WPI-CS-TR-01-04 March 2001

Sangam[1] : Modeling Transformations For Integrating Now and Tomorrow

by

Kajal T. Claypool and Elke A. Rundensteiner

Computer Science Technical Report Series

WORCESTER POLYTECHNIC INSTITUTE

Computer Science Department
100 Institute Road, Worcester, Massachusetts 01609-2280

$\mathfrak{Sangam}$[†]: Modeling Transformations For Integrating Now and Tomorrow

Kajal T. Claypool and Elke A. Rundensteiner

Department of Computer Science
Worcester Polytechnic Institute
Worcester, MA 01609–2280
{kajal|rundenst}@cs.wpi.edu

Abstract

Today many application engineers struggle to not only publish their relational, object or ascii file data on the Web but to also integrate information from diverse sources, often inventing and reinventing a suite of hard-wired integration tools. A model management system that supports the specification and manipulation of not only data models and schemata, but also mappings between the different models in a generic manner has the promise of solving these issues. However, support for modeling and managing such mappings as objects remains an unsolved challenge. In our work, we propose a powerful middleware tool that successfully tackles this challenge. For this, we propose a graph-theoretic framework that allows users to explicitly model mappings between different data models as well as re-structuring within one data model. Our map metamodel is based on a set of re-usable mapping constructs that can in principle be applied on any data model described in our framework. In our work, we have tested these operators for XML and relational model mappings. Using the description of maps at the model level, mappings between specific application schemas and transformations of associated application data can be automated by our framework. Our framework guarantees the correctness of the map, of the generated transformation code, of the output data model, and of the generated application schemas, based on the correctness criteria for the map metamodel. In this paper, we also introduce the model management system $\mathfrak{Sangam}$ that we are developing to realize our proposed map modeling theory. With $\mathfrak{Sangam}$ we show not only the feasibility of our approach but also demonstrate the re-usability and the ease of end-to-end development of modeling strategies. To further illustrate our ideas, we present a walk-through example of mapping an application DTD to a relational application schema using $\mathfrak{Sangam}$.

Keywords: Metamodel, Model Management, Integration, Schema Transformation, XML Mapping

1 Introduction

The Problem of Integration. Networked environments like the Internet have catalyzed a phenomenal growth in the publication of data bringing with it an increasing need to share and integrate information. Alas, the format of data to be integrated varies from company to company and sometimes from person to person. To accomplish tasks such as data sharing, exchange, and integration, we may need to map an XML schema to a relational schema to drive transformation of XML elements into relational data or map an XML schema

[†] $\mathfrak{Sangam}$ (Sangam) is a Hindi word meaning Junction.

of one application to that of another. Data sources may need to be mapped into data warehouse tables, or a query posed against a high-level semantic model may need to be mapped into an equivalent query posed against a logical database schema [HMN$^+$99].

Today with new data formats continuously emerging, research and industry likewise have had to visit and re-visit the issues of integration. And in the era of electronic information exchange, as current technology turns into legacy systems at an ever increasing pace, this data model impedance presents an increasingly critical challenge. To combat this recurring problem, researchers [BR00, AT96, GL98] have looked at model management as a possible solution. For over a decade, researchers have modeled data models and application schemas to facilitate the translation process between different models. The translation between the models, however, was and is restricted to hard-wired code interspersed with generic rules to allow for some flexibility [MR83, AT96, BR00, PR95]. Moreover, these systems are not designed to be extensible. Hence, the addition of a new data model typically requires integration engineers to write from scratch, the translation of the new model to all other data models as well as the re-structuring within the same data model. The only re-use here is perhaps that of a user's knowledge and experience.

Modeling Maps. The key thrust of our work now lies in the *modeling of maps* in a model management system. Our goal is to alleviate the necessity of writing hard-wired translation code and to reach a new era of tackling future legacy system integrations for now and tomorrow. We do so by using a graph-theoretic framework [AT96] to model not only the data models, but to now also explicitly model the maps between the data models and between the application schemas. For this, we provide a generic set of map metaconstructs to describe (1) transformation of one data model to another - for example, map a relation in the relational model to an element or sub-element in the XML model; (2) re-structuring within the same data model - for example, combine two elements in a DTD to produce a new merged element in an output DTD; (3) re-structuring across schema-data boundaries - for example, map an attribute in the input to a relation in the output. We guarantee the correctness of these mappings, i.e., we can establish the correctness of the output data model (or output application schema) when given a "correct" input data model (or application schema). In this paper, we limit our discussions to translation to and from the relational model and the XML model (DTD) and re-structuring within these models. However, mapping between other models can be done in a similar manner.

To show the feasibility of our approach, we present $\mathfrak{Sangam}$ our prototype system [CRZS01]. To the best of our knowledge, this is the first model management system that models and manages mappings and treats them as first-class objects. We illustrate the ease with which a user can not only map an application DTD to an relational schema using $\mathfrak{Sangam}$ but also have the system generate the corresponding code for the data transformation.

Advantages of Our Approach.

- *Automatic Generation* - $\mathfrak{Sangam}$ offers declarative support for handling transformation strategies from one data model to another. These strategies are declared once at the model level by a system administrator. A key advantage of $\mathfrak{Sangam}$ lies in the fact that mapping between application schemas and the subsequent mapping between the application data can be almost completely automated, thereby increasing user productivity.

- *Correctness* - Based on our theoretical framework, we can guarantee the correctness of the mapping strategies, the mapping code that is generated to transform one application schema to another, as well as the transformation of the application data. Correctness here is based on the conformation to the output data model.

- *Plug-n-Play* - An integration engineer can edit and customize a map, for example, by plugging in a new function to combine two attributes together in a domain-specific manner. The system would still guarantee the correctness of the output as stated above.

- *Extensibility* - As new, perhaps more optimized, mapping strategies are discovered between existing data models, they can be added in $\mathfrak{Sangam}$ with relatively little effort (by composing new map models), when compared to writing a transformation program from scratch.

- *Map Evaluation* - In contrast to often ad-hoc translation code, our graph-theoretic representation of maps provides a solid basis for reasoning about maps, such as their capacity, their similarity or their performance. This allows us to evaluate different mappings and select the right mapping for a given task.

- *Generic Map Operators* - A generic powerful set of operators [BR00] can now be defined for maps, allowing easier maintainability and powerful management of maps compared to working directly with translation programs. This set would include, for example, operators to copy portions of a map, to compose more complex maps, and to propagate schema changes on either the input or the output application schemas.

Overview. In Section 2, we present our four-tier map modeling architecture. Section 3 reviews background on modeling of data models. Section 4 gives formal descriptions of our map metaconstructs and Section 5 describes how these metaconstructs can be put together to form maps between data models and application schemas. In Section 6, we walk-through a XML to relational model example to show the feasibility and realization of our theoretical framework via $\mathfrak{Sangam}$ our prototype system. Section 7 presents some related work and we conclude in Section 8.

2 A Four Layer Architecture for Map Modeling

UML has introduced a four-tier UML Metamodel architecture [Boo94] for modeling of data models. We now put forth our novel extension of the architecture to also enable modeling of maps between data models and application schemas.

Modeling Data Models and Application Schemas. While our work focuses on *modeling the mappings* between data models and application schemas, declaratively describing the latter is a necessity as these are the inputs and outputs of our maps. Following the four layer UML Metamodel architecture [Boo94], the data models, application schemas and data are represented in a four-tier architecture depicted in Figure 1. The metamodel layer describes the metaconstructs that are used in the model layer to define the data models. The data models in turn define the structure of an application schema in the application layer. The application data resides in the bottom layer, and its structure is defined by the corresponding application schema in the application layer above. Figure 2 (a) represents a DTD model in the data model layer; Figure 2 (b) represents an application DTD (conforming to the DTD model) in the application layer. A detailed description of these appears in Section 3.

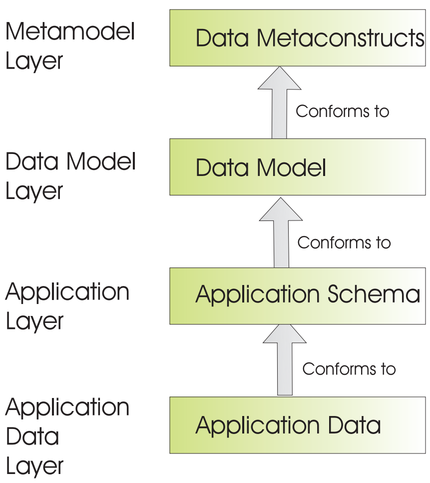

Figure 1: The Four Tier Architecture for Modeling Data Models and Application Schemas.

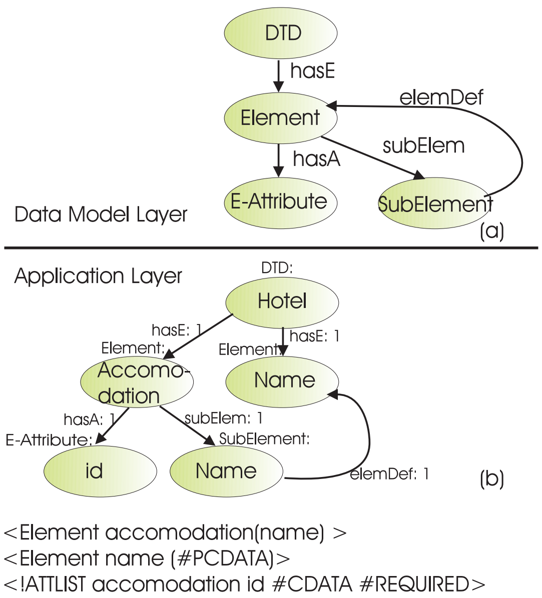

Figure 2: An Example Showing the DTD Model described in Data Model Layer and a DTD in the Application Layer.

Modeling Model Maps and Application Maps. We propose a novel extension to the familiar four tier architecture to now also model maps as shown in Figure 3. The map metaconstructs given in the top layer define the building blocks required for modeling maps. A map model composed of these map metaconstructs describes the translation of a given input data model to an output data model in the map model layer. For example, we can define a map model for translating the constructs of the XML (DTD) model (e.g., `element`) to constructs in the relational data model (e.g., `relation` or `attribute`).

In the application map layer, application maps conforming to the map model structure translate an input application schema to an output application schema as depicted in Figure 4. For example, an application map conforming to a given map model may map the element `Accommodation` to a relation `ACCOMMODATION`, or it may map the element `Name` to an attribute `name` in the relation `ACCOMMODATION`.

In the bottom layer, data maps describe the translation of the input application data to the output application data. This data transformation is guided by the information stored in the application map. Thus, for example, the values for element `Accommodation` for all XML documents conforming to the `Hotel` application DTD are translated to rows in the `ACCOMMODATION` relation; and the values for element `Name` are mapped to a value in a column of the relation `ACCOMMODATION`.

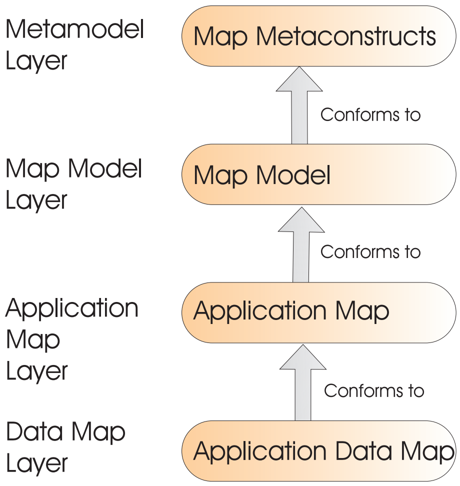

Figure 3: Our Proposed Four Tier Architecture for Modeling Map Models and Application Maps between Application Schemas.

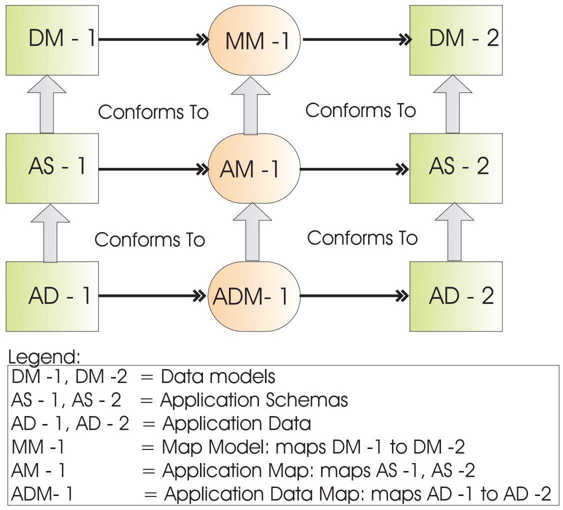

Figure 4: The Interaction between the Four-Tier Architecture for Data Models (Figure 1 and the Four-Tier Architecture for Map Modeling (Figure 3).

3 Background - Modeling Data Models and Application Schemas

In this section we briefly review the metamodel we use to express data models in the model layer and application schemas in the application layer. We utilize a subset of the graph-theoretic formalism as presented by Atzeni et al. [AT96]. Our metamodel is based on a fixed set of metaconstructs that are either nodes or edges. There are two node metaconstructs $\mathcal{N}$, namely *complex* ($\Box$) and *atomic* ($\bigcirc$) node types. Complex nodes have a set of outgoing edges while atomic nodes have no outgoing edges. There are two edge metaconstructs $\mathcal{E}$, namely *containment* ($\rightarrow$) and *property* ($\dashrightarrow$) edges. A containment edge exists between two complex nodes while a property edge stems from a complex node and ends on an atomic node.

Throughout $\mathtt{n} \in \mathcal{N}$ refers to a node and $\mathtt{e} \in \mathcal{E}$ refers to an edge. The notation $\mathtt{e}$:<$\mathtt{n}_1$,$\mathtt{n}_2$> refers to the two end-points of an edge. Here the edge $\mathtt{e}$ stems from $\mathtt{n}_1$ and ends on $\mathtt{n}_2$. Given edges $\mathtt{e}_1$:<$\mathtt{n}_1$, $\mathtt{n}_2$> and $\mathtt{e}_2$:<$\mathtt{n}_2$, $\mathtt{n}_3$>, we denote the path from n_1 to $\mathtt{n}_3$ by the path expression $\mathtt{n}_1.\mathtt{e}_1.\mathtt{n}_2.\mathtt{e}_2.\mathtt{n}_3$.

Patterns and Structures. As in [AT96], two main notions are utilized to describe data models as well as application schemas. A *structure* is a directed acyclic graph whose nodes and edges are metaconstructs of our metamodel. A *pattern* is a rooted tree whose nodes and edges are metaconstructs of the metamodel and whose edges have *quantifiers* as labels. A quantifier given as a pair of integers [x:y], with $0 \leq \mathtt{x} \leq \mathtt{y} < \infty$, specifies the minimum and maximum times an edge with the same label can appear in a structure.

Definition 1 (Structure) *A structure is a triple* $\mathtt{S} = (\mathtt{G}, \mu, \epsilon)$ *where* $\mathtt{G} = (\mathtt{N},\mathtt{E})$ *is a directed acyclic graph and* μ *and* ϵ *are typing functions:* μ: $\mathtt{N} \rightarrow \mathcal{N}$ *and* ϵ: $\mathtt{E} \rightarrow \mathcal{E}$.

Trees within the structure $\mathtt{S}$ composed of all the nodes and the edges reachable from the outer nodes[1] of $\mathtt{S}$ are referred to as *components* of $\mathtt{S}$. One structure $\mathtt{S}_1 = ((\mathtt{N}_1, \mathtt{E}_1), \mu_1, \epsilon_1)$ is mapped to another structure $\mathtt{S}_2 = ((\mathtt{N}_2, \mathtt{E}_2), \mu_2, \epsilon_2)$ by a pair of functions θ: $\mathtt{N}_1 \rightarrow \mathtt{N}_2$ and ϕ: $\mathtt{E}_1 \rightarrow \mathtt{E}_2$. Two structures are *isomorphic* if

[1]We use the term outer nodes to refer to nodes that have no edges incident on them.

for each $\mathtt{n} \in \mathtt{N}_1$, $\mu_1(\mathtt{n}) = \mu_2(\theta(\mathtt{n}))$ and for each edge $\mathtt{e}$:<$\mathtt{n}$, $\mathtt{n'}$> $\in \mathtt{E}_1$, $\phi(\mathtt{e}) = \mathtt{e'}$:<$\theta(\mathtt{n})$,$\theta(\mathtt{n'})$> $\in \mathtt{E}_2$ and $\epsilon_1(\mathtt{e}) = \epsilon_2(\phi(\mathtt{e}))$.

Definition 2 (Pattern) *A pattern is a pair* $\mathtt{P} = (\mathtt{S},\rho)$ *where* $\mathtt{S} = (\mathtt{G}, \mu, \epsilon)$ *is a structure such that* $\mathtt{G}$ *is a rooted tree, and* ρ *is a function that associates a quantifier with each edge of* $\mathtt{G}$.

A pattern describes a collection of structures that represent the same composition of the metaconstructs. The quantifier [x:y] associated with an edge in a pattern specifies the range, i.e., minimum (x) and maximum (y) number of times that the edge e can occur in a structure.

A structure S *matches* a pattern $\mathtt{P} = (\mathtt{S'}, \rho)$, if for a node $\mathtt{n} \in \mathtt{S}$ there is a set of edges in S denoted by E_n^e outgoing from n such $\phi(\mathtt{e}) = \mathtt{e'}$ for some edge $\mathtt{e'} \in \mathtt{S'}$; then, for each node n of S and for each set of edges E_n^e, $|E_n^e| \in \rho(\mathtt{e'})$.

A structure S is an *instanceOf* a set of patterns $\mathcal{P}$, if for each component $\mathtt{S}_i$ of a structure S there is a pattern $\mathtt{P}_i \in \mathcal{P}$ such that $\mathtt{S}_i$ matches $\mathtt{P}_i$. Given a set of patterns $\mathcal{P}$, *Inst(*$\mathcal{P}$*)* denotes the set of structures that are instances of $\mathcal{P}$. Figure 5 represents a set of patterns and Figure 6 shows a structure that is an instance of patterns P1 and P3 in Figure 5.

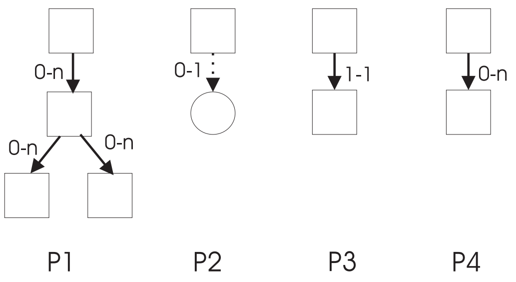

Figure 5: A Set of Patterns.

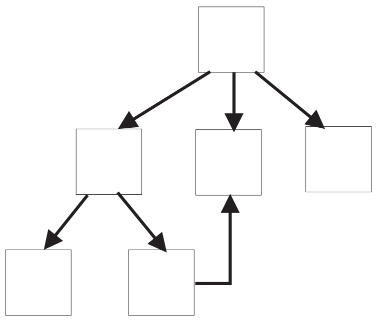

Figure 6: A Structure $\mathtt{S}_1$ - an Instance of Patterns P1 and $\mathtt{P}_3$ given in Figure 5.

Two patterns $\mathtt{P}_1 = (\mathtt{S}_1, \rho_1)$ and $\mathtt{P}_2 = (\mathtt{S}_2, \rho_2)$ are *isomorphic* if $\mathtt{S}_1$ and $\mathtt{S}_2$ are isomorphic and if for each edge $\mathtt{e}_1 \in \mathtt{P}_1$, there is $\mathtt{e}_2 \in \mathtt{P}_2$ such that $\phi(\mathtt{e}_2) = \mathtt{e}_1$ and $\rho(\mathtt{e}_1) = \rho(\mathtt{e}_2)$.

There is also a *subsumption* relationship between patterns, i.e., a set of patterns $\mathcal{P}_1$ is subsumed by another set of patterns $\mathcal{P}_2$, if *Inst(*$\mathcal{P}_1$*)* $\subseteq$ *Inst(*$\mathcal{P}_2$*)*.

Data Model. A data model $\mathtt{DM} = (\mathcal{P}, \gamma)$ is composed of a set of patterns $\mathcal{P}$ with a labeling function γ that maps each element of $\mathcal{P}$ of type $\mathcal{N} \bigcup \mathcal{E}$ to a label l from a set of labels $\mathcal{L}$. These labels correspond to the names of constructs in a particular data model.

Figure 2 shows the segment of the DTD model obtained by assigning labels *"DTD"*, *"Element"*, *"E-Attribute"*, and *"SubElement"* to the first, second, third and fourth □ node respectively in the the pattern P1; and the labels *"SubElement"* and *"Element"* to the first and second □ node respectively in pattern P3 given in Figure 5. Similarly, labels are assigned to all the edges.

Application Schema. An application schema AS is a pair $\mathtt{AS} = (\mathtt{S}, \lambda)$ composed of a structure S and a labeling function λ that maps each node and edge of S to a label, such that distinct labels are associated with different elements of the structure. These labels correspond to real-world entities modeled by the application schema. A schema $\texttt{AS} = (\texttt{S}, \lambda)$ is an *instanceOf* a model $\texttt{DM} = (\mathcal{P}, \gamma)$ if the structure $\texttt{S}$ is an *instanceof* $\mathcal{P}$.

Figure 2 (b) shows a slice of the application DTD `HOTEL` that is obtained by assigning labels to the structure in Figure 6.

4 Metaconstructs for Modeling Maps

In this section we now describe our map metamodel and in particular its map metaconstructs. We describe the functionality and correctness of these map metaconstructs using the concept of patterns from Section 3. These (instances of) map metaconstructs can map between patterns as well as structures, and hence can map from one data model to another and also from one application schema to another as defined below. In this section we show how the map metaconstructs work for patterns and structures using examples of patterns and structures that are part of a data model or an application schema (i.e., they have labels). We also assume here that there is a global universe of discourse of all allowable patterns in the system denoted by $\mathcal{P}$ and a set of all allowable structures in the system denoted by $\texttt{Inst}(\mathcal{P})$. This includes all sets of patterns that describe an individual data model and all structures that describe an application schema.

Our map metamodel is based on a fixed set of map node types $\mathcal{M}$ and a fixed set of map edge types ξ. We define five types of map nodes: *cross* node ($\otimes$), *merge* node ($\oplus$), *ident* node ($\ominus$), *project* node ($\ominus$) and *connect* node ($\ominus$). We also define three types of map edges: *input* ($\twoheadrightarrow$), *output* ($\twoheadrightarrow$) and *containment* ($\longrightarrow$).

4.1 Map Nodes

Each map node captures the semantics of some translation of a given input pattern to some output pattern, with their specific mappings and requirements for the input and output patterns as described below. We denote the mapping of a node $\texttt{n}$ in an input pattern $\texttt{P}$ to a node $\texttt{n}'$ in the output pattern $\texttt{P}'$ by $\theta(\texttt{n}) = \texttt{n}'$ as given in Section 3. Similarly, an edge $\texttt{e} \in \texttt{P}$ is mapped to an edge $\texttt{e}'$ in $\texttt{P}'$ by $\phi(\texttt{e}) = \texttt{e}'$. For each map node we now give more precise definitions of θ and ϕ.

4.1.1 Cross Node

Cross Node for Patterns. The cross map node is the simplest map node. It maps an input pattern consisting of one node (root) to an output pattern of exactly one node (root). Its functionality enables the mapping of one data model construct to another. For example, a cross node can map a node labeled *Element* in the XML model to a node labeled *Relation* in the relational model. Figure 7 (a) shows the cross mapping of a pattern $\texttt{P}$ to a pattern $\texttt{P}'$.

Definition 3 (Cross Node - Pattern) *Given an input pattern* $\texttt{P}$ *with* $\texttt{G} = (\texttt{N},\texttt{E})$*,* $|\texttt{N}| = 1$ *and* $|\texttt{E}| = 0$*, a cross node, denoted by* $\otimes$*, produces as output a pattern* $\texttt{P}'$ *with* $\texttt{G}' = (\texttt{N}',\texttt{E}')$*,* $|\texttt{N}'| = 1$ *and* $|\texttt{E}'| = 0$ *and function* θ*:* $\texttt{N} \longrightarrow \texttt{N}'$.

Pattern Correctness of Cross Node. Given an input pattern $\texttt{P}_1 \in \mathcal{P}$ with $\texttt{G}_1 = (\texttt{N}_1, \texttt{E}_1)$ a cross node $\otimes$ is said to be correct if it produces as output some pattern $\texttt{P}_2$ with $\texttt{G}_2 = (\texttt{N}_2, \texttt{E}_2)$, such that $\texttt{P}_2 \in \mathcal{P}$.

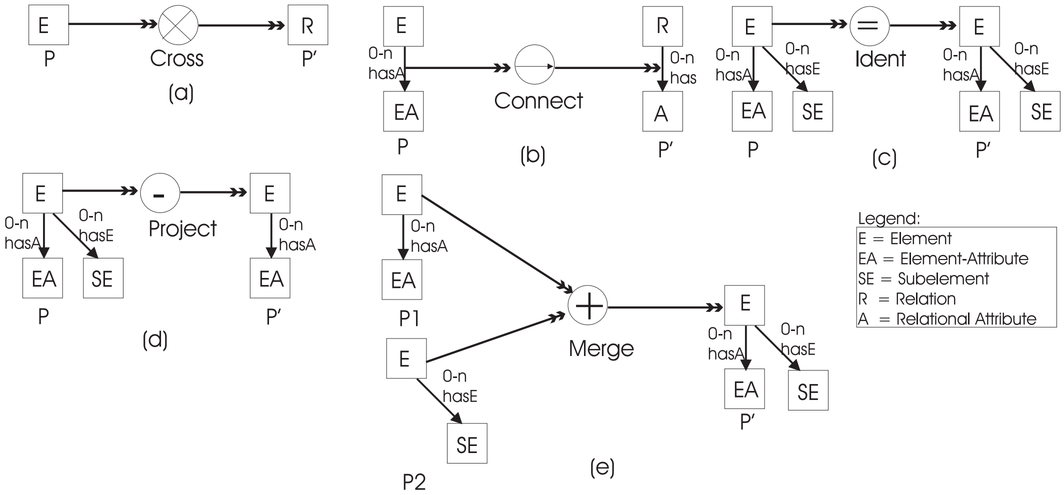

Figure 7: The Five Map Metaconstructs For Patterns.

Cross Node for Structures. At the application map layer, (an instance of) a cross node maps an input structure to produce an output structure, thereby allowing the mapping of one application schema to another. For example, Figure 8 (a) shows the mapping of an XML element `Accomodation` to a relation `ACCOMODATION`.

Definition 4 (Cross Node - Structure) *Given an input structure* $\mathtt{S}$ *with* $\mathtt{G} = (\mathtt{N},\mathtt{E})$*,* $|\mathtt{N}| = 1$ *and* $|\mathtt{E}| = 0$*, a cross node, denoted by* $\otimes$*, produces as output a structure* $\mathtt{S'}$ *with* $\mathtt{G'} = (\mathtt{N'},\mathtt{E'})$*,* $|\mathtt{N'}| = 1$ *and* $|\mathtt{E'}| = 0$ *and function* θ*:* $\mathtt{N} \longrightarrow \mathtt{N'}$.

Structure Correctness of Cross Node. Given an input structure $\mathtt{S}_1 \in \mathtt{Inst}(\mathcal{P})$ with $\mathtt{G}_1 = (\mathtt{N}_1, \mathtt{E}_1)$, a cross node $\otimes$ is said to be correct if it produces as output some structure $\mathtt{S}_2$ with $\mathtt{G}_2 = (\mathtt{N}_2, \mathtt{E}_2)$ such that $\mathtt{S}_2 \in \mathtt{Inst}(\mathcal{P})$.

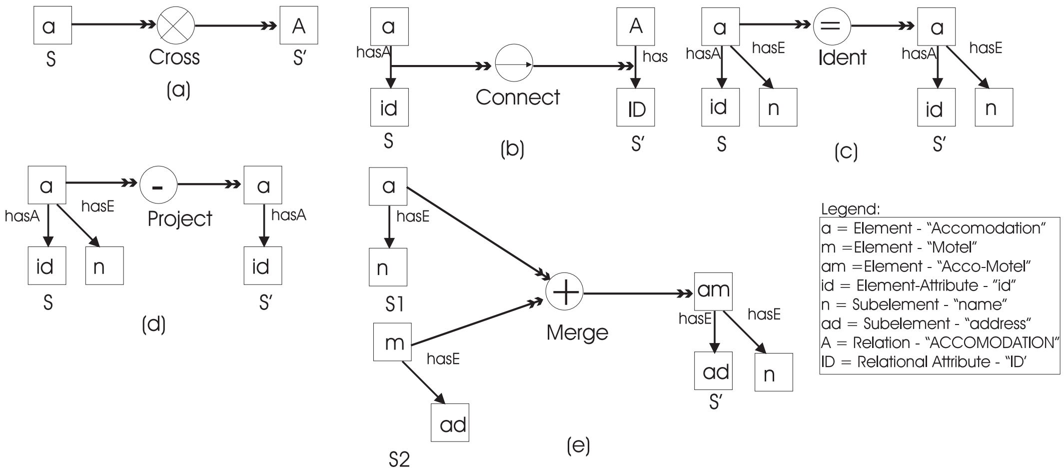

Figure 8: The Five Map Metaconstructs For Structures.

4.1.2 Connect Node

In general, the connect node enables the mapping of an edge $\mathtt{e}_1$:$<\mathtt{n}_1$, $\mathtt{n}_2>$ in an input pattern or structure to another edge $\mathtt{e}_2$: $<\mathtt{n}_3$, $\mathtt{n}_4>$ in the output pattern or structure respectively.

Connect Node for Patterns. The connect node enables the mapping of an edge $\mathtt{e}_1$:$<\mathtt{n}_1$, $\mathtt{n}_2>$ in an input pattern to an edge $\mathtt{e}_2$: $<\mathtt{n}_3$, $\mathtt{n}_4>$ in the output pattern. For example, a connect node can map the *Subelem* relationship between an *Element* and its *Subelement* in the XML model to the *ForeignKey* relationship between two *Relations* in the relational model. Input to a connect node can be a direct edge between two nodes such as *has*, *hasA*, or *Subelem*; or a path expression such as *Element.SubElem.SubElement.ElemDef* that refers to the definition of an *Element* that defines a *Subelement*. Output of an edge map node is restricted to be a direct edge[2].

For correctness of the connect node, we require that the end-points of the edge $\mathtt{e}_1$ be mapped to the end-points of the output edge $\mathtt{e}_1'$ to avoid dangling edges. Thus, node $\mathtt{n}_1$ must be mapped to either $\mathtt{n}_1'$ or $\mathtt{n}_2'$. If $\mathtt{n}_1$ is mapped to $\mathtt{n}_1'$, then $\mathtt{n}_2$ must be mapped to $\mathtt{n}_2'$; or vice versa. Hence, it is not meaningful to map the *has* edge between the *Relation* and the *Attribute* to the *hasA* edge between an *Element* and *E-Attribute* without also mapping the nodes appropriately.

Definition 5 (Connect Node - Pattern) *Given a pattern* $\mathtt{P}$ *with* $\mathtt{G} = (\mathtt{N},\mathtt{E})$, $|\mathtt{N}| = 2$, $|\mathtt{E}| = 1$, *and edge* $\mathtt{e}$:$<\mathtt{n}_1$, $\mathtt{n}_2> \in \mathtt{P}$, *such that* $\mathtt{n}_1$, $\mathtt{n}_2 \in \mathtt{N}$ *and* $\mathtt{n}_1$ *noteq* $\mathtt{n}_2$, *a connect node, denoted by* $\ominus$, *maps* $\mathtt{P}$ *to an output pattern* $\mathtt{P}'$, *with* $\mathtt{G}' = (\mathtt{N}',\mathtt{E}')$, *and* $|\mathtt{N}'| = 2$, $|\mathtt{E}| = 1$, *via function* ϕ: $\mathtt{E} \longrightarrow \mathtt{E}'$ *such that* $\phi(\mathtt{e}) = \mathtt{e}'$:$<\mathtt{n}_1'$, $\mathtt{n}_2'>$ *and* $\mathtt{e}' \in \mathtt{E}'$, $\mathtt{n}_1'$, $\mathtt{n}_2' \in \mathtt{N}'$. *Also we require* $\theta(\mathtt{n}_1) = \mathtt{n}_1'$ *and* $\theta(\mathtt{n}_2) = \mathtt{n}_2'$ *or vice versa.*

Pattern Correctness of Connect Node. Given an input pattern $\mathtt{P} \in \mathcal{P}$ with edge $\mathtt{e}$:$<\mathtt{n}_1$, $\mathtt{n}_2> \in \mathtt{P}$, the connect node is said to be correct if the output pattern $\mathtt{P}'$ it produces is in $\mathcal{P}$.

Connect Node for Structures. The connect node to map between two structures is similar in functionality to that between two patterns. Hence, a connect node maps edge $\mathtt{e}$:$<\mathtt{n}_1$, $\mathtt{n}_2>$ in an input structure to another edge $\mathtt{e}'$: $<\mathtt{n}_1'$, $\mathtt{n}_2'>$ in the output structure. Figure 8 shows a connect node that maps the edge between the XML element `Accomodation` and its attribute `id` to the edge between the relation `ACCOMODATION` and its attribute `ID`. Similar to the connect node between patterns, a direct edge or a path expression in the input structure is valid input for the (instance of) connect node in the application layer.

For correctness of the connect node here, we again stipulate as for the connect node between patterns, that the end-points of the edge in the input structure must be mapped to the end-points of the edge in the output structure.

Definition 6 (Connect Node - Structure) *Given a structure* $\mathtt{S}$ *with* $\mathtt{G} = (\mathtt{N},\mathtt{E})$, $|\mathtt{N}| = 2$, $|\mathtt{E}| = 1$, *and edge* $\mathtt{e}$:$<\mathtt{n}_1$, $\mathtt{n}_2> \in \mathtt{S}$, *such that* $\mathtt{n}_1$, $\mathtt{n}_2 \in \mathtt{N}$, *a connect node, denoted by* $\ominus$, *maps* $\mathtt{S}$ *to an output structure* $\mathtt{S}'$, *with* $\mathtt{G}' = (\mathtt{N}',\mathtt{E}')$, *and* $|\mathtt{N}'| = 2$, $|\mathtt{E}| = 1$, *via function* ϕ: $\mathtt{E} \longrightarrow \mathtt{E}'$ *such that* $\phi(\mathtt{e}) = \mathtt{e}'$:$<\mathtt{n}_1'$, $\mathtt{n}_2'>$ *and* $\mathtt{e}' \in \mathtt{E}'$, $\mathtt{n}_1'$, $\mathtt{n}_2' \in \mathtt{N}'$. *Also we require* $\theta(\mathtt{n}_1) = \mathtt{n}_1'$ *and* $\theta(\mathtt{n}_2) = \mathtt{n}_2'$ *or vice versa.*

[2]Path expressions on the output node can be simulated by a combination of map nodes. Thus, for example it is possible to map a direct input edge to an output path expression (splitting of an edge) by combining several map nodes.

Structure Correctness of Connect Node. Given an input structure $\mathtt{S} \in \mathtt{Inst}(\mathcal{P})$ with edge $\mathtt{e}$:<$\mathtt{n}_1$, $\mathtt{n}_2$> $\in \mathtt{S}$, the connect node is said to be correct if the output structure $\mathtt{S'}$ it produces is also in $\mathtt{Inst}(\mathcal{P})$.

4.1.3 Ident Node

The ident ($\ominus$) takes as input a pattern or a structure and produces as output an exact copy of the input pattern or structure respectively.

Ident Node for Patterns. The ident ($\ominus$) takes as input a pattern $\mathtt{P}$ with a complex root node and produces as output a pattern $\mathtt{P'}$ such that $\mathtt{P}$ and $\mathtt{P'}$ are isomorphic.

Definition 7 (Ident Node - Pattern) *Given an input pattern $\mathtt{P}$ with $\mathtt{G} = (\mathtt{N},\mathtt{E})$, $|\mathtt{N}| > 1$ and $\mathtt{nr} \in \mathtt{N}$ is the root, and output pattern $\mathtt{P'}$ with $\mathtt{G'} = (\mathtt{N'},\mathtt{E'})$, an ident node, denoted by $\ominus$, maps $\mathtt{P}$ to $\mathtt{P'}$ via a bijection θ: $\mathtt{N} \longrightarrow \mathtt{N'}$ such that for every $\mathtt{e}:<\mathtt{n}_1,\mathtt{n}_2> \in \mathtt{E}$ there exists an edge $\mathtt{e'}:(\theta(\mathtt{n}_1),\ \theta(\mathtt{n}_2)) \in \mathtt{E'}$ and $\mathtt{nr'} \in \mathtt{N'}$ is root of pattern $\mathtt{P'}$ with θ: $\mathtt{nr} \longrightarrow \mathtt{nr'}$ and quantifier $\rho(\mathtt{e}) = \rho(\mathtt{e'})$.*

Pattern Correctness of Ident Node. Given a pattern $\mathtt{P} \in \mathcal{P}$ as input, an ident node $\ominus$ is said to be correct if it maps to an output pattern $\mathtt{P'}$ such that $\mathtt{P}$ and $\mathtt{P'}$ are isomorphic[3].

Ident Node for Structures. The ident ($\ominus$) takes as input a structure $\mathtt{S}$ with a complex root node and produces as output a pattern $\mathtt{S'}$ such that $\mathtt{S}$ and $\mathtt{S'}$ are isomorphic.

Definition 8 (Ident Node - Structure) *Given an input structure $\mathtt{S}$ with $\mathtt{G} = (\mathtt{N},\mathtt{E})$, $|\mathtt{N}| > 1$ and $\mathtt{nr} \in \mathtt{N}$ is the root, and output structure $\mathtt{S'}$ with $\mathtt{G'} = (\mathtt{N'},\mathtt{E'})$, an ident node, denoted by $\ominus$, maps $\mathtt{S}$ to $\mathtt{S'}$ via a bijection θ: $\mathtt{N} \longrightarrow \mathtt{N'}$ such that for every $\mathtt{e}:<\mathtt{n}_1,\mathtt{n}_2> \in \mathtt{E}$ there exists an edge $\mathtt{e'}:(\theta(\mathtt{n}_1),\ \theta(\mathtt{n}_2)) \in \mathtt{E'}$ and $\mathtt{nr'} \in \mathtt{N'}$ is root of structure $\mathtt{S'}$ with θ: $\mathtt{nr} \longrightarrow \mathtt{nr'}$.*

Structure Correctness of Ident Node. Given a structure $\mathtt{S} \in \mathtt{Inst}(\mathcal{P})$ as input, an ident node $\ominus$ is said to be correct if it maps to an output structure $\mathtt{S'}$ such that $\mathtt{S}$ and $\mathtt{S'}$ are isomorphic.

4.1.4 Merge Node

The merge node ($\oplus$) maps either two input patterns or two input structures to produce as output one pattern or structure respectively.

Merge Node for Patterns. The merge node ($\oplus$) maps two input patterns $\mathtt{P}_1$ and $\mathtt{P}_2$ with roots $\mathtt{nr}_1$ and $\mathtt{nr}_2$ respectively to one output pattern $\mathtt{P'}$ with root $\mathtt{nr'}$. The set of patterns $\mathcal{P}_1$ and $\mathcal{P}_2$ reachable from the roots $\mathtt{nr}_1$ and $\mathtt{nr}_2$ and subsumed by the input patterns $\mathtt{P}_1$ and $\mathtt{P}_2$ respectively are mapped to the output pattern $\mathtt{P'}$ such that $\mathcal{P}_1 \bigcup \mathcal{P}_2$ are subsumed by $\mathtt{P'}$ and are reachable from its root $\mathtt{nr'}$. Consider the example given in Figure 7 (e). Here the merge node maps the patterns $\mathtt{P1}$ and $\mathtt{P2}$ to produce pattern $\mathtt{P'}$ such that all components stemming from the roots of $\mathtt{P1}$ and $\mathtt{P2}$ are subsumed in the output structure and are reachable from the root of the new pattern $\mathtt{P'}$.

[3] Isomorphism for patterns and structures is defined in Section 3.

Definition 9 (Merge Node - Pattern) [4] *Let* $\mathtt{P}_1$ *and* $\mathtt{P}_2$ *be two patterns. Here* $\mathtt{P}_1$ *has* $\mathtt{G}_1 = (\mathtt{N}_1,\mathtt{E}_1)$, $|\mathtt{N}_1| \geq 1$ *and* $\mathtt{nr}_1 \in \mathtt{N}_1$ *the root. Let* $\mathcal{P}_1$ *be the set of disjoint patterns rooted in* $\mathtt{nr}_1$ *and subsumed by* $\mathtt{P}_1$*. Let* $\mathbb{N}_1$ *and* $\mathbb{E}_1$ *represent the set of nodes and edges of* $\mathcal{P}_1$*. Let* $\mathtt{P}_2$ *and* $\mathcal{P}_2$ *be defined similar to* $\mathtt{P}_1$ *and* $\mathcal{P}_1$*.*

Given the patterns $\mathtt{P}_1$ *and* P_2 *as input, a merge node, denoted by* $\oplus$*, maps* $\mathtt{P}_1$ *and* $\mathtt{P}_2$ *to* $\mathtt{P'}$*, with* $\mathtt{G'} = (\mathtt{N'},\mathtt{E'})$ *and root node* $\mathtt{nr'}$*, by* θ: $\mathbb{N}_1 \bigcup \mathbb{N}_2 \longrightarrow \mathtt{N'}$ *such that* $\theta(\mathtt{nr}_1) = \mathtt{nr'}$; $\theta(\mathtt{nr}_2) = \mathtt{nr'}$; *and for every* $\mathtt{e:(nr,n_2)} \in \mathbb{E}_1 \bigcup \mathbb{E}_2$, $\mathtt{nr} = \mathtt{nr}_1$ *or* $\mathtt{nr} = \mathtt{nr}_2$*, there exists an edge* $\mathtt{e':(nr',}\theta\mathtt{(n_2))} \in \mathtt{E'}$; *and for every* $\mathtt{e:(n_1,n_2)} \in \mathbb{E}_1 \bigcup \mathbb{E}_2$*, there exists an edge* $\mathtt{e':(}\theta\mathtt{(n_1),}\theta\mathtt{(n_2))} \in \mathtt{E'}$*. If* $\mathtt{e}_1 \in \mathbb{E}_1$ *and* $\mathtt{e}_2 \in \mathbb{E}_2$ *such that* $\phi(\mathtt{e}_1) = \phi(\mathtt{e}_2) = \mathtt{e'}$, $\mathtt{e'} \in \mathtt{E'}$*, then* $\rho(\mathtt{e'}) = \mathtt{max}((\rho(\mathtt{e}_1)) + \mathtt{max}(\rho(\mathtt{e}_2))$[5]*.*

Pattern Correctness of the Merge Node. Let $\mathtt{P}_1, \mathtt{P}_2 \in \mathcal{P}$ with roots $\mathtt{nr}_1$ and $\mathtt{nr}_2$ respectively be two patterns. Let $\mathtt{P}_{r1}$ and $\mathtt{P}_{r2}$ be patterns subsumed by $\mathtt{P}_1$ and $\mathtt{P}_2$ respectively such that $|\mathtt{N}_{r1}| = 1$ and $\mathtt{nr}_1 \in \mathtt{N}_{r1}$, and $|\mathtt{N}_{r2}| = 1$ and $\mathtt{nr}_2 \in \mathtt{N}_{r2}$. Given two patterns $\mathtt{P}_1$ and $\mathtt{P}_2$, a merge node is said to be correct if it produces an output pattern $\mathtt{P'}$, such that $\mathtt{P'} \in \mathcal{P}$ and $\mathtt{P}_{r1}$ and $\mathtt{P}_{r2}$ are subsumed in $\mathtt{P'}$.

Merge Node for Structures. The merge node ($\oplus$) maps two input structure $\mathtt{S}_1$ and $\mathtt{S}_2$ with roots $\mathtt{nr}_1$ and $\mathtt{nr}_2$ respectively to one output structure $\mathtt{S}_o$ with root $\mathtt{nr}_o$. The set of components $\mathcal{S}_1$ and $\mathcal{S}_2$ reachable from the outer nodes (roots) $\mathtt{nr}_1$ and $\mathtt{nr}_2$ are mapped to the output structure $\mathtt{S}_o$ such that $\mathcal{S}_1 \bigcup \mathcal{S}_2$ are components reachable from the outer node (root) $\mathtt{nr}_o$ of the output structure $\mathtt{S}_o$. Consider the example given in Figure 8 (e). Here the merge node maps the structures S1 and S2 to produce structure S' such that all components stemming from the roots of S1 and S2 are components in the output structure S'.

Definition 10 (Merge Node - Structure) *Let* $\mathtt{S}_1$ *and* $\mathtt{S}_2$ *be two structures. Here* $\mathtt{S}_1$ *has* $\mathtt{G}_1 = (\mathtt{N}_1,\mathtt{E}_1)$, $|\mathtt{N}_1| \geq 1$ *and* $\mathtt{nr}_1 \in \mathtt{N}_1$ *the root. Let* $\mathcal{S}_1$ *be the set of components rooted in* $\mathtt{nr}_1$*. Let* $\mathbb{N}_1$ *and* $\mathbb{E}_1$ *represent the set of nodes and edges of* $\mathcal{S}_1$*. Let* $\mathtt{S}_2$ *and* $\mathcal{S}_2$ *be defined similar to* $\mathtt{S}_1$ *and* $\mathcal{S}_1$*.*

Given the structures $\mathtt{S}_1$ *and* S_2 *as input, a merge node, denoted by* $\oplus$*, maps* $\mathtt{S}_1$ *and* $\mathtt{S}_2$ *to* $\mathtt{S'}$*, with* $\mathtt{G'} = (\mathtt{N'},\mathtt{E'})$ *and root node* $\mathtt{nr'}$*, by* θ: $\mathbb{N}_1 \bigcup \mathbb{N}_2 \longrightarrow \mathtt{N'}$ *such that* $\theta(\mathtt{nr}_1) = \mathtt{nr'}$; $\theta(\mathtt{nr}_2) = \mathtt{nr'}$; *and for every* $\mathtt{e:(nr,n_2)} \in \mathbb{E}_1 \bigcup \mathbb{E}_2$, $\mathtt{nr} = \mathtt{nr}_1$ *or* $\mathtt{nr} = \mathtt{nr}_2$*, there exists an edge* $\mathtt{e':(nr',}\theta\mathtt{(n_2))} \in \mathtt{E'}$; *and for every* $\mathtt{e:(n_1,n_2)} \in \mathbb{E}_1 \bigcup \mathbb{E}_2$*, there exists an edge* $\mathtt{e':(}\theta\mathtt{(n_1),}\theta\mathtt{(n_2))} \in \mathtt{E'}$*.*

Structure Correctness of the Merge Node. Let $\mathtt{S}_1$, $\mathtt{S}_2$ with roots $\mathtt{nr}_1$ and $\mathtt{nr}_2$ respectively be two structures such that $\mathtt{S}_1$ and $\mathtt{S}_2 \in \mathtt{Inst(P)}$, i.e., they are instances of the same pattern P. Let $\mathtt{S}_{r1}$ and $\mathtt{S}_{r2}$ be components of $\mathtt{S}_1$ and $\mathtt{S}_2$ respectively such that $|\mathtt{N}_{r1}| = 1$ and $\mathtt{nr}_1 \in \mathtt{N}_{r1}$, and $|\mathtt{N}_{r2}| = 1$ and $\mathtt{nr}_2 \in \mathtt{N}_{r2}$. Given the two structures $\mathtt{S}_1$ and $\mathtt{S}_2$, a merge node is said to be correct if it produces an output structure S', such that S' is also an instance of the same pattern P and the set of components $\mathtt{S}_{r1}$ and $\mathtt{S}_{r2}$ are also components of S'.

4.1.5 Project Node

A project node maps a set of patterns subsumed in a pattern to an output pattern or a set of components of an input structure to an output structure.

[4] This is a very general definition of the merge node. More specific merge nodes can be described similarly.

[5] max returns the maximum quantifier ffrom the quantifier [min:max] pair.

Project Node for Patterns. Let $\mathtt{P}$ be a pattern with root $\mathtt{nr}$. The project node maps a set of patterns $\mathcal{P}_1$ subsumed in pattern $\mathtt{P}$ such that all patterns $\mathtt{p}_1 \in \mathcal{P}_1$ are rooted at $\mathtt{nr}$ to an output pattern $\mathtt{P'}$. Consider again the pattern in Figure 2 (a) where *Element* has two edges *hasA* and *Subelem*. A project can select an edge, for example *hasA*, and map it to the output pattern that now includes the root *Element*, the edge *hasA* and the sub-pattern on which edge *hasA* is incident, eliminating the edge *Subelem*. The project node is depicted in Figure 7 (d).

Definition 11 (Project Node - Pattern) *Let* $\mathtt{P}$ *be a pattern with* $\mathtt{G} = (\mathtt{N},\mathtt{E})$*,* $|\mathtt{N}| \geq 1$*, and* $\mathtt{nr} \in \mathtt{N}$ *be the root of* $\mathtt{P}$. *Let* $\mathcal{P}_1$ *be the set of disjoint patterns rooted in* $\mathtt{nr}$ *and subsumed in* $\mathtt{P}$. *Let* $\mathbb{N}_1$ *and* $\mathbb{E}_1$ *represent the set of nodes and edges respectively for* $\mathcal{P}_1$. *Given* $\mathtt{P}$ *as the input pattern, a project node, denoted by* $\ominus$*, maps* $\mathtt{P}$ *to an output pattern* $\mathtt{P'}$*, with* $\mathtt{G'} = (\mathtt{N'},\mathtt{E'})$*, by the bijection* θ*:* $\mathbb{N}_1 \longrightarrow \mathtt{N'}$ *such that for every* $\mathtt{e:(n_1,n_2)} \in \mathbb{E}_1$ *there exists an edge* $\mathtt{e':(\theta(n_1),\ \theta(n_2))} \in \mathtt{E'}$*, and* $\rho(\mathtt{e}) = \rho(\mathtt{e'})$*;* $\mathtt{nr'} \in \mathtt{N'}$ *is root of pattern* $\mathtt{P'}$ *and* $\theta(\mathtt{nr}) = \mathtt{nr'}$.

Pattern Correctness of Project Node. Given a pattern $\mathtt{P} \in \mathcal{P}$ as input, a project node is said to be correct if it maps to an output pattern $\mathtt{P'}$ such that $\mathtt{P'}$ is subsumed by $\mathtt{P}$ and $\mathtt{P'}$ *in* $\mathcal{P}$.

Project Node for Structures. Let $\mathtt{S}$ be a structure with root $\mathtt{nr}$. The project node maps a set of components $\mathcal{S}_1$ of structure $\mathtt{S}$ to an output structure $\mathtt{S'}$ such that the set of components $\mathcal{S}'_1$ of $\mathtt{S'}$ is a subset of $\mathcal{S}_1$. The project node is depicted in Figure 8 (d).

Definition 12 (Project Node - Structure) *Let* $\mathtt{S}$ *be a structure with* $\mathtt{G} = (\mathtt{N},\mathtt{E})$*,* $|\mathtt{N}| \geq 1$*, and* $\mathtt{nr} \in \mathtt{N}$ *be the root of* $\mathtt{S}$. *Let* $\mathcal{S}_1$ *be the set of components rooted in* $\mathtt{nr}$. *Let* $\mathbb{N}_1$ *and* $\mathbb{E}_1$ *represent the set of nodes and edges respectively for* $\mathcal{S}_1$. *Given* $\mathtt{S}$ *as the input structure, a project node, denoted by* $\ominus$*, maps* $\mathtt{S}$ *to an output structure* $\mathtt{S'}$*, with* $\mathtt{G'} = (\mathtt{N'},\mathtt{E'})$*, by the bijection* θ*:* $\mathbb{N}_1 \longrightarrow \mathtt{N'}$ *such that for every* $\mathtt{e:(n_1,n_2)} \in \mathbb{E}_1$ *there exists an edge* $\mathtt{e':(\theta(n_1),\ \theta(n_2))} \in \mathtt{E'}$*;* $\mathtt{nr'} \in \mathtt{N'}$ *is root of structure* $\mathtt{S'}$ *and* $\theta(\mathtt{nr}) = \mathtt{nr'}$.

Structure Correctness of Project Node. Given a structure $\mathtt{S} \in \mathtt{Inst}(\mathcal{P})$ as input, a project node is said to be correct if it maps to an output structure $\mathtt{S'}$ such that $\mathtt{S'}$ is subsumed by $\mathtt{S}$ and $\mathtt{S'}$ *in* $\mathtt{Inst}(\mathcal{P})$.

4.2 Map Edges

Input and Output Edges. Each map node is required to always have at least one input edge ($\twoheadrightarrow$) and one output edge ($\twoheadrightarrow$) that connect to patterns or structures of a data model or an application schema respectively. For example, when a cross node maps an input pattern $\mathtt{P}_1$ with a node labeled *Relation* to a pattern $\mathtt{P}_2$ with a node labeled *Element*, then the cross node has one input edge that ends at (node of) $\mathtt{P}_1$ and an output edge that is incident on (a node of) $\mathtt{P}_2$. The input and the output edges of a map denote here that a *Relation* is mapped to an *Element*.

Containment Edges. A containment edge from map node $\mathtt{mn}_1$ to map node $\mathtt{mn}_2$ indicates that the map node $\mathtt{mn}_2$ is part of the map node $\mathtt{mn}_1$. Containment edges between map nodes often (but not necessarily always) reflect containment edges between two nodes in the input pattern $\mathtt{P}_1$. For example, consider two cross map nodes such that $\mathtt{mn}_1$ maps an *Element* to a *Relation* (*Element*↠⊗↠*Relation*); and $\mathtt{mn}_2$ maps *E-Attribute* to *Attribute* (*E-Attribute*↠⊗↠*Attribute*). In that case, we may also have a connect node $\mathtt{mn}_3$ to map the edge *hasA* to *has* (*hasA*↠⊖↠*has*). Here $\mathtt{mn}_1 \rightarrow \mathtt{mn}_2$ and $\mathtt{mn}_1 \rightarrow \mathtt{mn}_3$, that is the nodes $\mathtt{mn}_2$ and $\mathtt{mn}_3$ are contained in map node $\mathtt{mn}_1$.

5 Map Model and Application Maps

In this section we give a brief overview of the graph-based map metamodel used to express map models and the application maps. Throughout we use $\mathtt{mn}$ to refer to a map node and $\mathtt{me}$ to refer to a map edge.

5.1 Map Patterns and Map Structures

5.1.1 Map Structure

We now extend and apply the concepts from Section 3 to maps, that is, we introduce the notion of *map patterns* and *map structures* for map modeling. A *map structure* is a directed acyclic graph whose nodes and edges are elements of our set of map metaconstructs $\mathcal{M} \bigcup \xi$. Roughly speaking, a map structure maps an input structure to an output structure. Figure 9 (b) depicts a map structure.

Definition 13 (Map Structure) *A structure is a five-tuple* $\mathtt{MS} = (\mathtt{MG}, \nu, \varepsilon, \mathtt{S}_1, \mathtt{S}_2)$ *where* $\mathtt{MG} = (\mathtt{MN}, \mathtt{ME})$ *is a directed acyclic graph, and* ν *and* ε *are typing functions:* ν: $\mathtt{MN} \rightarrow \mathcal{M}$ *and* ε: $\mathtt{ME} \rightarrow \xi$ *and* $\mathtt{S}_1$ *and* $\mathtt{S}_2$ *are the input and output structures as defined below.*

Let $\mathtt{ME}^i_{mn}$ be the set of input (↠) edges for map node $\mathtt{mn} \in \mathtt{MN}$. A structure $\mathtt{S}_1 = ((\mathtt{N}_1, \mathtt{E}_1), \mu_1, \epsilon_1)$ is defined as an *input structure* for a map structure $\mathtt{MS} = (\mathtt{MG}, \nu, \varepsilon, \mathtt{S}_1, \mathtt{S}_2)$ if for every map node $\mathtt{mn}$ each input edge in $\mathtt{ME}^i_{mn}$ is incident on either a node $\mathtt{n} \in \mathtt{N}_1$ or an edge $\mathtt{e} \in \mathtt{E}_1$. Moreover, every node and edge in $\mathtt{S}_1$ has at least one input edge from some map node $\mathtt{mn} \in \mathtt{MS}$ incident on it.

Let $\mathtt{ME}^o_{mn}$ be the set of output edges for map node $\mathtt{mn} \in \mathtt{MN}$. A structure $\mathtt{S}_2 = ((\mathtt{N}_2, \mathtt{E}_2), \mu_2, \epsilon_2)$ is defined as an *output structure* for a map structure $\mathtt{MS} = (\mathtt{MG}, \nu, \varepsilon, \mathtt{S}_1, \mathtt{S}_2)$ if for every map node $\mathtt{mn}$ each output edge in $\mathtt{ME}^o_{mn}$ is incident on either a node $\mathtt{n} \in \mathtt{N}_2$ or an edge $\mathtt{e} \in \mathtt{E}_2$. Moreover, every node and edge in $\mathtt{P}_2$ has at least one output edge from some map node $\mathtt{mn} \in \mathtt{MS}$ incident on it.

We use the same notion of *components* for map structures as defined for structures in Section 3.

Correctness of Map Structures. A map structure $\mathtt{MS}$ is said to be correct if for a given input structure $\mathtt{S}_1 \in$ *Inst*($\mathcal{P}$), it maps to an output structure $\mathtt{S}_2 \in$ *Inst*($\mathcal{P}$) and all map nodes $\mathtt{mn} \in \mathtt{MN}$ are correct by Section 4.

Properties of Map Structures. Consider two map structures $\mathtt{MS}_1 = (\mathtt{MG}_1, \nu_1, \varepsilon_1, \mathtt{S}_1, \mathtt{S}_2)$ and $\mathtt{MS}_2 = (\mathtt{MG}_2, \nu_2, \varepsilon_2, \mathtt{S}_3, \mathtt{S}_4)$, then $\mathtt{MS}_1$ can be *mapped* to $\mathtt{MS}_2$ via a pair of functions α: $\mathtt{MN}_1 \rightarrow \mathtt{MN}_2$ and β: $\mathtt{ME}_1 \rightarrow \mathtt{ME}_2$. Two map structures are *equivalent* if for each $\mathtt{mn} \in \mathtt{MN}_1$, $\nu_1(\mathtt{mn}) = \nu_2(\alpha(\mathrm{mn}))$ and for each edge $\mathtt{me}$:($\mathtt{mn}$, $\mathtt{mn'}$) $\in \mathtt{ME}_1$, $\varepsilon_1(\mathtt{me}) = \varepsilon_2(\beta(\mathtt{me}))$ and $\beta(\mathtt{me}) = (\alpha(\mathrm{mn}),\alpha(\mathtt{mn'}))$. Two map structures $\mathtt{MS}_1$ and $\mathtt{MS}_2$ are *isomorphic* if $\mathtt{MS}_1$ and $\mathtt{MS}_2$ are equivalent and if the two inputs $\mathtt{S}_1$ and $\mathtt{S}_3$ of the structures $\mathtt{MS}_1$ and $\mathtt{MS}_2$ respectively are isomorphic.

5.1.2 Map Pattern

Like a pattern defined in Definition 14, a map pattern describes a collection of structures that represents a specific composition of the map metaconstructs. The quantifier in a map pattern specifies the range, i.e., minimum and maximum number of times, an edge can occur in the corresponding map structure. Input and output of the map pattern are patterns as described in Section 3 implying that input and output edges from map nodes in the map pattern must be incident on nodes in the patterns. Figure 9 (a) depicts a map pattern.

Definition 14 (Map Pattern) *A map pattern is a four-tuple* $\mathtt{MP} = (\mathtt{MS}, \varrho, \mathtt{P}_1, \mathtt{P}_2)$ *where* $\mathtt{MS} = (\mathtt{MG}, \nu, \varepsilon, \mathtt{S}_1, \mathtt{S}_2)$ *is a map structure such that* $\mathtt{MG}$ *is a rooted tree, and* ϱ *is a function that associates a quantifier with each edge of* $\mathtt{MG}$*; and* $\mathtt{P}_1$ *and* $\mathtt{P}_2$ *are input and output patterns, respectively.*

Let $\mathtt{ME}^i_{mn}$ be the set of input edges for map node $\mathtt{mn} \in \mathtt{MN}$. A pattern $\mathtt{P}_1 = (\mathtt{S}_1, \rho_1)$ is defined as an *input pattern* for a map pattern $\mathtt{MP}$ if for every map node $\mathtt{mn} \in \mathtt{MN}$ each input edge in $\mathtt{ME}^i_{mn}$ is incident on either a node $\mathtt{n} \in \mathtt{N}_1$ or an edge $\mathtt{e} \in \mathtt{E}_1$. Every node and edge in the input pattern must have at least one input edge from some map node $\mathtt{mn} \in \mathtt{MP}$ incident on it.

Let $\mathtt{ME}^o_{mn}$ be the set of output edges for map node $\mathtt{mn} \in \mathtt{MN}$. A pattern $\mathtt{P}_2 = (\mathtt{S}_2, \rho_2)$ is defined as an *output pattern* for a map pattern $\mathtt{MP}$ if for every map node $\mathtt{mn}$, each output edge in $\mathtt{ME}^o_m$ is incident on either a node $\mathtt{n} \in \mathtt{N}_2$ or an edge $\mathtt{e} \in \mathtt{E}_2$. Moreover, every node $\mathtt{n}$ *in* $\mathtt{N}_2$ and every edge $\mathtt{e} \in \mathtt{E}_2$ has at least one output edge from some map node $\mathtt{mn} \in \mathtt{MP}$.

Correctness of Map Patterns. A map pattern $\mathtt{MS}$ is correct if for a given input pattern $\mathtt{P}_1 \in \mathcal{P}$, it outputs a pattern $\mathtt{P}_2$ such that $\mathtt{P}_2 \in \mathcal{P}$ and all map nodes $\mathtt{mn} \in \mathtt{MN}$ are correct by Section 4.

Properties of Map Patterns. Two map patterns $\mathtt{MP}_1$ and $\mathtt{MP}_2$ are *isomorphic* if $\mathtt{MS}_1$ and $\mathtt{MS}_2$ are isomorphic and if each map edge $\mathtt{me}_1$ in $\mathtt{MP}_1$, $\mathtt{me}_1 = \beta(\mathtt{me}_2)$ and $\mathtt{me}_2 \in \mathtt{MP}_2$ and $\varrho(\mathtt{me}_1) = \varrho(\mathtt{me}_2)$.

Map Structures and Map Patterns. The *instanceOf* relationship between a map structure and a map pattern is defined similar to that between a structure and a pattern. We first need to be able to show that a map structure $\mathtt{MS}_i$ can be represented by some pattern $\mathtt{MP}_i$ in a given set of patterns $\mathcal{MP}$. For this we first define a *map match.*

Definition 15 (Map Match) *A map structure* $\mathtt{MS} = ((\mathtt{MN}_s, \mathtt{ME}_s), \nu, \varepsilon, \mathtt{S}_1, \mathtt{S}_2)$ **matches** *a map pattern* $\mathtt{MP} = (\mathtt{MS}', \varrho', \mathtt{P}_1', \mathtt{P}_2')$ *where* $\mathtt{MS}' = (\mathtt{MG}', \nu', \varepsilon', \mathtt{S}_1', \mathtt{S}_2')$ *if for a map node* $\mathtt{mn} \in \mathtt{MN}_s$ *there is a set of edges* $\mathtt{ME}^{me}_{mn}$ *stemming from the* $\mathtt{mn}$ *such that they map to the same edge* $\mathtt{me}' \in \mathtt{ME}'$ *i.e.,* $\beta(\mathtt{me}) = \mathtt{me}'$ *for* $\mathtt{me}' \in \mathtt{ME}'$*, and* $|\mathtt{ME}^{\mathtt{me}}_{\mathtt{mn}}| \in \varrho(\mathtt{me}')$*;* $\mathtt{S}_1$ *is* instanceOf $\mathtt{P}_1$ *and* $\mathtt{S}_2$ *is* instanceOf $\mathtt{P}_2$.

A map structure $\mathtt{MS}$ is an *instanceOf* a set of map patterns $\mathcal{MP}_1$ if for each component $\mathtt{MS}_i$ of a map structure $\mathtt{MS}$, there is a pattern $\mathtt{MP}_i \in \mathcal{MP}_1$ such that $\mathtt{MS}_i$ *matches* $\mathtt{MP}_i$ by Definition 15.

We also define a *subsumption* relationship between map patterns, i.e., a set of map patterns $\mathcal{MP}_1$ is subsumed by another set of map patterns $\mathcal{MP}_2$, if for each map pattern $\mathtt{MP}_1 \in \mathcal{MP}_1$ there is a map pattern $\mathtt{MP}_2 \in \mathcal{MP}_2$ such that $\mathtt{MP}_1$ and $\mathtt{MP}_2$ are isomorphic; and the input pattern $\mathtt{P}_{i1}$ for $\mathtt{MP}_1$ is subsumed by the input pattern $\mathtt{P}_{i2}$ for map pattern $\mathtt{MP}_2$; and the output pattern $\mathtt{P}_{o1}$ for $\mathtt{MP}_1$ is subsumed by the output pattern $\mathtt{P}_{o2}$ for map pattern $\mathtt{MP}_2$.

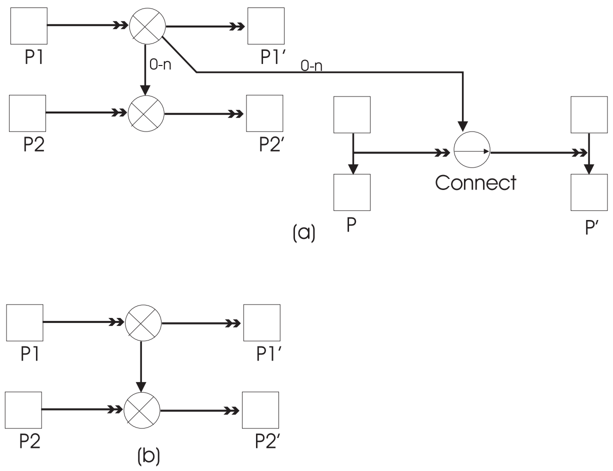

Figure 9: (a) An Example Map Pattern and (b) Map Structure.

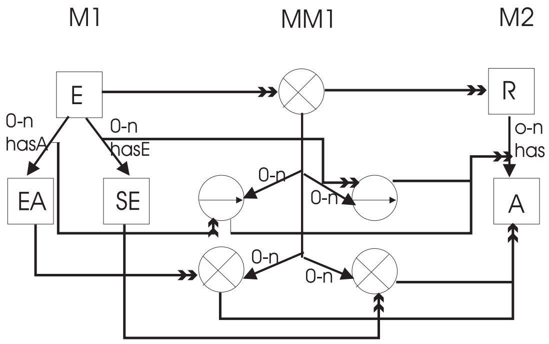

Figure 10: An Example Map Model to Map DTD to Relational Model.

5.2 Map Model and Application Model

Map Model. A map model $\texttt{MM} = (\mathcal{MP}, \eta, \delta)$ is composed of a set of map patterns $\mathcal{MP}$ with a labeling function η that maps the elements of $\mathcal{MP}$ which are of types $\mathcal{M} \bigcup \xi$ to a set of labels; and a constraint function δ that maps the map edges $\texttt{me} \in \mathcal{MP}$ to a set of constraints. Constraints are conditions applied primarily for the input and output edges of the map nodes. A constraint is satisfied if the condition holds true. The input and output patterns for each map pattern are also mapped via the labeling function γ to a set of labels (see the definition of a data model). Intuitively, a map model MM maps (a subset of) a data model $\texttt{DM}_1$ to (a subset of) another data model $\texttt{DM}_2$. Figure 10 depicts a map model that assigns labels to the map pattern represented in Figure 9 (a). This map model maps from DTD to the relational model.

Correctness of Map Model. A map model MM is correct from $\texttt{DM}_1$ and $\texttt{DM}_2$, if (1) all map patterns MP $\in \mathcal{MP}$ are correct; (2) all constraints mapped via the constraint function δ are satisfied; and (3) for a given input model $\texttt{DM}_1$ the map model produces a set of patterns $\mathcal{P}_1$ that is a subset of the set of patterns $\mathcal{P}_{DM2}$ for $\texttt{DM}_2$.

Application Map. An application map $\texttt{AM} = (\texttt{AS}, \kappa, \zeta)$ is defined by a map structure AS, a labeling function κ that maps each map node and map edge of AS to a label, a constraint function ζ that maps each map edge $\texttt{me} \in \texttt{AM}$ to a constraint, and a labeling function λ that maps each node and edge of $\texttt{S}_1$ and $\texttt{S}_2$ to a label. In other words, an application map defines a mapping between two application schemas. Figure 11 shows a segment of a mapping between an application DTD with an *Element*- Accommodation and the relational schema with *Relation*- ACCOMMODATION.

An application map AM is an *instanceOf* a map model MM if the structure MS that defines AM is an *instanceOf* $\mathcal{MP}$ that defines MM.

Correctness of Application Map. An application map AM is correct if (1) for a given correct input application schema $\texttt{AS}_1$, the application map produces a structure S that defines a correct application schema $\texttt{AS}_2$; (2) the application map AM is an instanceOf a map model MM; and (3) all constraints mapped via the constraint function ζ are satisfied.

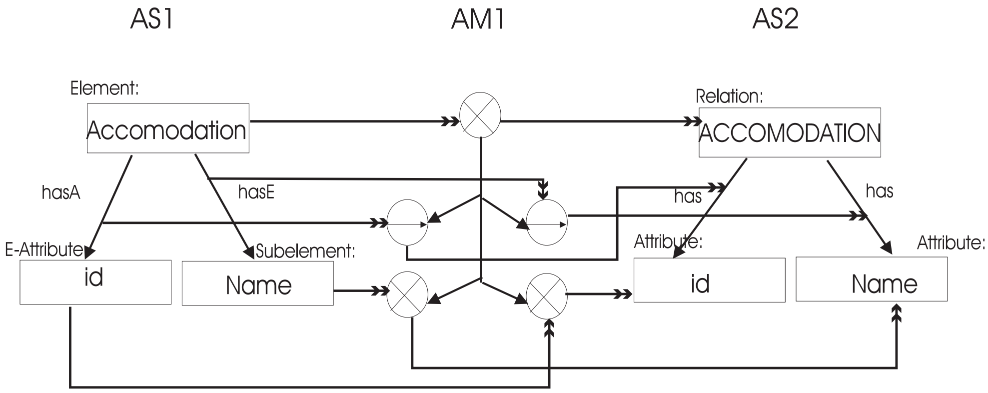

Figure 11: Example Application Map to Transform Application DTD to Relational Application Schema.

6 𝔖𝔞𝔫𝔤𝔞𝔪 - Our Model Management System

In the previous sections we have presented a theoretical framework that allows us to model maps between data models as well as between application schemas; and allows us to ensure that the outputs for the maps are correct. However, this is a far cry from the actual realization of the system that must now not only allow the modeling of these maps and ensure their correctness (as per the theory) but must also perform the data transformation and all the while making it easier for users to actually perform these tasks. Towards this goal, we now show how the proposed model management theory can be realized using a relational engine as data manager. Below we show the steps for defining a map between data models and between application schemas to illustrate the mapping theory in concrete terms. However, we would like to point out that once a map model is defined in 𝔖𝔞𝔫𝔤𝔞𝔪, most of the mapping steps illustrated here can be automated. Hence a user of 𝔖𝔞𝔫𝔤𝔞𝔪 while reaping benefits from its formal basis need not be aware of its details. In this section we describe the data model and map model layers and work our way down to the application data and data map layer (Figure 4). For simplicity, we use relational tables to represent the storage structures to capture models and maps, however an object-relational model is utilized in our implementation.

6.1 Setting up Data Models and Map Models

In this section we describe how a system administrator of 𝔖𝔞𝔫𝔤𝔞𝔪 would initialize the system by first entering data models and map models relevant for the domain.

Modeling Data Models. Table 1 describes the storage structure for explicitly modeling the data models in 𝔖𝔞𝔫𝔤𝔞𝔪. The `NODE` and `EDGE` tables represent the nodes and the edges respectively of patterns that define a data model (Section 3). Table 2 shows a subset of the DTD model captured as data in our metamodel tables in the data model layer (Figure 1). Other data models such as the relational model can be represented in a similar fashion.

Modeling Map Models. Similarly Table 3 stores a map model. The tables `MAP-NODE` and `MAP-EDGE` store the map nodes and map edges contained in the set of patterns that define a map model. Properties such

NODE

Field	Description
Node Name	Label of node.
Node Type	Complex (C) or atomic (A).
NodeID	Internal unique identifier.

EDGE

Field	Description
From-Node	Id of node that edge stems at.
Edge-Label	Label of edge.
To-Node	Id of node edge is incident on.
Edge-Type	Co - containment, P - property.
Quantifier	Range of edge.
EdgeID	Internal unique identifier.

Table 1: The Storage Structure for a Data Model in $\mathfrak{Sangam}$.

NODE

Node Name	Node Type	NodeID
DTD	C	N-1
Element	C	N-2
E-Attribute	C	N-3
SubElement	C	N-4
Group	C	N-5
Bool	A	N-6
String	A	N-7
Integer	A	N-8

EDGE

From-Node	Edge-Label	To-Node	Edge-Type	Quant.	EdgeID
DTD	Name	String	P	1-1	E-1
DTD	FullPath	String	P	1-1	E-2
DTD	HasE	Element	Co	1-n	E-3
Element	HasA	Attribute	Co	0-n	E-4
Element	Name	String	P	1-1	E-5
Element	Grps	Group	Co	0-n	E-6
Element	ID	String	P	1-1	E-7
Element	Subelem	SubElement	Co	0-n	E-9
E-Attribute	Required	Bool	P	0-1	E-10
E-Attribute	Name	String	P	1-1	E-11
E-Attribute	Type	String	P	1-1	E-12
E-Attribute	Fixed	String	P	0-1	E-13
Group	Required	Bool	P	0-1	E-14
Group	CanRepeat	Bool	P	0-1	E-15
Group	Order	Integer	P	0-1	E-16
Group	Subelem	SubElement	Co	1-n	E-17
SubElement	Required	Bool	P	0-1	E-18
SubElement	CanRepeat	Bool	P	0-1	E-19
SubElement	Order	Integer	P	0-1	E-20
SubElement	ElemDef	Element	Co	1-1	E-8

Table 2: The DTD Model Described Using the Structure in Table 1 in the Data Model Layer. We use the Name of the Node for Clarity. Here the Columns To-Node and From-Node are References (Foreign Keys) to the `NODE`.NodeID.

as label, type, quantifier, constraints, etc. are stored along with the map nodes and map edges in these tables. Table 4 captures the map pattern given in Figure 10. This map pattern maps a fragment of the DTD model (Table 2) to a fragment of the relational model.

6.2 Application Schema and Application Map Layer

In this section we describe how an application schema and a map between two application schemas are stored in the application layer of $\mathfrak{Sangam}$. Storage structures for the application schemas and the application maps are generated by our system directly from the data model (Table 1) and the map model (Table 3) respectively.

Application Layer. Data models described in the data model layer (such as Table 2) are used by the system to generate storage structures in the application layer. The Storage Structure Generator, `SSG` for short, in

MAP-NODE

Field	Description
Node Name	Label of map node.
Node Type	Ident (I), Cross (C), Merge (M), Project (P), Edge (E).
NodeID	Internal unique identifier.

MAP-EDGE

Field	Description
From-Map-Node	Id of map node edge stems from.
Map-Edge-Label	Label of map edge.
To-Map-Node	Id of map node edge is incident on.
Edge-Type	Containment (C), Input (I), Output (O)
Quantifier	Range for map edge.
MapConstraint	Constraint on map edge
EdgeID	Internal unique identifier.

Table 3: The Storage Structure for a Map Model.

MAP-NODE

Node Name	Node Type	NodeID
Cross-1	C	M-1
Cross-2	C	M-2
Cross-3	C	M-3
Edge-1	E	M-4
Edge-2	E	M-5

MAP-EDGE

From-M-Node	M-Edge-Label	To-M-Node	M-Edge-Type	M-Quant.	MapC	M-EdgeID
Cross-1	elem-in	Element	I	1-1	-	ME-1
Cross-1	rel-out	Relation	O	1-1	-	ME-2
Cross-1	att-con	Cross-2	C	0-n	-	ME-3
Cross-1	subE-con	Cross-3	C	0-n	-	ME-4
Cross-1	hasA-con	Edge-1	E	0-n	-	ME-5
Cross-1	se-con	Edge-2	E	0-n	-	ME-6
Cross-2	att-in	E-Attribute	I	1-1	-	ME-7
Cross-2	att-out	Attribute	O	1-1	-	ME-8
Cross-3	sube-in	SubElement	I	1-1	-	ME-9
Cross-3	att-out	Attribute	O	1-1	*	ME-10
Edge-1	hasA-in	hasA	I	1-1	-	ME-11
Edge-1	has-out	has	O	1-1	-	ME-12
Edge-2	sube-in	Subelem	I	1-1	-	ME-13
Edge-2	has-out	has	O	1-1	-	ME-14

Table 4: A Map Pattern describing the mapping of a DTD pattern to a relational pattern. This map pattern is part of the map model that maps an entire DTD to a relational model. We use the Name of the node for clarity purpose. * = If $|\texttt{SubElement.ElemDef.Element.hasA}| = 0$

Figure 13 is a two-pass algorithm. In the first pass, `COLLECT` (Figure 12) gathers the set of nodes that belong to a given data model by following the containment edges from a given root. In the second pass, we generate the application schema storage structure. Here we use the intuition that most nodes can have multiple instantiations. Thus given a data model `DM`, for every row in the `NODE` table representing a node `n` $\in$ `DM` we generate a table in the application layer with the table name equal to the node's label. All edges between the nodes are represented as attributes of the table generated for either the `From-Node` or the `To-Node` using the `Edge-Label` as the attribute name. Figure 13 gives the details of the algorithm.

Table 5 is gives a subset of tables generated from the Table 2 using the algorithms given in Figure 12 and 13 in the application layer (Figure 1). Table 6 depicts a segment of the sample application DTD (shown in Figure 2). The data in Table 6 is validated at loading time and thus ensures that it is an instanceOf the DTD model in Table 2. It thereby conforms to the structure of the DTD model and preserves the constraints and quantifiers specified in the DTD model.

```
COLLECT (Node: Root,
         Set<String>: TablesToCreate)
Root: root node (starting point)
TablesToCreate : Set of Node Labels
e: an edge, a row in the EDGE table
{
    if Root ∈ TablesToCreate:
        return;
    for all e in EDGE:
    {
        if (e.From-Node = Root)
            if (e.EdgeType) = Co
                TablesToCreate.add(e.To-Node)
                COLLECT(e.To-Node,
                        TablesToCreate)
    } }
```

Figure 12: First Pass: Collecting the names of all Types that must be Created in the Application Layer from the NODE and EDGE Tables.

```
SSG (Set<Strings>: TablesToCreate)
TablesToCreate : Node Labels to create tables
e: an edge, a row in the EDGE table
CreateDef(): creates empty types with name
addAttribute(): schema function to add an attribute
                to existing type
getType(): Helper function - returns handle on
           the type given a name.
{
  CreateDef(TablesToCreate)
  for all e in EDGE
    where e.From-Node ∈ TablesToCreate
{
      if (e.EdgeType = P)
        addAttribute(e.EdgeLabel,
            getType(e.From-Node),String)
      else if ((e.EdgeType = Co) &&
            (e.Quantifier = 0-n))
      {
        addAttribute(e.EdgeLabel,
            getType(e.To-Node),
            REF(getType(From-Node))
      }      else if ((e.EdgeType = Co) &&
            (e.Quantifier = 0-1))
      {
        addAttribute(e.EdgeLabel,
            getType(e.From-Node),
            REF(getType(To-Node))
      } } }
```

Figure 13: Second Pass: Creating Table Definitions Using Data in NODE and EDGE Tables.

DTD

Field	Description
Name	Name of the DTD
FullPath	Full path of the DTD
UID	Internal unique identifier

ELEMENT

Field	Description
Name	Name of the Element
ID	User assigned id for element
UID	Internal unique identifier
hasE	reference to DTD

E-ATTRIBUTE

Field	Description
Name	Name of the Attribute
Type	Type of the Attribute (String)
Required	Boolean
Fixed	Boolean
hasA	reference to Element.
ID	Internal unique identifier

SUBELEMENT

Field	Description
Required	Boolean
CanRepeat	Boolean
Order	Integer
ElemDef	reference to Element (element definition)
Subelem	reference to Element (parent element)
UID	Internal unique identifier

Table 5: Storage Structure in Application Layer for Application DTDs.

Application Map Layer. Similar to the data model, the map model is used to generate storage structures for the application map. Table 7 depicts the set of tables generated from the map model in Table 4. To generate this structure we use the map storage structure generator called MSSG. MSSG is also a two-pass algorithm. The first pass of the algorithm, similar to the COLLECT algorithm, collects a set of tables that

DTD

Name	FullPath	ID
Acc	url	D1

ELEMENT

Name	ID	hasE	UID
accommodation	E1	D1	D2
name	E2	D1	D3
PCDATA	E3	D1	D4

E-ATTRIBUTE

Name	Type	Required	Fixed	hasA	UID
id	CDATA	False	False	E1	D5

SUBELEMENT

Required	Can-Repeat	Order	Elem-Def	hasE	UID
True	False	1	E2	E1	D6
True	False	1	E3	E2	D7

Table 6: Application DTD Stored in the Application Layer.

need to be created. The second pass of the algorithm (`MSSG`) creates the table definitions as represented by the map model based on the algorithm given in Figure 14. Here each map node is represented by a table with the table name the same as the node's label. Different kinds of edges are treated differently. The details of this algorithm are given in Figure 14.

```
MSSG (Set<Strings>: MTablesToCreate)
MTablesToCreate : Map Node Labels to create types
e: an edge, a row in the EDGE table
CreateDef(): creates empty types with name
addAttribute(): schema function to add an attribute to existing type
getType(): returns handle on type given a name.
{
  CreateDef(MTablesToCreate)
  for all m in MTablesToCreate
    if m.NodeType = I or M or P
      createType((m.Name)-SECONDARY)

  for all e in EDGE        where e.From-Node ∈ MTablesToCreate
  {
      if (e.EdgeType = I or O)
        addAttribute(e.EdgeLabel, getType(e.From-Node), REF(getType(e.To-Node)))
      else if ((e.EdgeType = C) && (e.Quantifier = 0-n))
      {
        addAttribute(e.EdgeLabel, getType(e.To-Node),REF(getType(e.From-Node)))
      }
      else if ((e.EdgeType = C) && (e.Quantifier = 0-1))
      {
        addAttribute(e.EdgeLabel, getType(e.From-Node),
            REF(getType(To-Node))
      }
      else return;
      } } }
```

Figure 14: Second Pass MSSG: Creating Table Definitions Using Data in `MAP-NODE` and `MAP-EDGE` Tables.

Given the data in Table 4, the `MSSG` algorithm generates the set of tables for the application map layer given in Table 7. Data in this storage structure maps an application DTD to produce as output a relational application schema. Each table in this layer has an additional attribute which refers to the code segment (function) used for the corresponding data transformation as described next. Table 8 represents an application map, depicted in Figure 11, whose input is the application DTD given in Table 6. The application map in Table 8 is an instanceOf the map model in Table 4 and conforms to the structure and the constraints of the map.

It should be noted here that it is possible to automate the creation of the application map as depicted in Table 8. To automate this process a 𝔖𝔞𝔫𝔤𝔞𝔪 user needs to specify the map model as well as the input for the application map. It is a fairly straightforward process to then create the application map using this information.

CROSS-1

Field	Description
UID	Internal unique identifier
elem-in	Input for the Cross Node
rel-out	Output of the Cross Node
func	Data transformation function

CROSS-2

Field	Description
UID	Internal unique identifier
att-in	E-attribute input
att-out	Attribute output
att-con	Containment, Reference to Cross-1
func	Data transformation function

CROSS-3

Field	Description
UID	Unique Identifier
subelem-in	Sublement input
att-out	Attribute output
sube-con	Containment - reference to Cross-1
func	Data transformation function

EDGE-1

Field	Description
UID	Internal unique identifier
hasA-in	edge input
has	edge output
hasA-con	reference to Cross-1
func	Data transformation function

EDGE-2

Field	Description
UID	Unique Identifier
subelem-in	Subelement edge input
has-out	Attribute output
se-con	Containment - reference to Cross-1
func	Data transformation function

Table 7: Storage Structure for an Application Map Conforming to the Map Model in Table 4.

CROSS-1

UID	elem-in	rel-out	func
am1	accommodation	ACCOMMODATION	elem-rel

CROSS-2

UID	eatt-in	att-out	att-con	func
am2	id	id	am1	ea-a

CROSS-3

UID	subelem-in	att-out	sube-con	func
am3	name	name	am1	se-a
am4	PCDATA	name	am3	tostring

EDGE-1

UID	hasA-in	has-out	hasA-con	func
am5	hasA-id	ACC-has	am1	edgeT

EDGE-2

UID	subelem-in	has-out	sube-con	func
am6	se-name	ACC-has	am1	edgeT

Table 8: Application Map to Map the Application DTD in Table 6 to a Relational Structure.

6.3 Application Data and Map Data Transformers

Once the map between application schemas is in place, the next, perhaps the most crucial, step is to transform the input application data in the data layer to data in the desired output format. In 𝔖𝔞𝔫𝔤𝔞𝔪 we automate this process by encoding atomic units of transformation code as a part of each map node metaconstruct. For some map nodes there may be multiple default functions. A user or the system can select from these different choices at map generation time in the application layer. *Data transformers* in the data layer are generated using these atomic units of code and are composed as per the application map. While we are able to generate these transformers in a generic fashion, we rely on these pre-coded transformation functions to for instance extract data from XML documents or to load it into relational tables. The default functions may be overridden by a user-defined function such as a function that allows the user to merge two attributes, for example `First-Name` and `Last-Name` into one attribute `Full-Name` by concatenating the string values for the two attributes.

7 Related Work

Meta Data Modeling. Meta-modeling has been utilized to model many different types of information, data, roles and business rules [AT96, BR00, GL98]. In particular, it has been looked at as a middle-ware medium to handle schema integration and data transformation over the last twenty years [MR83, AT96, BR00, PR95]. Papazoglou et al. [PR95] propose a middle-layer meta-model to accomplish transformations between the OO and relational data models. The transformations are accomplished by a set of pre-defined translation rules that can convert the OO or relational data models to and from the middle-layer meta-model. Using translations as basic building blocks, they aim to automatically generate mappings from one given model to another at run-time. Atzeni et al. [AT96] have presented a framework to describe data models and application schemas. They focus on discovering translations between data models and hence application schemas. We make use of their graph model to express the data models and application schemas in our system. Commercially, Microsoft Repository [Ber99] and Rochade Information Model [Roc00] are meta-repositories that generically describe mainly data models for system integration purposes. To the best of our knowledge, these works all place the primary focus on the modeling of the data models and not of transformations at the meta level. In our work we now focus on modeling of maps between data models and between application schemas and the subsequent automatic generation of code to perform the data transformation.

Schema Integration and Data Transformation. The crux of MMS, the mapping, has been dealt with in the literature under the umbrella of schema transformation and integration [HMN$^+$99, MZ98, FK99, MIR93, CJR98]. However, this work is typically specific to either the application domain or to a particular data model and does not deal with meta-modeling [MR83, AT96, BR00, PR95]. Recent work related to ours are Clio [HMN$^+$99] a research project at IBM's Almaden Research Center and work by Milo and Zohar [MZ98]. Clio, a tool for creating mappings between two data representations semi-automatically (i.e., with user input) 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. [MZ98] have looked at the problem of data translations based on schema-matching. They follow an approach similar to Atzeni et al.[AT96] and Papazoglou et al. [PR95], but not at the meta-level, in that they define a set of translation rules to enable discovery of relationships between two application schemas. We can directly make use of translation algorithms from the literature, such as the algorithms for translating between an XML-DTD and relational schema [FK99] or mapping rules [MZ98], and model them as mappings in the MMS. However, our focus is not discovering such algorithms for mapping but rather on the generic expressibility of any possible (future) mapping and its management. Work on equivalence of the translations between models [MIR93] is of particular importance as such properties of maps can also be established.

8 Conclusions

In this paper we have presented a framework to allow modeling of mapping between different data models as well as to allow re-structuring within a data model. We are currently in the process of implementing $\mathfrak{Sangam}$ in Java using Oracle 8i (and its object extensions) as the MMS data store. A version of this system will be demonstrated at SIGMOD 2001 [CRZS01].

In conclusion, we would like to point out that while a model management approach may not be the quickest solution to mapping between data models, it still offers some immediate advantages in terms of flexibility and extensibility. Beyond its immediate advantages, we believe our approach has a more far reaching impact, for as today's data models become legacy models our framework does not become obsolete. But rather it is extensible in that we can, not only add other data models but we can also easily describe or re-target existing maps between such new data models using the map metaconstructs, thus getting actual transformation code for free with very little effort.

References

[AT96] P. Atzeni and R. Torlone. Management of Multiple Models in an Extensible Database Design Tool. In Peter M. G. Apers and et al., editors, *Advances in Database Technology - EDBT'96, Avignon, France, March 25-29*, LNCS. Springer, 1996.

[Ber99] Bernstein, P. A. and Bergstraesser, T. et al. Microsoft Repository Version 2 and the Open Information Model. *Information Systems*, 2(24):71–98, 1999.

[Boo94] G. Booch. *Object-Oriented Analysis and Design*. Benjamin Cummings Pub., 1994.

[BR00] P. A. Bernstein and E. Rahm. Data Warehouse Scenarios for Model Management. In *International Conference on Conceptual Modeling*, 2000.

[CJR98] K.T. Claypool, J. Jin, and E.A. 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.

[CRZS01] K.T. Claypool, E.A. Rundensteiner, X. Zhang, and H. et al. Su. Model Management - A Solution to Support Multiple Data Models, Their Mappings and Maintenance. In *Demo Session Proceedings of SIGMOD'01, to appear*, 2001.

[FK99] 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.

[GL98] S. Gbel and K. Lutze. Development of Meta Databases for Geospatial Data in the WWW. In *ACM-GIS*, pages 94–99, 1998.

[HMN+99] L.M. Haas, R.J. Miller, B. Niswonger, M.T. Roth, P. Schwarz, and E.L. Wimmers. Transforming Heterogeneous Data with Database Middleware: Beyond Integration. *IEEE Data Engineering Bulletin*, 22(1):31–36, 1999.

[MIR93] 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.

[MR83] L. Mark and N. Roussopoulos. Integration of Data, Schema and Meta-Schema in the Context of Self-Documenting Data Models. In Carl G. Davis, Sushil Jajodia, Peter A. Ng, and Raymond T. Yeh, editors, *Proceedings of the 3rd Int. Conf. on Entity-Relationship Approach (ER'83)*, pages 585–602. North Holland, 1983.

[MZ98] T. Milo and S. Zohar. Using Schema Matching to Simplify Heterogeneous Data Translation. In Ashish Gupta, Oded Shmueli, and Jennifer 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.

[PR95] M.P. Papazoglou and N. Russell. A Semantic Meta-Modeling Approach to Schema Transformation. In *CIKM '95*, pages 113–121. ACM, 1995.

[Roc00] Rochade Information Model. Rochade Information Model. http://support.viasoft.com/html/2RochDocDownload.html, 2000.