A high-level specification language for structured document transformation

XUERONG TANG and FRANK TOMPA
School of Computer Science
University of Waterloo

The purpose of this paper is to introduce and study the problem of automatic transformation of structured documents. We consider collections of documents where the instances in each collection share a common structure in the sense that they can all be characterized by grammar rules such as found in a context-free grammar (CFG) or forest-regular grammar (FRG). We extend the notation to a single XML (or SGML) document with accompanying DTD (document type definition) to say that it is structured. As long as documents do not conform to a single universal standard, the data transformation between them remains a problem. Thus in the absence of a universal tag set and schema, structured document transformation is important for XML to serve as the data interchange format for the Web. Recently, W3C proposed XSLT (Extensible Stylesheet Language Transformations) as a transformation language for XML data. This language has considerable computation power. However, it requires detailed and tedious programming to accomplish complex structure transformations. As alternatives, SDT (Syntax Directed Translation) and its extended form TT (Tree Transformation) grammar are widely used to specify transformations of source code in various programming languages, and they have been proposed as specification languages for structured document transformation. These languages are descriptive but have limited expressive power, which makes them unable to specify complex structure transformations. In this paper, we propose an approach based on syntax tree templates. We show that our language is both descriptive and expressive. We also provide algorithms to convert our specification to XSLT for executing the transformation. Based on the algorithms, we present a prototype implementation.

Categories and Subject Descriptors: I.7.1 [**Document and Text Processing**]: Document and Text Editing – *Document management*; I.7.2 [**Document and Text Processing**]: Document Preparation – *Markup languages; XML;* H.2.m [**Database Management**]: Miscellaneous

General Terms: Management, Algorithm, Experimentation
Additional Key Words and Phrases: Tree transformation, Forest-regular (regular hedge) grammar, Structured document, Specification language, Syntax tree (SynTree), XML, XSLT

1. INTRODUCTION AND MOTIVATION

1.1 Structured documents

In this paper, we introduce and study the problem of automatic transformation between structured documents. Generally speaking, a document is structured if it explicitly or implicitly contains extra information about its hierarchical composition. In this sense, the scope of the meaning of structured documents is quite broad. It could refer to a well-structured document conforming to a pre-defined grammar, which typically is a context-free grammar (CFG), perhaps with further constraints. SGML (Standard Generalized Markup Language) or XML (Extensible Markup Language) documents conforming to some Document Type Definition (DTD) are such examples. It could also be viewed as a tree structure or even graph structure, perhaps conforming to the constraints defined in a form other than a grammar. Semi-structured data [MAG+97, FFK+98, BDHS96] are such examples.

In this paper, we study structured documents of a more rigid form: A document is considered to be structured only if it can be characterized by grammar rules such as found in a context-free grammar and forest-regular grammar [Mur97, Mur98, GS84]. Thus the

structured documents discussed in this paper always have an underlying schema defined by grammar rules, which provide permissible tags (names for attributes and elements of the documents) and structures for documents. In terms of XML, we limit ourselves to "valid" documents rather then merely "well-formed" ones [BPSM00].

1.2 Transformation problem for structured documents

In many disciplines, it is quite common that multiple standards/schema coexist with overlapping functionalities, but one cannot replace another since each has its own unique characteristics. Repeatedly we find that documents need to be converted from one form to another as they are exchanged among various applications. Such transformations will be especially important in manipulating data that is encoded in XML for widespread interoperability. Even within one specific schema definition language such as XML DTD or XML schema, communities of users have the flexibility to develop their own vocabulary and structure for their documents. Unfortunately the transformation between any two user-defined schemas is not a trivial problem. For example, in the chemical industry, there are currently at least two markup languages: CML (Chemical Markup Languageg) and CIDX (Chemical Industry Data Exchange), both of which rely on XML DTDs to develop their own vocabulary and structure for chemical data [CML97, CIDX]. Data transformation will be required if one company wants to acquire and integrate data in both format, or if one CML-supported company wants to incorporate data in CIDX format into its database.

This paper addresses the transformation problem between different schemas developed from the same schema definition language such as XML Schema or XML DTD.

1.3 Transformation process

When structural conversion is needed, the ideal transformation process includes three distinct phases, as discussed in [KP96] and [Mur98]:

1) Determine the input and output constraints (e.g. grammars, DTDs, etc.) imposed on the document.
2) Specify the transformation in terms of input/output instances (e.g. tree patterns) or input/output constraints (e.g. grammars, DTDs, etc.) of the document.
3) Choose an appropriate sequence of operations to carry out the transformation according to the specification. Within this stage, the first step is to identify the parts to be transformed in the input document and the last step is to do the replacement accordingly.

In this paper, we consider several languages with transformation capabilities, served as powerful transformation languages, but we find none of them appropriate as a *specification* language dedicated for transformation, as explained below. On the other hand, SDT and its extended form TT grammar have long served as the dominant mechanisms to specify the translation between source codes in various programming languages [KPPM84]. Thus they have been proposed for structured document transformation as well [KP96]. They are descriptive but have limited expressive power, which makes them unable to specify complex structure transformations.

Therefore, we propose a new language: *Paired SynTrees*, which augments syntax tree templates by enriching both their syntactic and semantic rules. We show that our language is both descriptive and expressive as a specification language for transformation. We also provide algorithms to show how a specification in the SynTree language can be converted to XSLT for execution. Based on the algorithms, we have created a prototype implementation for the SynTree language, which takes the SynTree

specification and automatically generates XSLT scripts and then executes the transformation by using third-party XSLT engines.

2. APPROACHES TO SPECIFY THE TRANSFORMATION

It is natural to consider specifying a transformation using W3C's XSLT [Cla99, Kay01]. An XSLT program (called a stylesheet) is a set of template rules, each of which has two parts: a pattern that is matched against nodes in a source tree and a template that can be instantiated to form part of a result tree. XSLT and similar transformation languages, such as TXL [CP90], are functional programming languages which have powerful computational capability. But when dealing with complex transformations, their programs become complicated and operational in the sense that the programs provide detailed plans that are necessary for carrying out the transformation but not necessary for specifying it. Such languages are well-designed for carrying out a transformation, but their operational nature makes them less desirable candidates for a *specification* language.

Many XML query languages, such as XQuery [BCF+01] and XML-GL [CCD+00], provide transformation capabilities to varying degrees. Our belief is that a query language will seldom be a good transformation specification language. The primary design of these languages is query-oriented, so they seldom support a complex structure transformation that is to comply to an explicitly defined output DTD. Furthermore, in a query language such as XQuery, a program to perform even a simple change to the structure, such as renaming of a single nonterminal node or moving a small set of nodes, involves recursively rebuilding much of the tree. As a result, such programs are neither efficient nor concise as a specification mechanism for structural transformations.

We view all of the above candidates as good target languages to which a specification might be finally translated in an automatic manner. We will therefore focus on more descriptive specification mechanisms in this chapter, and come back to these languages in later chapters.

2.1 An example

For the convenience of discussion, we present an extended context-free grammar that defines a professor list, where the professors are grouped according to departments and within each department professors are grouped by their ranks (full professor, associate professor, assistant professor, etc.).

Example 2.1:

```
prof_list ::= department+
department ::= @title   head   section+
head ::= official+
official ::= title+  rank  professor
section ::= rank   professor+
professor ::= firstname   lastname   degree+   honors?
degree ::= type   univ?
```

We assume that all the nonterminals above that do not appear on the left-side of any production will conform to a production ***Nt ::= string*** where ***Nt*** is the nonterminal and ***string*** produces the terminal strings. These productions are implicitly included in the set of grammar rules.

One possible corresponding DTD can be as follows:

Example 2.2 (prof_list.dtd):

```
<!DOCTYPE  prof_list  [
```

```
    <!ELEMENT   prof_list (department+) >
    <!ELEMENT   department (head section+) >
    <!ATTLIST   department title CDATA #REQUIRED>
    <!ELEMENT   head (official+) >
    <!ELEMENT   section (rank professor+) >
    <!ELEMENT   official (title+ rank professor) >
    <!ELEMENT   professor (firstname lastname degree+ honors?) >
    <!ELEMENT   degree (type univ?) >
]>
```

Similarly as in example 2.1, all the undefined element names, such as *rank*, will conform to a rule <!ELEMENT ***x*** **#PCDATA** >, where #**PCDATA** are terminal strings and ***x*** is the corresponding generic identifier.

2.2 Syntax directed translation (SDT)

Aho and Ullman use input-output paired grammars to describe a syntax-directed translation (SDT), which combines a syntax analysis according to a grammar and code generation according to a second grammar [AU72].

A transformation is defined by a *syntax directed translation schema* (SDTS), which consists of a finite set of nonterminals, one of which is a start symbol and all of which are shared by both input and output grammars; a finite set of input terminals; a finite set of output terminals; and a set of paired grammar rules. Each paired rule comprises one input grammar rule and one output grammar rule. The two rules share the same nonterminal on the left side of the production. The nonterminals in the output rule are a permutation of the nonterminals in the input rule. If the same nonterminal exists more than once, integer superscripts are associated with different occurrences of the same nonterminal name to indicate the associations between the identical symbols in two rules.

The original definition for SDTS is too strict to specify complex structure changes in which we are interested; it cannot even allow renaming of the nonterminals. In order to make it more practical, various extensions have been made to SDTS. Some natural extensions include renaming of the paired associated nonterminals and adding or deleting some nonterminals. Kuikka and Penttonen call SDTS with these natural extensions ESDTS (extended SDTS) [KP96]. Furthermore, SDTS was originally designed for compiling a program strictly conforming to a traditional context-free grammar; one other extension is to let SDTS use extended CFGs in order to deal more naturally with document transformation. It is assumed that the form we consider in this report will implicitly refer to SDTS with all the above extensions.

Suppose we want to change the schema in example 2.1 such that within ***department***, ***head*** will always appear after ***section***, and within ***professor***, ***lastname*** will appear before ***firstname***. We can use the following paired grammar rules to specify this transformation:

Example 2.3:

```
department ::= @title   head   section+,      @title   section+   head
professor ::= firstname   lastname   degree+   honors?,      lastname   firstname   degree+   honors?
```

Aho and Ullman defined an algorithm for automatic transformation of a parse tree via such an SDT schema [AU72]. The algorithm performs a depth-first tree walk from the root of a parse tree to the leaves. Whenever the symbol of a node matches one of the rules in the SDT schema, the algorithm will remove all input terminal children of a node, reorder associated nonterminal children, and add new output terminal children.

The simplicity and regularity makes SDTS feasible in terms of implementation, but these characteristics also limit its expressive power. Because the SDTS method insists that the input grammar and output grammar are strictly paired, an SDT schema only

provides a "flat" description, and thus it is inherently difficult to describe hierarchical changes. For example, moving a nonterminal from one production to another production is disallowed by the SDTS definition. Accordingly, we hypothesize that it is not possible, by using this method, to move nodes up or down in the parse tree. Thus, for example, the following transformation is beyond SDTS's expressive power: reorganize the professor list to group professors by *rank* and indicate the department as one of the attributes within the *professor* subtree.

2.3 Tree transformation grammar (TT grammar)

TT grammar was originally introduced as a formal description technique for describing transformations from one well-defined programming language to another [KPPM84]. TT grammars extend SDTS by allowing users to specify the associations between the input/output grammar rules explicitly. This implies that an input nonterminal node can be associated with an output nonterminal with different name and in different level, thus increasing the expressive capacity as a specification language.

A TT-grammar is a sextuple (*Gi, Go, Si, So, PA, SA*). *Gi* and *Go* are the input and output grammars, respectively. *Si* and *So* are sets of input and output subgrammars, serving as patterns and replacements respectively. *PA* is a set of *production group associations*, and *SA* a set of *symbol associations*. A production group association is a pair (*Si, So*). A symbol association is a relationship between a symbol in *Si* and a symbol in *So* within one production group.

Consider the following transformation example for our professor list, which cannot be specified by SDTS: Modify *head* so that it directly contains *professor*. 2) use an extra element *name* to group *firstname* and *lastname*. Let us formulate this transformation by using a TT grammar.

Example 2.4:

Gi : prof_list ::= department+
department ::= @title head section+
Si[1] : head ::= official+
official ::= title+ rank professor
Si[2] : section ::= rank professor+
professor ::= firstname lastname degree+ honors?
degree ::= type univ?
Go : prof_list ::= department+
department ::= @title head section+
So[1] : head ::= professor+
So[2] : section ::= rank professor+
professor ::= name degree+ honors?
name ::= firstname lastname
degree ::= type univ?
PA : {(**Si**[1],**So**[1]) , (**Si**[2],**So**[2]) }

SA in this example is quite straightforward, nonterminals with the same name will have a natural mapping relationship, the mapping for *name* will be implied by its children mappings, and those nonterminals not showing up in the output grammar such as *official* will be deleted.

TT grammars were originally designed to deal with transformation for programming languages, where transformations are usually in the fashion of expression-to-expression, thus quite localized. Another limitation of this approach is that it cannot specify contextual conditions, which are important in expressing more complex structural transformation as indicated by Murata [Mur96, Mur98]. Lindén applied this TT grammar technique to structured document transformation in his Ph.D. work [Lin97]. There is no indication on how to formalize the associations to express more powerful and complex structure changes, though it is obvious there is such potential [KPPM84]. In conclusion,

we think TT grammar is a fairly convenient mechanism to express structure changes. But without formalizing the association rules and relating proper semantic actions with the rules, we are not clear how to express complex hierarchical changes, and even further from knowing if there is an efficient translation to carry out the transformation given such a specification. Our work is partially based on this approach with some substantial extensions in order to overcome its limitations.

2.4 Filters

Salminen and Tompa [ST99] introduced a pair of filters serving as the description of a parse tree transformation. A *filter* is a sequence of interconnected constraining context-free grammars. Constraining grammars allow boolean conditions on any non-terminal with respect to its context, so they are able to specify more complex structure changes than SDTS does. The input filter is used to mark the parse tree to be transformed, and the output filter describes the new structure to be assembled. In the input filters, both the nonterminals and the constraints in the properties are all selection criteria. In the output filters, however, the constraints in the properties can never be selection criteria but rather they are assembling criteria that are to be met by the result. Associated with each constraining grammar is also a context. For any nonterminal appearing within a filter, the transformation process will first try to find it within its context in the parse tree; otherwise, it is assumed that it is the sibling of a nearest possible ancestor for this nonterminal.

Example 2.5: Consider again the following transformation example: we want to reorganize the professor list according to ***rank*** and indicate ***department*** as one of the attributes within ***professor***. This transformation is basically a *partitioned by* operation in the p-string model [GT87], but we find it quite complex to express by using filters. In order to be consistent with the semantics mentioned above, we have to use two pairs of input/output filters, i.e., two consecutive transformations, to represent it. The first input filter for this example happens to be empty since the whole document is selected. Therefore, for the sake of illustration, we add one more condition to the transformation: only the CS and ECE departments will be selected and transformed. The first pair of the filters is as follows:

Input filters:
Context: ***prof_list***
department ::= @title { = “CS” or =“ECE”} head section+
professor {::it_prof} ::= firstname lastname degree+ honors?

Output filters:
Context: ***it_prof***
professor ::= rank firstname lastname title* @title degree+ honors?
Context: ***prof_list***
prof_list ::= section+
section ::= rankname {¬=rankname} it_prof {rank = rankname }+

The input filter can be used to select all the ***professor*** elements within the two matched ***department*** elements. Then the output filters can be used to reassemble the subtrees. The first output filter is within the context of ***it_prof***. In the parse tree, we can move nonterminal nodes with their subtrees both upward and downward (***title, @title*** in this example where ***title*** is for ***official*** and ***@title*** is an attribute for ***department***), and attach them to node ***professor***. The second filter, working within the context of ***prof_list***, attaches a few empty ***section*** elements to the ***prof_list*** node, and the number of the ***section*** elements is decided in the next constraining production, which imposes two conditions: each ***section*** node has a unique ***rankname*** value and all the ***it_prof*** nodes

whose rank value is equal to some ***rankname*** value are grouped and attached to this ***section*** node. Note, however, that the ***rank*** value of ***it_prof*** must equal the ***rankname*** value of some section within the context of the complete ***prof_list***. This assumes that all ***rank***s appear as ***rankname***s, but does not constrain the ***professor***s to lie within the matching ***section*** only. Therefore, we need to follow that conversion with one more selection condition to make sure that the ***rank*** value matches the ***rankname*** value within each ***section***, which can be specified with an second pair of filters as follows:

Input filters:
Context: ***professor***
professor{::sel_prof } ::= rank{=rankname} firstname lastname title* @title degree+ honors?

Output filters:
Context: ***prof_list***
section ::= rankname sel_prof +
prof_list ::= section+
professor ::= firstname lastname official_title* @ dept_title degree+ honors?

The output filter will only reassemble the "valid" section elements based on the selection results from the input filter, and thus it is safe to drop the ***rank*** elements from the ***professor*** subtree.
Having created such a two-step specification, we must still derive the output parse tree by applying the following five steps to the input parse tree:

1) Within the parse tree, delete any ***department*** node and its subtree if its value is not "CS" or "ECE". In the remaining parse tree , identify ***professor*** as ***it_prof***.
2) Within each ***department*** subtree, push @***title*** down to each node identified as ***it_prof***.
3) Within each ***official*** subtree, push any ***title*** and ***rank*** down to ***professor*** node.
4) Within each ***section*** subtree, push ***rank*** down to each ***professor*** node.
5) Within ***prof_list***, delete all the children and pull up ***section*** elements, combining ***section*** elements with similar ***rank***. Alternatively, create new ***section*** nodes, one per ***rank***, and partition ***it_prof*** elements into ***section*** by their ***rank*** value; then add ***rank*** to the ***section*** node and delete it from ***professor***.

The mechanism of paired filters is flexible in expressing structure changes. However, because filters are closely bound to grammar rules and the correlation between various constraining productions is not obvious, they are difficult to formulate correctly and do not map to the target parse tree very easily. Since the filter specification is quite succinct, we must pay attention to the exact semantics for it. Additional information may be needed to interpret the boolean conditions specified in the filters correctly. The semantics may be suitable for this particular example but not necessarily adequate for specifying other transformations. Secondly, if we consider the situation that the matching points may scatter anywhere in a parse tree with arbitrary levels, the task for assembling them may be arbitrarily complex. How to derive the proper actions to assemble the output parse tree (like the five steps we listed above) according to the filters is a difficult problem. How to prove the validity of the transformation in general needs further study.

Paired filters were introduced as a promising idea for specifying transformation, but they still need to be formalized with exact semantics in order to become an effective mapping language. In the next chapter, we will propose a new language which combines the ideas discussed above and tries to keep a manageable balance between complexity and expressiveness. The resulting language is called *Paired SynTrees*, and we show it overcomes some limitations in each of the approaches.

3. SYNTREE SPECIFICATION LANGUAGE

In this chapter, we introduce **Paired SynTrees**, a high-level specification language which is both expressive and descriptive in terms of specifying transformations. We use forest-regular grammars as the mechanism to express the document structure [Mur96, Mur98]. Compared with DTDs, forest-regular grammars describe structured document schemas more naturally and more expressively.

3.1 Grammar trees and syntax trees

We introduce some definitions which will be frequently used during the discussion for the SynTree language. We assume that conventional *regular* expression (herein called *string-regular* expression), *context-free grammar* (CFG) and *derivation* of the CFG are well known concepts.

Definition 3.1. A *forest-regular grammar* (FRG) is a 4-tuple <*S*, *N*, *P*, *rf*>, where *S* is a finite set of symbols; *N* is a finite set of non-terminals; *P* is a finite set of production rules of the form $A \rightarrow a < r >$, where $A \in N$, $a \in S$, and *r* is a non-empty string-regular expression over $N \cup S$; and *rf* is a string-regular expression over *N*. If *rf* is a single non-terminal, the 4-tuple describes a *tree-regular grammar* (TRG) and *rf* is called the *start nonterminal* of the grammar.

Note: We require that a FRG be normalized such that each nonterminal has exactly one production rule associated with it. Our simplification of the definition omits the set of variables that Murata uses to represent the external (leaf) nodes in a forest [Mur96], which are not needed to support the specification of a transformation.

Murata showed that tree-regular grammars are a better fit than context-free grammars for XML data [Mur96, Mur98, MLM01]. When parse trees, instead of the normal derivation strings, of a context-free grammar are considered as the instances of structured documents, as described in the *p-string* model for structured documents [GT87], a tree-regular grammar can derive those parse trees directly. It has been proven that any set of parse trees derived by a *regular right-part grammar* [Lal77] forms exactly a language defined by an analogous tree-regular grammar [Tha67]. The effect is similar to that achieved by xscheme [Beh00], but forest-regular languages have the advantage of being closed under set union as well as under intersection and difference. We therefore choose forest-regular grammars, also known as regular hedge grammars [BMW+01, Mur00], as the underlying schema language to develop *Paired SynTrees* in spite of the fact that most ideas we borrowed and extended were based on context-free grammars.

Definition 3.2. A *grammar tree* of the TRG is defined to be a tree-like structure derived by using the following steps:

1) Choose the start nonterminal of the TRG as the root.
2) Repeatedly replace the nonterminals with the right part of the corresponding production rule except that: after the first application of any rule $A \rightarrow a<r>$, other replacements of that same nonterminal may be bypassed. Before the replacement of any nonterminal, if the nonterminal is followed by a unary repetition operator (+, *, ?), move that operator to precede the nonterminal.

Because it is straightforward to recover the original grammar up to the renaming of non-terminals by reversing the process, we claim that the grammar tree precisely captures the corresponding TRG grammar.

Example 3.1: An equivalent tree-regular grammar for the DTD in example 2.2 is: G = <*S*, *N*, *P*, *rf*> where

S = { proflist, department, @title, head, official, title, rank, section, professor, lastname, firstname, honor, degree, type, univ}
N = { PROFLIST, DEPARTMENT, HEAD, SECTION, OFFICIAL, PROFESSOR DEGREE }

```
P = {
        PROFLIST →proflist <DEPARTMENT +>
        DEPARTMENT → department <@title  HEAD  SECTION+>
        HEAD → head <OFFICIAL+>
        OFFICIAL → official <title rank  PROFESSOR>
        SECTION → section < rank PROFESSOR+>
        PROFESSOR → professor <lastname  firstname  DEGREE+ honors?>
        DEGREE → degree < type  univ? >
    }
rf = { PROFLIST }
```

Note: In the production rules, the upper-case names represent nonterminals, and lower-case names represent terminals (symbols). In order to hide #PCDATA productions for elements or CDATA productions for attributes, we define those symbols with such productions as special terminals. In this TRG, all names with "@" are special terminals which hide CDATA productions; *rank* and *title* are special terminals which hide the #PCDATA productions.

Example 3.2: The grammar tree for G in example 3.1 can be represented as follows:

```
proflist<+department
          <@title
            head<+official<title rank professor<lastname firstname +degree<type ?univ> ?honors>>>
            +section<rank +PROFESSOR>
          >
        >
```

Note: Since a nonterminal ***Nt*** may appear more than once in a derived grammar tree, we use ***Nt***[i] to indicates the i-th occurrence of the nonterminal in the string representing the derived grammar tree. For example, *PROFESSOR* [1] refers to the first *PROFESSOR* in the grammar tree in example 3.2. This will become useful when we try to modify the content model of *department* by adding a new element containing *professor*.

As illustrated by the above example, we can see that a grammar tree is NOT a derivation tree which has a rigid tree structure, but a more general structure for the underlying data. It represents the complete data space, because it preserves structure information such as alternatives (|), optionalities (?) and multiple occurrences (* or +).

Definition 3.3. A *syntax tree* (SynTree) derived from the TRG is an incomplete grammar tree that can be produced by using the following steps:

1) Choose ANY nonterminal of the TRG as the root.
2) For any nonterminal *A*, whose production rule is $A \rightarrow a\langle r\rangle$, it can be operated in either of the two ways: a) substitute it by $a\langle\ldots\rangle$, $a\langle expr\ldots\rangle$, $a\langle\ldots\ expr\ \rangle$, $a\langle\ldots\ expr\ \ldots\rangle$, where *expr* can be the portion (a forest regular expression) under *a* that is relevant to the transformation, and will be further expanded recursively; notation … is used to indicate that the eluded portion remains unchanged during the transformation process. b) replace it with the right part of the corresponding production rule except that: after the first application of any rule $A \rightarrow a\langle r\rangle$, other replacements of that same nonterminal may be bypassed. Before the processing of any nonterminal, if the nonterminal is followed by a unary repetition operator (+, *, ?), move that operator to precede the nonterminal.
3) Repeat step 2 until each nonterminal is processed according to step 2 precisely once.

Example 3.3: Three of many possible syntax trees for G in example 3.1:

```
SynTree 1: head<+official<title rank professor<lastname firstname +degree<…> ?honors>>>
SynTree 2: department<…head<…>…>
SynTree 3: section<rank +professor<…>>
```

Definition 3.4. A SynTree rooted by symbol *s* is called an *s-SynTree*. Thus SynTree 3 in example 3.3 is a *section*-SynTree.

Note: We can use either *s*<…> or *S* in the SynTree where *S* → *s*<*r*> is the production rule, but with different semantics. We choose *S* to indicate that there is going to be structure changes inside but as specified somewhere else (the first occurrence) in the SynTree. We use *s*<…> to represent the *s*-SynTree in the corresponding grammar tree, which impiles that there will be no structure changes inside this *s*-SynTree. Thus In SynTree 2 of example 3.3, the expression *professor*<…> represents the *professor*-SynTree appearing in the grammar tree of example 3.2.

Definition 3.5. Each symbol or nonterminal appearing in a SynTree is called a *node* in that SynTree. A symbol node is called an *atomic* node because it represents a simple value such as a string or number. A nonterminal node is called a *complex* node because it represents a subtree structure.

Definition 3.6. The *scope* (for the transformation) of a SynTree node refers to the subtree structure associated with the node, including the node itself. For example, expression section<rank +professor<…>> represents the scope for the outermost node *section.*

Definition 3.7. The *envelope* (for the transformation) of a SynTree node refers to the remaining part of the SynTree when the transformation scope of that node is removed.

Before a transformation, we make the following assumptions:

1) A valid document conforms to a tree-regular grammar, so it is always singly-rooted.
2) The input of the transformation can be a collection of documents that can be defined by a forest-regular grammar, while the output must be a single document confirming to a tree-regular grammar. For an XML document with corresponding DTD, it is straightforward to convert its DTD into a corresponding tree-regular grammar.
3) The transformation attempts to keep the parent-child relationship among the translated nodes unless the specification explicitly dictates otherwise.

3.2 Description of the SynTree Language

The SynTree specification language contains four components: a pair of grammar trees, a pair of SynTrees, a set of boolean conditions and a set of mapping rules. In this section, instead of giving the formal syntax for the language, we will present a detailed description for each component, which will be used as the basis for developing the formal syntax and semantics of the SynTree language.

3.2.1 Grammar trees

The first component of the SynTree language is a pair of grammar trees, one represents the grammar or constraints for the input document, and the other represents the desired grammar or constraints for the output of the transformation.

3.2.2 Paired SynTrees

The next component of the SynTree language is a pair of SynTrees, one for the input documents and the other for the output. A SynTree, besides its definition, has further implications as follows:

1) Ellipses will be extensively used to represent subtree structure or a sequence of subtree structure in the SynTree. Those elided subtree structures will remain unchanged. For example, *a*<…> means the substructure of node *a* will remains the same after the transformation; *a*<…*expr*…> indicates only the substructure with

expression *expr* will be affected during the transformation, where *expr* could be a sequence of sub-SynTrees.

2) The *input SynTree* is one SynTree optionally followed by a sequence of additional SynTrees. With respect to semantics, the first SynTree will serve as the base context for the transformation, other SynTrees will be viewed as the subtree structure outside the context but partially or completely needed to be incorporated into the transformation result. The envelope of the first input SynTree root will be deleted as part of the transformation.
3) The *output SynTree* is precisely one SynTree which gives the context for the transformation result. The root of output SynTree is the same as the root of the output grammar tree.

Example 3.4 Suppose we want to make some changes (deleting the rank attribute) within each *head* section only. With the notation introduced so far, we can have the following paired SynTrees as our specification:

Input SynTree : prof_list<+department< … head<+official<+title rank professor<…>>> … >>
Output SynTree : prof_list**<+**department< … head<+official<+title professor<…>>> … >>

The paired SynTrees above make it quite clear what kind of transformation we want to specify: Ellipses are used to avoid presenting unrelated structures which will remain the same during the transformation; *professor* <…> is used to indicate that the *professor*-SynTree will remain unchanged.

In order to describe more complex transformations, we need to add more mechanisms than just a pair of SynTrees. For example, what if we want to keep only CS and ECE departments in the output? We achieve this kind of transformation by introducing *boolean conditions* associated with SynTree nodes. Furthermore, we need a mechanism to associate an input symbol with the corresponding output symbol, because in general it is not likely that the association will be obvious. For example, we may want to rename *professor* as *prof*. We therefore introduce *mapping rules* to fulfill such purpose. These two components make the SynTree language much more expressive.

3.2.3 Boolean Conditions

Boolean conditions are labelled predicate expressions for which the label is placed within square brackets [] attached to a node in the SynTree.

1) Predicates can take only one of the following forms:
   a. Existence testing expressions for a node selected by an XPath expression [Cla99a, BBC+01]. For example,*Cond1*: ../@title.
   b. Relation testing expressions between node sets selected by an XPath expression. For example, *Cond2*: ../i:department/i:@title=”CS” or *Cond3*: ../i:rank = ../o:rank. We use the namespace *i* to refer to the input grammar, i.e., the values before transformation, and *o* to refer to the output grammar, i.e., the values after transformation.
   c. Function constraints to be satisfied by the associated nodes. Currently these may be either *distinct* or *sort*.
   d. Expressions using boolean operators (*not*, *and*, *or*) to combine expressions of the above three types
2) Xpath expressions appearing in the boolean conditon must be localized, which means the selected node should be within the *local* context of the currrent node. More specifically, the selected node is not allowed to be outside the subtree rooted by the top node in the SynTree. Furthermore, “//” is not allowed in the path expression because it has the potential to refer to a node at an arbitrary distance from

the current node and makes the task of efficient translation to operational program more difficult.

3) Boolean conditions appearing in the input SynTree will serve as selection conditions. In other words, if there is a boolean condition associated with a node in the input SynTree, the matching subtrees (in the instance) satisfying the condition will be selected; the corresponding subtrees not satisfying the condition will be deleted.
4) Boolean conditions in the output SynTree are used as the construction constraints.

5) If no selection condition appears on nodes in a SynTree, the condition "TRUE" is assumed. Thus for the input SynTree, all the nodes are selected by default.

An abstract syntax and a clean formal semantics of XPath expression are provided by Wadler [Wad00]. The syntax we choose is based on both the syntax provided by Wadler and the one by Olteanu et al. [OMTB02].

3.2.4 Mapping rules

The mapping rules specify which nodes from the input are to be transformed to which nodes in the output. If input and output nodes have the same name and there is no explicit mapping rule to the output node, then there is an implicit mapping rule to copy the input to the corresponding output.

1) Each mapping rule uses the following syntax: *operator* [*parameter* *] : *S*1→ *S*2, where *S*1 is a set of nodes from the input SynTree and *S*2 from the output SynTree.
2) Both the operators and parameters are chosen from a predefined closed set, which currently includes:
   a. *add* (with parameters for passing constant values or subtrees), used to insert a new value as a new leaf node in the output.
   b. *update* (with parameters for passing constant values), used to update the value of an input leaf node and copy it over to an output leaf node.
   c. *copy* which takes no parameter and uses "union" semantics when more than one set of nodes are selected in the corresponding mapping rules.
   d. *aggregate* (with parameters to indicate aggregation functions), higher-order functions whose parameters themselves are functions. These parameters are either simple functions taking the selected nodes as input or more complex functions taking as input the selected structures within the subtree of the associated node. Each function application returns a single value.
3) In the input SynTree, any nodes associated by a mapping rule, and passing the selection condition if there is any, must be placed somewhere in the output document. Input nodes not associated with any implicit or explicit mapping rules will be deleted. This implies that the transformation assumes no information loss unless otherwise indicated by deletion.
4) In order to create a document that matches the output grammar, every node that must have at least one occurrence in the output instance must be the target of some mapping rule. Furthermore, any such node must also be instantiated by the conversion.

So far, we have described all the components of our SynTree language along with some default semantics that are implicit in the specification language of transformation. Note that such default behavior does not apply in typical query languages, where data is neither retrieved nor transformed unless explicitly specified.

In summary, a paired SynTree specification attaches contextual conditions and mapping rules to syntax tree structures. We have presented a description of the SynTree

language which gives the guideline for the formal syntax and semantics without explicitly separating them. The language is designed to make the set of operators compact, their semantics easy to follow, and common simple transformations easy to specify. Among the components of the SynTree language, the mapping component is the most flexible component. In principle, it can map an arbitrary number of input nodes to an arbitrary number of output nodes, and various operators associated with the mapping may add further computations to such mappings. This nature of the mapping rule complicates the task of figuring out the exact semantic meaning and doing the translation accordingly. Thus a reasonable approach is to restrict mapping rules to the simplest forms and to study the expressive power of the language and complexity involved in the translation algorithm.

3.3 Core (SynTree) language

In this section, we provide a formal syntax, along with a description for semantics, of the core SynTree language, which is a subset of the language described above with further restrictions on the mapping rules.

We use a forest-regular grammar to represent the complete syntax, which has an equivalent representation in BNF [Tha67].

SynTreeSpec → syntreespec<GrammarTrees, SynTrees, BooleanConditions, Mappings>

This rule indicates that the specification has four substructures corresponding to the four components of the SynTree language.

GrammarTrees → grammartrees<InputGrammarTrees, OutputGrammarTree>
InputGrammarTrees → inputgrammartrees<GrammarTree+>
OutputGrammarTree → outputgrammartree<GrammarTree>

This set of rules defines the first component: grammar trees. Note the actual grammar tree for the input and output is not presented because they depend on the transformation application and are provided by the user before the transformation.

SynTrees → syntrees <InputSynTrees, OutputSynTree>
InputSynTrees→inputsyntrees<SynTree+>
OutputSynTree→outputsyntree<SynTree>
SynTree → syntree<@name @id @occurrence? @terminaltype @conditionid SynTree*>

This set of rules defines the structure of the input/output SynTree. The *name* attribute indicates the name of the symbol. The occurrence attribute is used to specify ? , * and +. The terminaltype attribute is used to indicate three situations: *nonterminal*, *terminal*, *alternative*. When the terminaltype equals *alternative*, the corresponding SynTree node is a "fake" node that connect to two or more "real" SynTree nodes among which only one can be chosen when deriving instances.

BooleanConditions → booleanconditions<Condition+>
Condition → condition<@id>

Conditions are predicates described in the previous section.

MappingRules → mappingrules<Rule+>
Rule → rule<@id @name parameter*, (SourceNode*, DestinationNode+)>
SourceNode → sourcenode<@name @id? @appearanceOrder?>
DestinationNode → destinationnode<@name @id? @appearanceOrder?>

The mapping rule is a restricted version with the following restrictions:

1) The mapping operator must be chosen from the set : {*add, update, copy, aggregate*}.
2) Operator add and update allows one-to-one mapping only. When the mapping operator is *aggregate*, only a many-to-one mapping is allowed. In addition, the

parameters allowed for aggregate are simple functions only, chosen from the set: {min, max, avg, count, sum, concat}.

With the simplified mapping rules, a formal semantics of the SynTree specification can be derived in three steps. We first extend a tree-transformer called k-pebble transducer to include data values [AMN+01, MSV00, Suc02]. We then define a collection of tree operations on top of the extended k-pebble transducer. Finally, we define semantic functions to translate the specification into a sequence of such tree operations.

3.4 Specifying transformations

We collect a set of examples of structural transformations and demonstrate how to use the SynTree language to specify them. We first give a classification of transformations in terms of structure changes, then give an example for each possible classification. Most of the examples are based on a university catalog conforming to the DTD defined in example 2.2, whose equivalent tree-regular grammar (TRG) is shown in example 3.1.

3.4.1 Class One: expressible by SDT (no Boolean conditions)

Example 3.5.1 Switching the symbol. Suppose we want to switch the order of elements *lastname* and *firstname*. The SynTree specification will be:

Input SynTree : proflist<…professor< firstname lastname…>…>
Output SynTree : proflist<…professor< lastname firstname … >…>

Example 3.5.2 Removing an attribute.

Input SynTree : proflist<…official< @position professor<…> rank>…>
Output SynTree : proflist<…official< professor<…> rank>…>

All the examples within this class are expressible in SDT so our specification can be reduced to an equivalent form of SDT specification.

3.4.2 Class Two: expressible by SDT except for the boolean conditions

With boolean conditions, we can impose contextual conditions when specifying transformation, which beyond the capability of SDT.

Example 3.5.3 Extract cs professors only

Input SynTree : department[C1]<…>
Output SynTree : department<…>
Boolean Condition : C1: @title="CS"

We copy all the nodes that satisfy the selection condition.

Example 3.5.4 Delete professors whose firstname is John.

Input SynTree : proflist<…professor[C1]<…>…>
Output SynTree : proflist<…professor<…>…>
Boolean Condition : C1: not(@firstname="John")

We copy all the nodes that satisfy the complement of the deletion condition (i.e. do not match those to be deleted).

3.4.3 Class Three: updates not expressible by SDT

Examples in this class are a trivial because only leaf nodes will be affected. Though trivial, our specification language should support it so that the users can express simple update operations on specific elements in their structured documents.

Example 3.5.5 Add *age* element to professor named John in the list.

Input SynTree : proflist<…professor<…?age>…>
Output SynTree : proflist<…professor<… ?age>…>
Boolean Condition : C1: @firstname="John"

Mapping Rule : *add*("Ph.D"): age[C1]

Example 3.5.6 Updating the value of an attribute. Suppose professor John Smith is an official who just got promoted from vice president to president of the university. We assume both positions are unique in this catalog. The SynTree specification is as follows:

Input SynTree : proflist<…official[C1] < title…>…>
Output SynTree : proflist<…official< title …>…>
Boolean Condition : C1: @position="vice president"
Mapping Rule : *update*("president"): @title→@title

3.4.4 Class Four: non-nesting structure changes not expressible by SDT

Example 3.5.7 Divide professors within one section into two groups, those who received PhD degrees from UW and those who did not.

Input SynTree : proflist<…section<rank *professor<…>> …>
Output SynTree : proflist<…section<rank section_uw< *professor [C1]<…>>
section_outside< * PROFESSOR [C2]> >…>
Boolean Condition : C1: ./degree/type="PhD" and ./degree/univ="UW"
C2: not (./degree/type="PhD" and ./degree/univ="UW")
Mapping Rule : *copy*: professor → professor[1] , professor[2]

3.4.5 Class Five: nesting structure changes and aggregations

Example 3.5.8 Consider the transformation problem given by W3C's XML Query Working Group as **Use Case ``PARTS''** [CFMR01]: The input is a flat list of *part* elements, each of which has values for *partid* and *name* attributes. Each *part* may or may not be a component of a larger *part*, indicated by the value of the *partof* attribute. The transformation is to convert the flat representation into an explicit hierarchic representation, based on *partof* attributes.

Input DTD:

```
<!DOCTYPE partlist [
    <!ELEMENT partlist  (part*)>
    <!ELEMENT part      EMPTY>
    <!ATTLIST   part
        partid  CDATA   #REQUIRED
        partof  CDATA   #IMPLIED
        name    CDATA   #REQUIRED> ]>
```

Output DTD:

```
<!DOCTYPE parttree [
  <!ELEMENT parttree  (part*)>
  <!ELEMENT part      (part*)>
  <!ATTLIST   part
      partid CDATA  #REQUIRED
        name CDATA  #REQUIRED> ]>
```

The corresponding SynTree specification is as follows:

Input SynTree : partlist< *part<@partid ?@partof @name>
Output SynTree : partree< *part[C1]<@partid @name *Part[C2] >
Boolean Condition : C1: not(i:@partof)
C2: i:@partof =../o:@partid
Mapping Rule : *copy*: partlist → parttree
copy: part → part Part

Example 3.5.9 Consider again the list of university professors. Assume now that a transformed listing is to be produced to include only members of the Departments of Computer Science and of Electrical and Computing Engineering, but this list is to be strictly partitioned by rank, with the department name appearing as an attribute for each high-tech professor. We also want to add one more attribute to the newly generated section @num, which will store the number of professors in this section.

The transformation can be specified by the following paired SynTrees:

Input SynTree : proflist<*department[C1]< @title
head<*official<title+ rank professor<…> >>
*section<rank *professor<…> >>>
Output SynTree : proflist<*rank_section<@rank @num[C2] *hitech_prof[C2]<@dept_title…>>>
Boolean Condition : C1: i:@title="ECE or i:@title="CS"
C2: ../i:rank = ../o:@rank

Mapping Rule : *copy* : rank[1] rank[2] → rank[*distinct*()]
copy : @title → @dept_title
copy : professor[1] professor[2] →hitech_prof
aggregate(*count*) : professor[1] professor[2] → @num

Notice that the mapping rules, together with boolean condition C2, specify that the output listing is to include each rank exactly once, and that each professor is to appear under the correct rank name (according to the associated rank in the input document, whether or not that professor was an officer), and the number of professors under the same section is to be counted as the *num* attribute.

4. OPERATIONAL LANGUAGES FOR TRANSFORMATION

For paired SynTrees to be a useful specification mechanism, we need algorithms to convert from the description of a specification to a sequence of operations that carry out the transformation on document instances. In this section, we introduce some languages that are suitable for expressing the operational behavior of a specified transformation. In the next, we provide algorithms to translate our specification to one such language, XSLT.

TXL and XSLT are functional languages combined with pattern matching rules. Both are Turing-complete in terms of computation power. Forest (or Hedge) automata, derived from forest (or hedge) grammars, are capable of describing both patterns and contextual conditions and thus are fairly flexible ways of describing transformations [Mur96, Mur98]. Although it is more powerful than finite automata, its computation power is not Turing-complete. In fact, it can only capture a very restricted fragment of XSLT.

4.1 Forest Automata

Murata describes transformations by using forest (sequence of parse trees) conditions. Forest conditions include patterns and contextual conditions, where patterns are conditions on immediate or descendant subordinate nodes while contextual conditions are conditions on non-subordinates such as immediate or ancestor superiors, siblings, and subordinates of siblings [Mur98].

Each condition in Murata's approach can also be represented by a forest automaton, which can be strictly derived from the corresponding forest grammar. In this way, since a document and condition are both instances of forest-regular languages recognized by the corresponding automata, pattern matching and condition testing can be done by the intersection of the two corresponding automata.

Document transformation is defined as a composition of a marking function M_c^P and a linear tree homomorphism *H*. The function M_c^P marks a node if the subtree rooted by this node matches pattern *P* and the envelope (the rest of the tree) satisfies contextual condition *C*. The algorithm used by this marking function is basically a pattern matching algorithm or an algorithm for contextual condition testing. Homomorphism *H* is essentially a replacement algorithm which rewrites the tree, for example, by deleting or renaming marked nodes.

In terms of computation power, forest automata are not as expressive as Turing machines. An extended version of forest automata, called k-pebble transducers, are shown to be a rather restricted version of XSLT. In the restricted version, equality testing based on node values is not allowed, which limits the expressiveness [MSV00, Suc02]. Bex, Maneth and Neven provide a more expressive model for XSLT, which can completely simulate a k-pebble transducer, and the resulting language from this model is shown to be not Turing-complete [BMN00]. Murata develops a forest algebra based on this computation model and starts to develop a rule-based language called *forestlog* on

top of the algebra [Mur98]. When the language is fully defined, we will have a better target to analyze its expressive power. The examples demonstrated by Murata are fairly simple transformations, such as deleting nodes or relabeling nodes. The process for describing the transformation are complicated and operational [Mur96, Mur98], and thus will not be presented in this paper.

4.2 TXL and XSLT

Both XSLT (XML eXtensible Stylesheet Languague Transformation) and TXL (Turing eXtensible Language) are tree-manipulation programming languages which combine features of functional languages with pattern-matching rules [Cla99, CP90].

A TXL program takes as input a parse tree according to a given input grammar, and transforms it into an output parse tree by applying its pattern matching rules. A basic pattern matching rule in TXL looks like this:

```
rule name
  replace [type]
    pattern
          by
            replacement
end rule
```

Where *name* is a rule identifier, *type* is the nonterminal type designating the root of the input parse tree, *pattern* is a pattern which the input parse tree must match in order to be transformed, and *replacement* is the result of the corresponding transformation.

XSLT is proposed by W3C as an XML extensible stylesheet language (XSL) for transformation. Its primary role is to allow users to write transformations from one XML document to another. Very similar to TXL, an XSLT program (called *stylesheet*) is also a set of template rules. A *template rule* has two parts: a pattern which is matched against nodes in the source tree and a template which can be instantiated to form part of the result tree.

As indicated by others [BMN00], before entering its recommendation phase in November 1999 [Cla99], the database community [DFF+99, ABS00] viewed XSLT as a weak language in terms of expressive power and only recommended it for simple transformations such as HTML formatting. But in the recommendation version, with some important added features, XSLT has become a powerful general-purpose transformation language. Those added features include flexible control structures, variable binding for node sets, parameter passing between templates and template modes (to mimic the states of a tree transducer). These additions make XSLT a Turing-complete language. In its latest version [Kay01], XSLT has added more powerful features, such as the operator *groupby*.

Although TXL and XSLT look similar at the syntax level, they have some fundamental differences. First of all, TXL takes a tree-editing approach which continuously makes changes to the source tree whenever matching happens; on the other hand, XSLT only navigates the source tree and emits output whenever matching occurs; the output will not affect the input tree in any way. Secondly, a pattern in TXL could be any string generated by a context-free grammar, and needs to be parsed into a parse tree for pattern matching; while XSLT forms its pattern by using XPath expressions, which are not as expressive as TXL's tree patterns at a structural level but have extra equality-testing capabilities. Finally, TXL is mainly used as a tool for transforming between different programming languages or variants, so the patterns and replacement are usually expressions in various programming languages; XSLT, on the other hand, is dedicated for transformation of structured documents. These differences contribute to the fact that, for the same transformation task, the two programs may look different and also behavior differently.

We view both of the languages as powerful operational languages, and thus good candidates to which a specification could be translated. Since XSLT is widely supported and has open-source implementations, we choose XSLT as our target language. In order to develop the translation algorithm, we need to know a little more detail about how XSLT works.

4.3 Push and Pull technique of XSLT

The push and pull technique related to XSLT is briefly discussed by James Clark [Cla99b]. Generally speaking, push means emitting outputs whenever some condition is satisfied during the navigation of the source tree. Such a typical push model is deployed by the SAX2 specification. Pull usually refers to a process that walks through an output template and retrieves data from various input sources whenever necessary. A typical example is a JSP (Java server page), which usually defines an HTML template and then fills in data either dynamically generated on the fly or retrieved from databases by calling Java beans via JDBC drivers.

An XSLT program typically uses XPath expressions to navigate a static source tree up and down without modifying it, matching the pattern described in XPath expressions. During the tree-walking, it can either issue new templates or construction whenever specified pattern matches selected nodes (push technique) or generate query result of the source tree within construction elements (pull technique) To better understand these two techniques, we give two simple examples in XSLT.

An example of the push technique is the use of "match" to generate the output by further processing all the children of the matched *student* nodes from the input:

```
<xsl:template match="student">
    <xsl:apply-templates/>
</xsl:template>
```

An example of the pull technique is the use of "select" to query the source and extract the value of a selected source node back:

```
<newNode>
  <xsl:value-of select="./firstName">
</newNode>
```

Push is usually deployed in document transformation with a rule-based approach where the output structure is closely dependent on input structure. Pull, on the other hand, is widely used for data transformation, typically with a template which implies the output structure; therefore the output structure can be independent of the input structure.

With the aid of XPath expressions, XSLT allows the combination of both pushing and pulling in a single transformation. On one hand, XPath provides a query language to fulfill the task of data pulling; on the other hand, XPath expressions can serve as a pattern to be matched in the process of data pushing.

4.4 XSLT default templates

An XSLT template takes the following form:

```
<xsl:template match = pattern name = qname priority = number mode = qname>
        do some construction work during which possibly call/apply other templates…
</xsl:template>
```

Basically we have three kinds of template: the *pattern template* which does not need a *name* or *mode* attribute, the *named template* which must have a *name* attribute but does not require a pattern, and the *mode template* which must have a *mode* attribute and requires a pattern as well.

These three templates are called in different ways:

```
<xsl:apply-templates  select = node-set-expression  mode = qname>
  provide sorting criteria or template parameters if applicable…
</xsl:apply-templates>

<xsl:call-template name = qname>
  provide template parameters if applicable…
</xsl:call-template>
```

Pattern templates can be called by an *xsl:apply-template* element without mode or name attribute. Mode templates can only be called by *xsl:apply-templates* element with a mode attribute. Thus mode can be used to enforce a particular construction phase by restricting processing to a set of templates that will be called during that phase [BMN00]. Named templates can only be called by *xsl:call-template* with a matched name attribute. Named templates give the flexibility to call a specific template whenever necessary at any construction phase.

We can use XSLT templates to mimic the following generic tree operators: *fullTreeCopy*(…), *subTreeCopy*(…), *subTreeDelete*(…), *nodeCopy*(…), *nodeDelete*(…), *addLeafNode*(…) and *updateLeafNode*(…). In addition, we also introduce some common database operations that can be easily imitated by XSLT templates: *aggregate*(*count/max/min/sum/average/concat*).

To support the SynTree specification language, these templates can be pre-built and put into an *XSLT Library* which will be used by the translation algorithm. For each type of template, we can pass parameters (denoted by …) to control the actual behavior. For example, we can name a template by passing the name parameter. We can also assign a mode to any template to mimic the state of construction. We can pass the XPath expression on-the-fly to form the actual pattern in the template. We can also generate new source locations so that within any template we can apply other templates to these locations.

We should keep in mind that XSLT defines some default templates:

1) "continue-process" rule for document root and all the elements:

```
<xsl:template match = " * | / ">
   <xsl:apply-templates>
</xsl:template>
```

2) "produce-value" rule for text nodes and attributes:

```
<xsl:template match=" text() | @* ">
  <xsl:value-of select = "."/>
</xsl:template>
```

3) "do-nothing" rule for processing-instruction nodes and comment nodes:

```
<xsl:template match=" processing-instruction() | comment() "/>
```

These templates are automatically added to any generated stylesheet. Unless implicitly replaced by other templates, these templates will be processed whenever the matching happens.

Our algorithm takes an "implicit-deletion" approach: for those nodes without matching template rules, the deletion action is implicitly implied. For this purpose, we need to overwrite the "produce-value" rule so that the deleted elements will not generate unexpected values from their text children nodes or attribute children nodes. The overwritten rule is as follows:

```
<xsl:template match = " text() | @* "/>
```

It is essentially a "do-nothing rule" that implicitly implements *subTreeDelete*(…). Because the "produce-value" rule also applies to mode templates, for each mode template

whose value for mode attribute is *modeName*, our algorithm also needs to generate the following do-nothing template:

```
<xsl:template match = "text() | @*" mode = "modeName"/>
```

5. TRANSLATION ALGORITHM FOR THE CORE LANGUAGE

On this foundation, we are ready to introduce the translation algorithm targeting XSLT.

5.1 Tree model

Conceptually, the SynTree specification can be viewed as a pair of tree-like structures with the following characteristics:

1) The trees represent part of the syntactic structure of input/output documents that is relevant to the transformation. In addition, the full grammar trees of input/output documents are always available whenever necessary.
2) Each mapping rule is attached to some input nodes and some output nodes. Whenever a SynTree node is visited, we assume it is trivial to find its associated mapping rules.
3) Each boolean condition is attached to a SynTree node. When the SynTree node is visited, we assume it is also trivial to check its boolean conditions.

5.2 Description of the general algorithm

Here we present a general algorithm that conceptually describes how to translate a SynTree specification into XSLT templates. The detailed algorithm with its complexity analysis will be presented in a later section.

Input: iSynTree: input SynTree in the SynTree specification
oSynTree: output SynTree in the SynTree specification
Mappings: the mapping rules in the SynTree specification
Conditions: the boolean conditions in the SynTree specification
iGrammarTree: the complete grammar tree for the source doucment
oGrammarTree: the complete grammar tree for the destination document

Output: XSLT stylesheet consisting of a series of template rules, implementing the transformation

Step 1. Verify the SynTree Specification

1. Validate the correctness of the syntax.
2. Validate iSynTree with respect to iGrammarTree, oSynTree with respect to oGrammarTree.
3. Validate and preprocess Mappings. This makes sure that each oSynTree node must have at least one associated mapping rule, whereas each iSynTree node may be (but not necessarily) associated with one or more rules. (iSynTree nodes without mappings will be implicitly deleted).
4. Validate boolean conditions conforming to XPath syntax.

Step 2. Initialization for the translation

1. Locate the root of oSynTree: currentOutputNode ← oSynTree.getRoot();
2. Construct a stylesheet containing only default templates initially:
   XSLTStylesheet xsltStylesheet = new XSLTStylesheet();
3. Initialize the *nodesToProcess* Queue to be empty

Step 3. Traverse the oSynTree in a top-down breadth-first manner

1. Generate a set of current *bindings* from currentOutputNode, including currentInputNodes, currentMapping, currentConstructionCondition and currentSelectionCondition.
2. Generate a construction template for current node by using currentMapping and currentConstructionCondition.
3. For each child, oChild, of the currentOutputNode, adjust the above template by inserting more construction or apply-template rules whenever necessary:
   a. Get bindings from oChild, including iChildren, childMapping
   b. Produce Xpath expression from currentInputNodes to iChildren
   c. If the mapping is *copy*, put oChild into the NodesToProcess queue and insert an applyTemplate rule with *mode* attribute into the template in order to further process this node. Otherwise, for the case of leaf operation (*add*, *update*) or aggregation (*min*, *max*,…), insert construction rule for oChild and no further process is needed for this node.
4. Add adjusted template into xsltStylesheet.
5. if NodesToProcess queue is non-empty, dequeue one node as currentOutputNode and loop back to step 3.1, else continue to step 4.

Step 4. Return the generated xsltSylesheet;

The detailed algorithm is in Appendix. This function implements a tree walker that descends from the root of the output SynTree. In a breadth-first manner, it processes each node in the output SynTree exactly once.

The heart of the algorithm is in step 3.3 where 3.3(b) is a navigation process which starts from the set of current input nodes defined by current mapping rule and computes the possible XPath expressions to the children input nodes defined by the mapping rule of the corresponding child output node. It is also the most complicated procedure in terms of computation complexity. In general, if we know the depth of each node in the SynTree, it takes at most twice the height of the tree to find a path between any two nodes in the tree. Let us look at the complexity of the whole algorithm based on the assumption that the size and height of the SynTrees are bounded by n and h respectively and the number of input nodes involved in each mapping rule is bounded by m:

Step 1 of the algorithm needs several tree traversals which takes O(n) time where n is the size of the output SynTree. Step 2 needs constant time for initialization. Step 3 needs one tree traversal for its main procedure, which takes O(n); sub-procedure 3(b) can take O(hm^2) time in the worst case; all the rest of the sub-procedures take constant time; so the total cost for this step is O(nhm^2). Step 4 requires constant time for returning the results. Therefore the complexity of the whole algorithm is O(nhm^2).

6. SYNTREE IMPLEMENTATION

In this section, we give an introduction to the prototype implementation for the SynTree language and its translation algorithm. The entire software is written in pure Java, which makes the system portable to various platforms.

6.1 Software components

There are four modules in our SynTree software:

1) *Paired SynTree Parser* converts the SynTree specification into an in-memory tree structure.

2) *Paired SynTree Visualizer* displays the input/output SynTrees graphically. Ideally a user could interactively modify the SynTrees, but this feature has not been implemented yet.
3) *Stylesheet Generator* implements the translation algorithm.
4) *SynTree Transformer* reads the input document and transforms it into an output document by calling the XSLT engine to execute the XSLT script.

In Figure 6.1 we give a graphical presentation of our system, which indicates the relationship among different modules:

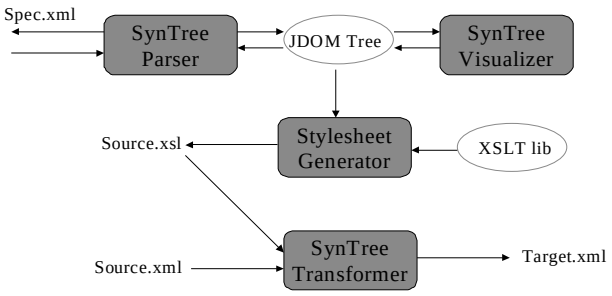

Fig. 6.1 System components

6.2 Target languages and tools

Many open-source implementations around XML specifications/applications have been developed in Java, and they are available for downloading over the internet. We designed an XML format to represent SynTree specifications so that we can benefit from the availability of XML parsers. We use the SAX-compatible XML parser Crimson (or Xerces) to do the syntax validation for our specification [Meg00]. Because we need an in-memory representation for the SynTree specification, we chose JDOM to load our specification into a JDOM tree [HM02, BE01]. JDOM is not compatible with W3C DOM specifications [ABC+98], but it is optimized for Java so that it avoids the heavy memory-print due to DOM's language-neutrality.

Our target language is XSLT. We use Xalan as our XSLT engine, but the user can switch to any other XSLT engine such as Saxon or XT. In our system, the XSLT engine will also use either Crimson or Xerces to parse input documents and then carry out the translation according to the stylesheet generated by the Stylesheet Generator.

6.3 Applications

In this section we use examples to show how the SynTree system works.

6.3.1 Partition

Consider again the transformation from example 3.7 that will divide professors within one section into two groups, those who received PhD degrees from UW and those who did not.

The *preprocessor* will take this specification and produce an augmented specification in XML format. Additional information, such as the depth of each SynTree node, the type of the symbol, the unique id associated with each SynTree node as well as some default mapping rules, will be added. It is this XML file that will be processed by the translation

algorithm. The following is the screen shot for this transformation in the SynTree System:

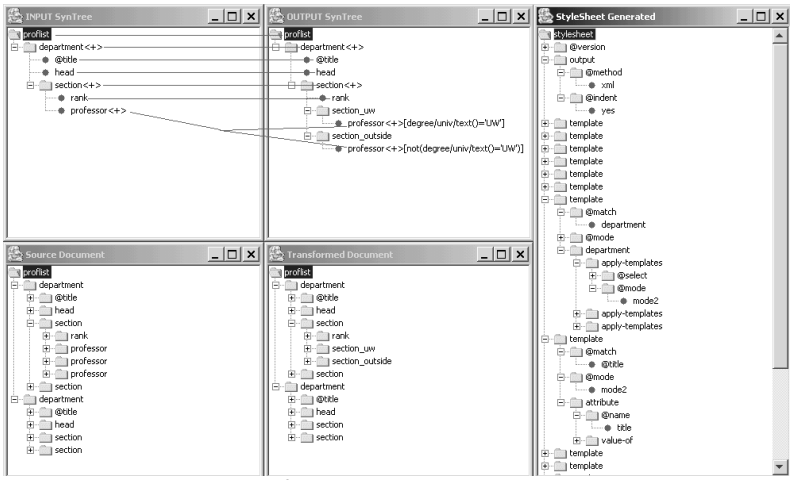

Fig. 6.2 Specifying Partition in Paired SynTrees System

The corresponding specification in XML format is as follows:

```
<?xml version="1.0"?>
<!DOCTYPE synTreeSpec SYSTEM "synTreeSpec.dtd">
<synTreeSpec>

  <iSynTree>
    <synTree  id="1" level="1" name="proflist">
      <synTree id="2" level="2" name="department" occurrence="oneOrMore">
        <synTree id="3" level="3" name="@title"/>
        <synTree id="4" level="3" name="head"/>
        <synTree id="5" level="3" name="section" occurrence="oneOrMore">
          <synTree id ="6" level="4" name="rank"/>
          <synTree id ="7" level="4" name="professor" occurrence="oneOrMore"/>
        </synTree>
      </synTree>
    </synTree>
  </iSynTree>


  <oSynTree>
    <synTree  id="1001" level="1" name="proflist">
      <synTree id="1002" level="2" name="department" occurrence="oneOrMore">
        <synTree id="1003" level="3" name="@title"/>
        <synTree id="1004" level="3" name="head"/>
        <synTree id="1005" level="3" name="section" occurrence="oneOrMore">
          <synTree id ="1006" level="4" name="rank"/>
          <synTree id = "1008" level ="4" name="section_uw">
            <synTree id ="1009" level="5" name="professor" occurrence="oneOrMore"
                                          condition  = "degree/univ/text()='UW'" />
          </synTree>
          <synTree id = "1010" level ="4" name="section_outside">
            <synTree id ="1011" level="5" name="professor" occurrence="oneOrMore"
                                        condition  = "not(degree/univ/text()='UW')" />
          </synTree>
        </synTree>
      </synTree>
    </synTree>
  </oSynTree>

  <mRules>
    <rule   id="m1" name="copy">
```

```
      <sourceNode  id="1" name="proflist"/>
      <destinationNode  id="1001" name="proflist"/>
    </rule>
    <rule  id="m2" name="copy">
      <sourceNode  id="2" name="department"/>
      <destinationNode  id="1002" name="department"/>
    </rule>
    <rule  id="m3" name="copy">
      <sourceNode  id="3" name="@title"/>
      <destinationNode  id="1003" name="@title"/>
    </rule>
    <rule  id="m4" name="copy">
      <sourceNode  id="4" name="head"/>
      <destinationNode  id="1004" name="head"/>
    </rule>
    <rule  id="m5" name="copy">
      <sourceNode  id="5" name="section"/>
      <destinationNode  id="1005" name="section"/>
    </rule>
    <rule  id="m6" name="copy">
      <sourceNode  id="6" name="rank"/>
      <destinationNode  id="1006" name="rank"/>
    </rule>
    <rule  id="m7" name="copy">
      <sourceNode  id="5" name="section"/>
      <destinationNode  id="1008" name="section_uw"/>
    </rule>
    <rule  id="m8" name="copy">
      <sourceNode  id="5" name="section"/>
      <destinationNode  id="1010" name="section_outside"/>
    </rule>

    <rule  id="m9" name="copy">
      <sourceNode  id="7" name="professor"/>
      <destinationNode  id="1009" name="professor"/>
    </rule>
    <rule  id="m10" name="copy">
      <sourceNode  id="7" name="professor"/>
      <destinationNode  id="1011" name="professor"/>
    </rule>


  </mRules>

</synTreeSpec>
```

We give a brief description on how the translation algorithm generates the stylesheet. First it constructs a new stylesheet with some default templates:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <xsl:output method="xml" indent="yes"></xsl:output>
  <xsl:template match="text()|@*"></xsl:template>
   … more templates need to be inserted here…
</xsl:stylesheet>
```

Then it generates a set of templates for the construction of the root of the output SynTree. Meanwhile it also prepares for the further construction of the children nodes:

```
<xsl:template match="proflist">
   <proflist>
     <xsl:apply-templates select="./department" mode="mode1"></xsl:apply-templates>
   </proflist>
</xsl:template>
```

Then it continues to process the root's child node *department*. In addition, it also produces a default mode template:

```
<xsl:template match="text()|@*" mode="mode1"></xsl:template>
<xsl:template match="department" mode="mode1">
  <department>
    <xsl:apply-templates select="./@title" mode="mode2"></xsl:apply-templates>
    <xsl:apply-templates select="./head" mode="mode3"></xsl:apply-templates>
    <xsl:apply-templates select="./section" mode="mode4"></xsl:apply-templates>
  </department>
</xsl:template>
```

Next, for the first child node of *department*, it generates a template using a similar process.

```
<xsl:template match="text()|@*" mode="mode2"></xsl:template>
<xsl:template match="@title" mode="mode2">
  <xsl:attribute name="title">
    <xsl:value-of select="."></xsl:value-of>
  </xsl:attribute>
 </xsl:template>
```

Templates are generated in the same way for the second and third children nodes. The algorithm issues a subtree copy for *head* as indicated in our specification.

```
<xsl:template match="text()|@*" mode="mode3"></xsl:template>
<xsl:template match="text()|@*" mode="mode4"></xsl:template>
<xsl:template match="head" mode="mode3">
  <head>
    <xsl:for-each select="./@* | ./node()">
      <xsl:copy-of select="."></xsl:copy-of>
    </xsl:for-each>
  </head>
</xsl:template>
<xsl:template match="section" mode="mode4">
  <section>
    <xsl:apply-templates select="./rank" mode="mode5"></xsl:apply-templates>
    <xsl:apply-templates select="." mode="mode6"></xsl:apply-templates>
    <xsl:apply-templates select="." mode="mode7"></xsl:apply-templates>
  </section>
</xsl:template>
```

The algorithm iteratively generates more templates to achieve the remaining constructions:

```
<xsl:template match="text()|@*" mode="mode5"></xsl:template>
<xsl:template match="text()|@*" mode="mode6"></xsl:template>
<xsl:template match="text()|@*" mode="mode7"></xsl:template>
<xsl:template match="rank" mode="mode5">
  <rank>
    <xsl:for-each select="./@* | ./node()">
      <xsl:copy-of select="."></xsl:copy-of>
    </xsl:for-each>
  </rank>
</xsl:template>
<xsl:template match="section" mode="mode6">
  <section_uw>
    <xsl:apply-templates select="./professor" mode="mode8"></xsl:apply-templates>
  </section_uw>
</xsl:template>
<xsl:template match="section" mode="mode7">
  <section_outside>
    <xsl:apply-templates select="./professor" mode="mode9"></xsl:apply-templates>
  </section_outside>
</xsl:template>
```

```
<xsl:template match="text()|@*" mode="mode8"></xsl:template>
<xsl:template match="text()|@*" mode="mode9"></xsl:template>
<xsl:template match="professor[degree/univ/text()='UW']" mode="mode8">
   <professor>
     <xsl:for-each select="./@* | ./node()">
       <xsl:copy-of select="."></xsl:copy-of>
     </xsl:for-each>
   </professor>
</xsl:template>
<xsl:template match="professor[not(degree/univ/text()='UW')]" mode="mode9">
   <professor>
     <xsl:for-each select="./@* | ./node()">
       <xsl:copy-of select="."></xsl:copy-of>
     </xsl:for-each>
   </professor>
</xsl:template>
```

The whole stylesheet mimics a depth-first tree transducer, which is equivalent to a k-pebble transducer [MSV00] plus more powerful condition testing capabilities.

6.3.2 Intersection

In this example, we assume a simple grammar to represent a student list where students can show up in either a club or a CS department or both. We also assume that the *id* is the unique identification for student. We want to form a document in which there is one student list containing all the students who are both in the club and in the CS department.

Input Grammar Tree : univ< club< +student<@id @name> > csdept<+STUDENT > >
Output Grammar Tree : csclubmember < +student<@id @name> >

The rest of the SynTree specification is as follows:

Input SynTree : univ< club< +student<…>> csdept<+student[C1]<…> > >
Output SynTree : csclubmember< +student<…> >
Boolean Condition : C1: ./ @ id = ../../club/student/@id
Mapping Rule : *copy*: univ → csclubmember
copy: student[2] → student

The corresponding screen shot is as follows:

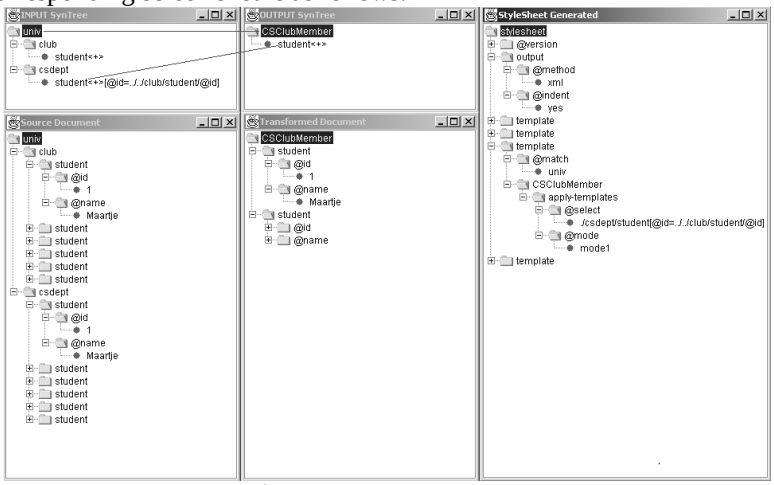

Fig. 6.3 Specifying intersection in Paired SynTrees System

The corresponding specification in XML format is as follows:

```
<?xml version="1.0"?>
<!DOCTYPE synTreeSpec SYSTEM "synTreeSpec.dtd">
<synTreeSpec>
   <iSynTree>
```

```
      <synTree  id="1" level="1" name="univ">
         <synTree id="2" level="2" name="club">
          <synTree id ="4" level="3" name="student" occurrence="oneOrMore"/>
          </synTree>
         <synTree id="3" level="2" name="csdept">
            <synTree id ="6" level="3" name="student" occurrence="oneOrMore"
                          sCondition = "@id=../../club/student/@id" />
            </synTree>
      </synTree>
   </iSynTree>

   <oSynTree>
      <synTree  id="1001" level="1" name="csclubmember">
         <synTree id ="1006" level="3" name="student" occurrence="oneOrMore" />
      </synTree>
   </oSynTree>

   <mRules>
      <rule   id="m1" name="copy">
         <sourceNode  id="1" name="univ"/>
         <destinationNode  id="1001" name="csclubmember"/>
      </rule>
      <rule   id="m2" name="copy">
         <sourceNode  id="6" name="student"/>
         <destinationNode  id="1006" name="member"/>
      </rule>

   </mRules>

</synTreeSpec>
```

The following stylesheet is generated by the translation algorithm:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <xsl:output method="xml" indent="yes"></xsl:output>
  <xsl:template match="text()|@*"></xsl:template>
  <xsl:template match="text()|@*" mode="mode1"></xsl:template>
  <xsl:template match="univ">
    <CsPetClubMember>
      <xsl:apply-templates select="./csdept/student[@id=../../club/student/@id]" mode="mode1"/>
    </CsPetClubMember>
  </xsl:template>
  <xsl:template match="student" mode="mode1">
    <student>
      <xsl:for-each select="./@* | ./node()">
        <xsl:copy-of select="."></xsl:copy-of>
      </xsl:for-each>
    </student>
  </xsl:template>
</xsl:stylesheet>
```

In the current implementation, we also support transformations requiring the difference of two or more node sets, and simple aggregation functions over selected nodes.

6.4 Observations – potential optimization

In this section, we present a few experiments to indicate some potential optimizations. The experiments were carried out in Windows XP professional running on a 600Mhz PC with 448M RAM. Our SynTree software was compiled and executed under SUN JDK1.3.1.

6.4.1 Number of template calls in XSLT stylesheet

In the algorithm, each construction template is used to build up just one node type. If we can combine templates to form one template with the same construction capability, then we can reduce the template calls during the transformation process, and thereby reduce the total running time. For example, the construction for the leaf nodes of one common parent can be combined together and put into the common parent's construction template, which results in fewer construction templates, thus reducing the template calls during the actual XSLT transformation process.

Suppose our output SynTree is a complete binary tree, which means there are as many leaf nodes as internal nodes. By applying the above technique, the number of templates generated will be reduced by half. For more general cases, if each node has more than one child, by applying the above technique, the number of templates generated will be reduced at least by half.

We use a trivial transformation (renaming symbols) to illustrate the performance gain:

InputSynTree : doc< +b1<c1 c2> +b3<c3 c4> >
Output SynTree : newDoc< +bb1<cc1 cc2> +bb3<cc3 cc4> >
Mapping rule : *copy*: doc → newDoc
copy: b1 → bb1
copy: b3 → bb3
copy: c1 → cc1
copy: c2 → cc2
copy: c3 → cc3
copy: c4 → cc4

We use one input data file of size 226KB (one *doc* element, one thousand *b1* elements and ten thousand *b3* elements) and test for 30 times using the Xalan XSLT engine. With the default approach, the algorithm generates seven templates for the corresponding seven output node types. The average execution time of this set of templates is roughly 1.48 seconds with an average deviation 0.12 seconds. With the optimized approach, the algorithm generates 3 templates, the average execution time is reduced to roughly 1.07 seconds with an average deviation 0.10 seconds, a considerable performance gain. The experiment result roughly confirms our reasoning above (see Figure 6.4).

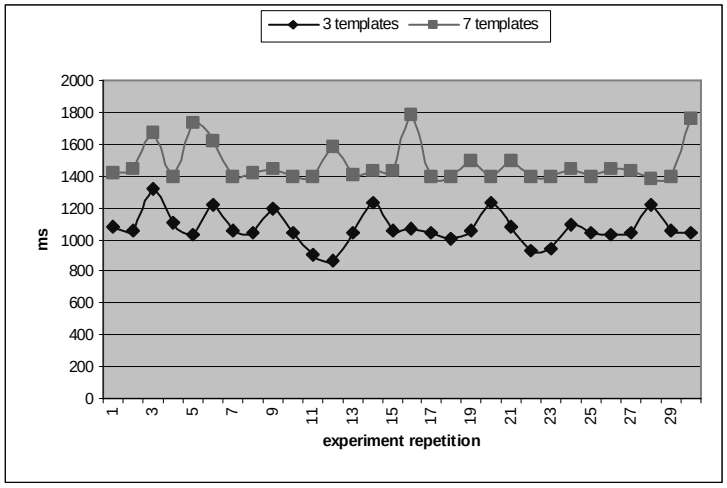

Fig. 6.4 Reduce Template Calls

6.4.2 Join on common attributes

In relational database systems, the size of the input tables affects how join operations can be optimized. Join operation can occur in the transformation process as well. In the core language, we support natural join on common attributes with exactly the same name only.

Our extended language will allow more general join operations which will take parameters to indicate the attributes and conditions to be used for join operation.

Consider a simple transformation requiring a natural join on the *student* nodes and *course* nodes:

InputSynTree : university< *student<@studentId @studentName @courseId>
*course<@courseId @courseName>>
Output SynTree : univ< *courseSelection<@studentId @studentName @courseId @courseName >>
Mapping Rule : *copy*: university → univ
copy: student, course → courseSelection

The straightforward translation for this specification is a nested-loop approach that iterates both *student* nodes and *course* nodes and constructs a new *courseSelection* node whenever the natural join condition is satisfied.

If the input file is large then the underlying operational language may not fetch the whole document into memory. In this situation, picking up the small subtree first is usually a good choice for performance, as proved for relational databases. We suspect this is true for the Xalan XSLT engine with SAX parser because, with their DTM (Document Table Model), it will not fetch a whole large document at once but rather only fetch the relevant portion whenever necessary (lazy fetching) [HM02]. To confirms this, we used a file of size 650KB in which there are just three *course* nodes and around thirteen thousand *student* nodes. We provide two specifications for the above transformation. One specification chooses the *student* nodes first whereas the other chooses the *course* nodes first. The two generated XSLT scripts are almost identical except for the nested loop structure, as shown in the following:

This one iterates *course* nodes first:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- this is hand maded version, not from translation algorithm -->
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <xsl:output method="xml" indent="yes"></xsl:output>
  <xsl:template match="text()|@*"></xsl:template>
  <xsl:template match="text()|@*" mode="mode1"></xsl:template>
  <xsl:template match="university">
    <univ>
        <xsl:variable name="VarS" select = "./studentEnrollment"/>
        <xsl:variable name="VarC" select = "./course"/>
        <xsl:for-each select = "$VarC">
        <xsl:variable name="VarC1" select "." />
          <xsl:for-each select = "$VarS">
            <xsl:variable name="VarS1" select = "."/>
            <xsl:if test="$VarS1/@cId=$VarC1/@cId">
              <CourseSelection>
                <xsl:element name="sid"><xsl:value-of select="$VarS1/@sId"/></xsl:element>
                <xsl:element name="sName"><xsl:value-of select="$VarS1/@sName"/></xsl:element>
                <xsl:element name="cId"><xsl:value-of select="$VarS1/@cId"/></xsl:element>
                <xsl:element name="cId"><xsl:value-of select="$VarC1/@cId"/></xsl:element>
                <xsl:element name="cName"><xsl:value-of select="$VarC1/@cName"/></xsl:element>
              </CourseSelection>
            </xsl:if>
          </xsl:for-each>
        </xsl:for-each>
  </univ>
 </xsl:template>
</xsl:stylesheet>
```

The second one iterates *student* nodes first:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- this is hand maded version, not from translation algorithm -->
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <xsl:output method="xml" indent="yes"></xsl:output>
  <xsl:template match="text()|@*"></xsl:template>
```

```
 <xsl:template match="text()|@*" mode="mode1"></xsl:template>
 <xsl:template match="university">
  <univ>
      <xsl:variable name="VarS" select = "./studentEnrollment"/>
      <xsl:variable name="VarC" select = "./course"/>
      <xsl:for-each select = "$VarS">
      <xsl:variable name="VarS1" select = "."/>
        <xsl:for-each select = "$VarC">
        <xsl:variable name="VarC1" select ="." />
          <xsl:if test="$VarS1/@cId=$VarC1/@cId">
            <CourseSelection>
              <xsl:element name="sid"><xsl:value-of select="$VarS1/@sId"/></xsl:element>
              <xsl:element name="sName"><xsl:value-of select="$VarS1/@sName"/></xsl:element>
              <xsl:element name="cId"><xsl:value-of select="$VarS1/@cId"/></xsl:element>
              <xsl:element name="cId"><xsl:value-of select="$VarC1/@cId"/></xsl:element>
              <xsl:element name="cName"><xsl:value-of select="$VarC1/@cName"/></xsl:element>
            </CourseSelection>
          </xsl:if>
        </xsl:for-each>
    </xsl:for-each>
  </univ>
 </xsl:template>
</xsl:stylesheet>
```

The difference in running time between the above two scripts is significant. The first template runs for less than twenty seconds whereas the second one runs for more than four hundred seconds, which confirms that choosing the smaller set for the outer loop is important for efficient execution.

6.4.3 Keys in XSLT

Performance can be further improved by using an index if indexing is supported by the underlying operational language. Current XSLT implementations seldom support an index so it is not a choice when XSLT is the target language for the translation algorithm. But XSLT provides keys for efficient fetching when there is a cross-reference in an XML document. For example, a user can define a key on the attribute of an element type, then fetch a subset of those elements later according to the predefined key. If the implementation of XSLT is capable of dealing with keys efficiently, defining a key should be a good choice to achieve better performance.

6.4.4 Subtree copies

BizTalk Mapper is a software tool that uses a similar approach to generate an XSLT script from a specification [Biz02]. The GUI allows users to draw *link*s, each of which specifies a mapping rule corresponding to a node copy. Figure 6.5 shows a screen shot from BizTalk Mapper. In our language, users can specify the above transformation in exactly the same way as in BizTalk Mapper, but we also allow users to specify it in a more efficient way by using *subtree* copy. Figure 6.6 shows our corresponding specification which uses subtree copy.

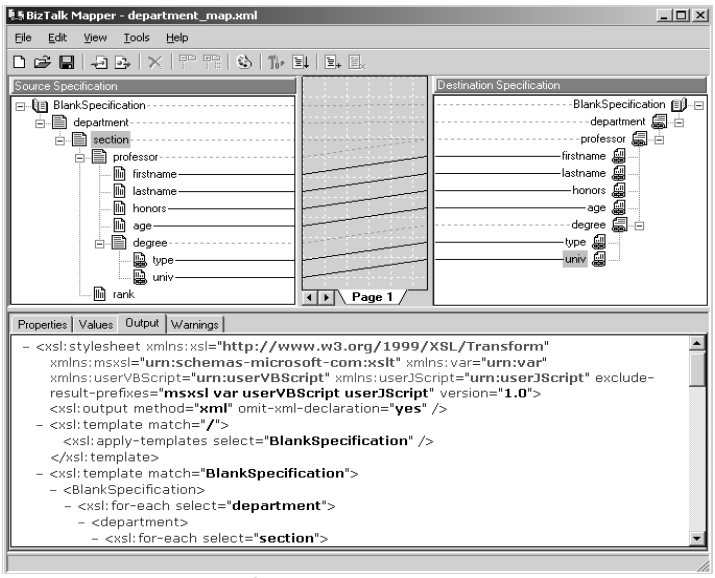

Fig. 6.5 Specifying Subtree Copy in BizTalk Mapper

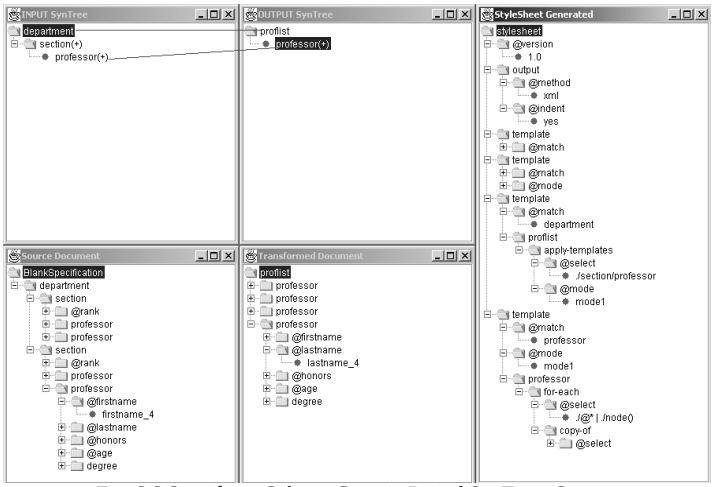

Fig. 6.6 Specifying Subtree Copy in Paired SynTrees System

Thus BizTalk Mapper forces users to specify all the mappings for all the descendants to mimic a subtree copy, which is quite inconvenient and the generated XSLT script is much less efficient since it explicitly copies every node. In our approach, our generated XSLT script uses <xsl:copy-of> to copy the whole subtree, resulting in more succinct specification and efficient code. We use Xalan to execute both stylesheets ten times to transform a source xml file of five hundred *section* elements and one thousand *professor* elements. The average running time for the stylesheet generated by BizTalk is 1.09 seconds whereas the stylesheet generated with subree copies requires running time of only 0.67 seconds on average.

7. FUTURE WORK AND CONCLUSION

Prior to developing paired SynTrees, we examined several approaches for specifying transformations for structured documents. None of them seemed suitable as a high-level specification language: some are too operational in nature and others can describe local

transformations only. We therefore propose a new approach, paired syntax tree templates, which is both descriptive and expressive.

Many XML query languages, such as XQuery and XML-GL, provide transformation capabilities to varying degrees. Similar to the XSLT solution, the XQuery solution for the ``PARTS'' Use Case example [CFMR01] also uses a recursive function:

```
<parttree>
    FOR $p IN //part[NOT @partof]
    RETURN one_level($p)
</parttree>

FUNCTION one_level($p element)  RETURNS element
{
  <part>
    $p/@partid
    $p/@name
    FOR $s IN //part[@partof=$p/@partid]
    RETURN one_level($s)
  </part>
}
```

Both solutions explicitly specify in which order to carry out the transformation (via recursive calls), and in this sense they are clearly operational rather than descriptive.
In XQuery, a program to perform even a simple change to the structure, such as renaming of a single nonterminal node or moving a small set of nodes, also involves recursively rebuilding much of the tree. Let us return to example 3.5.1, which specifies a simple transformation that only switches *lastname* and *firstname* of the element *professor*. The corresponding XQuery solution is as follows:

```
<proflist>
  FOR $varDept IN ./department
  RETURN
  <department>
    {$varDept/@title}
    <head>
      FOR $varOffi IN $varDept/head/official
      RETURN
      <official>
        {$varOffi/title}
        {$varOffi/rank}
        FOR $varProf IN $varOffi/professor
        RETURN
        <professor>
          {$varProf/firstname}
          {$varProf/lastname}
          {$varProf/degree}
          {$varProf/honor}
        </professor>
      </official>
    </head>
    FOR $varSect IN $varDept/section
    RETURN
    <section>
      {$varSect/rank}
      FOR $varProf1 IN $varSect/professor
      RETURN
      <professor>
          {$varProf/firstname}
          {$varProf/lastname}
          {$varProf/degree}
          {$varProf/honor}
```

```
      </professor>
    </section>
          </department>
</proflist>
```

Such programs are neither efficient nor concise as a specification mechanism for structural transformations.

We believe that a separate, more dedicated transformation language can complement a query language and that paired SynTrees is such a language. Any operational language supports the following functions can be the target language: navigation and pattern matching, basic tree construction and duplicate elimination, sorting, and aggregation.

It is worthwhile to investigate how to translate Paired SynTree specifications into XQuery.

We intend to extend the core language to cover more complex transformation. One such extension is to define the *aggregate* operator as a higher-order function:
aggregate (*f1*, *f2*(*n*)): where the operator will take two parameters: *f1* refers to those simple aggregation functions defined in its original form. *f2*(*n*) is a user-defined function that returns a single value based on the input *n*-SynTree. In its original form, *f1* is applied to current input nodes; in the extended form, *f1* is applied to the results of the second function, which gives considerable expressive power.

For example, let us define *f1* as concat(",") which concatenates multiple strings with "," as the separator, and define *f2*(*n*) as *extract-text-of-subtree*(*n*) which is supported in operational language such as XSLT and XQuery. Now we can specify a transformation as follows:

```
Input Grammar Tree    : department<+professor<first last> +student<first last> +program >
Output Grammar Tree   : department<professorList studentList +program>

Input SynTree   : department<+professor<first last> +student<first last>… >
Output SynTree  : department<proessorList studentList…>
Mapping         : aggregate(concat(",",)extract-text-of-subtree(./professor)):professor → professorList
                  aggregate(concat(","), extract-text-of-subtree(./student)):student → studentList
```

Another extension is to introduce *join* operator with parameters to represent more general join operations.

Finally, our ongoing research will continue to study various optimization potentials to incorporate into our template generating algorithm.

Reference:

[ABC+98] Vidur Apparao, Steve Byrne, Mike Champion, Scott Isaacs, Ian Jacobs, Arnaud Le Hors, Gavin Nicol, Jonathan Robie, Robert Sutor, Chris Wilson, Lauren Wood. Document Object Model (DOM) Level 1 specification. Version 1.0. W3C Recommendation, October 1998. http://www.w3.org/TR/REC-DOM-Level-1

[ABS00] S.Abiteboul, P.Buneman, and D.Suciu. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann, 2000.

[AMN+01] Noga Alon, Tova Milo, Frank Neven, Dan Suciu, and Victor Vianu. XML with data values: typechecking revisited. In *Symposium on Principles of Database Systems*, 2001.

[AU72] A.V. Aho and J.D. Ullman. *The theory of Parsing, Translation, and Compiling, Vol 1: Parsing*. Prentice-Hall (1972).

[BBC+01] Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernandez, Michael Kay, Jonathan Robie, Jérôme Siméon. XML Path language (XPath) 2.0. W3C Working Draft 2.0 December 2001. http://www.w3.org/TR/xpath20.

[BCF+01] Scott Boag, Don Chamberlin, Mary F. Fernandez, Daniela Florescu, Jonathan Robie, Jérôme Siméon, Mugur Stefanescu. XQuery 1.0: an XML query language. W3C Working Draft 2.0 December 2001. http://www.w3.org/TR/xquery.

[BDHS96] Peter Buneman, Susan Davidson, Gerd Hillebrand, Dan Suciu. A query language and optimization techniques for unstructured data. *Proceedings of ACM-SIGMOD International Conference on Management of Data*, 1996.

[BE01] Wes Biggs, Harry Evans. Simplify XML programming with JDOM. May, 2001. http://www-106.ibm.com/developerworks/java/library/j-jdom/

[BMD01] Anne Brüggemann-Klein, Makoto Murata, and Derick Wood. Regular tree and regular hedge languages over unranked alphabets: Version 1. *HKUST Theoretical Computer Science Center Research Report*: HKUST-TCSC-2001-05, 2001.

[BMN00] Geert Jan Bex, Sebastian Maneth, Frank Neven. A formal model for an expressive fragment of XSLT. *Information Systems*, 27(1): 21-39 (2002)

[BPSM00] Tim Bray, Jean Paoli, C.M. Sperberg-McQueen, Eve Maler. Extensible markup language (XML) 1.0 (second edition). W3C Recommendation 6 October 2000. http://www.w3.org/TR/1999/REC-xml.

[Beh00] Ralf Behrens. A grammar based model for XML schema integration. *Brithish National Conf. On Databases*, pp. 172-190, 2000.

[CCD+00] Stefano Ceri, Sara Comai, Ernesto Damiani , Piero Fraternali, Stefano Paraboschi, Letizia Tanca. XML-GL: a graphical language for querying and restructuring XML documents. *Proc. of WWW8, Toronto, Canada* (1999). http://www8.org/w8-papers/1c-xml/xml-gl/xml-gl.html

[CFMR01] Don Chamberlin, Peter Fankhauser, Massimo Marchiori, Jonathan Robie. XML query use cases. W3C Working Draft 20 December 2001. http://www.w3.org/TR/xmlquery-use-cases.

[CP90] James R. Cordy, Eric Promislow. Specification and qutomatic prototype implementation of polymorphic objects in TURING using the dialect processor. *Proc. IEEE International Conference on Computer Languages*, New Orleans (1990).

[CIDX] The chemical industry data eXchange. http://www.cidx.org, as at April 2002.

[CML97] P. Murray-Rust. Chemical markup language. World Wide Web Journal, 135-147 (1997). http://www.xml-cml.org.

[Cla99] James Clark. XSL transformations (XSLT) version 1.0. W3C Recommendation 16 November 1999. http://www.w3.org/TR/1999/REC-xslt-19991116.

[Cla99a] James Clark. XML path language (XPath) Version 1.0. W3C Recommendation 16 November 1999. http://www.w3.org/TR/1999/REC-xpath-19991116.

[Cla99b] James Clark. XSLT in perspective. Slides of a talk on XSLT, July 1999. http://www.jclark.com/xml/xslt-talk.htm.

[DFF+99] A.Deutsch, M.Fernandez, D.Florescu, A.Levy, D.Maier, and D.Suciu. Querying XML data. *Data Engineering Bulletin*, 22(3):10-18, 1999.

[FFK+98] M. Fernandez, D. Florescu, J. Kang, A. Levy, and Dan Suciu. Catching the boat with strudel: experience with a web-site management system. *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pp.414-425 (1998).

[GS84] Ferenc Gécseg, Magnus Steinby. *Tree automata*. Akadémiai Kiadó, Budapest (1984).

[GT87] G. Gonnet and F. Tompa. Mind your grammar: a new approach to modelling text. *International Conference on Very Large Data Bases* (*VLDB'87*), pp. 339-346, Brighton, England, 1987.

[HM02] Jason Hunter, Brett McLaughlin. JDOM API Javadoc materials, 2002. http://www.jdom.org/docs/apidocs

[KP96] E.Kuikka, M.Penttonen. Transformation of structured documents. *Processing of Structured Documents Using a Syntax-directed Approach*. Ph.D. thesis, Computer Science and Applied Mathematics, University of Kuopio (1996).

[KPPM84] S.E.Keller, J.A.Perkins, T.F.Payton, S.P.Mardinly. Tree transformation techniques and experiences. *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction.* SIGPLAN Notices 19(6), (1984).

[Kay01] Michael Kay. XSL Transformations (XSLT) Version 2.0. W3C Working Draft 20 December 2001. http://www.w3.org/TR/xslt20.

[Lal77] W.R. Lalonde. Regular right part grammars and their parsers. *Communications of the ACM*, 20(10):731-741 (1977).

[Lin97] Greger Lindén. Structured document transformations. *Report A-1997-2.* CS Department of University of Helsinki, Finland. (1997)

[MAG+97] J.McHugh, S. Abiteboul, R. Goldman, D.Quass, and J.Widom. Lore: a database management system for semistructured data, *SIGMOD Record* 26(3):54-66 (1997).

[MLM01] Makoto Murata, Dongwon Lee and Murali Mani. Taxonomy of XML schema languages using formal language theory. *Extreme Markup Languages*, Montreal, Canada, August 2001.
[MSV00] Tova Milo, Dan Suciu, Victor Vianu. Typechecking for XML transformers. In *Proceedings of the ACM Symposium on Principles of Database Systems* (PODS), pp11-22 (2000).
[Meg00] D. Megginson. SAX 2.0: The simple API for XML, May 2000. http://www.megginson.com/SAX/index.html and http://www.saxproject.org (latest)
[Mur96] Makoto Murata. Transformation of documents and schemas by patterns and contextual conditions. Lecture Notes in Computer Science, 1293:153-169(1997). Also appears in PODP(1996).
[Mur98] Makoto Murata. Data model for document transformation and assembly (extended abstract). *Principle on Digital Document Processing* (1998).
[Mur00] Makoto Murata. Hedge automata: a formal model for XML schemata. Web pages, 2000. http://www.horobi.com/Projects/RELAX/Archive/hedge_nice.html.
[OMTB02] Dan Olteanu, Holger Meuss, Tim Furche, and Francois Bry. XPath: looking forward. In *Workshop on XML-Based Data Management* (XMLDM), 2002.
[ST99] Airi Salminen, Frank W. Tompa. Grammars++ for modelling information in text. *Information Systems*, 24(1):1-24 (1999).
[Suc02] Dan Suciu. The XML typechecking problem. *SIGMOD Record* 31(1):89-96 (2002).
[Tha67] J.W. Thatcher. Characterizing derivation trees of context-free grammars through a generalization of finite automata theory. *Journal of Computer and System Sciences*, 1:317-322, 1967.
[Wad00] Philip Wadler. A formal semantics of patterns in XSLT and XPath. *Markup Languages: Theory and Practice*, 2(2):183-202, 2000.

Appendix : Top-down Translation Algorithm

```
topDownProcess(xsltStylesheet, oSynTree.root)
{

ContextBinding currentBindings ← setCurrentBindings(oSynTree.root);
XSLTTemplate template = generateTemplate(currentBindings);
Queue queue = new Queue();

//Adjust template by inserting more construction or apply-template rules if necessary
for (childNode ∈ currentOutputNode.getChildren())
{
  mappingName ← childNode.getMapping().getName();  //copy or aggregate with parameters
  inputNodes[] ← currentMapping.getInputNode();  //a sequence of input nodes
  for (int i=0; i<currentInputNodes.length; i++)//one path for each inputNode
    relativePaths[i] ← calculateXPaths(currentInputNodes[i], inputNodes);

    switch(mappingName)
  {
    case copy: //leave it for next recursive call to construct
      queue.enQueue(chileNode);
      template.insertApplyTemplatesElement(template, relativePaths);
      break;

        case add:  //add a constant value to an output leaf node
      if ( ! childNode.hasChild() )   //childNode is a leaf node in oSynTree
        template.insertAddConstructionElement(template, relativePaths);
      else  //this scenario should not appear
        report error("no further downward processing allowed after add");
      break;

        case update:  //update a input leaf node with given value
      if ( ! childNode.hasChild() )   //childNode is a leaf node in oSynTree
        template.insertUpdateConstructionElement(template, relativePaths);
      else  //this scenario should not appear
        report error("no further downward processing allowed after update");
```

```
        break;

      case aggregate:  //max, min, average, sum, count…
        if ( ! childNode.hasChild() )   //childNode is a leaf node in oSynTree
          template.insertAggregateConstructionElement(template, relativePaths);
        else  //this scenario should not appear
          report error("no further downward processing allowed after aggregation");
        break;

          default:
        report error("no such operation");
        break;
      }//endSwitch
    }//endFor

xsltStylesheet.addTemplate(template);

//notice all the input children without mapping rule will be implicitly deleted during this process

//go ahead with children mappings if necessary
while( !queue.isEmpty())
{
      childNode ← queue.deQueue();

      //recursively buildup more templates for descendents whenever possible
      topDownProcess(xsltStylesheet, childNode);

}//endWhile

}//endFunction


//Some private methods used by topDownProcess()

//generate a set of bindings for a given output SynTree node
ContextBinding setCurrentBindings(oNode)
{
    ContextBinding binding= new  ContextBinding();
      binding.mapping ← oNode.getMapping();
      binding.oNode ← oNode;
      binding.iNodes[] ← mapping.getInputNodes();
      binding.constructionCondition ← oNode.getCondition();
      binding.selectionCondition ← iNode.getCondition();
      return binding;
}

XSLTTemplate generateTemplate(bindings)
{
    XSLTTemplate template = new XSLTTemplate();

    //use bindings to get construction condition, then construct current output
      template.insertSelfConstructionElement(bindings);

    //if leaf node represents a subtree, copy the whole subtree as well
    if (bindings.oNode.isLeafComplexNode())
      template.insertSubtreeCopyElement();
}

RelativePaths calculateXPaths(currentInputNode, inputNodes[])
{
    //for every node i in inputNodes[], calcaulate the path
    //then union all of the valid path
    return union calculateXPath(currentInputNode, inputNodes[i]);
}
```

```
RelativePaths calculateXPath(iRoot, iNode)
{
  //anchor point is the nearest ancestor for currentInputNode and  the two node
  Anchor anchor = null;

  //searching for the anchor point in a bottom-up manner
  while ( anchor == null)
    {
      //fail to find anchor, error case
      if (iRoot.depth() == iNode.depth() ==1)    break;

      //bottom-up to search for the anchor
        if (iRoot.depth() == iNode.depth())
      {
        iRoot.goUpOneLevel(); //go up to its parent, record its move meanwhile
        iNode.goUpOneLevel(); //go up to its parent
        break;
      }
      else if (iRoot.depth() < iNode.depth())
      {
        iNode.goUpOneLevel();  //go up to its parent
      }
      else if (iRoot.depth() > iNode.depth())
      {
        iRoot.goUpOneLevel();
      }

      //check if anchor is found
      if (iRoot.equals(iNode))
      {
        anchor = iRoot;
        break;
      }
    }//endwhile

    //if anchor found, generate the path going from iRoot to iNode
    if (anchor !=null)
      return generateXpath(iRoot.getTrace(), anchor, iNode.getTrace());
    else
        return "";

}//endfunction
```