Pattern-based Information Integration in Dynamic Environments

Jürgen Göres
University of Kaiserslautern, Heterogeneous Information Systems Group
goeres@informatik.uni-kl.de

Abstract

The convenient availability of information is an essential factor in science and business. While internet technology has made large amounts of data available to the general public, the data is largely provided in human-readable format only. New technologies are now making direct access to millions of structured or semi-structured databases possible, but only through integration of these data sources maximum benefit can be gained. Traditional approaches to information integration, which involve human development teams and work in a controlled environment with a stable set of data sources, are not applicable due to the dynamic nature of such an environment. Therefore a higher degree of automation of this process is required. We present the PALADIN project (Pattern-based Architecture for LArge-scale Dynamic INformation integration), that uses machine-understandable patterns to capture and apply expert experience in the integration planning process.

1. Introduction

Information integration technology is already applied successfully to consolidate the IT infrastructure within many organisations. It provides tools to bridge the diverse forms of heterogeneity encountered between the systems deemed for integration, like different hardware platforms, operating systems, programming languages, and APIs (technical heterogeneity) as well as diverging data models and structuring of the data sources (logical heterogeneity). Planning and setting up an integration system is, however, to a large part still a manual process: It encompasses determining the data sources of interest that can contribute to the final system and the analysis of their schemas and technical properties. In a next step, correspondencies between the data sources are identified and mappings from the source data models and structures to those of the desired integration system are developed. This mapping is then used to set up the chosen runtime platform for the integration system. Since this process relies on human experts, it is both time-consuming and costly. Still it is adequate for many of the current intra-organisational scenarios, which are characterised by a limited and well-known set of source systems. These sources are usually under a single administrative control and have therefore only restricted autonomy.

New, still evolving forms of inter-organisational cooperation, however, are often more short-term in nature and require a more ad-hoc kind of integration: Foster et al. [7] describe *virtual organisations* (VOs), which aim at sharing computing resources like CPU cycles and main memory for a common goal. They describe the concept of *grid computing*, which was born out of this idea. A grid makes the heterogeneous distributed systems, which supply these resources, available over a network in a manner transparent for users and application programmers. Our work concentrates on information integration in *data grids*. This term originally stood for purely file-based grids, which were used to transfer input data and executables to the grid nodes in a computing grid and were later extended to provide distributed storage space for large amounts of raw scientific data. We refer to them as *storage grids*, to distinguish them from the true data or information grids where not CPUs or flat persistent storage space, but data from structured or semi-structured data sources is considered the resource to share. These scenarios are characterised by a large, ever changing set of autonomous data sources. Only the proper integration of these sources will make their use beneficial. However, their dynamic nature makes the traditional, manual integration methods unsuitable, as they rely on human developers and are therefore too slow and too expensive for ubiquitous deployment. Consider a virtual organisation that uses data found on the grid. As the data sources are provided by different organisations like companies or research institutes and are distributed globally, it is likely that sources become temporarily or permanently unavailable, while at the same time new sources have to be included. A conventional approach would obviously not be able to react timely to such changes. An integration planning system is required that can create a mapping of the data sources into the desired target schema with as little human interaction as possible.

The PALADIN project aims at reducing the amount of human expertise needed in setting up an integration system, enabling the use of integration technology in these dynamic

scenarios. While application domain experts are unlikely to be replaced even in the long term, we substitute the human information integration expert in those scenarios, where the application of integration technology has been impractical to date. We accomplish this by conveying expert knowledge in a machine-processable format, as *integration patterns*. A pattern basically consists of the description of a recurring atomic or larger-scale integration problem and directions on how to solve it. A pattern does not provide a specific solution to a specific problem. Instead, it supplies a generic description of the general problem situation to which it is applicable and a generic guideline for an appropriate solution that has to be adopted to the peculiarities of the concrete problem. For example, a pattern might describe how to map an SQL type into an appropriate XML Schema type or how to restructure a set of tables into a single table, if the desired target schema is denormalised.

PALADIN patterns work on a data model that describes source and target schemas and their data as attributed, typed multigraphs. The patterns consist of production rules that describe the transformation of a subgraph of schema elements and the respective transformation of the data held within these elements. Essentially, the source schema graphs have to be understood as a set of axioms, while the target schema is a hypothesis. The patterns form a kind of deduction rules. Integration planning can now be seen as the problem of proving the hypothesis, based on the axioms and using the deduction rules. If a sufficiently exhaustive library of patterns is provided, their repeated application will yield a successful deduction. This deduction can then be used as an integration plan by concatenating the operations on the data described by the patterns, resulting in what is essentially a tree of logical operators. This operator tree can then be mapped to the physical operators of a suitable runtime environment.

Unlike other attempts on automated information integration, PALADIN's set of available patterns is easily extensible via the declarative pattern language it provides. The description of patterns as graph transformations enables an expressiveness that goes far beyond the operator algebras of common data models and query languages. Where the additional expressiveness is not required, high-level languages can be used to describe the operation and then be compiled into the internal representation.

While our focus is on the planning phase, PALADIN also aims at supporting an increased degree of automation for the other essential steps of information integration, like the specification of the requirements (i.e., the target schema), the discovery of adequate data sources and the deployment of the integration plan into a runtime environment.

The remainder of this paper is structured as follows: Section 2 continues with a description of related work. Section 3 gives an overview of the PALADIN integration process and briefly describes the PALADIN metamodel that is used to convey metadata and data. Section 4 elaborates on the concept of integration patterns. Section 5 concludes with a summary and a perspective on remaining problems and future work.

2. Related Work

The grid and web services communities are both working on a standardisation of protocols and formats that solve many aspects of technical heterogeneity. The emerging web services technology [1] provides methods and languages to define and use platform, operating system, and programming language independent interfaces to services available over networks. Web services themselves serve as a vantage point for the efforts of the Global Grid Forum, which use their facilities to standardise interfaces for all services required in a grid. The Data Access and Integration Working group is defining Grid Data Service interfaces for accessing data and metadata of structured and semi-structured data sources [3]. These approaches, however, only tackle issues of technical heterogeneity, but offer nothing to resolve the diverse forms of logical heterogeneity found between different data sources. Still they will eventually yield standard interfaces for data source access that can be used by PALADIN.

Inspired by the precise classification of the different forms of heterogeneity encountered in information integration defined by Busse et al [6], we divide logical heterogeneity, whose resolution is the primary focus of PALADIN, into three forms: *Data model heterogeneity* subsumes fundamental differences in the data model used to describe the data source schemas like the differences between the relational world of SQL, the hierarchical nature of XML, or the nets of interwoven objects in the object-oriented paradigm, as well as the subtle differences between different implementations of the same data model. *Schematic heterogeneity* describes scenarios where identical domain concepts are modelled using the same data model, but different data model elements. Even if two data sources agree on a common data model and represent the domain using identical data model concepts, these identical schema elements can still be used differently, a situation which is referred to as *structural heterogeneity*. A well-known example is a varying degree of normalisation.

These forms of heterogeneity have been subject of diverse research projects. Numerous mappings between different data models, with a strong emphasis on translating between XML and the relational world, have been defined (e.g. [10]). These mappings, however, are commonly defined imperatively as algorithms, which limits their ability to handle more complex situations. They often yield unnat-

ural mappings that are not a good starting point for further integration planning. Instead of supplying a set of mapping algorithms that can only be applied as a whole, we will use our pattern concept to describe and reuse atomic data model mapping operations that can be used and combined with respect to the semantics of each data source schema.

The problem of schematic heterogeneity has been identified by [9]. The authors define SchemaSQL, a language that extends standard SQL with capabilities to transform data to meta-data and vice versa. Their approach, however, is limited to the relational context, and they do not aim at using it in automated schema integration.

Sauter et al. [8] describe BRIITY, a language and system to define mappings between differently structured database schemas. They put considerable effort in the definition of updatable mappings. As their understanding of structural heterogeneity includes data model heterogeneity, they also describe mapping between the relational and an object-oriented data model. While they list a large number of common structural problems and exemplarily show how to map them using the BRIITY language, they do not create these mappings with automated support.

SchemaSQL and BRIITY can serve as an inspiration for a high-level language to allow an easier description of schematic and structural patterns. BRIITY's large number of samples are essentially exemplified patterns, which we plan to reuse by describing them in a generic way.

While, due to the limitations of standard SQL, structural and schematic heterogeneity is a considerable problem in relational systems, emerging query languages for XML [4] are generally sufficiently expressive to describe most patterns in the XML data model.

To successfully discover a mapping between different schemas, the inherent semantic heterogeneity among them has to be identified. It results from the use of different terms for identical concepts (synonyms), broader or narrower terms (hyper- or hyponyms) or identical terms that have different meaning depending on domain and context (homonyms). As true machine-understandable semantics are out of reach at the current state-of-the-art in artifical intelligence, we will use schema matching techniques (see [13] for an overview) that identify correspondencies between the schema elements. Existing techniques usually operate on identifiers and structure of schema elements. Approaches to automated schema matching like Cupid [11] show promising results. These systems, however, are usually so-called *hybrid* schema matchers, i.e. they implement a limited range of different techniques and provide it as a monolithic component. Our framework instead encapsulates these individual techniques as atomic operators that can be independently implemented and chained together via standard interfaces. This allows the reuse of common techniques and algorithms and gives way to an empirical analysis of new matching methods and the best way to combine them. Matching operators include preprocessing steps (e.g. stemming of schema identifiers, synonym lookup in dictionaries etc.) and different matching strategies as well as methods to combine their results via *composite matchers*. However, the quality of the resulting correspondencies or matches is generally insufficient to serve as a reliable basis for integration. We therefore explicitly include the user in the schema matching process, allowing him to inspect, correct and amend the results of the matching process if desired.

Existing integration tools like IBM's Information Integrator [5] provide access to different data sources using wrapper technology and use SQL views to transform and integrate schemas. While limited to SQL operations, they can still be used as a runtime environment for PALADIN-generated integation plans, where their expressiveness is sufficient.

3. Architecture and infrastructure

In this section, we present the essential elements of the PALADIN architecture and describe the PALADIN metamodel (PMM).

3.1. Conceptual Architecture

The PALADIN architecture is presented in figure 1. The desired target schema and the schemas of the data sources, both represented in their respective native formats (e.g. SQL DDL, DTD, XML-Schema etc.), are first converted into a PMM representation. These (still unconnected) schemas are then matched semi-automatically using the PALADIN schema matching framework. The matched schemas are the vantage point for the actual integration planning, which uses integration patterns to identify a sequence of logical operations that transform the instances of the source schemas into instances of the desired integrated target schema. The details of this planning process will be presented in section 4.

The resulting logical operator tree will not be used directly. Instead it serves as a high-level, logical representation that can be translated into physical plans for concrete runtime environments, which can be chosen by the user or determined by other constraints. The potential spectrum of runtime environments ranges from existing integration tools over a purpose-built PALADIN runtime environment to a distributed integration architecture, where the individual operators are available as web or grid services, connected or choreographed using a web service flow language like BPEL [2]. With our focus on the planning phase, the necessary tooling and rules for physical plan generation and deployment are future work.

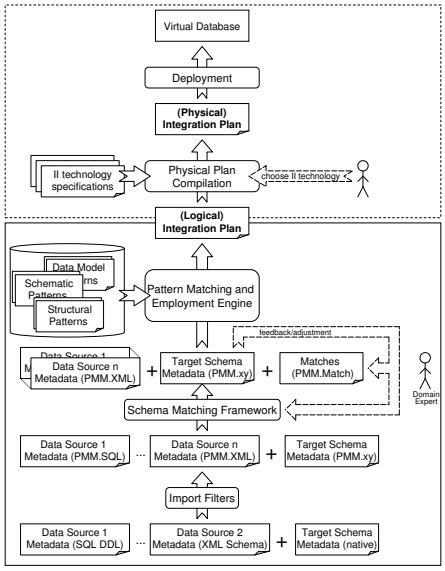

Figure 1. PALADIN conceptual Architecture

3.2. Metamodel

An essential component of the PALADIN infrastructure is a metamodel that is able to hold all schema and matching information throughout the planning process. Schemas of different data models are usually described using their own native (usually textual) representation, like SQL's Data Definition Language or one of the many XML schema languages. These specialised notations have to be replaced by a unified metamodel that recognises their respective special features, but also captures common properties of elements of the same or different data models. The Common Warehouse Metamodel (CWM) [12] provides such a unified metamodel. It uses the Meta Object Facility (MOF) to define metamodels for SQL, XML, the object model and many others. However, the CWM does not satisfy all our requirements on a metamodel. A general problem for practical use is the inherent complexity and fine granularity of the full CWM specification. Additionally, some of the predefined metamodels do not provide sufficient expressiveness. The CWM XML metamodel, for example, is based on Document Type Definitions (DTDs) and does not support many of the advanced constructs offered by XML Schema. The most severe limitation of CWM is, however, the data layer, which we need for instance-based schema matching and for defining how patterns handle data. Although CWM allows instance handling, it does so by providing an instance *meta*model, i.e. the actual instances themselves are located on the model layer, together with the models they instantiate. This does not only violate CWM's own fundamental concept (where every object on layer M_i is an instance of an object on layer M_{i+1}), but makes handling of instances extremely cumbersome and therefore unsuitable for describing a pattern's operational effects on the data.

Based on CWM, we define a simplified PALADIN metamodel (PMM), which fulfills our requirements and at the same time retains general compatibility to CWM models by borrowing CWM's concept of four meta layers, namely data (M_0), model (M_1), metamodel (M_2) and metametamodel (M_3), as well as the separation into different packages for each supported metamodel. We provide an XML metamodel that supports XML Schema. The main extension is, however, a conceptionally valid data layer (M_0), where every data element is a true instance of the respective element of the schema.

Figure 2 gives a brief overview of the central parts of PMM and its four meta layers and shows a subset of the model elements. Note that the figure omits most attributes for simplicity. We have also changed the way cardinality constraints are represented from the UML to the Entity-Relationship style, which allows us to naturally represent them in n-ary associations where $n > 2$. Elements on one layer are used as classes on the layer below and are themselves instances of the classes on the layer above.

The top layer (M_3) of PALADIN's metamodel provides the elements that are used to describe the different metamodels. It is self-describing (i.e. it can be described using its own elements) and enables the definition of arbitrary metamodels, therefore permitting the addition of new metamodels without hard-coding their semantics. All M_3 elements are derived from a common superclass. The main elements are *Class* and the *Association*s between classes. Associations are n-ary, i.e. they can connect an arbitrary number of classes, which requires the explicit modeling of *AssociationEnd*s to support rolenames and cardinality constraints. Both Classes and Associations can carry unstructured attributes that are typed using a subset of the built-in simple types provided by XML Schema.

On the metamodel (M_2) layer, we provide classes that represent the elements of the data models we wish to support in our system. The M_2 classes are separated into packages. The core package provides common functionality, which is utilised in the other metamodel packages by extending the core classes. With the specialised classes of the different data models like XML and SQL represented as subclasses of the core, we gain a natural mechanism for their integrated and uniform handling during the integration process. For example, a central element of the SQL package is the *Table* class, which conceptually shares many properties with the *ComplexType* class of the XML package. By letting both classes inherit from the abstract core class *Classifier*, we can handle them uniformly where their special properties are of no concern, e.g. during schema matching, which is usually based on the labels alone. To

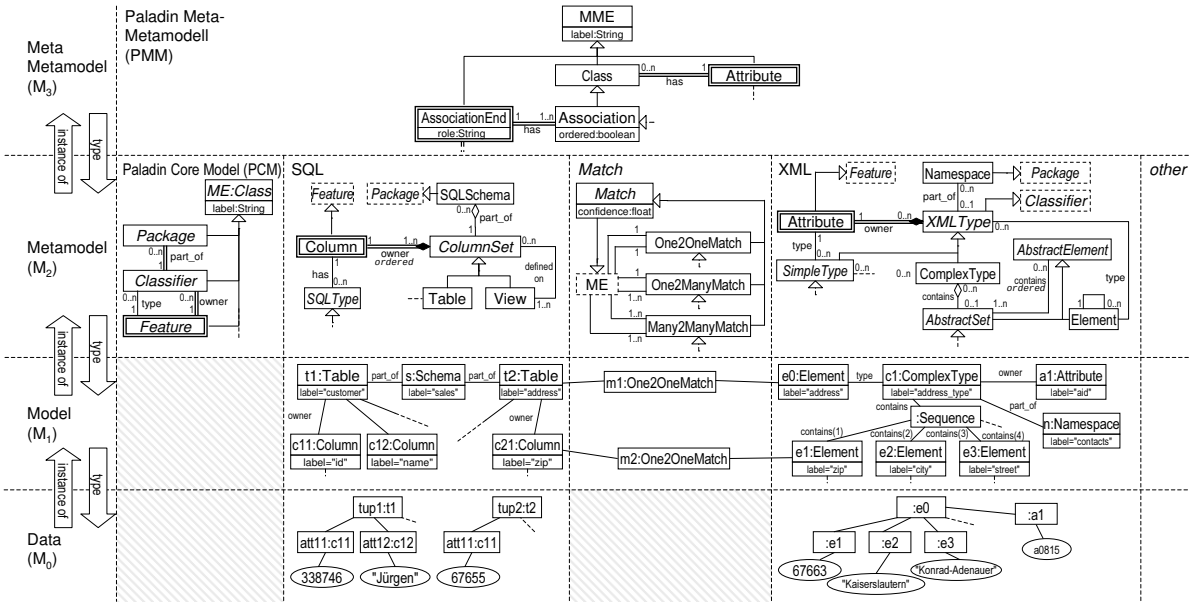

Figure 2. Simplified overview of the PALADIN metamodel (PMM)

capture the results of the schema matching process, we provide a dedicated match metamodel that provides classes and associations whose instances can describe correspondencies both between elements in the same schema and between different schemas, which can be based on identical or different data models. Inheriting from the basic *Match* class, we discern matches on an abstract level by arity, i.e. matches between individual elements, matches between a set and an individual element and matches between sets. These generic matches are subclassed by specific types of matches that carry semantics, like subclass-relationships, complete and incomplete unions, etc.

The model or metadata layer contains the instances that represent concrete schemas, shown exemplarily in figure 2. Unlike CWM, we explicitly include the data layer, which allows the representation of data with the means of our metamodel, therefore avoiding a technological gap that would otherwise result by using data source and data model specific interfaces. We require handling of data for two purposes: Instance-based schema matching, which actually looks into the data to determine correspondencies, can often improve the quality of the purely schema-based approaches and validate their results. We also need a way to define how patterns handle data, i.e. their operational semantics, which is described in detail in section 4. The M_0 layer is not intended to completely materialise a data source, but instead to provide data on demand, e.g. by wrapping a source specific interface. Consistent with the concept of different meta layers, each data element is represented as an instance of its model element. For example, a relational tuple is an instance of a table and is composed of attributes, which are themselves instances of the table's columns. The values are represented using special literal nodes, which use the aforementioned built-in types.

4. Integration Patterns

The central concept that distinguishes our approach from others is the idea that we do not simulate an information integration expert's work by providing algorithms that perform the necessary mappings and transformations, as they usually suffer from a lack of extensibility and realise only a limited set of standard solutions. Instead, we use graph transformations to specify machine-understandable integration patterns that allow the description of the isolated integration problem and its solution in a declarative way. This approach also makes expansion of the set of patterns possible without programming effort.

Each pattern contains a description of the changes on the schemas that result from its application, as well as how it is applied operationally, i.e. how it transforms the data. In essence, each pattern therefore describes a specialised operator. By defining the operator's effects as a graph transformation on a graph model that can handle arbitrary data models and their respective data, we gain an enormous amount of flexibility: We are not limited to the operators provided by existing algebras, which usually work only within a single data model and often cannot bridge the gap between data and metadata, i.e. handle schematic heterogeneity.

If the set of patterns is sufficiently large, an appropriate chaining of the patterns can yield a non-trivial transformation from a set of source schemas to the desired target schema. This tree of operators is essentially a query plan that defines the target schema as a view on the data sources. Like a common view definition, it is prefixed to the plan of every query on the target schema at runtime.

4.1. Graph Transformations

Graph transformation is a well-established concept. A graph transformation system operates on a graph-oriented data model, on which transformations are described as *productions* or rules. Many formalisms or languages for rule definitions exist. We chose a language loosely based on the PROGRES approach [15]. PROGRES uses a hybrid visual language, consisting of a mixture of graphical and textual elements. This allows both an easily comprehensible graphical representation of the essential aspects of every rule (having, of course, precisely defined semantics as well as a textual representation) and at the same time the expression of complex pre- and postconditions and transformation operations, which would be hard to capture graphically, by resorting to the textual elements of the notation. The graphical notation is essentially an alternative to path expressions on the associations between the nodes. Paths allow the definition of node sets and are the basic method for the definition of integrity constraints and production rules. Schürr [14] describes the theoretical foundations of this transformation language using predicate calculus.

The main components of a production are left- and right-hand side (LHS and RHS) graph patterns. Node sets are used on the left-hand side to select the subgraph which is subject to a transformation. The right-hand side shows the resulting subgraph when applying the production. Binding variables are used as identifiers to bind nodes on the LHS to nodes on the RHS. A bound LHS node that does not appear on the RHS is deleted. A RHS node without a corresponding LHS node signals the creation of a new node. A LHS node with a bound RHS node is preserved on the RHS, together with all attributes and context edges that are not explicitly manipulated on the RHS.

Figure 3 shows a simple production rule using only PROGRES's graphical notation to introduce these basic elements. The LHS defines a mandatory node of type *B*, which is connected to a mandatory set of nodes of type *A* via a *y* edge. Both are preserved by binding them to the variables *i2* and *i1*, respectively, and repeating them on the RHS. The optional set of type *C* nodes is given a binding variable but is not repeated on the RHS, indicating a deletion of these nodes if any exist. The crossed out node of type *D* is a negative node, i.e. for the pattern to be applicable, no nodes of type *D* must be connected to the *B* node via

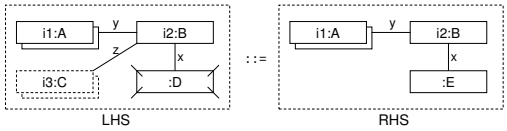

Figure 3. A production rule

an edge of type *x*. The node of type *E* has no counterpart on the LHS and indicates the creation of a new node.

4.2. Representing Integration Patterns as Graph Transformations

When imported from the native format into a PMM representation, the source and target schemas are represented as objects and associations on the model layer. This representation is essentially an attributed, typed multigraph, with objects and associations of the respective metamodel forming the nodes and edges of the graph. Initially, the graphs of the individual schemas are unconnected. Schema matching connects them using objects and associations of the Match metamodel.

The objective is to find a sequence of production rules that transform those parts of the source schema graphs that correspond to elements of the target schema into the target schema structure and, of course, to determine corresponding transformations for the schema instances (i.e. the data). A pattern describes an elementary or complex graph transformation, both on the model layer M_1 (i.e. the schema constellation before and after applying the pattern) as well as on the data layer M_0 (i.e. the operation required to transform the data within the transformed part of the schema). While the transformations on the model layer are used to guide the planning process, their corresponding operations on the data layer describe the operations for the runtime phase, i.e. the deployment of the plan. As an alternative to using a graph transformation on the data layer to describe the operations required for a pattern, it is often possible to use existing operators of an algebra for the respective data model (e.g. relational algebra or one of several XQuery algebras). Naturally, these algebras are limited to operations within their respective datamodel. They also often do not offer much with respect to schematic transformations (e.g. turning tables into attributes). For these cases, the graph-oriented description offers the possibility to describe a pattern's operational semantics without the need to actually define and implement an operator. This description can then be executed by a generic graph-based operator. Figures 4a–d show four sample patterns dealing with different forms of logical heterogeneity. Note that some details like identifier mapping and the resolution of naming conflicts are omitted for clarity.

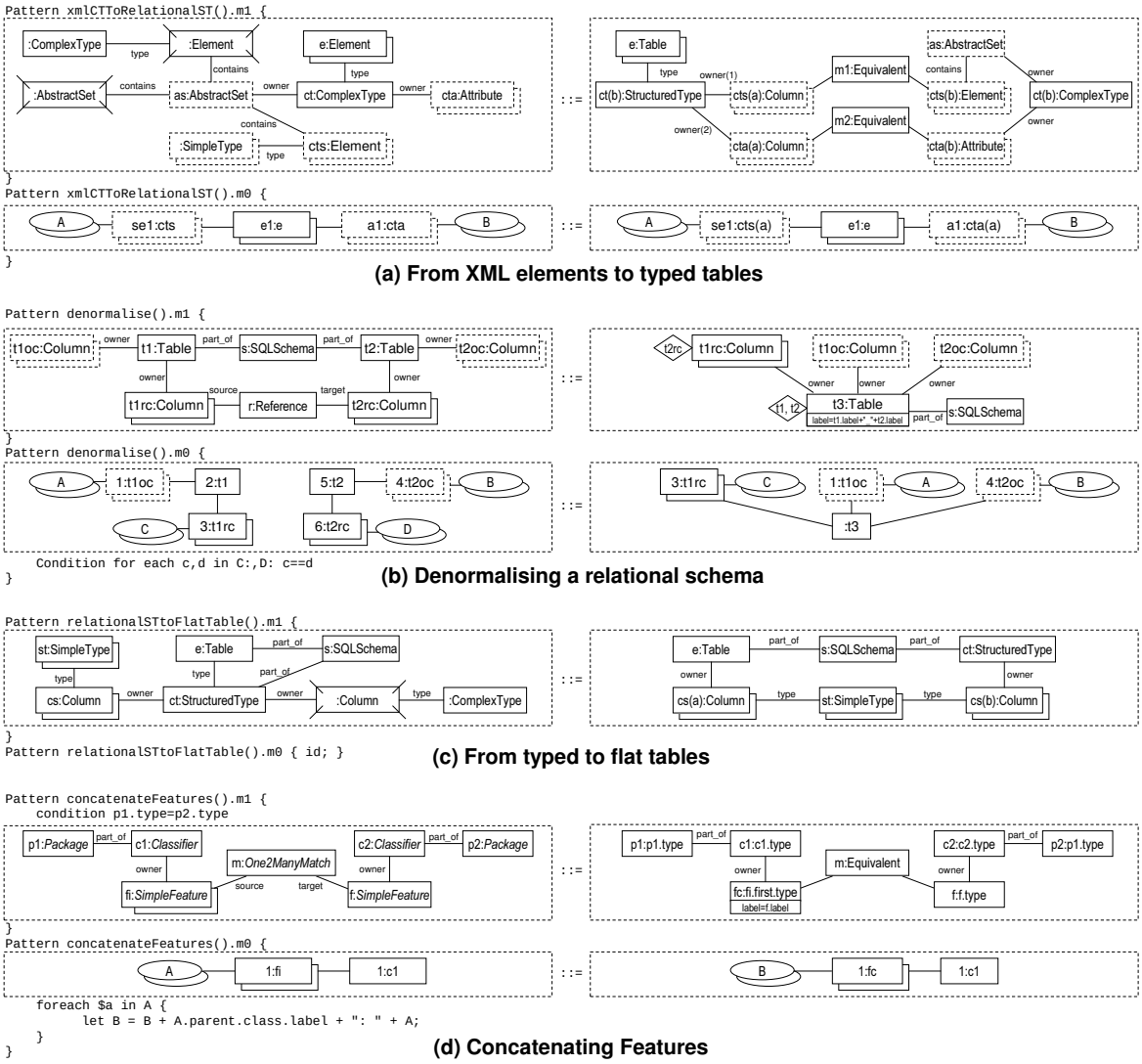

Figure 4. Example patterns

The data model pattern shown in figure 4a describes how to map an XML element based on a complex type, which may contain any number of attributes and any number of subelements of a simple type, but no complex subelements, to an SQL table based on a structured type, by creating a column out of every attribute and simple subelement. Notice that the XML complex type is preserved (as it might still be referenced by other elements) and how the pattern adds match information as annotations to the resulting schema. The pattern does not handle the mapping of the XML simple types to equivalent SQL types, as type conversions are needed in many other constellations and are therefore delegated to dedicated patterns.

The structural pattern of figure 4b maps two tables on the LHS that are connected via a reference (which can be a simple reference match, a foreign key etc.), to a single table on the RHS, i.e. it denormalises the two relations by joining them. Notice the diamonds on the RHS which indicate that all inbound edges of the deleted LHS elements *t1*, *t2* and *t2rc* are redirected to the new table *t3* and to the *t1rc* columns, respectively. The M_0 facet uses a condition to express the equi-join. A second structural pattern in figure 4c shows how to change a typed table without nested complex types into an untyped one. The structured type is preserved, its columns are cloned for the new flat table.

Another common problem is handled by the pattern depicted in figure 4d: Two or more features (i.e. columns, XML attributes etc.) correspond to a single feature in the target schema. The pattern resolves this problem by concatenating the features' values.

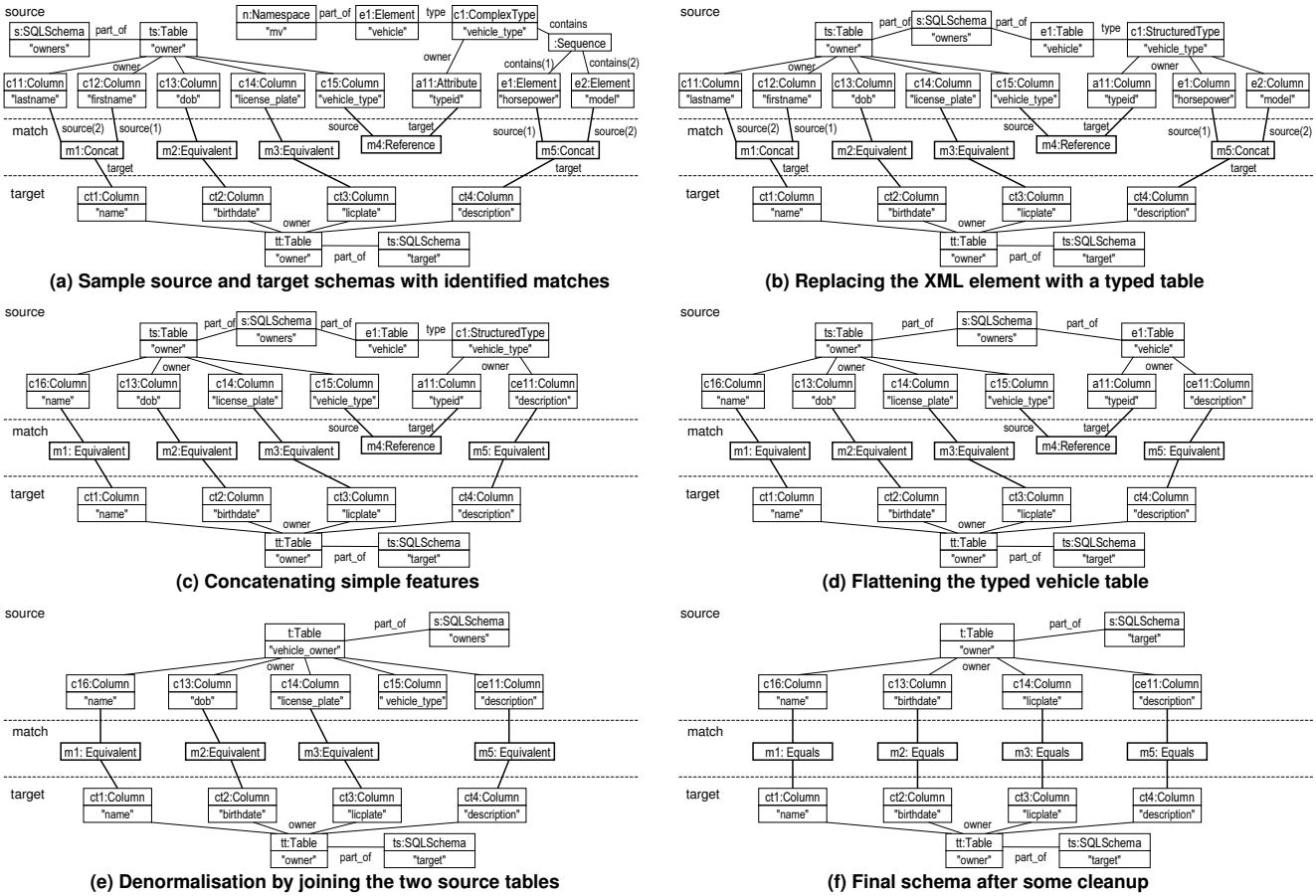

Figure 5. Application of Integration Patterns

4.3. Applying Patterns

Using the patterns and the graph interpretation of models and data described in the previous section, we will now illustrate the application of patterns on the source and target schemas shown in figure 5a. An XML schema stores vehicle types in elements based on an XML complex type. A relation holds information about vehicle owners, i.e. the type ID and license plate of each vehicle they own. The user, however, is interested in a target schema consisting of a single table which lists vehicle owners with license plate and a description of their vehicles. Schema matching has already identified (perhaps with the help of the end user) correspondencies among the elements of the schemas. The *typeid* of the XML complex type is referenced by the *vehicle_type* column of table *ts*. Other matches connect elements of the source and target schemas. Several 1:1 correspondencies exist, other attributes are in 1:n correspondency.

The schemas and the pattern applications are simplified to be presentable. For brevity, we omit types that are handled by the aforementioned type mapping patterns. Clean-up and conflict resolution patterns will be used implicitly on several occasions, like removing types that are no longer needed, deleting colums not required for the target schema (i.e. a projection operation) or renaming those schema elements that are identified as equivalent to a target element with a sufficient degree of confidence.

Our current pattern selection mechanism uses a simple greedy approach, i.e. it will generally chose the pattern that yields the maximum gain in similarity between the intermediate schema resulting from its application and the target schema. It will use backtracking if it reaches a dead end, i.e. a situation where no pattern is applicable and the transformation to the target schema is not complete. Similarity is defined based on the matches between the source and target schemas. In general a *One2OneMatch* (cf. figure 2) is preferable to a *One2ManyMatch*, which itself is better than a *Many2ManyMatch*. By considering the concrete type of match and by using the confidence in each match as modifier, we calculate a similarity measure. Pattern application stops when each target schema element is connected to a target schema element via an *Equals* match.

To further optimise pattern matching, integration planning is split into phases, which reduces the set of patterns that are tested. Initially, while elements of data models other than the target model remain, data model patterns of the proper type are prioritised. After all schema elements have been transformed into the target data model, first schematic and then structural patterns have precedence.

Of the set of patterns defined in figures 4a–d, several are applicable in the initial state. Since one of the source schemas is of a type different from that of the target schema, data model patterns receive higher priority. Figure 5b shows the resulting schema after applying the *xmlCTToRelationalST* pattern of figure 4a. The attributes and subelements of the complex XML type are converted into columns of a newly created SQL structured type.

The maximum increase in similarity between source and target schemas is gained by unifying the 1:n matches between features. Therefore, the *concatenateFeatures* pattern of figure 4d is applied twice, yielding the intermediate state shown in figure 5c.

As the target table is untyped, the next step flattens the typed vehicle table using the *relationalSTtoFlatTable* pattern (figure 4c), which leads to the schema of figure 5d.

Since the user is interested in a detailed description of the model for every owned vehicle, the two tables have to be joined, i.e. denormalised with the *denormalise* pattern introduced in figure 4b. The resulting schema in figure 5e is already close to what the user specified.

After the application of some clean-up patterns, the columns receive their proper names and the unneeded *vehicle_type* column is dropped. The result of these final steps is shown in figure 5f: *Equals* matches between all relevant schema elements indicate that planning has succeeded.

Figure 6 shows the resulting tree of operators, again skipping several clean-up operations and leaving out operator parametrisation. This logical representation of the required mapping operations can then be mapped to a concrete set of physical operators that are available in the chosen runtime environment. The resulting physical operator tree is then deployed to the runtime environment.

4.4. Schema Matching using Patterns

We have demonstrated how patterns can be used to find a sequence of operations that resolve data model, schematic and structural heterogeneity, thereby integrating a set of source schemas into a desired target schema. The principal idea of graph transformations can easily be extended to help resolving semantic heterogeneity as well, i.e. for schema matching. Even without extending the expressiveness of the pattern language, it is straightforward to define a pattern that adds an *equivalent* match node between two

Figure 6. The resulting operator tree

schema element nodes with literally identical labels. Dictionaries or thesauri can be supported by modelling them as a schema, either using an existing or a dedicated metamodel (like a simplified OO model with only generalisation/specialisation relationships) that captures the terms of the respective domain and their relationships. This reference schema can then be used in patterns like the one shown in figure 7. The reference schema *t* describes a subclass relationship between two terms, the pattern uses it to add an appropriate match between two concrete schema elements whose labels are identical to those of the thesaurus terms.

Similar patterns can be constructed for dictionary and acronym lookups. Complex preprocessing steps like stemming algorithms for the node and edge labels, however, have to be added as functions.

5. Conclusion and Future Work

We have motivated how new forms of cooperation based on sharing structured data in an ad-hoc manner in data grids require a new approach to information integration that removes the overhead of a human-driven and costly integration process. Such an approach has to account for the diverse nature of these data sources and must therefore be able to bridge many forms of logical heterogeneity, requiring assistance only to solve application-domain-related ambiguities during schema matching, which can easily be provided by the end-user. The creation of an integration plan, i.e. a mapping from the sources to the target schema has to be done by the system. PALADIN uses an extensible collection of integration patterns, which describe both common and specialised problems encountered during the integration process, as well as their solution, essentially capturing expertise in information integration in a machine-processable way. The patterns are represented declaratively as transformations on schema and data graphs, thus we avoid hardcoding a fixed set of solutions and allow the simple expansion of the pattern base. The graph-based metamodel these graph transformations operate on can itself be expanded to embrace more metamodels by defining them using a meta-metamodel. We have showcased, how graph transformations can be used to describe the effects of a pattern application on both the schemas and the data. We can describe the semantics of arbitrary operations without the need to provide a unified operator algebra, while at the same time including existing operators where they are ap-

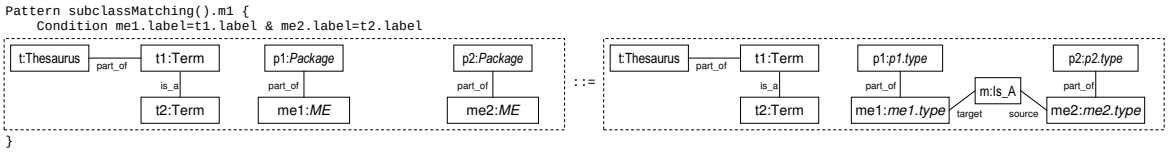

Figure 7. A simple schema matching pattern

plicable to improve runtime efficiency. We then demonstrated the validity of our approach by applying these patterns to two simple source schemas, which were successfully mapped to the desired target schema.

To minimise user interaction during schema matching, PALADIN goes beyond existing approaches to automatic schema matching by using an extensible framework for composite matchers. It is even possible to describe automated schema matching itself as a set of patterns, i.e. graph transformations, requiring support by imperatively defined (i.e. programmed) operations only for preprocessing steps like tokenisation or stemming.

Our current work focuses on the mapping of our graph transformation paradigm to existing transformation engines like PROGRES. Due to the limitations of these engines, we also pursue the development of a transformation engine specialised for schema and data graph transformations, to allow a better evaluation of approaches and heuristics for selecting the optimal pattern for a given situation.

While we concentrate on the creation of the logical integration plans starting from a preselected set of data sources and using a well-defined target schema, we also pursue approaches that support the discovery phase, i.e. the inference of the desired target schema and the selection of appropriate data sources. The deployment of the logical integration plans is another central issue. By providing mappings of the logical operators defined in the integration plan to concrete physical operators of different runtime environments, PALADIN can create plans for a variety of integration systems that currently use manually created integration plans. The potential spectrum of runtime platforms reaches from existing tools like IBM's Information Integrator to a BPEL choreography that uses distributed web services as operators. Here, the principal idea of patterns can be reused to define the basic mappings between logical and physical operators and to capture the special properties of each platform. If a mapping for a pattern's logical operator to a physical operator of the respective runtime environment is missing or if it simply provides no suitable physical operator implementation, the definition of the graph-based operator can be used as a fallback: Using extension mechanisms present in many of the potential runtime environments, a generic operator can be provided that can be configured using the transformational description in the logical integration plan alone.

References

[1] G. Alonso. Myths around Web Services. *Data Engineering Bulletin*, 25:3–9, December 2002.

[2] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. *Business Process Execution Language for Web Services version 1.1*, May 5, 2003.

[3] M. Antonioletti, M. Atkinson, S. Malaika, S. Laws, N. W. Paton, D. Pearson, and G. Riccardi. The Grid Data Service Specification, September 19, 2003.

[4] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, and J. Siméon. XQuery 1.0: An XML Query Language, April 4, 2005.

[5] P. Bruni, F. Arnaudies, A. Bennett, S. Englert, and G. Keplinger. *Data Federation with IBM DB2 Information Integrator V8.1*, October 16, 2003.

[6] S. Busse, R.-D. Kutsche, U. Leser, and H. Weber. Federated Information Systems: Concepts, Terminology and Architectures. Technical report, TU Berlin, April 1999.

[7] I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid – Enabling Scalable Virtual Organizations. In *Proceedings of CCGrid 2001*. IEEE Computer Society, May 15-18, 2001.

[8] T. Härder, G. Sauter, and J. Thomas. The Intrinsic Problems of Structural Heterogeneity and an Approach to their Solution. *The VLDB Journal*, 8:25–43, 1999.

[9] L. V. S. Lakshmanan, F. Sadri, and I. N. Subramanian. SchemaSQL – A Language for Interoperability in Relational Multi-Database Systems. In *Proceedings of the 22nd VLDB Conference*, pages 239–250, 1996.

[10] D. Lee, M. Mani, and W. W. Chu. Schema Conversion Methods between XML and Relational Models. *Knowledge Transformation for the Semantic Web.*, 95:1–17, 2003.

[11] J. Madhavan, P. A. Bernstein, and E. Rahm. Generic Schema Matching with Cupid. In *Proceedings of the 27th VLDB Conference*, pages 49–58, 2001.

[12] Object Management Group. *Common Warehouse Metamodel (CWM) Specification Version 1.1*, March 2003.

[13] E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *The VLDB Journal*, 10:334–350, 2001.

[14] A. Schürr. Programmed Graph Replacement Systems. In *Handbook of Graph Grammars and Computing by Graph Transformations*, pages 479–546, 1997.

[15] A. Schürr, A. W. Winter, and A. Zündorf. The PROGRES Approach: Language and Environment. In *Handbook of Graph Grammars and Computing by Graph Transformation*, pages 487–550, 1997.