Sangam: A Transformation Modeling Framework

Kajal T. Claypool
Department of Computer Science
University of Massachusetts, Lowell
kajal@cs.uml.edu

Elke A. Rundensteiner
Department of Computer Science
Worcester Polytechnic Institute
rundenst@cs.wpi.edu

Abstract

Integration of multiple heterogeneous data sources continues to be a critical problem for many application domains and a challenge for researchers world-wide. One aspect of integration is the translation of schema and data across data model boundaries. Researchers in the past have looked at both customized algorithmic approaches as well as generic meta-modeling approaches as viable solutions. We now take the meta-modeling approach the next step forward. In this paper, we propose a flexible, extensible *and* re-usable *transformation modeling framework which allows users to (1) model their transformations; (2) to choose from a set of possible execution strategies to translate the underlying schema and data; and (3) to access and re-use a library of transformation generators. In this paper, we present the core of our modeling framework - a set of cross algebra operators that covers the class of linear transformations, and two different techniques of composing these operators into larger transformation expressions. We also present an evaluation strategy to execute the modeled transformation, and thereby transform the input schema and data into the target schema and data assuming that data model wrappers are provided for each data model. To show reusability in our framework, we also present one transformation generator and show how the generator can produce a transformation model for any given input schema and data. The proposed framework has been implemented, and we give an overview of this prototype system.*

Keywords: Cross Model Mapping Algebra, Heterogeneous System Integration, Schema Transformation

1. Introduction

Integration of multiple heterogeneous data sources continues to be a critical problem for many application domains and a challenge for researchers world-wide [5]. Each database brings with it its own concepts, semantics, data formats, and access methods. Currently, the burden falls on the human to manually resolve conflicts, integrate the data, and interpret the results. More often than not, this barrier proves too difficult or too time-consuming to overcome and data hence often is under-exploited.

Data integration as a research field looks at automating as many of the tasks related to the above process, and hence aims to provide better and painless access to data no matter what data source or format it is stored in. One aspect of data integration is schema matching. Schema matching is the task of finding semantic correspondences between elements of two schemas [16]. Many researchers have addressed the schema matching problem either for a specific domain [6, 3, 4] or in a generic domain-independent way [14, 16, 5, 2, 12, 17, 20]. Another aspect of the integration problem with respect to the heterogeneity of information, i.e., the different data models, is the *translation* of schema (and data) from one data model to another. Solutions for this include customized algorithmic approaches [26, 11, 22, 7, 23] and meta-modeling approaches [2, 12, 20, 5, 17]. A customized algorithmic approach provides fixed translation algorithms that convert schema and data between a given pair of data models. The meta-modeling approach provides a more general technique that goes beyond translations for a given pair of data models. The translations themselves are generally expressed either via rules [17, 2] or via fragments of code [5, 12].

In our work we focus on the meta-modeling approach for the translation of schemas across data model boundaries. In particular, we focus on the explicit *modeling* and the subsequent execution of the transformations themselves, an aspect not addressed by previous research [2, 12, 20, 5, 17]. While models, such as the UML model, for static concepts like schemata or data models have been much studied, models for the more dynamic aspects such as for transformations have been largely overlooked. The goal of our work thus is to provide a *flexible*, *extensible* and *re-usable* translation modeling framework wherein users can (1) explicitly model the translations between schemas; (2) choose automated execution strategies to execute the modeled translations that would transform the source schema and data to the desired

target schema and data; and (3) choose and compose transformations from an existing library of translations. Such a framework offers many advantages over previous translation approaches. In particular it allows for (1) optimized execution strategies as each modeled transformation can be reasoned over to determine the optimal execution plan similar to the algebraic query optimization; (2) development of a *generic* tool set for facilitating activities such as maintenance; and (3) query merging and translation in a multi-tier environment.

To enable this translation modeling framework, we identify (1) the fundamental operations required to express and model the translation process (Section 3); and (2) the flexible techniques necessary for composing these core operations into larger meaningful translations (Section 4). A modeled transformation can be executed using any one of the many possible execution strategies, to perform the requisite data transformation process. These execution strategies range from the mapping of the cross algebra expressions to full-fledged query languages such as SQL [1] and XQuery [10] to having customized execution algorithms. In this paper, we briefly sketch out a customized algorithm (Section 5) for executing the cross algebra expressions to illustrate the simplicity of this task.

2. Background: Sangam Graph Model

```
<!ELEMENT item (location,
        mailbox, name)>
<!ATTLIST item id ID #REQUIRED
        featured CDATA #IMPLIED>
<!ELEMENT location (#PCDATA)>
<!ELEMENT mailbox (mail*)>
<!ATTLIST mailbox id CDATA>
<!ELEMENT mail (from, to, date)>
<!ATTLIST mail text CDATA>
<!ELEMENT from (#PCDATA)>
<!ELEMENT to (#PCDATA)>
<!ELEMENT date (#PCDATA)>
<!ELEMENT name (firstName,
        lastName)>
<!ELEMENT firstName (#PCDATA)>
<!ELEMENT lastName (#PCDATA)>
```

Figure 1. A Fragment of the XMark Benchmark DTD.

We assume, as in previous modeling approaches [2, 12, 20, 5, 17], that schemas from different data models are first represented in one common data model. In our work we assume that all schemas are represented by a simple graph called a *Sangam graph*, an instance of the Sangam graph model [8]. A Sangam graph $\mathtt{G} = (\mathrm{N}, \mathrm{E}, \lambda)$ is a directed graph of nodes N and edges E, and a set of labels λ. Each node has an associated type *complex* ($\Box$) or *atomic* ($\bigcirc$); and each edge is either a *containment* ($\rightarrow$) or a *property* ($\dashrightarrow$) edge. A *containment* edge is an edge between two complex nodes, while a *property* edge exists between a complex node and an atomic node. Each node $\mathtt{n}$ has associated with it a set of objects, its *extent* denoted as I ($\mathtt{n}$). Each object $\mathtt{o} \in \mathrm{I}$ ($\mathtt{n}$) is a pair $<\mathtt{id}, \mathtt{v}>$ where $\mathtt{id}$ is a globally unique identifier and $\mathtt{v}$ is its data value. Each edge $\mathtt{e}$:$<\mathtt{n}_1, \mathtt{n}_2>$ also has associated with it a set of objects, termed its *extent* R ($\mathtt{e}$). Each object $\mathtt{o}_e \in \mathrm{R}$ is a triple $<\mathtt{id}, \mathtt{o}_1, \mathtt{o}_2>$ where $\mathtt{id}$ is a system-generated identifier, object $\mathtt{o}_1 \in \mathrm{I}\,(\mathtt{n}_1)$ and object $\mathtt{o}_2 \in \mathrm{I}\,(\mathtt{n}_2)$. There may be zero to multiple edges between the same two nodes. In addition, each edge $\mathtt{e}$ is annotated with a set of properties ζ, possibly empty. This set of properties includes a *local order*, denoted by ρ, and *quantifier* annotation, denoted by Ω. The local order ρ gives the relative local ordering for all outgoing edges from a given node $\mathtt{n}$ in the Sangam graph. A *quantifier* is a pair of integers [min:max], with $0 \leq \mathtt{min} \leq \mathtt{max} < \infty$ where $\mathtt{min}$ specifies the minimum and $\mathtt{max}$ the maximum occurrences of objects of a node $\mathtt{n}_2$ for a given object $\mathtt{o}$ of node $\mathtt{n}_1$ associated via the relationship (edge) $\mathtt{e}$.

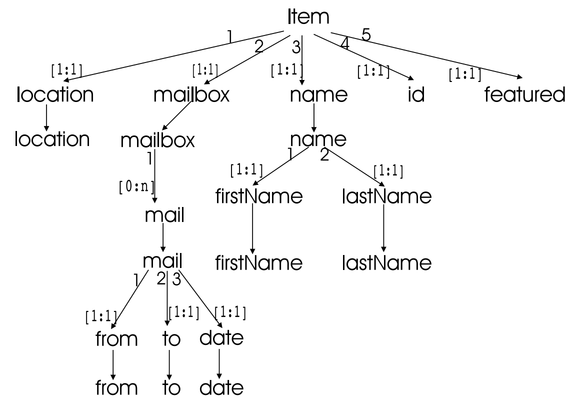

Figure 2. A Fragment of the XMark Benchmark DTD as shown in Figure 1 depicted as a Sangam Graph.

Example: Representing XML Schema as a Sangam Graph Figure 2 shows the Sangam graph for the XMark benchmark schema of Figure 1. Here each element and attribute is represented by a Sangam graph node. For example, the edge $\mathtt{e}_1$ between the node labeled `item` and `location` represents a relationship between the `item` element and its sub-element `location`. The edge $\mathtt{e}_1$ has an order annotation of `1`. The quantifier annotation [1:1] on

edge $\mathtt{e}_1$ denotes a functional edge, i.e., that there must be exactly one object of `location` that can participate in a binary relationship with one object of `item`.

3. The Bricks: Cross Algebra Operators

The key factors that influence the achievement of a flexible and extensible translation framework are the building blocks that would enable users to model different translations in order to transform a schema. To enable the modeling of such translations we provide two main building blocks: (1) the bricks: the cross algebra operators which allow the user to express a variety of linear transformations; and (2) the mortar: different techniques that allow users to compose the operators together to represent larger translation units. In this section we present the first building blocks, i.e., the cross algebra operators.

In our work we have identified four basic transformation operators, `cross`, `connect`, `smooth`, and `subdivide`. These operators, termed the *cross algebra operators*, represent the primitive set of operations in the class of linear graph transformations [13] on the basis of which larger more complex linear transformations can be defined. In this section we briefly describe the semantics of these operators. For more details refer to [8].

3.1. Cross Operator

The `cross` algebra operator $\otimes$ takes as input a node `n` in `G` and produces as output a node `n'` in `G'`. The cross operator is a total mapping, i.e., the objects in the extent of `n` given by I (`n`) are mapped one-to-one to the objects in the extent of `n'` given by I (`n'`) in the output Sangam graph . Figure 3 (a) depicts the cross operator. We use the notation $\otimes_{\mathtt{n'}}$ (`n`) to depict a cross operator with input `n` and output `n'`.

3.2. Connect Operator

A `connect` algebra operator ($\ominus$) corresponds to an edge creation in `G'`. It takes as input an edge `e` between two nodes $\mathtt{n}_1$ and $\mathtt{n}_2$ in `G` and produces an edge `e'` between two nodes $\mathtt{n}_1$' and $\mathtt{n}_2$' in `G'`. All objects $\mathtt{o} \in$ I (`e`) are also copied as part of this process. The connect operation succeeds if and only if nodes $\mathtt{n}_1$ and $\mathtt{n}_2$ have already been mapped to the nodes $\mathtt{n}_1$' and $\mathtt{n}_2$' respectively using two cross operators. The connect operator preserves the annotations of the edge `e`, i.e., the output edge `e'` will have the same quantifier and local ordering annotation as the input edge `e`. Figure 3 (b) gives an example of the connect operator. We use the notation $\ominus_{e'}$ (`e`) to depict a connect operator that maps the edge `e`:$<\mathtt{n}_1, \mathtt{n}_2>$ to edge `e'`:$<\mathtt{n}_1', \mathtt{n}_2'>$.

3.3. Smooth Operator

A `smooth` operator ($\circleddash$) models the combination of two relationships in `G` to form one relationship in `G'`. Let `G` be a Sangam graph with three nodes $\mathtt{n}_1$, $\mathtt{n}_2$, and $\mathtt{n}_3$, and two relationships represented by edges `e1`:$< \mathtt{n}_1, \mathtt{n}_2 >$ and `e2`:$< \mathtt{n}_2, \mathtt{n}_3 >$. The `smooth` ($\circleddash$) operator replaces the relationships represented by edges `e1` and `e2` in `G` with a new relationship represented by edge `e'`:$< \mathtt{n}_1', \mathtt{n}_3' >$ in `G'`. The smooth operator can only be applied when $\otimes(\mathtt{n}_1) = \mathtt{n}_1'$ and $\otimes(\mathtt{n}_3) = \mathtt{n}_3'$. The *local order* annotation on the edge `e'` is set to the *local order* annotation of the edge `e1`. However, as the edge `e'` has a larger information capacity than the edges `e1` and `e2`, the *quantifier* annotation of the edge `e'` is given as: $\rho(\mathtt{e'}) = \rho(\mathtt{e1}) * \rho(\mathtt{e2})$. Figure 4 gives an example of the smooth operator. We use the notation $\circleddash_{e'}(\mathtt{e}_1, \mathtt{e}_2)$ to depict a smooth operator that maps edges $\mathtt{e}_1$:$<\mathtt{n}_1, \mathtt{n}_2>$ and $\mathtt{e}_2$:$<\mathtt{n}_2, \mathtt{n}_3>$ to edge `e'`:$<\mathtt{n}_1', \mathtt{n}_3'>$ in the output.

3.4. Subdivide Operator

A `subdivide` operator $\oslash$ intuitively performs the inverse operation of the `smooth` operator, i.e., it splits a given relationship into two relationships connected via a node. Let `G` have two nodes $\mathtt{n}_1$ and $\mathtt{n}_3$ and edge `e`:$< \mathtt{n}_1, \mathtt{n}_3 >$. The `subdivide` operator introduces a new node $\mathtt{n}_2'$ in `G'` such that the edge `e` in `G` is replaced by two edges `e1'`:$<\mathtt{n}_1', \mathtt{n}_2'>$ and `e2'`:$<\mathtt{n}_2', \mathtt{n}_3'>$ in `G'`. The `subdivide` operator is only valid if $\otimes(\mathtt{n}_1) = \mathtt{n}_1'$ and $\otimes(\mathtt{n}_3) = \mathtt{n}_3'$. The local order annotation for the edge `e1'`:$< \mathtt{n}_1', \mathtt{n}_2' >$ is the same as the local order annotation of the edge `e` as $\otimes(\mathtt{n}_1) = \mathtt{n}_1'$. The edge `e2'` is the only edge added for the node $\mathtt{n}_2'$ and thus has a local order annotation of `1`. To preserve the extent I (`e`), the edges `e1'` and `e2'` are assigned quantifier annotations as follows. If $\mathtt{min}(\rho(\mathtt{e}))$ = 0, then the quantifier range for `e1'` is given as $[0 : 1]$, else it is always set to $[1 : 1]$. The quantifier of edge `e2'` is set equal to the quantifier of edge `e`. We use the notation $\oslash_{e1', e2', n2'}(\mathtt{e})$ to depict a subdivide node that maps edge `e`:$<\mathtt{n}_1, \mathtt{n}_3>$ to `e1'`:$<\mathtt{n}_1', \mathtt{n}_2'>$ and `e2'`:$<\mathtt{n}_2', \mathtt{n}_3'>$ in the output.

4. The Mortar: Composition Techniques

Cross algebra operators can be composed into larger transformations using two techniques: (1) context dependency; and (2) derivation. The context dependency composition enables several algebra operators to collaborate and jointly operate on sub-graphs to produce one combined output graph. The derivation composition enables the nesting of several algebra operators wherein output of one or more operators becomes the input of another operator. Derivation

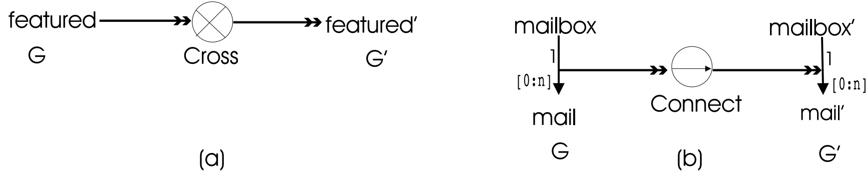

Figure 3. (a) Example of Cross Algebra Operator; (b) Example of Connect Algebra Operator.

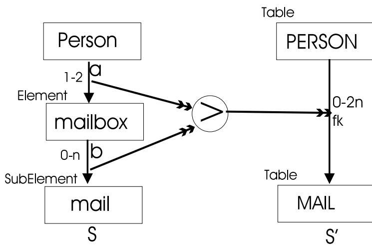

Figure 4. Example of Smooth Operator.

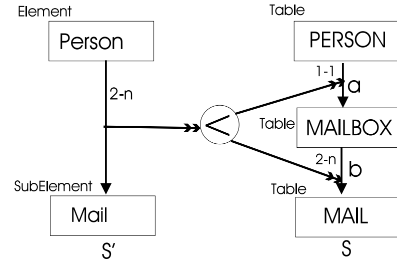

Figure 5. Example of Subdivide Operator.

and context dependency can in turn be combined together to produce larger, more complex transformations. In this section, we present the rules governing their combination.

4.1. Context Dependency Composition

The first composition technique, called the context dependency composition, enables several algebra operators to collaborate and jointly operate on sub-graphs to produce one combined output graph. Figure 7 denotes such a context dependency composition CT of three cross algebra operators. Here, the algebra operators $\texttt{op1}_{\texttt{A}'}$(A) and $\texttt{op2}_{\texttt{B}'}$(B) are cross operators that map the nodes A and B in G to nodes A' and B' respectively in the output Sangam graph G'. The algebra operator $\texttt{op3}_{\texttt{e}'}$(e), a $\ominus$ operator is the root of CT and maps the edge e:<A, B> between the nodes A and B in the input Sangam graph G to the edge e':<A', B'> between the nodes A' and B' in the output Sangam graph G'. Here the outputs of all operators op1, op2, and op3 together produce G'.

Definition 1 *Given an input Sangam graph* G*, a context dependency expression* $\texttt{CT}_o$ *is specified as:*

$$\texttt{CT}_{\texttt{o}(\texttt{out}_\texttt{o})}(\texttt{in}_\texttt{o}) \quad = \quad \begin{cases} op_{i(out_i)}(in_i) \\ op_{i(out_i)}(in_i), \\ (CT_{k(out_k)}(in_k))[\circ \\ (CT_{l(out_l)}(in_l))]^{+}) \end{cases}$$

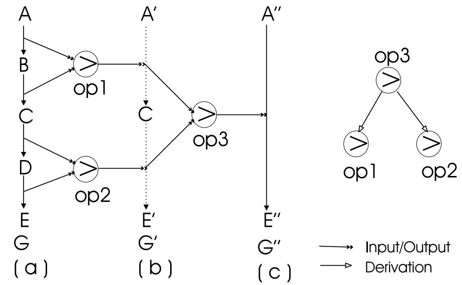

Figure 6. Derivation Composition.

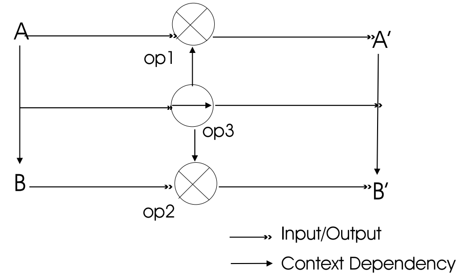

Figure 7. Context Dependency Composition Example.

where op_i *is the parent operator of* CT_k *and* CT_l *denoting that* op_i *must be executed after* CT_k *and* CT_l*.* op_i *uses outputs, inputs and mapping of* CT_k *and* CT_l *and* $out_o = out_j \bigcup out_k \bigcup out_l$*. The context dependency expression* $\texttt{CT}_o$ *operates on nodes* $\texttt{n}_i$ *and edges* $\texttt{e}_i \in \texttt{G}$*, and produces as output a Sangam graph* G' *such that all nodes* $\texttt{n}_i$*' and/or edges* $\texttt{e}_i$*' produced as output by any of the individual op-*

erators $\texttt{op}_i \in \texttt{CT}$ *are in* G′. *Here the symbol "+" is part of the BNF grammar syntax to indicate that the expression contained in "[ ]" may occur one or more times.*

As an example, the expression for the context dependency composition in Figure 7 is given as: $\texttt{CT}_{out}\texttt{(in)}$ = $\texttt{op3}_{e'}$(e), ($\texttt{op1}_{A'}$ (A) ∘ $\texttt{op2}_{B'}$(B)). Here the operator op3 is the root of the composition, and op1 and op2 are the children operators. Here e:<A, B> and e′:<A′, B′>. The final output out = e′ $\bigcup$ A′ $\bigcup$ B′ corresponds to the final output G′.

The context dependency expression $\texttt{CT}_o$ is evaluated from right to left and from inside out. That is all children operators are evaluated prior to the evaluation of their parent operator. The order of evaluation between the sibling operations is immaterial. Beyond the order of evaluation, the context dependency relation between two operators op3→op1 (Figure 7) implies that the operator op3 uses the following three pieces of information in its calculation: (1) the input of op1; (2) the output of op1; and (3) the mapping ϕ of op1 as established by the type ⊗, ⦵, ⧁ and ⧀ of op1.

4.2. Derivation Composition Technique

The second composition technique is the *derivation composition*. This technique enables the nesting of several algebra operators wherein output of one or more operators becomes the input of another operator. Figure 6 gives an example of the modeling of a derivation composition that transforms the path in the Sangam graph G shown in Figure 6 (a) to the edge in the Sangam graph G′ given in Figure 6 (c) by applying three smooth nodes ⧁. Let $\texttt{e}_1$:<A,B>, $\texttt{e}_2$:<B, C>, $\texttt{e}_3$:<C, D> and $\texttt{e}_4$:<D, E> be edges in G. Operators $\texttt{op1}_{e1'}(\texttt{e}_1, \texttt{e}_2)$ and $\texttt{op2}_{e2'}(\texttt{e}_3, \texttt{e}_4)$ are applied to the input edges $\texttt{e}_1$ and $\texttt{e}_2$, and $\texttt{e}_3$ and $\texttt{e}_4$ respectively to first produce the intermediate edges $\texttt{e}_1$′:<A′, C′> and $\texttt{e}_2$′<C′, E′> as shown in Figure 6 (b)[1]. The operator $\texttt{op3}_{e3''}(\texttt{e}_1', \texttt{e}_2')$ operates on these intermediate edges $\texttt{e}_1$′ and $\texttt{e}_2$' and produces the desired output edge $\texttt{e}_3$′′:<A′′, E′′> as shown in Figure 6 (c). One approach to achieving this is to first produce the intermediate edges and then the final output edge $\texttt{e}_3$". Equivalently we can express this by nesting. Thus, the output of the algebra expression $\texttt{op3}_{e3''}(\texttt{op1}_{e1'}(\texttt{e}_1, \texttt{e}_2),\texttt{op2}_{e2'}(\texttt{e}_3, \texttt{e}_4))$ is the output edge $\texttt{e}_3$". The output of the operator op3 is said to be *derived* from the outputs of operators op1 and op2, or put differently, the output of operators op1 and op2 are the *inputs* of operator op3 and are consumed by op3.

Definition 2 *Given an input Sangam graph* G*, a derivation expression* $\texttt{DT}_o$ *is given as:*

[1]We do not include here the context dependency composition that would be needed to map the nodes of the graph.

$$\texttt{DT}_{\texttt{o(out}_\texttt{o})}(in_o) \quad = \quad \begin{cases} op_{i(out_i)}(in_i) \\ op_{i(out_i)}(\texttt{DT}_{j(out_j)(in_j)}) \\ op_{i(out_i)}((\texttt{DT}_{k(out_k)}(in_k)), \\ (\texttt{DT}_{l(out_l)}(in_l))) \end{cases}$$

where DT_k *and* DT_l *are derivation trees and* $\texttt{in}_i = \{\texttt{out}_j\}$ *or* $\{\texttt{out}_k, \texttt{out}_l\}$. *The expression* $\texttt{DT}_o$ *produces as output* $\texttt{out}_o$ *node and edge elements for an output Sangam graph* G′*, such that* $\texttt{out}_o$ *is the output of the root operator* $\texttt{op}_i$ *and thus also of the complete derivation tree* $\texttt{DT}_o$*. Here, "(" and ")" pairs denote nesting and the symbol "," separates input arguments of an operator* $\texttt{op}_i$.

4.3. Cross Algebra Graphs (CAG): Combining Context Dependency and Derivation Compositions

Derivation and context dependency compositions can be combined in one *cross algebra graph* (CAG) to model a complex transformation of a Sangam graph G into into a graph G′. See for example Figure 8. Here let e1:<A, B>, e2:<B, C>, e3:<C, D> and e4:<D, E> represent edges in G, and let e′:<A′, E′> represent an edge in G′. The expression for Figure 8 is CAT = CAT3.

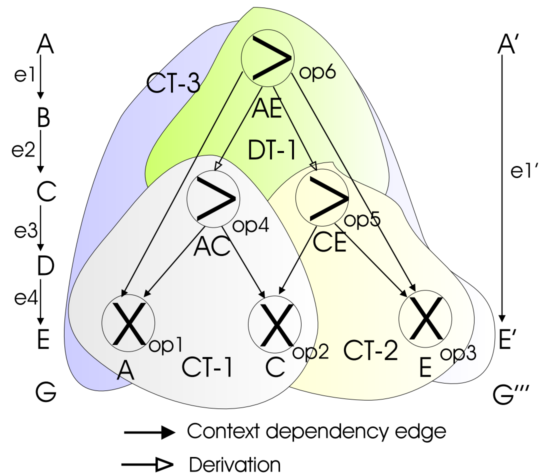

Figure 8. A Cross Algebra Graph.

The output of CAT3 (CAT3 = DT1, ($\texttt{op1}_{\texttt{A}'}$(A) ∘ $\texttt{op3}_{\texttt{E}'}$(E))) is produced by the evaluation of its three inputs, the derivation composition DT1, and the two cross algebra operators op1 and op3. The evaluation of the two primitive operators op1 and op3 produces the nodes A′ and E′

respectively. The expression DT1 produces two intermediate Sangam graphs G' and G'' (outputs of CT1 and CT2 respectively) that smooth the edges e1 and e2 to produce edge eTemp1 and smooth edges e3 and e4 to produce edge eTemp2 respectively. The operator op6 then gets its inputs from CT1 and CT2 and produces the edge e':<A', E'>. The operator op6 also participates in the composition CAT3, the output of which is the final Sangam graph G''' with nodes A', E' and the edge e':<A', E'>. Thus, the CAT in Figure 8 operates on the input Sangam graph G and produces as output the Sangam graph G'''.

Definition 3 (CAT) *A CAT is an expression that operates on one or more input Sangam graphs* G *and produces one or more output Sangam graph* G' *such that:*

$$\mathtt{CAT}_{\mathtt{o(out_o)}}(\mathtt{in_o}) = \begin{cases} DT_{i(out_i)}(in_i) \\ CT_{i(out_i)}(in_i) \\ (CAT_{j(out_j)}(in_j)) \quad , \\ ((CAT_{k(out_k)}(in_k)))^{\dagger}. \\ op_{j(out_j)} \\ (CAT_{k(out_k)}(in_k))^{\ddagger}. \\ op_{j(out_j)}((CAT_{k(out_k)}(in_k)), \\ (CAT_{l(out_l)}(in_l)))^{\gamma}. \end{cases}$$

$\dagger$ = *A context dependency edge is added from the root* $\mathtt{op}_j$ *of* $\mathtt{CAT}_j$ *to the root* $\mathtt{op}_k$ *of* $\mathtt{CAT}_k$.
$\ddagger$ = *A derivation edge is added from* $\mathtt{op}_j$ *to* $\mathtt{op}_k$*, the root of* $\mathtt{CAT}_k$.
γ = *A derivation edge is added from* $\mathtt{op}_j$ *to* $\mathtt{op}_k$ *and* $\mathtt{op}_l$*, the roots of* $\mathtt{CAT}_k$ *and* $\mathtt{CAT}_l$ *respectively.*

Based on the definition of a CAT (Definition 3), we now define a cross algebra graph (CAG). Intuitively, a CAG is a collection of cross algebra trees that may operate on possibly disjoint input graphs to produce possibly disjoint output graphs.

Definition 4 (CAG) *A cross algebra graph (CAG) is an expression that operates on one or more input Sangam graphs* G *and produces one or more output Sangam graph* G' *such that:*

$$\mathtt{CAG}_{\mathtt{o(out_o)}}(\mathtt{in_o}) = \begin{array}{l} (CAT_{j(out_j)}(in_j)) \quad [\circ \\ (CAT_{k(out_k)}(in_k))]^{+} \end{array}$$

where CAT_j *and* CAT_k *are sibling CATs such that* $out_o = out_j \bigcup out_k$.

5. Execution Strategies for Modeled Transformations

In Sections 3 and 4 we have introduced the bricks and the mortar of our translation modeling framework, and shown how large complex transformations can be modeled in the same. In this section, we briefly describe how the cross algebra expressions representing the modeled transformations can be executed.

```
function EvaluateCAG (input: CAG cag, Sangam graph G,
          output: Sangam graph G')
{
    List roots ← cag.getRoots()
    while (roots != null) {
      operator op ← roots.getNext()
      EvaluateCAT (op, G, G')
    }
}

function EvaluateCAT (input: Operator op, Sangam graph G,
          output: Sangam graph G')
{
    if (!op.hasChildren())
      Sangam graph G' ← op.evaluate(G, G')
      op.markDone()
      Sangam graph out ← G' // cache the local output
      return localG'

    while (op.hasChildren()) {
      operator opC ← op.getNextChild()
      if (e:<op, opC> = derivation)
          SAG G_local ← EvaluateCAT (opC, G, G')
          SAG G' ← op.evaluate(G_local, G')
          op.markDone()
          SAG out ← G' // local cached output
          return G'

      elseif (e:<op, opC> = context dependency)
          SAG G_local ← EvaluateCAT (opC, G, G')
          SAG G'_local ← op.evaluate(G, G')
          SAG G' ← G_local ⋃ G'_local
          op.markDone()
          SAG out ← G'_local // local cached output
          return G'
    }
}
```

Figure 9. The Evaluation Algorithm for a Cross Algebra Graph.

The execution of the cross algebra expression here implies the transformation of the source schema and data as per the modeled transformation to produce the target

schema and data. As stated in Section 1 there are many possible strategies for executing these modeled transformations. These strategies range from mapping the modeled transformation into query languages such as SQL [1] or XQuery [10] to applying execution algorithms directly using the transformation model. To keep the discussion simple we now sketch out a customized algorithm for executing the cross algebra expressions as shown in Figure 9.

A cross algebra graph (CAG) can be viewed as a forest of nested context dependency and derivation compositions. Each composition $\texttt{CAT}_i$ of the CAG can be evaluated independently of any other composition $\texttt{CAT}_j$ of the CAG. To evaluate each individual CAT, we use post-order evaluation, i.e., all children operators $\texttt{op}_j$ of an operator $\texttt{op}_k$ are evaluated prior to the evaluation of $\texttt{op}_k$.

Figure 9 gives the algorithm for evaluating the cross algebra graph. Here, to facilitate evaluation of shared operators, each operator $\texttt{op}_j$ is marked “visited” the first time it is evaluated, and its local output is cached. If the operator $\texttt{op}_j$ is re-visited, no further evaluation of $\texttt{op}_j$ is done, instead its cached output is returned to the invoking parent operator $\texttt{op}_i$. Evaluation of the tree terminates with the evaluation of the root operator[2].

6. Architectural Overview of 𝔖𝔞𝔫𝔤𝔞𝔪

The 𝔖𝔞𝔫𝔤𝔞𝔪 system incorporating all the techniques discussed in the paper has been developed using Java technology and a variety of tools such as the JAXP [24] for parsing XML documents and the DTD-Parser [25] for parsing the DTDs. Figure 10 gives an architectural overview of the system. Here **SAG-Loader** translates XML and relational schema and data into Sangam graphs ; **CAG-Builder** houses the library of transformations and builds transformation models based on the chosen generators for the given input Sangam graph ; the **CAG-Evaluator** evaluates the CAG; and lastly the **CAG-Generator** is able to translate the given output Sangam graph into a relational or XML schema and data.

7. Related Work

Schema Integration and Data Transformation. There is extensive literature under the umbrella of schema transformation and integration [15, 19, 11, 18, 9, 21]. However, this work is typically specific to either an application domain or to a particular data model and does not deal with meta-modeling [2, 5, 20, 21]. Recent work related to ours are Clio [15] a research project at IBM's Almaden Research Center and work by Milo and Zohar [19]. Clio, a tool for creating mappings between two data representations semi-automatically with user inputs, focuses on supporting querying of data in either the source or the target representation and on just in time cleansing and transformation of data. Milo et al. [19] have looked at the problem of data translations based on schema-matching. They follow an approach similar to Atzeni et al.[2] and Papazoglou et al. [20], but not at the meta-level, in that they define a set of translation logic rules to enable discovery of relationships between two application schemas. Bernstein et al. [5] have also proposed a meta-modeling framework to represent schemas in a common data model to facilitate a host of tools such as the *match* operator which produces a set of matches between two given schemas. While we share this vision, our focus is more on the representation and the execution of transformations that may be produced either by such matches or by a user via a GUI. In fact, based on the discussion with the authors of [5], their meta-model can be extended with a `makemap` meta-operator to capture the semantics of our work.

We can directly make use of translation algorithms from the literature, such as the algorithms for translating between an XML-DTD and relational schema [11] or mapping rules [19] or output from match operators [5]. That is, we can develop generators that capture such specific algorithms, and then generate separate transformation models that can be executed. Work on equivalence of the translations between models [18] is of particular importance as such properties could also be established for the cross algebra.

8. Conclusions

In this paper, we have presented our *flexible*, *extensible* and *re-usable* transformation modeling framework. This framework has the advantage of allowing users to model complex translations using the basic building blocks that we have presented in this paper, and then with a click of a button generating and executing code that would perform the data translation. While in this paper, we make substantial contributions towards providing a transformation framework, we believe this is just the tip of the iceberg. Some possible extensions to this work are handling of non-linear transformations and optimization of evaluation strategies.

References

[1] ANSI Standard. The SQL 92 Standard. http://www.ansi.org/, 1992.

[2] P. Atzeni and R. Torlone. Management of Multiple Models in an Extensible Database Design Tool. In P. M. G. Apers and et al., editors, *Advances in Database Technology - EDBT'96, Avignon, France, March 25-29*, LNCS. Springer, 1996.

[3] S. Bergamaschi, S. Castano, M. Vincini, and D. Beneventano. Semantic integration of heterogeneous information

[2] Proof of termination can be found in [8].

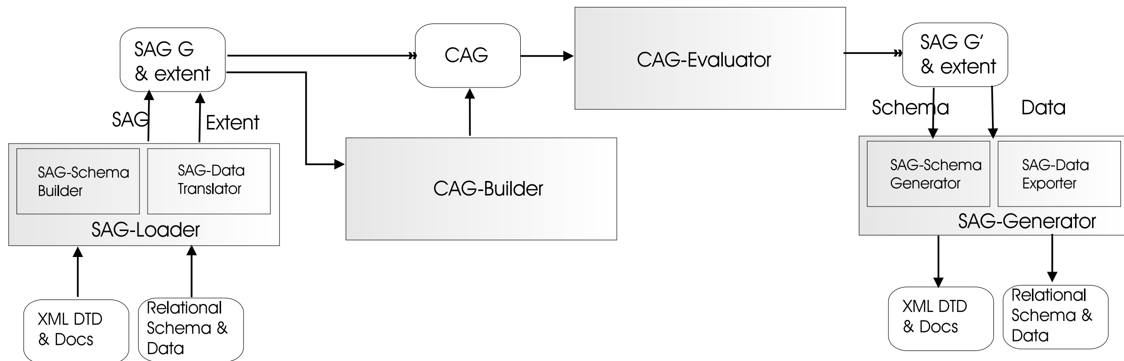

Figure 10. Architecture of Sangam

sources. *Data and Knowledge Engineering*, 36(3):215–249, 2001.

[4] J. Berlin and A. Motro. AutoPlex: Automated Discovery of Content for Virtual Databases. In *CoopIS*, pages 108–122, 2001.

[5] P. A. Bernstein and E. Rahm. Data Warehouse Scenarios for Model Management. In *International Conference on Conceptual Modeling*, 2000.

[6] M. Bright, A. Hurson, and S. H. Pakzad. Automated Resolution of Semantic Heterogeneity in Multidatabases. *TODS*, 19(2):212–253, 1994.

[7] B. Catania, E. Ferrari, A. Levy, and A. Meldelzon. XML and Object Technology. In *ECOOP Workshop on XML and Object Technology, LNCS 1964*, pages 191–202, 2000.

[8] K. Claypool. *Managing Change in Databases*. PhD thesis, Worcester Polytechnic Institute, May 2002.

[9] K. Claypool, J. Jin, and E. Rundensteiner. SERF: Schema Evolution through an Extensible, Re-usable and Flexible Framework. In *Int. Conf. on Information and Knowledge Management*, pages 314–321, November 1998.

[10] P. Fankhauser, M. Fernandez, A. Malhotra, M. a. Rys, J. Simeon, and P. Wadler. XQuery 1.0 Formal Semantics. http://www.w3c.org/TR/2001/query-semantics, June 2001.

[11] D. Florescu and D. Kossmann. Storing and Querying XML Data Using an RDBMS. In *Bulletin of the Technical Committee on Data Engineering*, pages 27–34, Sept. 1999.

[12] S. Göbel and K. Lutze. Development of Meta Databases for Geospatial Data in the WWW. In *ACM-GIS*, pages 94–99, 1998.

[13] J. Gross and J. Yellen. *Graph Theory and it Applications*. CRC Press, 1998.

[14] L. Haas, R. Miller, B. Niswonger, M. Roth, P. Schwarz, and E. Wimmers. Transforming Heterogeneous Data with Database Middleware: Beyond Integration. *IEEE Data Engineering Bulletin*, 22(1):31–36, 1999.

[15] L. Haas, R. Miller, B. Niswonger, M. Roth, P. Schwarz, and E. Wimmers. Transforming Heterogeneous Data with Database Middleware: Beyond Integration. *IEEE Data Engineering Bulletin*, 22(1):31–36, 1999.

[16] J. Madhavan, P. Bernstein, and E. Rahm. Generic Schema Matching with Cupid. In *vldb*, pages 49–58, 2001.

[17] L. Mark and N. Roussopoulos. Integration of Data, Schema and Meta-Schema in the Context of Self-Documenting Data Models. In C. G. Davis, S. Jajodia, P. A. Ng, and R. T. Yeh, editors, *Proceedings of the 3rd Int. Conf. on Entity-Relationship Approach (ER'83)*, pages 585–602. North Holland, 1983.

[18] R. J. Miller, Y. E. Ioannidis, and R. Ramakrishnan. The Use of Information Capacity in Schema Integration and Translation. In *Int. Conference on Very Large Data Bases*, pages 120–133, 1993.

[19] T. Milo and S. Zohar. Using Schema Matching to Simplify Heterogeneous Data Translation. In A. Gupta, O. Shmueli, and J. Widom, editors, *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*, pages 122–133. Morgan Kaufmann, 1998.

[20] M. Papazoglou and N. Russell. A Semantic Meta-Modeling Approach to Schema Transformation. In *CIKM '95*, pages 113–121. ACM, 1995.

[21] A. Rosenthal and D. Reiner. Theoretically Sound Transformations for Practical Database Design. In S. T. March, editor, *Entity-Relationship Approach, Proceedings of the Sixth International Conference on Entity-Relationship Approach, New York, USA, November 9-11, 1987*, pages 115–131, 1987.

[22] J. Shanmugasundaram, G. He, K. Tufte, C. Zhang, D. DeWitt, and J. Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *Proceedings of 25th International Conference on Very Large Data Bases (VLDB'99)*, pages 302–314, 1999.

[23] T. Shimura, M. Yoshikawa, and S. Uemura. Storage and Retrieval of XML Documents using Object-Relational Databases. In *Int. Conference and Workshop on Database and Expert Systems Applications*, pages 206–217. Springer-Verlag, 1999.

[24] J. Systems. The jaxp 1.1.1parser. http://java.sun.com, November 2001.

[25] M. Wutka. The dtd parser. http://www.wutka.com, March 2001.

[26] X. Zhang, W.-C. Lee, G. Mitchell, and E. A. Rundensteiner. Clock: Synchronizing Internal Relational Storage with External XML Documents. In *ICDE-RIDE 2001*, 2001.