Towards an Integrated Model for Data, Metadata, and Operations

Jürgen Göres
goeres@informatik.uni-kl.de

Stefan Dessloch
dessloch@informatik.uni-kl.de

Abstract: Information integration requires manipulating data and metadata in ways that in general go beyond a single existing transformation formalism. As a result, a complete source-to-target mapping can only be expressed by combining different techniques like query languages, wrappers, scripting, etc., which are often specific to a single integration platform or vendor. Such a mapping is not portable across different alternative deployment scenarios, thus limiting the mapping's reusability and putting the considerable investment required to create it at risk. To avoid this vendor lock-in, we define an integrated representation for operations on arbitrary data and metadata that is independent of any specific metamodel or transformation language. Using it, we can express mappings in an abstract, vendor-neutral form, improving the interoperability of integration tools and the flexibility for the deployment of mappings.

1 Introduction

The goal of information integration is to provide a homogeneous, integrated view over multiple heterogeneous data sources. To overcome heterogeneity, a mapping or *integration plan* has to be developed that transforms the data found in the sources into the data model, structures, and formats of a desired target schema. Requiring a thorough understanding of data source semantics, integration planning remains essentially a manual process, making it perhaps the most time-consuming and expensive aspect of information integration. Once an integration plan has been created, it has to be deployed on a *runtime platform*. This can be full-fledged integration middleware like a federated DBMS or an ETL tool, but can also be a generic programming tool, a stylesheet, scripts used in a hand-crafted solution, or a combination of all of the above. Together, integration plan and runtime platform constitute the *integration system* that provides the target schema.

Information integration subsumes numerous tasks, which have to represent and modify metadata and data in some way: Metadata and data have to be translated between different metamodels or have to be analyzed for semantical correspondencies (schema and record matching), mappings that translate data between different schemas have to be developed, etc. While powerful integration tools and information system implementations exist for specific problems, the interoperability between these systems remains limited. E.g., there is no agreed standard for the exchange of discovered matches between schema matching and mapping tools. Even worse, the mapping tools themselves are commonly built to create mappings for a single integration platform only – consequently, the resulting integration plan is specific to the platform's internal data model (IDM) and infrastructure

(e.g., the wrappers). This tight coupling of integration tools, integration plan, and runtime platform is often an accepted fact in the traditional integration scenarios within a single organization. Here the requirements on the integrated view, the available data sources, and the resources dedicated to the integration system are assumed to be stable. But once new requirements go beyond what the original integration platform can handle, a migration to a new platform may be unavoidable.

In dynamic, open-world integration settings such as a grid environment, such restrictions are even less acceptable. With sources and users coming from different contexts, the degree of heterogeneity between data sources and user requirements is likely to be even greater than in a static, closed-world setting. Furthermore, grid data sources remain autonomous and can join and leave the grid at any time, making the tedious and expensive manual integration approach even less feasible. To enable the use of integration technology in such scenarios, the PALADIN project (Pattern-based Approach to LArge-scale Dynamic INformation integration) [Gö05b] explores concepts and techniques to bring cost-effective services for the planning and operation of integration systems to these new environments. After an integration plan has been created, a redeployment of a grid integration system can be necessary for several reasons: Service providers compete for customers in both price and performance. Grid services – being inherently unreliable – can become unavailable. Varying loads or changes in available data replicas can also require a migration.

To gain the necessary flexibility for redeployment, both the static and the dynamic integration scenarios require an abstract, vendor-independent formalism to develop and describe mappings. This allows not only to choose the best tools for each task during development, but also gives flexibility for the deployment of the result.

In this paper, we introduce a metamodeling approach based on typed, attributed multigraphs that – unlike existing approaches – is well suited not only for the efficient representation of metadata, but also for the representation of data. This is important, as many integration tasks do not operate on metadata alone, e.g., operators that turn data to metadata or vice versa, or instance-based schema matching. We then introduce a formalism based on graph transformations that is capable of describing arbitrary operations on our data/metadata representation, as well as sequences of these operations, which we refer to as *transformation plans*. We will demonstrate the formalism's practical use by describing the operational semantics of well-known data management operators and the chaining of these operators to form an abstract integration plan. We will then discuss how these abstract plans give us the desired flexibility, as they can be transformed into concrete plans for deployment into specific runtime environments.

The remainder of this paper is structured as follows: Section 2 discusses the requirements on an integrated method to represent data and metadata independently of specific metamodels and introduces the Paladin Metamodeling Architecture (PMA), our graph-based approach to satisfy these requirements. Section 3 introduces our graph transformation formalism. Section 4 classifies the types of operations it has to support. Section 5 provides an example of its use for data management. Section 6 briefly discusses how abstract integration plans can be mapped to concrete plans. Section 7 gives an overview of related work. Section 8 closes with a summary and an outlook on future work.

2 Graph-based Data and Metadata Representation

As the data/metadata representation plays a pivotal role for all subsequent operations, it has to fulfill a number of sometimes contradictory requirements: (1) It should be able to *represent data and metadata naturally*. This is required both as a conceptual foundation to describe the semantics of data management operations which modify this data representation (and its associated metadata), and to support model management operations, which also often depend on data samples or statistics. (2) It has to be *extensible* to support any existing or future metamodel, but (3) should ideally be able to *express common aspects* between the different metamodels, to allow their uniform handling. At the same time, the representation must be (4) *lossless*, i.e., preserve metamodel-specific characteristics. Finally, the representation of both data and metadata should be (5) *efficient*.

2.1 Paladin Metamodeling Architecture

Like similar approaches (e.g., [OMG06]), our graph model for generic data/metadata representation, the Paladin Metamodeling Architecture (PMA) (Figure 1), is based on the concept of metamodeling. The principal idea is to define a stack of (meta)layers. By instantiating elements on a layer M_n, elements of the layer M_{n-1} are modeled. Most approaches use four layers: A meta-metamodel (M3), which is usually fixed, provides the central concepts for defining different (new or existing) metamodels (M2), like SQL or XML. The instances of the metamodels represent concrete application models or schemas (M1), whose instances finally represent the application data (M0). The Paladin meta-metamodel (PMM) (M3) defines all the concepts available to define concrete metamodels in a graph model: A TypeGraph consists of GraphElements, the most important being NodeType and EdgeType. AttributeTypes can be attached to any GraphElement, their Domain is defined by refering to one of the built-in simple types. Note that the PMM is self-defining, i.e., every PMM element is also an instance of a PMM element. The PMM can therefore be understood as a type graph for type graphs. Examples of type graphs representing concrete metamodels (the SQL and XML metamodels) are shown in Figure 1 (M2). Any existing or future metamodel can be introduced into the PMA by specifying a suitable type graph (req. 2), which can be as detailed as required for a lossless representation (req. 4).

Each metamodel's type graph elements should inherit from the elements of the Core type graph, which capture common properties of existing data models like typing, inheritance, nesting of features or namespaces (req. 3). This later allows any model to be interpreted in terms of the Core metamodel, when model-specific aspects are not relevant. An essential differentiating feature is the availability of the PMM as a base of inheritance for concrete metamodels. Note how all elements of the Core, SQL and XML metamodels are both instances (indicated by the underlined part of their label), as well as subclasses of PMM elements (indicated in angle brackets). This allows the instances of type graphs (i.e, the concrete models or schemas, like the excerpt of a small human resources SQL schema shown in Figure 1 (M1)), to be themselves interpreted as type graphs that finally define the structure of those graphs that represent data, as illustrated in Figure 1 (M0), fulfilling

req. 1. This way we receive a model-specific data representation, which is – like that of the metadata – very efficient when compared to the generic representations found in the literature (req. 5): Schema, Table and Column are subclasses of NodeType, so their instances define nodes on the data layer M0. SQLType and its subclasses are instances of NodeType and a subclass of AttributeType. An instance of SQLType can therefore appear generically as a node in the model diagram, but also in its specific interpretation as an attribute of a node type instance to which it is connected by a hAtt edge (or one of its subclasses). The SQL schema in Figure 1 shows the two value SQLTypes in both interpretations. Instantiating the specific Table and Column instances with their attributes and the model-specific EdgeTypes between them, we can represent the actual tuples and the values of the tuple attributes. We reuse XML Schema's [BM04] powerful simple type system for atomic domains and for expressions and functions on them.

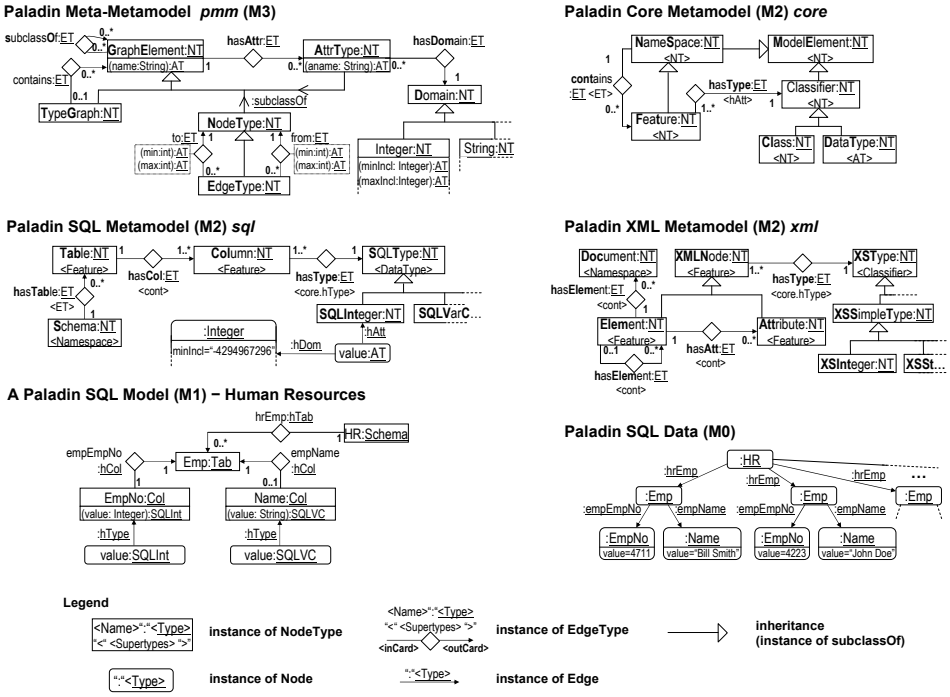

Figure 1: The PMA metalayer stack

2.2 Related metamodeling approaches

Existing metamodeling approaches have inspired the PMA. However, all these approaches have shortcomings, which we have addressed in PMA: The Common Warehouse Metamodel (CWM) [OMG03a] aims to ease the exchange of metadata in data warehousing scenarios. With its MOF meta-metamodel it is extensible (req. 2) and lossless (req. 4).

A Core metamodel, from which concrete metamodels inherit, captures common aspects of different metamodels (req. 3). However, to represent data, the CWM uses a generic instance *meta*model. This violates the metalayer concept and is not natural (violating req. 1), because data now resides on the same layer as metadata. This also results in a very verbose data representation (see [OMG03a] 4.6), violating req. 5.

The Graph eXchange Language (GXL) [HSSW06] is primarily an XML format for the exchange of graphs between different applications. But GXL also defines the language's underlying metamodel (the GXL metaschema, M2) of directed, nested hypergraphs, with typed and attributed edges and nodes. Type graphs (M1) are used to define the valid structure of graphs, and to specify the available node and edge types with their attributes. To fulfill req. 2, type graphs have to be used to represent different metamodels. While this allows the lossless representation of metadata (req. 4), GXL type graphs cannot import and inherit from other type graphs. It is therefore impractical to define something like a Core model (not satisfying req. 3). As actual metadata already resides on M0, we cannot instantiate concrete models again, as they are not type graphs themselves. This leaves us with no further metalayer to represent data, violating req. 1. A generic representation like CWM's instance model could be defined, which would, however, violate req. 5.

The semantic web standard RDF [MM04] provides the schema language RDFS, which allows to define complete RDF schemas for any metamodel (fulfilling reqs. 2 and 4). Arbitrary inheritance is possible between schemas, so a Core model can be defined (fullfilling req. 3). RDF(S) abandons the rigid separation of metalayers. As a consequence, an RDF Schema instance can inherit from RDF Schema constructs and can itself be interpreted as an RDF Schema. This essentially allows an arbitrary number of metalayers (fulfilling req. 1). However, RDF models tend to quickly become very complex and thus difficult to understand and modify, as RDF does not have attributes for resources and properties. While attributes for resources can be adequately represented with literal nodes, adding attributes to properties requires the cumbersome reification mechanism (violating req. 5).

3 Graph Transformations

With the generic data/metadata representation introduced in the previous section, we will now discuss how to describe operations that manipulate this representation by using a formalism based on graph transformations (GTs) (see [Hec06] for an overview, [AEH+99] for an in-depth discussion). Graph transformations have been subject to research for approximately 30 years. However, despite being a powerful tool to describe modifications of graph-like structures, they are still rarely used outside the graph community. On the one hand, this is due to the lack of an accepted standard for intuitive specification of graph transformations and differences in the operational semantics of these systems. On the other hand, few graph transformation systems (GTSs) have reached a state of maturity beyond that of research prototypes. Formalisms for specifying rules vary widely in their general approach, their formal properties and in their expressiveness (which is closely related to intellectual manageability of the resulting rule set). In this paper, it is not our intension to discuss their individual benefits and drawbacks, but to propose the use of the Paladin Graph

Transformation Algebra (GTA), our variant of *algebraic graph grammars* (AGGs), both for its expressiveness and – in our opinion – superior clarity of the resulting specification. For a comparison of the different approaches refer to [BFG96]. AGGs are a generalisation of Type-0 Chomsky grammars to non-linear structures. Like string grammars, an AGG consists of a set of *production rules*, which specify individual graph transformations.

3.1 Production Rule Language

To meet our specific requirements, we defined a variant of existing production rule formalisms. We will illustrate its essential semantics by means of the simple production rule shown in Figure 2. A production rule consists of a *left-* and a *right-hand side* (LHS, RHS), a *morphism* M and an optional *application condition* (AC). The LHS consists of a graph pattern, an abstract subgraph that describes which elements (nodes and edges) have to be found in the input graph (often called *host graph*) for the production rule to be applicable. To differentiate nodes and edges in a host graph from the nodes and edges on the LHS, we refer to the latter as *pattern nodes* and *pattern edges*. In a GTS with typed graphs, the LHS elements can specify a type test. The type test is given as the part of an element's label behind a colon. A subgraph of the host graph that matches a rule's LHS is called a *witness graph* or *occurence* of the LHS. In general, a rule can have several witness graphs in a given host graph. A witness graph can be expressed as an *occurence morphism*, i.e., a mapping of host graph elements to the pattern elements on the LHS of the rule. Morphisms are expressed by *binding variables*. In general, occurence morphisms can be non-injective, i.e., map a host graph element to more than one LHS element. Since arbitrary homomorphism can result in unexpected matches, groups of elements on an LHS among which general morphisms are allowed have to be explicitely specified. These groups are indicated by the homomorphic clause.

The right-hand side describes how a matched witness graph is modified by the rule. To connect LHS and RHS of a rule, we use a *rule morphism* to determine corresponding elements on the LHS and RHS, indicated by binding variables. By using non-injective rule morphisms, several LHS nodes can be merged into one, maintaining all edges that originally connected to one of the LHS nodes.

In the usual interpretation, a production rule is understood as replacing the structure in the host graph identified on the LHS with the structure found on the RHS, i.e., the host graph is modified *in-place*: Elements appearing on the LHS but not on its RHS are deleted, while elements on the RHS that have no corresponding element on the LHS are created. Since we often want to simulate algebra operators that are free of side-effects, we would have to repeat the elements of the LHS (i.e., the input for an operator) on the RHS, as the input elements (e.g., relations) would otherwise be directly modified by the transformation. This will not only lead to a lot of redundancy in rules, but also make specification of the newly created elements on the RHS more difficult. To suit this special requirement, we split the LHS elements into an *in-place* and a *copy* set. The in-place set has the usual semantics and can be used to do modifications on the host graph. The copy set indicates that corresponding elements on the RHS represent copies of this induced subgraph of the

witness graph. With the copy set, we can easily use host graph elements as templates for new graph elements. Copied RHS elements refer to the template LHS element via a morphism and can specify a new label. By connecting elements of the copy set to those of the in-place set, we can indicate an embedding of created and copied elements into the host graph.

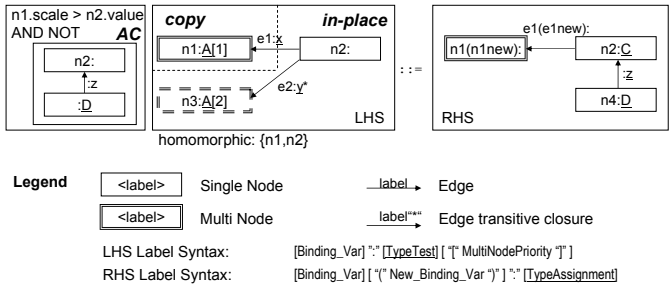

Figure 2: Overview of our graph transformation formalism

Application conditions are a generalization of various mechanisms found in other graph transformation languages to restrict a rule's applicability to those subgraphs of a host graph that meet additional structural and non-structural restrictions. An AC is essentially a boolean expression with the usual conjunction, disjunction, and negation junctors. Its elements can be "flat" expressions on attributes of the elements identified on the LHS, as well as graph patterns that must or must not appear together with a potential witness graph. Morphisms are used to identify elements in these graphs with those on the LHS of a rule.

Beyond the basic elements for specifying production rules that GTA has in common with many GTSs, different language extensions can increase the degree of expressive power of individual rules, but not of the GTS as a whole. E.g., optional elements (indicated by dashed lines) help to reduce the number of production rules required, as they allow to capture simple variants within a single rule. *Multinodes* on a LHS can match a set of nodes in the host graph and are indicated by double lines. However, the match semantics of two multinodes connected directly by an edge are not clear. One option is to prohibit edges between multinodes, as done, e.g., by PROGRES [Zü96]. Instead, we use cardinality constraints on edges to indicate whether, e.g., each of the matches to the first multinode must have exactly one corresponding match in the other multinode, or whether they have to be fully connected. Another problem occuring when using multinodes is *binding precedence*. Often a given host graph node can be mapped to more than one multinode in a rule. By specifying priorities for multinodes, the assignment becomes unambiguous.

3.2 Operational Semantics of Graph Transformations

So far, we have only discussed the elements used to construct individual production rules. Another important aspect of GTSs – and another aspect where they often differ considerably – is their operational semantics, i.e., the sequence in which rules are applied.

The basic usage method for graph grammars is to apply the rules non-deterministically and as long as possible until no more rules are applicable. Non-determinism is caused by two aspects, (1) choosing the rule and (2) choosing a concrete LHS occurence (i.e., a witness graph or morphism). However, for most practical applications of graph transformation systems, a way to enforce a strict or partial order in which rules are applied is needed, as well as a way to restrict the possible occurences to certain parts of the graph. While arbitrary control flows can be enforced by adding control flow elements to the host graph, this results in a tight coupling of rule and host graphs and therefore prohibits reuse. Approaches to structuring rule application go from a simple layering to full-fledged graph programming languages.

Rules for managing data/metadata are rarely applied completely non-deterministically: Since most of the operations on data/metadata graphs will only create new elements, a rule would be applicable to the same occurence in the host graph again and again. ACs can prohibit the rule if the new elements already exist. To express an actual use of an operator, we need a mechanism to apply rules in a strict or partial ordering, and to narrow down or determine where in a host graph they can be applied. E.g., to represent a more complex operator, it could be desirable to split its description into a strict or partial sequence of rules and connect them by passing the output of one rule as input to the next. On a coarser level, i.e., to describe an entire transformation plan, different operators (which could themselves be either atomic rules or a sequence of several rules) have to be connected in a similar fashion. Both aspects are covered by morphisms: A morphism connecting elements of a host graph with the pattern elements of a LHS pattern graph can be used to indicate complete bindings of the rule (i.e., to enforce a certain witness graph) as well as partial bindings to limit its applicability to parts of the host graph. Morphisms between one rule's RHS pattern elements and the LHS elements of another rule indicate both the sequence in which they are applied, as well as bindings between rules that indicate how the results of the first rule are to be used as input to the second. Together, production rules and their operational semantics constitute GTA, our *Graph Transformation Algebra* for modifying data and metadata represented using the PMA.

4 Types of Operations on Metadata and Data

Information integration deals with two general categories of operations on data/metadata: *Data management operations* are those classical operations that form the *parts* of an integration plan, e.g., SQL queries, relational algebra operator plans, wrapper configurations, stylesheets, etc. They focus on querying and modifying data and its associated metadata. *Model management operations* subsume those steps that aid the *creation* of an integration plan. We argue that our graph transformation approach is powerful enough to describe the operational semantics of any existing data or model management language in one uniform framework.

Existing data management languages are commonly mapped to or defined in terms of an underlying algebra, or have a minimal subset of elements that can be interpreted as an algebra. We will therefore use the terms algebra and language as synonyms.

The specific requirements of information integration have widened the scope and type of data management operations: One deficit of both relational algebra and SQL is the lack of operations to turn metadata into data or vice versa, i.e., to perform inter-layer transformations. Such functionality is often required during integration to resolve problems of schematic heterogeneity [BKLW99]. For example, a source table could model data category information as attribute values, while the target schema requires this information to be represented as individual columns. To resolve this, a pivot operation is needed, which is not supported in SQL, making it an intra-layer language. SchemaSQL [LSS01] addresses some of these problems by extending SQL, but lacks a proper theoretical foundation. FIRA and FISQL [WR05] continue where SchemaSQL left off and provide a complete set of data to metadata operators for RA and SQL, respectively, creating what the authors refer to as a *transformationally complete* algebra. An algebra for a given data model has this property, if it has the ability to transform arbitrary data into metadata, or vice versa. For the relational model, this requires to transform column and relation names to tuple values, or vice versa. FIRA and FISQL can be classified as inter-layer languages for the relational and SQL models, resp. For the XML metamodel, both XSLT and XQuery can be classified as inter-layer languages as well.

Another common problem is not solved by inter-layer languages: Often, data coming from sources of different data models has to be integrated, which requires a transformation of data and metadata between data models. Most existing approaches use a variant of the wrapper/mediator approach [Wie92]. A wrapper is a configurable software component specific to an integration platform that translates from the source's data model to the IDM of the platform. Since the operations performed by a wrapper cannot be mapped to any data-model-specific algebra, their effects cannot be described transparently in an integration plan, but only as a black box with their configuration parameters. For some types of mappings between data models, declarative languages have been proposed, often coming as an extension to an existing language. The most prominent example is SQL/XML, which allows to create XML fragments from SQL data. Together, these languages and wrappers represent *inter-metamodel* data management operations.

In section 5, we will illustrate how the operational semantics of individual data management operations can be represented by using GTA production rules, and how we can use this representation in conjunction with the rule chaining mechanism introduced in section 3.2 to describe sequences of these operators, i.e., abstract transformation plans.

Recently, the area of model management, introduced by the work of Bernstein, Melnik et al. [MRB03a, MRB03b], has received increasing recognition. [MRB03b] defines logical operators (i.e., an algebra) that work on models or schemas. Besides precisely defined concrete operators like the merging of models, or the deletion or extraction of parts of a model, some abstract operators stand for a number of alternative realizations. For example, *Match* is an abstract operator representing different schema matching heuristics to discover semantic correspondencies. Our graph transformation formalism is able to represent both concrete operations and specific realizations of abstract operators. Schema matching is an additional example of how model management can benefit from the PMA's ability to generically represent data (either completely or aggregated): Many *instance-based* approaches to schema matching use statistical information on the distribution of

data values. The PMA allows instance-based matching to be done independently of the concrete implementation and metamodel of the data sources to be matched.

Going beyond the existing model management operations that can support the manual creation of integration plans, our pattern-based approach to information integration presented in [Gö05a] aims at the automated creation of integration plans. It makes intensive use of graph transformations to represent both patterns and the resulting integration plans. Patterns are a machine-understandable, reusable representations of information integration problems and of knowledge on how to recognize and resolve them.

5 Graph Transformations for Generic Data Management

We will now demonstrate the use of GTA with a small example scenario. We will use GTA to describe (sequences of) production rules that represent operators and sequences of these operators, i.e., transformation plans. Figure 3 shows the schema and data of a small human resources database *HR*, with tables for employees and departments. Consider an application that needs an integrated relational view on this data source and has some specific requirements regarding its schema: The information about an employee's department should be combined into a single table row with his or her other data. The combined employee-department information should then be divided into individual tables for each city that is the location of a department. As departments open, close or are relocated, this information can change at any time, so the number of resulting tables is not predetermined.

The denormalization could be done with a simple join. The partitioning of the table, in order to be independent of the actual values for *city* found in the database, can be expressed with the FIRA partition operator $\wp$.

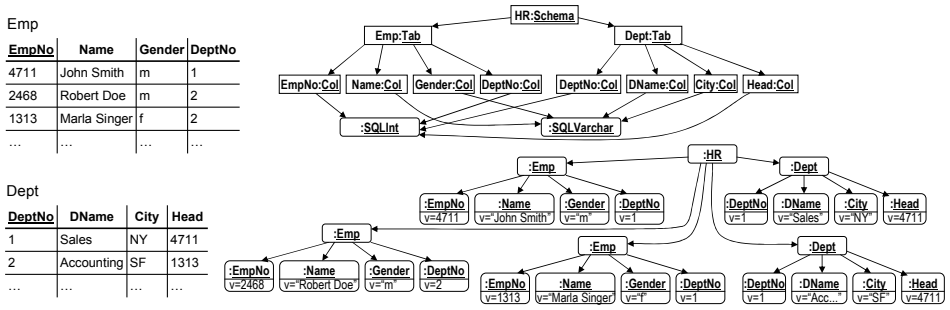

Figure 3: The human resource database represented using the PMA

5.1 Simulating a Basic Data Modeling Operation

This simple example shows that even though information integration has to deal with operations that go beyond the capabilities of basic data modeling languages, they are still the mainstay of many integration tasks. Usually, a considerable part of an integration plan will likely be modeled in a single data model and will only require basic functionality, like that provided by SQL. The left of Figure 4 shows how the relational join operator (the generalized θ-join) can be represented using the GTA. The operator is split into two production rules, one for the schema and one for the data part. On the LHS, the Join_Schema rule selects several elements out of the host graph: two tables t1 and t2 with multinodes for all their respective columns c1 and c2, the data types of the columns ty1 and ty2, and the (shared) schema the two tables reside in. Note that a *homomorphic* group is declared for the tables (to allow self-joins) and for the column types (to allow several columns to have the same type). The column sets c1 and c2 and table t1 are chosen for copying, while the other elements are used in-place on the RHS. A copy of t1 labeled tn is created, which is attached to the same schema to which t1 and t2 already belong. tn's name is set to the concatenation of the names of t1 and t2. All columns of t1 and t2 are copied and attached to the new table. The copied columns are connected to the same type nodes as their respective template.

The LHS of the rule handling the data part of our join (Join_Data) selects pairs of instances of the tables t1 and t2 (i.e., the tuples), together with instances of their respective columns (i.e., the tuples' attributes). Note that the only binding between the two rules is actually the type test of the LHS and the type assignments of the RHS. Copies of the instances of columns are attached to a new instance for the new table tn (i.e., a tuple of tn). The application condition consists of a placeholder which is bound to a boolean expression that represents the join condition parameter, so that only those pairs of tuples (tu1, tu2) that fulfill the join condition result in the creation of a tn tuple for the result table.

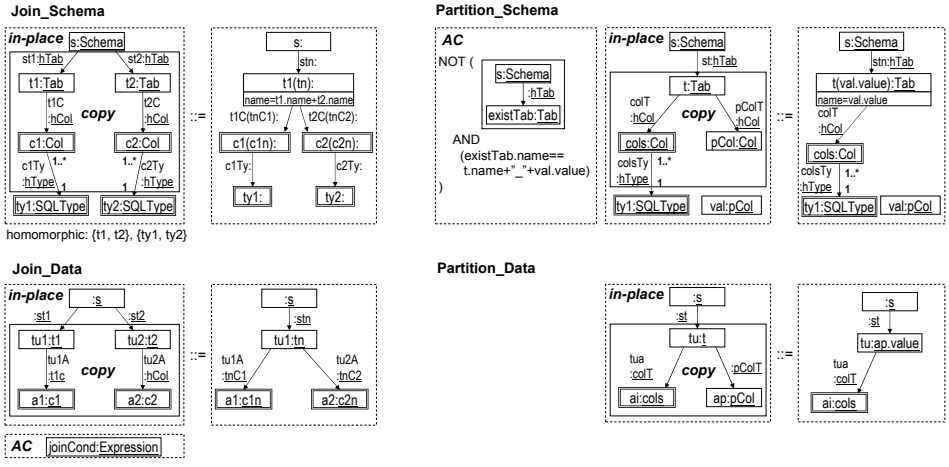

Figure 4: The θ-join ($t1 \bowtie_{joinCond} t2$) and partition ($\wp_{pCol}$) operators

Using morphisms between the LHS of operator rules and the host graph, and between the rules and the graph representing the join condition expression, we can now parametrize the GTA join operator to use it on the emp and dept tables (emp $\bowtie_{\text{emp.deptno=dept.deptno}}$ dept) of the HR database of Figure 3. Variable t1 is bound to the emp table, t2 to dept, s to the HR schema node. The join condition, given as an expression graph, is bound to the application condition of the Join_Data rule. This binding describes the first operator of our abstract integration plan shown in Figure 5.

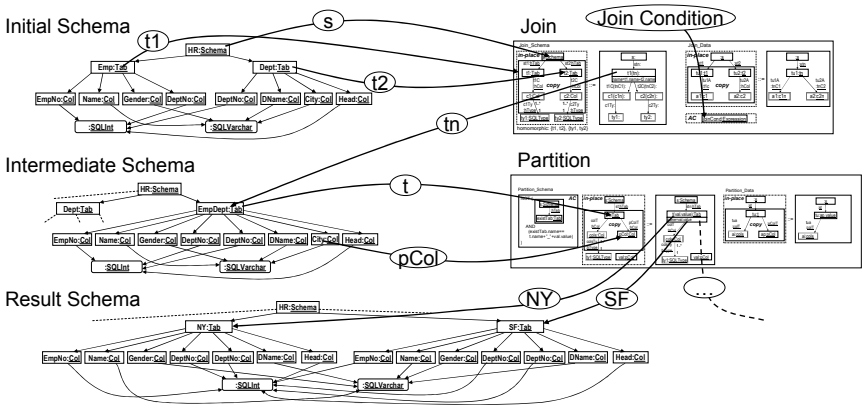

Figure 5: The abstract integration plan represented as chained graph transformations

5.2 Translating Data to Metadata

FIRA [WR05] defines operations that can translate data to metadata and vice versa. The partition operator $\wp_{\text{pCol}}\text{R}$ partitions an input relation R into newly created output relations based on the values contained in a column pCol given as parameter. Partitioning is generally used on *categorical attributes*, which have few distinct values when compared to the cardinality of the input relation. In our example HR database, we use the partition operator $\wp_{\text{City}}$empdept to split the result of the previous join, the empdept table, into separate tables depending on department's location. Its graph transformation representation is shown on the right in Figure 4. The operator is again split into a schema and a data rule. For each value of the partition column pCol, we create a new table that contains identical columns as the input relation, except for the partitioning column. Its respective value is used to define the name of the resulting table. As this rule would create a new table for each *occurence* of the value (not for each distinct value) we define a NAC stating that a table based on this value must not already exist in the schema. The data part of the operator takes instances of the input relation (tuples and their attributes) and creates an instance of the specific output relation that corresponds to the input tuple's value in the partition column.

We bind the GTA partition operator to the table node that represents the result table empdept of the previous join operation. To parametrize the operator, the pCol pattern

node is bound to the city column node of the same table. This binding gives us the second operator of our abstract integration plan in Figure 5.

6 Deployment of Mappings

An abstract representation for integration plans such as the one described above is a valuable tool: It allows us not only to describe integration plans in a way that is independent of a specific platform. Even more important, we can now express integration plans that require functionality going beyond that of a single platform (and thus also beyond the expressiveness of a single operator language), i.e., plans that would otherwise require a combination of languages and integration platforms. But with the graph transformation formalism, we not only have described an abstract operator algebra, but also a directly executable specification for a generic graph transformation system. While such a system is certainly useful for the creation and testing of mappings or the development of new operators, such a generic system could never be as efficient or be optimized to the same degree as the mature, but specialized existing information systems.

Therefore, once the tasks that benefit from a generic representation have been accomplished, e.g., the integration plan has been created, their results represented in the generic representation should be deployed to a suitable platform or a federation of platforms, if no single platform offers the full expressive power that would be needed. This is analogous to the idea of the Model Driven Architecture (MDA) [OMG03b]: GTA represents a domain-specific language for the domain of data and metadata management. A generic transformation plan described with GTA represents a platform-independent transformation model (PIM), which is then transformed into a platform-specific transformation model (PSM). Since we already have a graph representation for graph transformations and their combination, it is natural to integrate the deployment task into the overall PMA framework: For each concrete integration platform, a metamodel that represents its respective native transformation language is defined. Instances of this model represent concrete transformation sequences. For example, a metamodel for relational algebra would have a node type for each of the operators, with attributes representing parameters and edges indicating how the results of one operator are input for the next. To deploy an abstract plan to the specific representation, we can again use graph transformations to define a deployment specification, i.e., the mapping from abstract to concrete operators. A deployment rule's LHS refers to operators and morphisms of the abstract GTA representation, and on its RHS creates one or more appropriately configured platform-specific operators.

7 Related Work

As motivated in section 2.2, existing metamodeling approaches like the CWM [OMG03a], the Graph Exchange Language [HSSW06] and the Resource Description Framework RDF [MM04], have deficits when it comes to the efficient representation and handling of data,

which are corrected by the Paladin Metamodeling Architecture (PMA). Existing data and metadata management formalisms and systems that go beyond the expressiveness of established query languages, e.g., the FIRA/FISQL languages [WR05], or the Rondo model management framework [MRB03a, MRB03b] all focus only on supporting specific aspects of the general data/metadata management problem. To the best of our knowledge, our approach is the first to subsume the expressiveness of all these approaches.

A fundamental concept of the Model Driven Architecture [OMG03b] is the transformation of platform-independent models (PIM) of application logic to platform-specific models (PSM). This is comparable to our concept for deploying generic mappings to concrete runtime platforms. A number of approaches considers the use of graph transformations to perform these model-to-model transformations (e.g., [GLR$^+$02]). However, none of these approaches includes the modeling or transformation of data.

8 Conclusion and Outlook

We motivated how the area of data/metadata management in general, and information integration in particular, could benefit from a generic formalism for describing all possible operations on data and metadata. We introduced a generic data/metadata representation using an attributed, typed graph model. We defined a graph transformation formalism that allows us to define arbitrary operations on this representation in a natural way. We illustrated how this formalism can be used to represent data of existing data models and to mimic existing operations of the data model. We further motivated how the graph transformation formalism can also be used to specify transformations of transformation plans, e.g., to map the platform-independent transformation plans to platform-specific plans during deployment.

The power of our formalism can be used to perform a number of other data/metadata management tasks that we have yet to explore in more detail. For example, graph transformations can describe algebraic rewriting rules to optimize both abstract and the concrete transformation plans, structural schema matching approaches, or metadata management operations. While our current graph transformation formalism and its concrete syntax have shown to have adequate expressiveness, they need further streamlining to ease the specification of operators. At the same time, the amount of commonalities between different concrete metamodels that are captured by the core metamodel has to be increased, to enable the specification of more generic, metamodel-independent operators.

While our formalism is primarily intended as a theoretical framework, a reference platform for further refinement, like the refactoring of metamodels and developing operators, is needed. Existing graph transformation systems have proven to be inadequate for mapping some of the advanced constructs of our language and its copy semantics to their respective language concepts. Therefore, a prototype graph transformation system custom-tailored to our requirements is under development in our group.

Some open questions regarding the deployment process have still to be answered: The flexibility of GTA leaves many equivalent options to specify the same operator. This makes

their identification during deployment difficult. Besides the obvious solution of establishing standard representations for operators, one possibility we are looking into is to define a canonical form for graph transformations.

References

[AEH+99] Marc Andries, Gregor Engels, Annegret Habel, Berthold Hoffmann, Hans-Jörg Kreowski, Sabine Kuske, Detlef Plump, Andy Schürr, and Gabriele Taentzer. Graph transformation for specification and programming. *Sci. Comput. Program.*, 34(1):1–54, 1999.

[BFG96] Dorothea Blostein, Hoda Fahmy, and Ann Grbavec. Issues in the Practical Use of Graph Rewriting. In Janice E. Cuny, Hartmut Ehrig, Gregor Engels, and Grzegorz Rozenberg, editors, *Proc. 5th Int. Workshop on Graph Grammars and their Application to Computer Science*, volume 1073, pages 38–55. Springer-Verlag, 1996.

[BKLW99] Susanne Busse, Ralf-Detlef Kutsche, Ulf Leser, and Herbert Weber. Federated Information Systems: Concepts, Terminology and Architectures. Technical Report Forschungsberichte des Fachbereichs Informatik 99-9, 1999.

[BM04] Paul V. Biron and Ashok Malhotra. XML Schema Part 2: Datatypes Second Edition, October 2004.

[Gö05a] Jürgen Göres. Pattern-based Information Integration in Dynamic Environments. In *Proceedings of the 9th International Database Engineering Applications Symposium (IDEAS 2005)*, pages 125–134, July 25–27 2005.

[Gö05b] Jürgen Göres. Towards Dynamic Information Integration. In *Proceedings of the First VLDB Workshop on Data Management in Grids (DMG05), Trondheim Norway, September 2005*, number 3836 in LNCS, pages 16–29, 2005.

[GLR+02] Anna Gerber, Michael Lawley, Kerry Raymond, Jim Steel, and Andrew Wood. Transformation: The Missing Link of MDA. In *ICGT*, pages 90–105, 2002.

[Hec06] Reiko Heckel. Graph Transformation in a Nutshell. *Electr. Notes Theor. Comput. Sci.*, 148(1):187–198, 2006.

[HSSW06] Ric Holt, Andy Schürr, Susan Elliott Sim, and Andreas Winter. Graph eXchange Language. web site, 2006.

[LSS01] Laks V. S. Lakshmanan, Fereidoon Sadri, and Subbu N. Subramanian. SchemaSQL: An extension to SQL for multidatabase interoperability. *Database Systems*, 26(4):476–519, 2001.

[MM04] Frank Manola and Eric Miller. Resource Description Framework (RDF) Primer. web site, February 10th 2004.

[MRB03a] Sergey Melnik, Erhard Rahm, and Philip A. Bernstein. Developing metadata-intensive applications with Rondo. *J. Web Sem.*, 1(1):47–74, 2003.

[MRB03b] Sergey Melnik, Erhard Rahm, and Philip A. Bernstein. Rondo: A Programming Platform for Generic Model Management. In *SIGMOD 2003*, 2003.

[OMG03a] OMG. Common Warehouse Metamodel Spec., March 2003.

[OMG03b] OMG. The MDA Guide v1.0.1, June 2003.

[OMG06] OMG. Meta-Object Facility (MOF) 2.0 Specification. Specification, January 2006.

[Wie92] Gio Wiederhold. Mediators in the Architecture of Future Information Systems. *Computer*, 25(3):38–49, 1992.

[WR05] Catharine M. Wyss and Edward L. Robertson. Relational languages for metadata integration. *ACM Trans. Database Syst.*, 30(2):624–660, 2005.

[Zü96] Albert Zündorf. *PROgrammierte GRaphErsetzungsSysteme*. PhD thesis, 1996.