Scenario-Based Connector Optimization
An XML Approach

Welf Löwe[1] and Markus Noga[2]

[1] Växjö universitet, MSI, Software Tech. Group, 351-95 Växjö, Sweden, welf.lowe@msi.vxu.se
[2] Universität Karlsruhe, Program Structures Group, Adenauerring 20a, 76133 Karlsruhe, Germany, noga@ipd.info.uni-karlsruhe.de

Abstract. Software components can be connected by XML processing pipelines, which may perform adaptations. In our model, individual pipeline stages serialize source data structures to XML, perform one or multiple XSL transformations, transport the message to its destination and finally deserialize it to target data structures. Implementation of this model is open to optimizations. The present paper discusses two such optimizations: symbolic execution and lazy evaluation.

1 Introduction

With new problems at hand, management used to ask, "Build or Buy" a solution? Today, this question only applies to smallish problems. For entire systems, "Buy, Build and Integrate" has become the method of choice. That is, suitable components are bought, the missing remainder is built and the entire system is subsequently integrated.

This strategy separates system integration spatially and temporally from component design. Mismatches between components from different vendors, and between bought and custom-built components inevitably result. Thus, adaptation of components has become essential.

In [8], we presented a lightweight XML middleware architecture that explicitly addresses the adaptation problem. We also described basic optimizations like generator usage and intermediate structure omission. This article covers major new optimizations for the system, symbolic execution and lazy evaluation. Both are scenario-based: the former depends on the adaptation scenario, the latter on the communication scenario at hand.

The next section briefly revisits the middleware architecture in [8] and basic technologies it employs. Section 3 covers symbolic execution, while section 4 deals with lazy evaluation. Section 5 summarizes our results and outlines directions for future work.

2 Related Work

The first subsection summarizes basic XML technologies, most prominently XPath [17] and XSLT [20]. The second subsection briefly revisits our middleware architecture [8].

2.1 Basic Technologies

XML is a well-known storage format for depth-first preorder traversals of trees [16]. There are various type description languages for XML tree nodes, among them DTDs and the more expressive XML Schemas [18,19].

The Document Object Model, or DOM, provides an abstract, non-typed interface to XML tree nodes [15]. Its operations realize basic tree traversal and manipulation operations. E.g., child, parent and sibling nodes can be accessed, as well as attributes.

XPath is a query language for XML document trees [17]. It is inspired by the concept of path languages. XPath expressions are sequences of steps. Each step projects a set of source nodes onto a new set, which is subsequently filtered.

Syntactically, steps consist of an optional axis, a selection and optional additional filters. Axes determine the projection direction: onto child nodes, parent nodes, siblings, descendants, ancestors etc. If an axis is not specified, the child axis applies by default. Selection filters the projected set by node name or type. E.g., the Xpath expression *parent:A/B/C* selects C-children of B-children of A-parents of the current node. The optional additional filters can be arbitrarily complex predicates. Additionally, there are wildcards for element names "*" and entire path fragments "//".

XSL transformations, or XSLTs, perform maps on XML trees [20]. They are inspired by rewrite systems: an XSLT is an ordered set of rules. They consist of an applicability test, called match expression, and a body, which may contain output statements, recursive rule applications and some additional elements of functional programming.

Rules operate on a current node. They are checked for applicability in order of their definition. The body of the first matching rule is executed. At points of recursive rule applications, a sequence of new current nodes is selected according to a select expression. These new nodes are processed in the same manner. The transformation is initiated by applying the rules to the tree root.

In practice, the match expression is an XPath expression without axes. For a positive match, it must return the current node if applied to the current node or any ancestors thereof. Select expressions in recursive rule applications are also specified as XPath expressions. In this paper, we simplify the rule body to output statements and a single recursive rule application. We disregard the inner structure of output statements.

2.2 Connection and Adaptation

For the purpose of this paper, we define components to be software artifacts with typed input and output ports linked by communication channels called connectors. The notions of ports and connectors are known from architecture systems [13,3]. The problems solved by connectors are wide spread in general; [10] gives an overview. They may be as complex as most components, and thus require the same amount of consideration in design and implementation, cf. [14]. On the design level, we have explicit connector entities with a formal semantic allowing for consistency checking, cf. [1]. On the implementation level, connectors are executable first class entities allowing for reuse and composition, cf. [4].

However, we focus on special connectors with limited problems to solve. We assume connectors to be

– point-to-point data paths with in- and out ports known at compile time,
– executable in the sense that they implement stateless data transformation functions (adaptations) also known at compile time, and
– language independent as they do not require sender and receiver component to be implemented in the same programming language.

In this limited scenario, there is no need for explicit connector objects in the production code. Instead, we try to eliminate them to increase the system performance. Therefore, connector code fragments can be generated from the connector specification and merged into the sender and receiver component code. This meta-programming technique is known as "grey-box-connection", cf. [2] where it is used to adapt method calls. It is generalized in [6]. That work shows the generation of connectors adapting the synchronization and activity of components using abstract specifications. The generated adapter fragments are woven into the sender and receiver components and thus disappear from the production code as first class objects.

In the present paper, we use a similar approach for connectors adapting the exchanged data. The required and provided parameters of a communication are specified with XML Schema specifications, the adaptation with XSLT scripts. The specific runtime environment and the generator tools are described below.

2.3 Middleware Architecture and Generator Tools

The middleware architecture in [8] builds on the above model. Components are software artifacts with typed input and output ports. Connectors can perform adaptations. The system is strongly typed and statically type safe.

At runtime, our middleware serializes output port data to an XML wire format. Adapting connectors perform XSLTs on the wire format. Input ports parse the wire format and reconstruct the corresponding object graphs. Fig. 1 shows adapted communication between two components at runtime.

The wire format is not generic, but derived from the port types using one of the schemes in [7] or [11], which map data types to DTDs and XML Schema, respectively. This approach preserves strong typing and static type safety in the XML representation.

As types are known at deployment time, our middleware analyzes the component sources at that point. Using a metaprogramming system, we generate

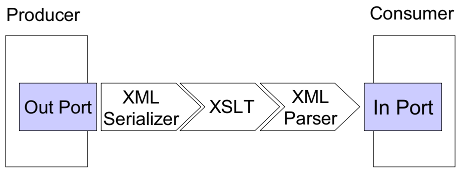

Fig. 1. Adapted communication between components.

specific wire format descriptions, as well as serializers, transformers and deserializers optimized for these formats. These fragments are woven into the component sources. [9] discusses code analysis and generation in detail.

3 Symbolic Execution

Symbolic execution jointly preprocesses transformations and document types given as DTDs or Schemas. If static analysis can guarantee that parts of the document type are never visited by a transformation, serialization may safely omit those parts of the document. Conversely, if static analysis detects that a rule cannot match on documents of the given type, that rule test can safely be omitted. The efficacy of these omissions depends on the adaptation scenario. Their impact is obviously the largest for filtering adaptations which omit large amounts of data.

We start with an overall view. Static preprocessing initially clones generic programmatic representations of the rules for each node type and possible grammatical context. We then eliminate inapplicable rules based on match expressions. Using tight conservative approximations with regular grammars, the select statements are then refined to precisely defined document traversals. Invocations of locally inapplicable match rules are removed. Together, this yields a conservative approximation of the document type parts actually visited.

Now, we are ready to discuss the process in more detail. Given routines implementing the XSLT `match` and `select` operations described in 2.1, each rule r with match m_r, output o_r and select expression s_r can be expressed as in Algorithm 1, which operates on a current node `n`. The `main` routine performs the entire XSL transformation when invoked on the document root node `d`.

If our document nodes are typed, unlike DOM, we can clone these algorithms with respect to node types. For each node type E we define a class $\texttt{class}_E$. In object-oriented terms, the `transform` m_r functions become methods operating on the current object `this` instead of a parameter node `n`, cf. Algorithm 2. Note that the recursive invocation of `transform` is now restricted to transform methods actually defined in the class of the target node `n'`, cf. the `inner` loop.

Now we are ready to specialize the $\texttt{class}_E$. We initially analyze match expressions. As stated in 2.1, a match expression matches a given node if the corresponding XPath expression returns the node when applied to the node or any of its ancestors. Thus, based on the node type, we can determine whether

Algorithm 1 (XSL Transformation Schema)

```
boolean transform$_r$(Node n){
   if match(n,$m_r$){
      output(n,$o_r$);
      NodeList nl := select(n,$s_r$);
      for (Node n' in nl) {
         inner: for ($r'$ in $rules$) {
            if transform$_{r'}$(n') break inner;
         }
      }
      return true;
   }
   return false;
}
void main(Document d){
   for ($r'$ in $rules$) {
      if transform$_{r'}$(d.rootNode) break;
   }
}
```

Algorithm 2 (Specialized XSL Transformation Schema)

```
class$_E$ is
   boolean transform$_r$() {
      if this.match($m_r$) {
         this.output($o_r$);
         NodeList nl := new NodeList := this.select($s_r$);
         outer: for (Node n' in nl) {
            inner: for ($r'$ in n'.$rules$) {
               if n'.transform$_{r'}$() break inner;
            }
         }
         return true;
      }
      return false;
   }
   boolean transform$_{r_2}$(){ ...
```

a given match expression must, may, or must not match. Algorithm 5 in the appendix defines this analysis. As they are guaranteed not to be invoked, we can safely eliminate the transformation methods for all must not matches. Similarly, all rules defined later than a must match can be eliminated.

If we clone a class for different document type contexts, may matches can turn into must or must not matches. This reduces the number of methods. We clone as long as the method count decreases. This procedure terminates even for recursive transformations due to the bounded size of match expressions and the bounded number of methods per class.

We turn our attention to the select expressions. The `select` routine implements the selection of nodes for a given XPath expression and context node. It proceeds by steps, projecting the current node sequence along the given axis, e.g., to children, and filtering down the projection by name, type and additional criteria. `select` returns a sequence of nodes `ns` ordered in document order, to which recursive transformations are applied.

We conservatively approximate `ns` for each context node type E and all contexts of E with a set of formal languages, one per context. If the select expression contains only *this*, *child* and *descendants* axes, our approximation is context *in*sensitive and the set contains exactly one langauge.

First, we consider individual axes. Let E be a node type and a be an axis. Let $\mathit{approx}(E, a)$ be the context-free language that conservatively approximates the mapping of E along a given the information in the document type. For the default axis *this*, e.g., $\mathit{approx}(E, \mathit{this}) = \{r \rightarrow E\}$ with starting symbol r. The algorithms 4 in the appendix compute approximations for nontrivial axes.

A step consists of an axis, a select expression and optional filters. If the select expression specifies a concrete element type E, we can specialize the above axis approximation for this step by replacing all terminals for element types $E' \neq E$ with ϵ. If filters are present, we similarly replace all terminals E with $E \mid \epsilon$ as they may be filtered out.

A select expression consists of multiple steps. We combine step approximations into select expression approximations by successively replacing element type terminals with the grammar rules for the respective next step. If that grammar defines the empty language, we effectively remove the corresponding terminals completely. Algorithm 6 in the appendix defines the approximation of steps and select expressions precisely.

If the approximation for a select expression is the empty language, recursive invocations of `transform` are not required. We may skip the `for` loops in Algorithm 2. In general, we could replace the `outer: for` loop by an acceptor for the selection sequence language and specialize the recursive invocations in the `inner: for` loop according to the rule set of the accepted element node type. However, the efficacy of these optimizations is quite low. Experience with database systems teaches that selection is by far the most expensive operation. We will therefore use the above approximations to optimize select operations.

The bigger the node set generated by projection on an axis, the more expensive it is to compute. Thus, *ancestor, descending, preceding, following* are generally more expensive than *parent, child, sibling.* We will attempt to replace expensive axes with cheaper operations.

A node can only be part of a selection path if its approximation language is non-empty. Otherwise, the final step in a selection cannot match, although intermediate steps may generate large node sets. With our analyses, we can safely skip their computation. Algorithm 7 in the appendix defines iterators searching only those nodes that are potentially selected. The sequence `ns` is replaced by such an iterator. The final result of all optimizations defined in this section is sketched in Algorithm 3.

Algorithm 3 (Optimized XSL Transformation Schema)

```
class$_E$ is
   boolean transform$_r$(){
      if this.match($m_r$){
         this.output($o_r$);
         Iterator a := new AxesIterator($axis(s_r)$); //Algorithms 7
         a.init(this);
         Iterator ns := new SelectionIterator(); //Algorithm 7
         ns.init(a,$selection\ path\ set(s_r)$);
         while ((n':=ns.next()) != null){
            for ($r'$ in n'.$rules$){
               if n'.transform$_{r'}$() break this loop;
            }
         }
         return true;
      }
      return false;
   }
   boolean transform$_{r_2}$(){ ...
```

4 Lazy Evaluation

If only stochastic data about access profiles are available, static analysis fails. Consider a common case: a component transfers documents to a viewer and requires adaptation. A human navigating the transformed document visits only fractions of it. However, we cannot statically determine *which* fractions.

In these stochastic cases, lazy evaluation is an alternative to eagerly transforming and transmitting all outgoing data. The producing component is lazy and transmits only a handle to the consumer. Upon actual access, the consumer requests the required fragments from the producer, who partially transforms the source and transmits the results.

The performance of this approach hinges on hardware scenarios and application profiles. Before we consider how the XML transformation pipeline allows for partial processing, we estimate the potential of the lazy approach with a model.

When comparing one large with many smaller messages, we must account for the lag l of a transmission in seconds and the throughput t of the channel in kBit/s. A message of size m kBits requires τ seconds where:

$$\tau(m) = \tau_{l,t}(m) = l + \frac{m}{t} \tag{1}$$

The application determines the data size m in kBits, the fraction f of the data required on the remote side and the number n of unique accesses, i.e. the number of fragments to transmit. With (1), we determine lazy and eager processing times

$$\tau_{\text{lazy}}(m, f, n) = n\tau\left(\left\lceil \frac{mf}{n} \right\rceil\right) \tag{2}$$

$$\tau_{\text{eager}}(m) = \tau(m) = \tau_{\text{lazy}}(m, 1, 1) \tag{3}$$

for a given hardware scenario (l, t).

Perhaps surprisingly, the tradeoff between eager and lazy processing is almost independent of the hardware scenario for a given data size. Solving $\tau_{\text{lazy}}(m, f, n) = \tau_{\text{eager}}(m)$ for a given m, we realize that only the product lt is relevant, which varies little between LAN, WAN and modem scenarios. In short, what counts is the amount of data that can be transmitted instead of waiting for the network.

Now that lazy evaluation is demonstrated to be effective, we still have to show that it applies to the XML processing pipeline. Let us initially assume simple connectors without transformation. Then, processing is limited to serialization, transport and reification of an object graph.

Let d_s and d_t, resp., of class D be the root of the source and target data graphs in question. On the remote side, D and recursively all depending classes are extended by a private attribute c_a and an access method m_a per attribute a of a class. c_a indicates complete transmission and reification of a. m_a checks c_a before access and triggers transport and reification of the object if necessary. Additionally, we add remote access stubs to the producing side.

Initially, we serialize a shallow of d_s, transport it to the consuming component and reify it to d_t. Whenever we initially access an attribute of d_t, transport and reification are triggered. Subsequent accesses are local.

Depending on the memory consistency model and on the communication semantics, we may have to deep copy d_s or block the execution of the source component. Those requirements are independent of the communication optimization, but outside our current focus. For a discussion, we refer to [6,5].

Objects accessed via different paths should be transported only once. This is guaranteed by the same bookkeeping approach used in complete depth-first transmission: the producer component maintains a hash table of serialized objects for the active session. The initial stub access (on the consumer side) retrieves an object o by triggering serialization of o (on the producer side). The producer retains an id, the XML mapping of the shallow of o and a hash table entry. The receiver maintains an array of deserialized objects by id.

Whenever the receiver accesses an object o' containing an alias to o, we find the the corresponding id in our hash table. Together with the shallow of o' we transmit the id of o. An receiver side access $o'.o$ does not trigger a communication as o has been reified already. A simple lookup in the array of deserialized objects with the id gets the required object. Figure 2 sketches this bookkeeping.

As our initial performance estimate shows, there are minimum transmission sizes for any given hardware scenario. Thus, instead of transmitting a single object shallow, we serialize and deserialize a copy of some level.

Finally, we consider the general case with transformations. For the sender component, very little changes – it is irrelevant if a remote component access or a transformation access triggers the serialization and transmission of some parts of the data structure in question. We only transmit the id together with the first serialization of an object.

Initially, we serialize a shallow of the root object d_s, and start the transformation until the remote root object d_t can be reified by the consumer. Depending

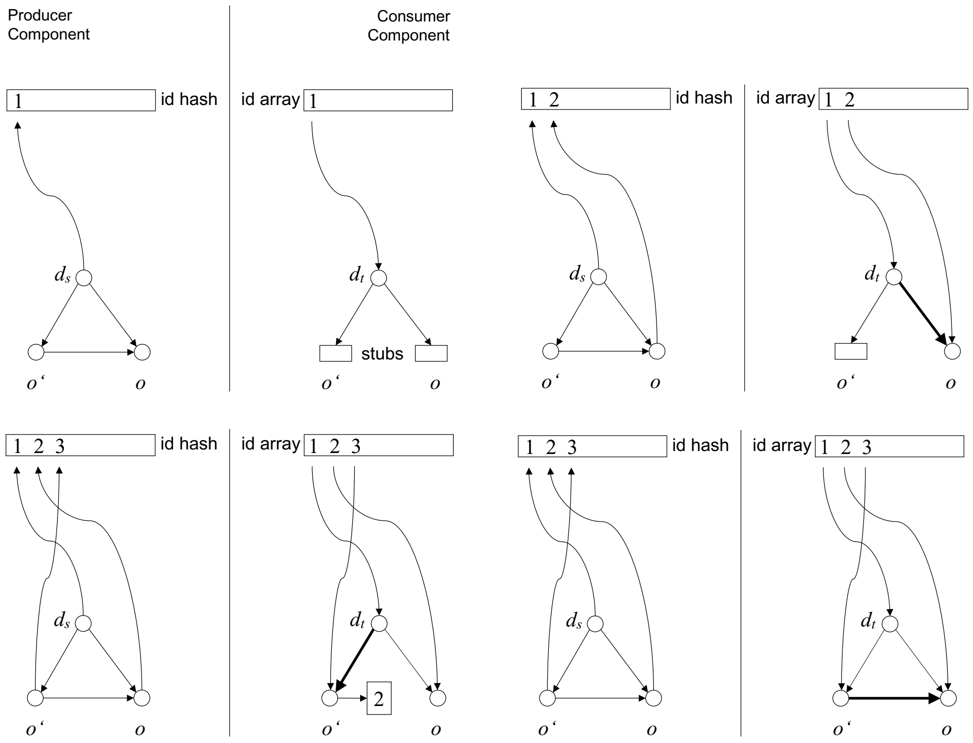

Fig. 2. Initial situation after root object transmission (top, left): d_s is serialized and refers to id 1, d_t is deserialized and accessible by id 1. After the first access to $d_t.o$ (top, right): o (id 2) is serialized and deserialized, respectively. After the first access to $d_t.o'$ (bottom, left): o' (id 3) is serialized and deserialized, respectively. Instead of a stub for the access to o, o' contains already the id of o (id 2). After the first access to $d_t.o'.o$, i.e. the second access to o (bottom, right): with a look up in the id array, id 2 is replaced by the actual reference to o.

on the transformation script, this may already trigger the serialization of some further parts of the sender side data structure.

Three observations lead to the lazy evaluation of the transformation: (i) XSL transformations are functional. Invocations of the `transform` method, cf. Algorithm 2 in the previous section, are free of side effects except output. (ii) XSL may traverse the input document in an arbitrary fashion, although depth-first is an important special case. They may cache intermediate results of any size, but the output document is written in depth-first order. (iii) Our mapping of data structures to XML traverses the data structures in depth-first order.

When we access an object o the first time, we trigger the transformations to reify it. The next accesses to o are local (as discussed above). The transformation is performed only partially: the recursive call to `transform` is stopped (and stored as a bound routine for later resume), whenever the `output`, cf.in

Algorithm 2, indicates that we do not generate the shallow of o but some descendants. We stop the main transformation process and store it as a bound method if the shallow of o is complete.

Observation (i) guarantees we can postpone the recursive decent, observations (ii) and (iii) ensure a transformation will produce few other objects than o. The candidates are objects accessible by an attribute of the current object (siblings of o) and objects accessible by an attribute of o (children of o). For those objects, we captured a bound routine that can be resumed to get the actual object if accessed the first time. If we try to access an object for which a bound routine was not captured, we resume the main transformation process.

5 Perspectives

Symbolic execution and lazy evaluation are highly promising optimizations. They both eliminate transmission overheads. In a sense, the approaches are complimentary: Symbolic execution applies to scenarios with static overhead, whereas lazy evaluation applies to scenarios with stochastic overhead.

We have started to implement lazy evaluation. Although transformations are not fully realized yet, initial measurements for lazy deserialization are highly promising. We are confident that benefits will continue to unfold.

Symbolic execution is still pending implementation. Measurements on our early XSLT compilers indicate that bookkeeping for node sets alone accounts for some 40% of total execution time. Replacing them with simple iterators and eliminating superfluous match tests should increase transformation performance by a factor of two [12].

Future work will focus on implementation and full evaluation of our optimizations. Moreover, we aim to determine worthwhile optimizations by automatic analysis of access profiles. Finally, we want to combine both approaches: lazy evaluation could benefit from splitting the global adaptation into a sequence of partial ones. To determine effective splits, we have to analyze the access profile as well. Each partial adaptation could again be optimized with a more precise approximation of the data structures and transformation rules involved.

References

1. Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, July 1997.
2. U. Aßmann, T. Genßler, and H. Bär. Meta-programming Grey-box Connectors. In *Proceedings of the 33rd TOOLS (Europe) conference*, 2000.
3. Len Bass, Paul Clement, and Rick Kazman. *Software Architecture in Practice.* Addison Wesley, 1998.
4. Stéphane Ducasse and Tamar Richner. Executable Connectors: Towards Reusable Design Elements. *ACM SIGSOFT*, 22(6):483 – 499, November 1997.
5. Dirk Heuzeroth, Thomas Holl, and Welf Löwe. Combining static and dynamic analyses to detect interaction patterns. In *IDPT*, 2002. (submitted to).

6. Dirk Heuzeroth, Welf Löwe, Andreas Ludwig, and Uwe Aßmann. Aspect-oriented configuration and adaptation of component communication. In Jan Bosch, editor, *Third International Conference on Generative and Component-Based Software Engineering, GCSE*, page 58 ff. Springer, LNCS 2186, 2001.
7. W. Löwe and M. Noga. Component communication and data adaptation. In *IDPT*, 2002.
8. W. Löwe and M. Noga. A lightweight xml-based middleware architecture. In *20th International Multi-Conference Applied Informatics, AI.* IASTED, 2002.
9. W. Löwe and M. Noga. Metaprogramming applied to web component deployment. In *ETAPS Workshop on Software Composition*, 2002.
10. Nikunj R. Mehta, Nenad Medvidovic, and Sandeep Phadke. Towards a Taxonomy of Software Connectors. In *International Conference on Software Engineering, ICSE 2000.* ACM, 2000.
11. M. Noga and W. Löwe. Data types and XML schema. *Journal of Markup Languages - Theory and Practice (to appear)*, 2002.
12. Tobias Schmitt-Lechner. *Entwicklung eines XSLT–Übersetzers.* Universtität Karlsruhe, IPD Goos, May 2001.
13. M. Shaw and D. Garlan. *Software Architecture in Practice – Perspectives on an Emerging Discipline.* Prentice Hall, 1996.
14. Mary Shaw. Procedure calls are the assembly language of software interconnection: Connectors deserve first-class status. In D.A. Lamb, editor, *Studies of Software Design, Proceedings of a 1993 Workshop*, pages 17–32. Springer, LNCS 1078, 1996.
15. *Document Object Model.* W3C, `http://www.w3.org/DOM/`, 2000.
16. *Extensible Markup Language (XML) 1.0.* W3C Recommandation, `http://www.w3.org/TR/1998/REC-xml-19980210`, 1998.
17. *XML Path Language.* W3C Rec., `http://www.w3.org/TR/xpath`, 1999.
18. *XML Schema Part 1: Structures.* W3C Recommendation 2 May 2001, `http://www.w3.org/TR/2001/REC-xmlschema-1-20010502`, 2001.
19. *XML Schema Part 2: Datatypes.* W3C Recommendation 2 May 2001, `http://www.w3.org/TR/2001/REC-xmlschema-2-220010502`, 2001.
20. *XSL Transformations (XSLT).* W3C Rec., `http://www.w3.org/TR/xslt`, 1999.

A Appendix with Algorithms

The following algorithms compute context free grammar rules that conservatively approximate the result of mapping E along the axis. We denote terminals representing element node types with capital letters.

The *child* axis of a node contains all its children in document order. Let cm be the content model for E, and $G(cm)$ the grammar rules generating cm, then:

$$approx(E, child) := G(cm) \tag{4}$$

Let $D ::= cm(\ldots, E, \ldots)$ denote a DTD rule defining D whose content model contains E. Then, the *parent* sequence is given by the following set productions with start symbol r:

$$approx(E, parent) := \{r \rightarrow D \mid \text{“}D ::= cm(\ldots, E, \ldots)\text{”} \in DTD - rules\} \tag{5}$$

Let $F(cm)$ be the finite acceptor for a regular context model cm and $G(F)$ the corresponding grammar productions. The *preceding-sibling* (*following-sibling*) axis contains siblings of the current node occurring before (after) in the document. The *ancestor* (*descendant*) axis of a node contains all transitive parents up to the root (transitive children). Algorithm 4 computes their approximations. More complex axes can be built from these simple ones.

Algorithm 5 conservatively estimates the matches for individual transformation rules. It symbolically executes the matches on the DTD, traversing the path expressions step by step starting with the last step e. $e.sel$ denotes the selection of this step, $e.axis$ the axis and $e.filter$ the presence of optional filter operations. The algorithm simultaneously traverses the DTD from the element node type E of the context node up to the root R. It checks if a certain path must, may or cannot match in a document conforming to the DTD.

Algorithm 6 approximates possible element node sequences of selections s_r relatively to a sequence of root nodes defined by some axes. In- and output sequences are given as grammar productions. The algorithm symbolically executes selection on the DTD, traversing the selection path step by step starting with the first step e. For each step, the productions R are updated. Node types filtered out by $e.sel$ are removed. If $e.filter$ marks the step as optional, all productions are marked as optional. Now, the grammar reflects the sequence of selections up to this step. If there are no steps left, we are done. Otherwise, we have to execute the next step symbolically on this sequence. Therefore, we compute a new root sequence by applying the next axis on each element. This is done by replacing the each remaining node types E with the axiom r_E of productions R_E corresponding to the next axis. Then we enter recursion.

If we hit a node of type E in the actual selection process at runtime, we can stop further searching if the symbolic execution of selection on the DTD indicates future selection steps will fail. I.e., we simply check if the sequence S_E generated by R_E is empty. This idea is exploited in the iterators defined in Algorithms 7. They select the root node sequences for the *ancestor* and *descendant* axes, respectively. Other axes iterators are computed analogously. Both algorithms compute the sequence in an initialization phase and store it in an internal container. Thereby they skip those element nodes n that can be excluded by the observations above, i.e. elements with a type $E = Type(n)$ where $S_E = \varepsilon$.

The finally iterator in Algorithm 7 computes a selection s_r composed of multiple steps and a given current node set. It proceeds step by step, starting with the first step e. For each step, nodes n in the current set are checked for conformance with $e.sel$. Also, nodes that failed the static execution, e.g., whose $S_{Type}(n) = \varepsilon$, are eliminated. If there are no steps left, the current set is the result of the selection and the algorithm terminates. Otherwise, each node is replaced with the result of iterating over it according to the next axis. Then we enter recursion.

Algorithm 4 (Axis approximations)

```
preceedingSibling( element node type $E$ ) is
   compute $F = \{F(cm(\ldots, E, \ldots)) \mid \text{“}D ::= cm(\ldots, E, \ldots)\text{”} \in \mathit{DTD\text{-}rules}\}$
   forall $f \in F$ {
      mark all states with outgoing $E$ transition as $E$ states;
      delete all states not reaching an $E$ state and their transitions;
      set all $E$ states final;
      delete unreachable states and their transitions;
   }
   return $G(\bigcup F)$

followingSibling( element node type $E$ ) is
   compute $F = \{F(cm(\ldots, E, \ldots)) \mid \text{“}D ::= cm(\ldots, E, \ldots)\text{”} \in \mathit{DTD\text{-}rules}\}$
   forall $f \in F$ {
      mark all states with incoming $E$ transition as $E$ states;
      create a new starting state and add $\varepsilon$ transitions to all $E$ states;
      delete unreachable states and their transitions;
   }
   return $G(\bigcup F)$

ancestorOrSelf( element node type $E$ ) is
   nodeTypes := $\{E\}$
   rules := $\{r_E \to \varepsilon\}$
   loop
      forall $X \in$ nodeTypes {
         add $D$ to nodeTypes if $\text{”}D ::= cm(\ldots, X, \ldots)\text{”} \in \mathit{DTD\text{-}rules}$
         add $r_D \to X\ r_X$ to rules
      }
   until nodeTypes stable;
   return rules $\cup \{r \to R\ r_R\}$ where $R$ is the document root element type

descendantOrSelf( element node type $E$ ) is
   nodeTypes := $\{E\}$
   rules := $\{r \to E\ r_E\}$
   loop
      forall $X \in$ nodeTypes {
         add $r_X \to \varepsilon$ to rules if $\text{”}X ::= \mathit{EMPTY}\text{”} \in \mathit{DTD\text{-}rules}$
         add $E_1, \ldots, E_n$ to nodeTypes if $\text{”}X ::= cm(E_1, \ldots, E_n)\text{”} \in \mathit{DTD\text{-}rules}$
         compute $G(cm(E_1, \ldots, E_n))$ and rename its axiom $r_X$
         in the result, replace $E_i$ by $(E_i\ r_{E_i})$
         add the result to rules
      }
   until nodeTypes stable;
   return rules
```

Algorithm 5 (Approximate Matches)

```
approximateMatches( element node type $E$, DTD root node type $R$,
                    match expression $m$) is
   class:= "must match"
   $e$ := last step of $m$
   $a$ := remainder of $m$
   if ($e.filter$) class:= "may match"
   if ($e.sel \neq$ "$*$" $\wedge\, e.sel \neq E$) return "no match"
   nodeTypes := $approx(E, e.axis)$
   if ($a$ empty) {
      if ($m$ absolut $\wedge R \notin$ nodeTypes) class:= "no match"
      return class;
   }
   for all $E'$ in nodeTypes {
      class($E'$) := approximateMatches($E', R, a$);
   }
   if (for all $E'$ in nodeTypes: class($E'$)="no match") return "no match"
   if (for all $E'$ in nodeTypes: class($E'$)="must match") return class
   return "may match"
```

Algorithm 6 (Approximate Selections)

```
typedef Rules = grammar rules for sequences of element node types;

// Input  Rules define the sequence of relative root element node types
// Output Rules define the sequence of selected element node types

Rules approximateSelections( Rules $R$, selection path expression $s_r$) is
   $e$ := first step of $s_r$
   $m$ := remainder of $s_r$
   if ($e.sel = A$) replace occurrences of $E \neq A$ in $R$ by $\varepsilon$;
   if ($e.filter$) replace occurrences of $E$ in $R$ by $(E|\varepsilon)$;
   if ($m$ empty) return $R$;
   else {
      replace occurrences of element node types $E$ in $R$ by a unique $r_E$;
      for all replaced element node types $E$ {
         $R_E = approx(E, m.axis)$;
         replace axiom of $R_E$ by $r_E$;
      }
      return approximateSelections($\bigcup R_E \cup R$, $m$);
   }
```

Algorithm 7 (Iterators)

```
class ChildIterator  extends Iterator is
   Queue Node nodes := new Queue();
   void init(Node $n$){
      for $cn$ in children($n$) if ($S_{Type(cn)} \neq \varepsilon$) nodes.enqueue($cn$);
   }
   Node next(){if (nodes.empty) return null; else return nodes.dequeue();

class AncestorOrSelfIterator  extends Iterator is
   Stack [Node] nodes := new Stack();
   void init(Node $n$){
      while ($n \neq R$){
         if ($S_{Type(n)} \neq \varepsilon$) nodes.push($n$);
         $n$ := parent($n$);
      }
   }
   Node next(){if (nodes.empty) return null; else return nodes.pop();}

class DescendantOrSelfIterator extends Iterator is
   Queue [Node] nodes := new Queue();
   void init(Node $n$){
      if ($S_{Type(n)} \neq \varepsilon$) nodes.enqueue($n$);
      for $cn$ in children($n$) init($cn$);
   }
   Node next(){if (nodes.empty) return null; else return nodes.dequeue();}

class SelectionIterator is
   Iterator [Node] nodes;
   void init(Iterator a, SelectionPath $s_r$){
      nodes := new Iterator();
      $e$ := first step of $s_r$
      $m$ := remainder of $s_r$
      while (($n$ := a.next()) != null)
         if ( conforms($n,e$) $\wedge$ $S_{Type(n)} \neq \varepsilon$ ) nodes.add($n$);
      if ($m$ empty) return;
      else {
         Iterator [Node] newNodes:= new Iterator();
         for ( $n$ in nodes ) {
            generate iterator $i$ for axis of $e$ and initialize over $n$
            newNodes.concat($i$);
         }
         init(newNodes, $m$);
      }
   }
   Node next(){if (nodes.empty) return null; else return nodes.next();}
```