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. 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 schemata; (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 trans-
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 \textit{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

![Figure 1. A Fragment of the XMark Benchmark DTD.](image)

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 \textit{Sangam graph}, an instance of the Sangam graph model [8]. A Sangam graph \(G = (N, E, \lambda)\) is a directed graph of nodes \(N\) and edges \(E\), and a set of labels \(\lambda\). Each node has an associated type \textit{complex} (\(\square\)) or \textit{atomic} (\(\circ\)); and each edge is either a \textit{containment} (\(\rightarrow\)) or a \textit{property} (\(\rightarrow\rightarrow\)) edge. A \textit{containment} edge is an edge between two complex nodes, while a \textit{property} edge exists between a complex node and an atomic node. Each node \(n\) has associated with it a set of objects, its \textit{extent} denoted as \(I(n)\). Each object \(o \in I(n)\) is a pair \(<\text{id}, \nu>\) where \(\text{id}\) is a globally unique identifier and \(\nu\) is its data value. Each edge \(e:<n_1,n_2>\) also has associated with it a set of objects, termed its \textit{extent} \(R(e)\). Each object \(o_e \in R\) is a triple \(<\text{id},o_1,o_2>\) where \(\text{id}\) is a system-generated identifier, object \(o_1 \in I(n_1)\) and object \(o_2 \in I(n_2)\). There may be zero to multiple edges between the same two nodes. In addition, each edge \(e\) is annotated with a set of properties \(\zeta\), possibly empty. This set of properties includes a \textit{local order}, denoted by \(\rho\), and \textit{quantifier} annotation, denoted by \(\Omega\). The local order \(\rho\) gives the relative local ordering for all outgoing edges from a given node \(n\) in the Sangam graph. A \textit{quantifier} is a pair of integers \([\text{min:max}]\), with \(0 \leq \text{min} \leq \text{max} < \infty\) where \(\text{min}\) specifies the minimum and \(\text{max}\) the maximum occurrences of objects of a node \(n_2\) for a given object \(o\) of node \(n_1\) associated via the relationship (edge) \(e\).

![Figure 2. A Fragment of the XMark Benchmark DTD as shown in Figure 1 depicted as a Sangam Graph.](image)
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 ⊙ 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 ⊙n’(n) to depict a cross operator with input n and output n’.

3.2. Connect Operator

A connect algebra operator (⊗) corresponds to an edge creation in G’. It takes as input an edge e between two nodes n1 and n2 in G and produces an edge e’ between two nodes n1’ and n2’ in G’. All objects o ∈ I(e) are also copied as part of this process. The connect operator succeeds if and only if nodes n1 and n2 have already been mapped to the nodes n1’ and n2’ 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 ⊗n’1,e,e2’n(1, e2’) to depict a connect operator that maps the edge e:<n1, n2> to edge e’:<n1’, n2’>.

3.3. Smooth Operator

A smooth operator (⊖) models the combination of two relationships in G to form one relationship in G’. Let G be a Sangam graph with three nodes n1, n2, and n3, and two relationships represented by edges e1:<n1, n2> and e2:<n2, n3>. The smooth (⊖) operator replaces the relationships represented by edges e1 and e2 in G with a new relationship represented by edge e’:<n1’, n3’> in G’. The smooth operator can only be applied when ⊖(n1) = n1’ and ⊖(n3) = n3’. 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: ρ(e’) = ρ(e1) * ρ(e2). Figure 4 gives an example of the smooth operator. We use the notation ⊖n’1,e,e2’n(e1, e2) to depict a smooth operator that maps edges e1:<n1, n2> and e2:<n2, n3> to edge e’:<n1’, n3’> in the output.

3.4. Subdivide Operator

A subdivide operator ⊗ 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 n1 and n3 and edge e:<n1, n3>. The subdivide operator introduces a new node n2’ in G’ such that the edge e in G is replaced by two edges e1’:<n1’, n2’> and e2’:<n2’, n3> in G’. The subdivide operator is only valid if ⊗(n1) = n1’ and ⊗(n3) = n3’. The local order annotation for the edge e1’:<n1’, n2’> is the same as the local order annotation of the edge e as ⊗(n1) = n1’. The edge e2’ is the only edge added for the node n2’ 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 m1n(ρ(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 ⊗n’1,e,e2’n(e1, e2) to depict a subdivide node that maps edge e:<n1, n3> to e1’:<n1’, n2’> and e2’:<n2’, n3’> 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 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 \(\text{CT}^o\) of three cross algebra operators. Here, the algebra operators \(\text{op}_1, \text{op}_2\) 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 \(\text{op}_3\) is the root of \(\text{CT}\) and maps the edge \(e: <A, B>\) 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 \(\text{op}_1, \text{op}_2, \text{op}_3\) together produce \(G'\).

Definition 1 Given an input Sangam graph \(G\), a context dependency expression \(\text{CT}^o\) is specified as:

\[
\text{CT}^o_{\text{out}}(\text{in}) = \begin{cases} 
\text{op}_1(\text{out}_1)(\text{in}_1) \\
\text{op}_2(\text{out}_2)(\text{in}_2) \\
(\text{CT}_k(\text{out}_k)(\text{in}_k)) + \\
(\text{CT}_l(\text{out}_l)(\text{in}_l)) + 
\end{cases}
\]

where \(\text{op}_i\) is the parent operator of \(\text{CT}_k\) and \(\text{CT}_l\) denoting that \(\text{op}_i\) must be executed after \(\text{CT}_k\) and \(\text{CT}_l\). \(\text{op}_i\) uses outputs, inputs and mapping of \(\text{CT}_k\) and \(\text{CT}_l\) and \(\text{out}_o = \text{out}_1 \cup \text{out}_2 \cup \text{out}_3\). The context dependency expression \(\text{CT}^o\) operates on nodes \(n_i\) and edges \(e_i\) in \(G\), and produces as output a Sangam graph \(G'\) such that all nodes \(n_i'\) and/or edges \(e_i'\) produced as output by any of the individual operators \(\text{op}_i \in \text{CT}\) 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: \(\text{CT}^o(\text{in})\).
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 $\varnothing$. Let $e_1:<A, B>$, $e_2:<B, C>$, $e_3:<C, D>$ and $e_4:<D, E>$ be edges in G. Operators $\text{op1}_{e_1}(e_1, e_2)$ and $\text{op2}_{e_2}(e_3, e_4)$ are applied to the input edges $e_1$ and $e_2$, and $e_3$ and $e_4$ respectively to first produce the intermediate edges $e_1':<A', C'>$ and $e_2':<C', E'>$ as shown in Figure 6 (b). The operator $\text{op3}_{e_3'}(e_1', e_2')$ operates on these intermediate edges $e_1'$ and $e_2'$ and produces the desired output edge $e_3'':<A'', E''>$ as shown in Figure 6 (c). One approach to achieving this is to first produce the intermediate edges and then the final output edge $e_3''$. Equivalently we can express this by nesting. Thus, the output of the algebra expression $\text{op3}_{e_3''}(\text{op1}_{e_1'}(e_1, e_2), \text{op2}_{e_2'}(e_3, e_4))$ is the output edge $e_3''$. The output of the operator $\text{op3}$ is said to be derived from the outputs of operators $\text{op1}$ and $\text{op2}$, or put differently, the output of operators $\text{op1}$ and $\text{op2}$ are the inputs of operator $\text{op3}$ and are consumed by $\text{op3}$.

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

\[
\text{DT}_o(\text{out}_o)(\text{in}_o) = \begin{cases} 
\text{op1}_{(\text{out}_1)}(\text{in}_1) \\
\text{op2}_{(\text{out}_2)}(\text{DT}_{(\text{out}_2)}(\text{in}_2)) \\
\text{op3}_{(\text{out}_3)}(\text{DT}_{(\text{out}_3)}(\text{in}_3)) \\
\text{DT}_{(\text{out}_4)}(\text{in}_4)
\end{cases}
\]

where $\text{DT}_k$ and $\text{DT}_1$ are derivation trees and $\text{in}_i = \{\text{out}_j\}$ or $\{\text{out}_k, \text{out}_l\}$. The expression $\text{DT}_o$ produces as output $\text{out}_o$ node and edge elements for an output Sangam graph $G'$, such that $\text{out}_o$ is the output of the root operator $\text{op}_1$, and thus also of the complete derivation tree $\text{DT}_o$. Here, "(" and ")" pairs denote nesting and the symbol "," separates input arguments of an operator $\text{op}_i$.

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

Derivation and context dependency compositions can be combined in one cross algebra tree (CAT) or a cross algebra graph (CAG) to model a complex transformation of a Sangam graph G into a graph $G'$. Figure 8 represents an example of a CAT. Here let $e_1:<A, B>$, $e_2:<B, C>$, $e_3:<C, D>$ and $e_4:<D, E>$ represent edges in G, and let $e'':<A', E''>$ represent an edge in $G'$. The expression for Figure 8 is $\text{CAT} = \text{DT}_3$.

**Figure 8. A Cross Algebra Graph.**

The output of $\text{CAT}_3$ ($\text{CAT}_3 = \text{DT}_1$, $(\text{op1}_A(A) \circ \text{op3}_E(E))$ is produced by the evaluation of its three inputs, the derivation composition $\text{DT}_1$, and the two cross algebra operators $\text{op1}$ and $\text{op3}$. The evaluation of the two primitive operators $\text{op1}$ and $\text{op3}$ produces the nodes $A'$ and $E'$.
respectively. The expression $DT_1$ produces two intermediate Sangam graphs $G'$ and $G''$ (outputs of $CT_1$ and $CT_2$ respectively) that smooth the edges $e_1$ and $e_2$ to produce edge $e_{Temp1}$ and smooth edges $e_3$ and $e_4$ to produce edge $e_{Temp2}$ respectively. The operator $op_6$ then gets its inputs from $CT_1$ and $CT_2$ and produces the edge $e' < A', E' >$. The operator $op_6$ also participates in the composition $CT_3$, 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 $G'''$. Formally, we define a CAT as follows.

**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:

$$CAT_{out}(in) = \begin{cases} DT_{i(out)}(in_i) \\ CT_{i(out)}(in_i) \\ (CAT_{j(out)}(in_j)) \cup (CAT_{k(out)}(in_k))) \\ op_{j(out)} \\ op_{k(out)} \\ (CAT_{j(out)}(in_j)) \cup (CAT_{k(out)}(in_k))) \\ \end{cases}$$

where $\downarrow$ = A context dependency edge is added from the root $op_j$ of CATj to the root $op_k$ of CATk.

$\uparrow$ = A derivation edge is added from $op_j$ to $op_k$, the root of CATk.

$\gamma$ = A derivation edge is added from $op_j$ to $op_k$ and $op_i$, the roots of CATk and CATj 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:

$$CAG_{out}(in) = (CAT_{j(out)}(in_j)) \cup (CAT_{k(out)}(in_k)))$$

where CATj and CATk are sibling CATs such that $out_j = out_j \cup out_k$.

### 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.

```plaintext
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'
    }
    operator opC ← op.getNextChild()
    if (e <: op, opC = derivation)
    {
        localG' ← EvaluateCAT (opC, G, G')
        G' ← op.evaluate(localG', G')
        op.markDone()
        Sangam graph out ← G' // local cached output
        return G'
    }
    else if (e <: op, opC = context dependency)
    {
        localG' ← EvaluateCAT (opC, G, G')
        G' ← op.evaluate(localG', G')
        G' local ← G' local \ G' \\
        op.markDone()
        Sangam graph 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 $\mathcal{CAT}_i$ of the CAG can be evaluated independently of any other composition $\mathcal{CAT}_j$ of the CAG. To evaluate each individual CAT, we use post-order evaluation, i.e., all children operators $\mathcal{OP}_j$ of an operator $\mathcal{OP}_k$ are evaluated prior to the evaluation of $\mathcal{OP}_k$.

Figure 9 gives the algorithm for evaluating the cross algebra graph. Here, to facilitate evaluation of shared operators, each operator $\mathcal{OP}_j$ is marked “visited” the first time it is evaluated, and its local output is cached. If the operator $\mathcal{OP}_j$ is re-visited, no further evaluation of $\mathcal{OP}_j$ is done, instead its cached output is returned to the invoking parent operator $\mathcal{OP}_i$. Evaluation of the tree terminates with the evaluation of the root operator $\mathcal{OP}_0$.

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

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

Figure 10. Architecture of Sangam