho and J. D. Ullman. *The theory of parsing, translation and compiling*, Volume I: Parsing. Prentice-Hall, Englewood Cliffs, 1972.

[Bak78] B. S. Baker. Generalized syntax directed translation, tree transducers, and linear space. *SIAM Journal of Computing*, 7(3):376–391, 1978.

[BBT92] G. E. Blake, T. Bray, and F. W. Tompa. Shortening the OED: Experience with a grammar-defined database. *ACM Transactions on Information Systems*, 10(3):213–232, 1992.

[Ber96] Berger-Levrault/AIS. *Balise Reference Manual, Release 3*, 1996.

[BF61] M. P. Barnett and R. P. Futrelle. Syntactic analysis by digital computer. *Communications of the ACM*, 5(10):515–526, 1961.

[BK94] A. Brüggemann-Klein. Compiler-construction tools and techniques for SGML parsers: Difficulties and solutions. To appear in *Electronic Publishing – Origination, Dissemination and Design*. Available from URL: `ftp://ftp.informatik.uni-freiburg.de/documents/reports/.index.html`, 1994.

[BR84] F. Bancilhon and P. Richard. Managing texts and facts in a mixed database environment. In G. Gardarin and E. Gelenbe, editors, *New Applications of Data Bases*, pages 87–107. Academic Press, 1984.

[Bro88] K. P. Brooks. A two-view document editor with user-definable document structure. Technical Report No. 33, Digital Systems Research Center, USA, 1988.

[Bro91] K. P. Brooks. Lilac: A two-view document editor. *IEEE Computer*, 24(6):7–19, 1991.

[BSM96] T. Bray and C. M. Speerberg-McQueen. Extensible Markup Language (XML). URL: `http://www.w3.org/ pub/WWW/TR/WD-xml-961114.html`, 1996. Draft.

[CB90] F. Cole and H. Brown. Editing structured documents with classes. Technical Report No. 73, Computing Laboratory, University of Kent at Canterbury, UK, 1990.

[CB92] F. Cole and H. Brown. Editing structured documents — problems and solutions. *Electronic Publishing – Origination, Dissemination and Design*, 5(4):209–216, 1992.

[CBG+82] D. D. Chamberlin, O. P. Bertrand, M. J. Goodfellow, J. C. King, D. R. Slutz, S. J. P. Todd, and B. W. Wade. JANUS: An interactive document formatter based on declarative tags. *IBM Systems Journal*, 21(3):250–271, 1982.

[CCH86] P. Chen, J. Coker, and M. A. Harrison. The VORTEX document preparation environment. In Désarménien [Dés86], pages 45–55.

[CH88] P. Chen and M. A. Harrison. Multiple representation document development. *IEEE Computer*, 21(1):15–31, 1988.

[Cha88] D. D. Chamberlin. An adaptation of dataflow methods for WYSIWYG document processing. In *Proceedings of the ACM Conference on Document Processing Systems, Santa Fe, New Mexico* [ACM88], pages 101–109.

[Cha90] D. D. Chamberlin. Managing properties in a system of cooperating editors. In Furuta [Fur90], pages 31–46.

[Che88] P. Chen. *A Multiple-representation Paradigm for Document Development.* PhD thesis, Report No. UCB/CSD 88/436, Computer Science Division, University of California, Berkeley, USA, 1988.

[CHL+88] D. D. Chamberlin, H. F. Hasselmeier, A. W. Luniewski, D. P. Paris, B. W. Wade, and M. L. Zolliker. Quill: An extensible system for editing documents of mixed type. In *Proceedings of the 21st Hawaii International Conference on System Sciences, Kailu-Kona, USA*, pages 317–325, Los Alamitos, 1988. IEEE Computer Society Press.

[CHM88] P. Chen, M. A. Harrison, and I. Minakata. Incremental document formatting. In *Proceedings of the ACM Conference on Document Processing Systems, Santa Fe, New Mexico* [ACM88], pages 93–100.

[Cho56] N. Chomsky. Three models for the description of language. *IRE Transactions on Information Control*, 2(3):113–124, 1956.

[CHP88a] D. D. Chamberlin, H. F. Hasselmeier, and D. P. Paris. Defining document styles for WYSIWYG processing. In van Vliet [vV88], pages 121–137.

[CHP88b] J. R. Cordy, C. D. Halpern, and E. Promislow. TXL: A rapid prototyping system for programming language dialects. In *Proceedings of the 1988 IEEE International Conference on Computer Languages, Miami Beach, USA*, pages 280–285, Los Alamitos, 1988. IEEE Computer Society Press.

[CIV86] G. Coray, R. Ingold, and C. Vanoirbeek. Defining document styles for WYSIWYG processing. In van Vliet [vV86], pages 154–170.

[CK95] K. Chiba and M. Kyojima. Document transformation based on syntax-directed tree translation. *Electronic Publishing – Origination, Dissemination and Design*, 8(1):15–29, 1995.

[CKS+81] D. D. Chamberlin, J. C. King, D. R. Slutz, S. J. P. Todd, and B. W. Wade. JANUS: An interactive system for document composition. In *Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation, Portland, USA, ACM SIGPLAN Notices* 16(6), pages 82–91, New York, 1981. ACM, ACM.

[Cla96] J. Clark. *SP, An SGML System Confining to International Standard ISO 8879 — Standard Generalized Markup Language*, 1996. URL: `http//www.jclark.com/sp/`.

[Cla97] J. Clark. Jade — James' DSSSL engine, 1997. URL: `http://www.jclark.com/jade/`.

[Cor90] J. R. Cordy. Specification and automatic prototype implementation of polymorphic objects in TURING using the TXL processor. In *Proceedings of the 1990 IEEE International Conference on Computer Languages, New Orleans, USA*, pages 145–154, Los Alamitos, 1990. IEEE Computer Society Press.

[Dés86] J. Désarménien, editor. *TEX for Scientific Documentation*. Number 236 in Lecture Notes in Computer Science. Springer-Verlag, Berlin, 1986.

[DJL88] P. Deransart, M. Jourdan, and B. Lorho, editors. *Attribute Grammars. Definitions, Systems and Bibliography.* Lecture Notes in Computer Science 323. Springer-Verlag, Berlin, 1988.

[DMW93] J. Domingue, E. Motta, and S. Watt. The emerging VITAL workbench. In N. Aussenac, G. Boy, B. Gaines, M. Linster, J.-G. Ganascia, and Y. Kodratoff, editors, *Knowledge Acquisition for Knowledge-Based Systems, 7th European Knowledge Acquisition Workshop, EKAW '93*, pages 320 – 339, Berlin, 1993. Springer-Verlag.

[Dra96] N. Drakos. All about LaTeX2HTML. URL: `http://cbl.leeds.ac.uk/nikos/tex2html/doc/latex2html/latex2html.html`, 1996.

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

[FQA88] R. Furuta, V. Quint, and J. André. Interactively editing structured documents. *Electronic Publishing – Origination, Dissemination and Design*, 1(1):9–44, 1988.

[FS88] R. Furuta and P. D. Stotts. Specifying structured document transformations. In van Vliet [vV88], pages 109–120.

[FSS82] R. Furuta, J. Scofield, and A. Shaw. Document formatting systems: Survey, concepts, and issues. *ACM Computing Surveys*, 14(3):417–472, 1982.

[Fur86] R. Furuta. An integrated, but not exact-representation, editor/formatter. In van Vliet [vV86], pages 246–259.

[Fur87a] R. Furuta. Complexity in structured documents: User interface issues. In J. J. H. Miller, editor, *Protext IV Proceedings of the Fourth International Conference on Text Processing Systems, Boston, USA*, pages 7–22, Dublin, 1987. Boole Press.

[Fur87b] R. Furuta. A grammar for representing documents. Technical Report UMIACS-TR-87-67 or CS-TR-1959, Department of Computer Science, Institute for Advanced Computer Studies, University of Maryland, USA, 1987.

[Fur90] R. Furuta, editor. *EP90 — Proceedings of the International Conference on Electronic Publishing, Document Manipulation & Typography, Gaithersburg, Maryland*, The Cambridge Series on Electronic Publishing, Cambridge, 1990. Cambridge University Press.

[Fur92] R. Furuta. Important papers in the history of document preparation systems: basic sources. *Electronic Publishing – Origination, Dissemination and Design*, 5(1):19–44, 1992.

[FW93] A. Feng and T. Wakayama. SIMON: A grammar-based transformation system for structured documents. In Hüser et al. [HMQ93], pages 361–372.

[GG84] H. Ganzinger and R. Giegerich. Attribute coupled grammars. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction, SIGPLAN Notices* 19(6)*, Montreal, Canada* [ACM84], pages 157–170.

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

[Gra91] J. O. Graver. T-gen user's guide. Technical Report SERC-TR-50-F, Software Engineering Research Center, University of Florida, USA, 1991.

[Gra92] J. O. Graver. T-gen: a string-to-object translator generator. *Journal of Object-oriented Programming*, 5(5):35–42, 1992.

[GT87] G. H. Gonnet and F. W. Tompa. Mind your grammar: A new approach to modelling text. In P. M. Stocker, W. Kent, and P. Hammersley, editors, *Proceedings of the Thirteenth International Conference on Very Large Databases, Brighton, England*, pages 339 – 346, Los Altos, 1987. Morgan Kaufmann.

[Har93] K. Harbo. CoST Version 0.2 – Copenhagen SGML Tool. Technical report, Department of Computer Science & Euromath Center, University of Copenhagen, 1993.

[Hec88] R. Heckmann. A functional language for the specification of complex tree transformations. In H. Ganzinger, editor, *Proceedings of the 2nd European Symposium on Programming (ESOP '88), Nancy, France*, number 300 in Lecture Notes in Computer Science, pages 175–190, Berlin, 1988. Springer-Verlag.

[HMQ93] C. Hüser, W. Möhr, and V. Quint, editors. *EP94 —Proceedings of the Fifth International Conference on Electronic Publishing, Document Manipulation & Typography, Darmstadt, Germany, April 1994, Electronic Publishing – Origination, Dissemination and Design,* 6(4), Chichester, 1993. Wiley.

[Iro61] E. T. Irons. A syntax directed compiler for ALGOL 60. *Communications of the ACM*, 4(1):51–55, 1961.

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

[ISO89] ISO – International Standards Organization. *Information Processing – Text and Office Systems – Office Document Architecture (ODA) and Interchange Format, ISO 8613*, 1989.

[ISO92] ISO – International Standards Organization and IEC – International Electrotechnical Commission. *Information Technology — Hypermedia — Time-based Structuring Language (HyTime), ISO/IEC DIS 10744*, 1992.

[ISO96] ISO – International Standards Organization and IEC – International Electrotechnical Commission. *Information technology – Processing Languages – Document Style Semantics and Specification Language (DSSSL) ISO/IEC DIS 10179*, 1996.

[JKL96a] J. Jaakkola, P. Kilpeläinen, and G. Lindén. TranSID: A language for transforming SGML documents. Technical report, Department of Computer Science, University of Helsinki, 1996.

[JKL96b] J. Jaakkola, P. Kilpeläinen, and G. Lindén. TranSID reference manual. Technical report, Department of Computer Science, University of Helsinki, 1996.

[JKL97] J. Jaakkola, P. Kilpeläinen, and G. Lindén. TranSID: An SGML tree transformation language. In J. Paakki, editor, *The Fifth Symposium on Programming Languages and Software Tools, Jyväskylä, Finland*, pages 72–83, 1997. Available as Technical report C–1997-37, Department of Computer Science, University of Helsinki, URL: `http://ftp.cs.helsinki.fi/pub/Reports/`.

[Joh75] S. C. Johnson. Yacc — yet another compiler compiler. Technical Report Computer Science Technical Report No. 32, AT & T Bell Laboratories, Murray Hill, USA, 1975.

[Kas80] U. Kastens. Ordered attributed grammars. *Acta Informatica*, 13:229–256, 1980.

[Kil92] P. Kilpeläinen. *Tree Matching Problems with Applications to Structured Text Databases*. PhD thesis, Report A–1992–6, Department of Computer Science, University of Helsinki, 1992.

[KLMN90] P. Kilpeläinen, G. Lindén, H. Mannila, and E. Nikunen. A structured document database system. In Furuta [Fur90], pages 139–151.

[KM95] P. Kilpeläinen and H. Mannila. Ordered and unordered tree inclusion. *SIAM Journal on Computing*, 24(2):340 – 356, 1995.

[KN94] E. Kuikka and E. Nikunen. Rakenteisten tekstien käsittelyjärjestelmistä (Processing systems for structured texts, in Finnish). Report A/1994/4, Department of Computer Science and Applied Mathematics, University of Kuopio, Finland, 1994. A summary and the system descriptions are available in English at URL `http://www.cs.uku.fi/~kuikka/systems.html`.

[Knu65] D. E. Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607–639, 1965.

[Knu68] D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968. Correction in *Mathematical Systems Theory*, 5(1):95–96, March 1971.

[Knu87] D. E. Knuth. *The TEXbook*. Addison-Wesley, Reading, 1987.

[KP91] E. Kuikka and M. Penttonen. Designing a syntax-directed text processing system. In K. Koskimies and K.-J. Räihä, editors, *Proceedings of the Second Symposium on Programming Languages and Software Tools, Pirkkala, Finland*, Technical Report A–1991–5, pages 191–204, Finland, 1991. University of Tampere.

[KP93] E. Kuikka and M. Penttonen. Transformation of structured documents with the use of grammar. In Hüser et al. [HMQ93], pages 373–383.

[KP95] E. Kuikka and M. Penttonen. Transformation of structured documents. *Electronic Publishing – Origination, Dissemination and Design*, 8(4), 1995. To be published; the number 4 issue of volume 8 has not yet been published in June 1997.

[KPPM84] S. E. Keller, J. A. Perkins, T. F. Payton, and S. P. Mardinly. Tree transformation techniques and experiences. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction, SIGPLAN Notices* 19(6)*, Montreal, Canada* [ACM84], pages 190–201.

[Kui96] E. Kuikka. *Processing of Structured Documents Using a Syntax-Directed Approach.* PhD thesis, Publications C, Department of Computer Science and Applied Mathematics, University of Kuopio, 1996.

[Lam86] L. Lamport. *A Document Preparation system. LATEX User's Guide & Reference Manual.* Addison-Wesley, Reading, 1986.

[Lin92] M. Linster. Sisyphus'91, Part 2: Comparison of different knowledge engineering approaches each based upon models of problem-solving. In M. Linster, editor, *Sisyphus '92: Models of Problem Solving*, Arbeitspapiere der GMD 663, pages 1 – 5. Gesellschaft für Mathematik und Datenverarbeitung MBH, 1992.

[Lin93] G. Lindén. Incemental updates in structured documents. Phil. Lic. Thesis, Report C-1993-19, Department of Computer Science, University of Helsinki, Finland, 1993.

[LMQ+95] G. Lindén, L. Montero, J. M. Quesada, H. Tirri, and A. I. Verkamo. OCML to FND — CASE integration transformations, Technical description. Deliverable T444/DS/1, ESPRIT-II Project 5365 VITAL, 1995.

[LMW88] P. Lipps, U. Möncke, and R. Wilhelm. OPTRAN – a language/system for the specification of program transformations: System overview and experiences. In D. Hammer, editor, *Proceedings of the 2nd Workshop on Compiler Compilers and High Speed Compilation (CCHSC), Berlin, Germany*, number 371 in Lecture Notes in Computer Science (LNCS), pages 52–65, Berlin, 1988. Springer-Verlag.

[LMW91] P. Lipps, U. Möncke, and R. Wilhelm. An overview of the OPTRAN system. In H. Alblas and B. Melichar, editors, *Proceedings of the International Summer School on Attribute Grammars, Applications and System (SAGA), Prague, Czechoslovakia*, number 545 in Lecture Notes in Computer Science, pages 505–506, Berlin, 1991. Springer-Verlag.

[LQV95] G. Lindén, J. M. Quesada, and A. I. Verkamo. OCML to FND — CASE integration transformations, User's guide. Deliverable T444/DS/2, ESPRIT-II Project 5365 VITAL, 1995.

[LRS74] P. M. Lewis, D. J. Rosenkrantz, and R. E. Stearns. Attributed translations. *Journal of Computer and System Sciences*, 9(3):279–307, 1974.

[LS68] P. M. Lewis and R. E. Stearns. Syntax-directed transduction. *Journal of the ACM*, 15(3):465–488, 1968.

[LT95] G. Lindén and H. Tirri. ALCHEMIST — The handbook. Version 1.08. Deliverable T416/DS/2, ESPRIT-II Project 5365 VITAL, 1995.

[LTV95a] G. Lindén, H. Tirri, and A. I. Verkamo. ALCHEMIST: A general purpose transformation generator. Technical Report C–1995–43, Department of Computer Science, University of Helsinki, Finland, 1995.

[LTV95b] G. Lindén, H. Tirri, and A. I. Verkamo. The VITAL transformation assistant. In A. Rouge, editor, *VITAL Project Final Report*, Chapter 4. Deliverable SYSECA/DD71.5, ESPRIT Project 5365 VITAL, 1995.

[LTV96] G. Lindén, H. Tirri, and A. I. Verkamo. ALCHEMIST: A general purpose transformation generator. *Software — Practice and Experience*, 26(6):653–675, 1996.

[Lun88] A. W. Luniewski. Intent-based page modelling using blocks in the Quill document editor. In van Vliet [vV88], pages 205–221.

[LV95] G. Lindén and A. I. Verkamo. An interface between different software development environments. In *Proceedings of the Tenth Annual Knowledge Based Software Engineering Conference (KBSE '95), Boston, USA*, pages 79–87, Los Alamitos, 1995. IEEE Computer Society Press.

[MBO93] S. A. Mamrak, J. Barnes, and C. S. O'Connell. Benefits of automating data translation. *IEEE Software*, 10(4):82–88, 1993.

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

[MKNS89] S. A. Mamrak, M. J. Kaelbling, C. K. Nicholas, and M. Share. Chameleon: A system for solving the data-translation problem. *IEEE Transactions on Software Engineering*, 15(9):1090–1108, sep 1989.

[MOB94] S. A. Mamrak, C. S. O'Connell, and J. Barnes. *Integrated Chameleon Architecture*. Prentice Hall, Englewood Cliffs, USA, 1994.

[Möl94] A. Möller. *SGML — en introduktion till Standard Generalized Markup Language*. Studentlitteratur, Lund, 1994.

[MPP+97] O.-P. Mahlamäki, K. Paasiala, S. Pienimäki, T. Sarajisto, and J. Sievänen. SGML-muunnoskielen toteutus (Implementation of an SGML transformation language, in Finnish). Project work report, Department of Computer Science, University of Helsinki, 1997.

[MR90] N. Major and H. Reichgelt. ALTO — An automated laddering tool. In B. Wielinga, J. Boose, B. Gaines, G. Schrieber, and M. van Someren, editors, *Current Trends in Knowedge Acquisition*, Volume 8 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, Amsterdam, 1990.

[Nel85] G. Nelson. Juno, a constraint-based graphics systems. In *SIGGRAPH '85 Conference Proceedings, SIGGRAPH Computer Graphics* 19(3)*, San Fransisco, USA*, pages 235–243, New York, 1985. ACM.

[Nik90] E. Nikunen. Views in structured text databases. Phil. Lic. Thesis, Report C-1990-60, Department of Computer Science, University of Helsinki, Finland, 1990.

[NM89] E. Nikunen and H. Mannila. Defining and inverting textual views of structured texts. In T. Gyimóthy, editor, *Proceedings of the First Finnish-Hungarian Workshop Symposium on Programming Languages and Software Tools, Szeged, Hungary*, pages 108–120, Szeged, 1989. Research Group on the Theory of Automata, Hungarian Academy of Sciences.

[Oxf96] The Oxford English Dictionary Online, 1996. URL: `http://www.oed.com/`.

[Pay88] T. F. Payton. SSAGS. In Deransart et al. [DJL88], pages 125–127.

[PB78] A. Pyster and H. W. Buttelmann. Semantic-syntax-directed translation. *Information and Control*, 36:320–361, 1978.

[PDC92] T. J. Parr, H. G. Dietz, and W. E. Cohen. PCCTS reference manual, Version 1.0. *ACM SIGPLAN Notices*, 27(2):88–165, 1992.

[PKP+82] T. F. Payton, S. Keller, J. A. Perkins, S. Rowan, and S. P. Mardinly. SSAGS: A syntax and semantics analysis and generation system. In *Proceedings of the IEEE Computer Society's Sixth International Computer Software and Applications Conference (COMPSAC '82), Chicago, USA*, pages 424–432, Los Alamitos, 1982. IEEE Computer Society Press.

[PKPM83] T. F. Payton, S. Keller, J. A. Perkins, and S. P. Mardinly. The DIANA interfacer. In P. J. L. Wallis, editor, *Proceedings of the Workshop on Ada Software Tools Interfaces, Bath, UK*, number 180 in Lecture Notes in Computer Sciences, pages 88–103, Berlin, 1983. Springer-Verlag.

[QV86] V. Quint and I. Vatton. GRIF: An interactive system for structured document manipulation. In van Vliet [vV86], pages 200–213.

[QV87] V. Quint and I. Vatton. An abstract model for interactive pictures. In H.-J. Bullinger and B. Shackel, editors, *Human Computer Interaction — INTERACT '87*, pages 643–647, Amsterdam, 1987. IFIP, Elsevier Science Publishers.

[QVB86a] V. Quint, I. Vatton, and H. Bedor. Grif: An interactive environment for TEX. In Désarménien [Dés86], pages 145–158.

[QVB86b] V. Quint, I. Vatton, and H. Bedor. Le système Grif. *T.S.I – Technique et Science Informatiques*, 5(4):337–341, July – August 1986.

[Rei83] S. T. Reiss. PECAN: Program development systems that support multiple views. Technical Report CS–83–29, Brown University, 1983.

[RT89] T. Reps and T. Teitelbaum. *The Synthesizer Generator. A System for Constructing Language-Based Editors.* Springer-Verlag, 1989.

[Shi84] Q. Y. Shi. Semantic-syntax-directed translation and its application to image processing. *Information Sciences*, 32:75–90, 1984.

[SMB93] C. M. Speerberg-McQueen and L. Burnard, editors. *Guidelines for Electronic Text Encoding and Interchange*, Chapter 2: A Gentle Introduction to SGML. Text Encoding Initiative (TEI), Chicago, 1993. Draft Version 2.

[SMR93] N. Shadbolt, E. Motta, and A. Rouge. Constructing knowledge-based systems. *IEEE Software*, 10(6):34–39, 1993.

[TL94a] H. Tirri and G. Lindén. ALCHEMIST — an object-oriented tool to build transformations between heterogeneous data representations. In *Proceedings of the Twenty-Seventh Annual*

Hawaii International Conference on System Sciences (HICSS '94), Volume II: Software Technology, pages 226–235, Los Alamitos, 1994. IEEE Computer Society Press.

[TL94b] H. Tirri and G. Lindén. VITAL transformation approach. Deliverable UH/DD415, ESPRIT-II Project 5365 VITAL, 1994.

[Tri81] S. Trimberger. Combining graphics and a layout language in a single interactive system. In *Proceedings of the 18th ACM/IEEE Design Automation Conference, Nashville, USA*, pages 234–239, Los Alamitos, 1981. IEEE Computer Science Press.

[Ver94] A. I. Verkamo. Cooperation of KBS development environments and CASE environments. In *Proceedings of the Sixth International Conference on Software Engineering and Knowledge Engineering (SEKE '94), Jurmala, Latvia*, pages 358–365, Skokie, USA, 1994. Knowledge Systems Institute.

[VL94] A. I. Verkamo and G. Lindén. CASE tool interface. Internal deliverable UH/T443/ID003, ESPRIT-II Project 5365 VITAL, 1994.

[VL95] A. I. Verkamo and G. Lindén. Problems in interfacing tools of different development environments. In *Proceedings of the Seventh International Conference on Software Engineering and Knowledge Engineering (SEKE '95), Rockville, USA*, pages 429–437, Skokie, 1995. Knowledge Systems Institute.

[vV86] J. C. van Vliet, editor. *EP86 — Proceedings of the International Conference on Text Processing and Document Manipulation, Nottingham, UK*, British Computer Society Workshop Series, Cambridge, 1986. Cambridge University Press.

[vV88] J. C. van Vliet, editor. *EP88 — Proceedings of the International Conference on Document Manipulation, and Typography, Nice, France*, The Cambridge Series on Electronic Publishing, Cambridge, 1988. Cambridge University Press.

[vVW86] H. van Vliet and J. B. Warmer. An annotated biography on document processing. In van Vliet [vV86], pages 261–277.

[Yel88] D. M. Yellin. *Attribute Grammar Inversion and Source-to-Source Translation.* Lecture Notes in Computer Science 302. Springer-Verlag, Berlin, 1988.

ISSN 1238-8645
ISBN 951-45-7766-3
Helsinki 1997
Helsinki University Printing House

Greger Lindén: Structured Document Transformations **A-1997-2**