Orchid: Integrating Schema Mapping and ETL

Stefan Dessloch*, Mauricio A. Hernández†, Ryan Wisnesky‡, Ahmed Radwan§, Jindan Zhou§

*Department of Computer Science, University of Kaiserslautern
Kaiserslautern, Germany
dessloch at informatik.uni-kl.de

†IBM Almaden Research Center
San Jose, CA, US
mauricio at almaden.ibm.com

‡School of Engineering and Applied Sciences, Harvard University
Cambridge, MA, US
ryan at eecs.harvard.edu

§Department of Electrical and Computer Engineering, University of Miami
Miami, FL, US
{a.radwan,j.zhou} at umiami.edu

***Abstract*— This paper describes Orchid, a system that converts declarative mapping specifications into data flow specifications (ETL jobs) and vice versa. Orchid provides an abstract operator model that serves as a common model for both transformation paradigms; both mappings and ETL jobs are transformed into instances of this common model. As an additional benefit, instances of this common model can be optimized and deployed into multiple target environments. Orchid is being deployed in *FastTrack*, a data transformation toolkit in IBM Information Server.**

I. Introduction

Over the last few years declarative schema mappings have gained popularity in information integration environments [1]. In schema mapping tools such as IBM Rational Data Architect (RDA)[1], or research prototypes like Clio [2], users see a representation of a source and a target schema side-by-side. Users specify transformations by specifying how each target object corresponds to one or more source objects, usually by drawing lines across the two schemas. Users annotate these lines with functions or predicate conditions that enrich the transformation semantics of the mapping. For example, column-to-column mapping lines are annotated with transformation functions and table-to-table mapping lines are annotated with filtering predicates. In many cases, complex transformation functions written in a host language are attached to lines enabling users to "escape" to more procedural specifications while maintaining the higher-level declarative specification of the mappings.

Mapping tools are used for two main reasons [3]: generating a transformation query or program that captures the semantics of the mapping specification (e.g., a SQL query that populates target tables from source tables), and providing meta-data that captures relationships between source and target schema elements. The latter functionality is useful when, for example, users need to discover relationships between two related schemas without regard to transformation semantics.

On the other hand, ETL (Extract - Transform - Load) tools [4], which are commonly used in data warehousing environments, allow users (often called ETL programmers) to express data transformations (often called jobs) as a flow of data over a graph of operators (often called stages). Each stage performs a piece of the transformation and passes the resulting data into the next stage. In effect, users construct a directed graph of these stages with the source schemas appearing on one side of the graph and the target schemas appearing on the other side of the graph. Stages in ETL jobs range from simple data mappings from one table to another (with renaming of fields and type conversion), to joining of data from two or more data paths, to complex splitting of data into multiple output paths that depend on input conditions and merging of those data paths into existing data.

ETL jobs and mappings are widely used in information integration tools to specify data transformations. IBM alone supports a number of mapping tools across several products (e.g., Rational Data Architect (RDA), Rational Application Development[2], and WebSphere Integration Developer[3]). IBM also supports at least two ETL tools: IBM WebSphere DataStage, and another in DB2 Warehouse Enterprise Edition.

In this paper we describe Orchid, a system originally designed to convert declarative Clio schema mappings [1] into IBM WebSphere DataStage ETL jobs and vice versa. Orchid provides an abstract operator model that serves as a common model for both transformation paradigms; both mappings and ETL jobs are transformed into instances of this common model. As an additional benefit, instances of this common model can be optimized and deployed into multiple target environments. For example, instead of converting an ETL job into a mapping, Orchid can rewrite the job and deploy it back as a sequence of combined SQL queries and ETL jobs. This rewrite and deployment of ETL jobs occurs automatically and

[1]http://www.ibm.com/software/data/integration/rda/

[2]http://www.ibm.com/software/awdtools/developer/application/

[3]http://www.ibm.com/software/integration/wid/

reduces the workload of (highly expensive) ETL programmers.

Mapping and ETL tools are aimed at different sets of users. In general, mapping tools are aimed at data modelers and analysts that want to express, at a high-level, the main components of a data transformation or integration job. In this kind of scenario, declarative specifications and simple GUIs based on lines work well. ETL tools are aimed at developers interested in the efficient implementation of the data exchange/integration task. Since designers and developers work as a team when implementing a task, collaboration is facilitated if the tools used can interoperate. However, mapping and ETL tools do not directly interoperate, and users often require manual processes to support the following features:

- Starting from declarative mappings, generate ETL jobs reflecting the mapping semantics, which can then be further refined by an ETL programmer.
- Starting from an ETL job, extract a declarative mapping that represents the logical aspects of the ETL operations as a source-to-target schema mapping.
- Support "round-tripping" for the different data transformation representations, allowing incremental changes in one representation to propagate into the other.

To illustrate the use of the above features, let us take a look at how Orchid's capabilities are utilized in an industrial product. Orchid's technology is now part of *FastTrack*, a component of IBM Information Server[4]. IBM Information Server is a new software platform providing a host of tools for enterprise information integration, including IBM WebSphere DataStage. FastTrack uses IBM Information Server's metadata repository to facilitate collaboration between designers of a data integration or data exchange application. For instance, tools are provided for system analysts to enter relationships between data sources in a declarative way (e.g., as correspondences between two schema elements, as business rules, etc.). These declarative specifications are captured and stored as schema mappings. The mappings are often incomplete, only partially capturing the transformation semantics of the application. In fact, some of the rules can be entered in a natural language such as English.

FastTrack converts these mappings into IBM WebSphere DataStage (ETL) job skeletons that contain some unresolved place-holder stages that are not completely specified. For example, an analyst might not know how to join two or more input tables, but FastTrack, nevertheless, detects that the mapping requires a join and creates an empty join operation (no join predicate is created) in the ETL job. Similarly, business rules entered in English are passed as annotations to the appropriate ETL stage. These annotations guide ETL programmers writing the actual transformation functions.

Once the programmers are done refining the generated ETL job, they communicate the refinements back to the analysts for review. The programmers can regenerate the mappings based on the refined ETL jobs; unless the users radically modify the ETL jobs, the regenerated mappings will match the original mappings but will contain the extra implementation details just entered by the programmers. In our example, the analyst will now see the join condition used for those input tables. In an alternative scenario, users can convert existing ETL jobs into a flow of mappings and send them to analysts for review.

Converting between mappings and ETL systems raises a number of challenges. The first is the different levels of abstraction between mappings and ETL jobs. Because mappings are declarative specifications, they do not capture (by design) the exact method by which their transformation semantics are to be implemented. For example, a mapping from two relational tables into a target table is often specified with a join operation between the two tables. As with SQL and other declarative languages, mappings do not capture how this join operation is implemented and executed. In general, ETL systems have more operations and richer semantics than mapping tools. ETL systems usually provide several operators that implement the join operation, each with a different implementation (e.g., depending on the selected operator, the runtime engine executes the join using nested-loops, sort-join, or hash-join). That is, ETL programmers can and often choose the specific implementation for the required transformation. This leads into the second challenge: ETL systems often provide operators whose transformation semantics overlap (i.e., some data transformation tasks can be implemented using different combinations of ETL operators). To convert ETL jobs into mappings, it is necessary to first compile them into an intermediate representation that better exposes elementary data transformations. As such, we require a common model that captures the primitive transformation operations of mapping systems and the main transformation operations of ETL systems. Finally, ETL systems support operators with semantics orthogonal to mapping systems, such as data cleansing and update propagation. These operators therefore do not have counterparts in mapping systems, and while such operators cannot be converted into equivalent mapping specifications, their presence must not interfere with our transformation and their presence must be preserved during transformation.

The rest of this paper is organized as follows. In the next section we give a brief survey of related work. We then provide an overview of Orchid and the internal common model used to capture the transformation semantics of ETL jobs and mapping specifications. In Section IV we describe this model in more detail and explore the generality and limitations of our approach. Sections V and VI discuss example transformations of ETL jobs to and from mappings. We conclude by describing our current status and future work.

II. Related Work

ETL systems are traditionally viewed as tools that load, cleanse, and maintain data warehouses. However, ETL systems have evolved to allow a variety of source and target data formats and are used in many data transformation and integration tasks.

Although ETL has received much attention in the commercial data integration arena, this attention has not been matched

[4]http://www.ibm.com/software/data/integration/info_server/

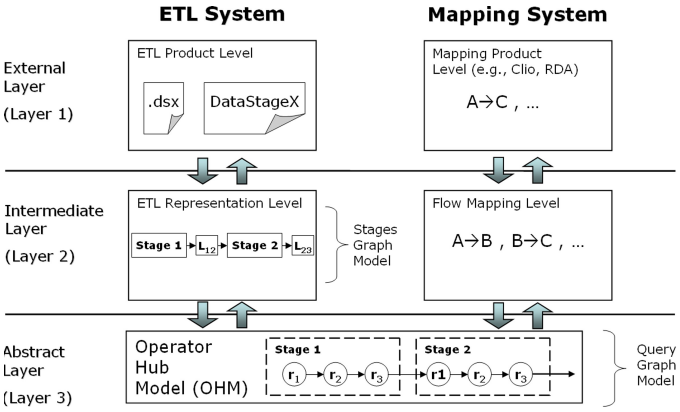

Fig. 1. Orchid Components and Representation Layers

by the database research community. The most extensive study of common models for ETL and ETL job optimization is by Simitis, et. al. [5][6]. Their work proposes a multi-level workflow model that can be used to express ETL jobs. ETL jobs expressed in their model can be analyzed and optimized using well-understood logical inference rules. Orchid uses a simplified version of this common model tailored to deal with mappings. Furthermore, the Simitis, et. al. work is not tied to any particular ETL system and the authors do not discuss how the set of ETL operators from a particular vendor can be converted into instances of their model and vice-versa. Orchid compiles real ETL jobs into a common model and can deploy that abstract model instance into a valid job in an ETL system or other target platform.

In Section III we describe how to transform ETL jobs and mappings into a common model that captures the transformation semantics. This transformation is somewhat similar to compiling declarative queries into query graphs. Techniques for optimizing query graphs by means of rewrite rules are well known [7].

Unlike ETL, schema mapping has received considerable attention by the research community [2][8]. Mappings are constraints between source and target data instances expressed in a logical notation. These relatively simple logical expressions can be generated semi-automatically from the schemas involved in a mapping and the simple correspondences that a user draws across them [2]. Furthermore, because the semantics of mappings are known, they can be converted into queries (or data transformation programs) expressed in several query languages. For instance, Clio can produce XQuery and XSLT scripts from the same mapping specification.

III. Overview

Orchid is built around a multi-layered representation model, where both ETL job and schema mapping descriptions are represented at an external, an intermediate, and an abstract layer. The representation layers are illustrated in Figure 1.

The External layer characterizes the description of ETL and mapping information in an external format specific to a data processing product or system. For example, IBM WebSphere DataStage uses proprietary file formats to represent and exchange ETL jobs. The representation model used by Orchid at this layer directly reflects the artifacts of the exchange format and makes the information available for further processing. In the same way, mapping related information is stored in a product-specific manner by systems such as Clio or RDA, and similar import/export capabilities are implemented in Orchid to exchange mapping information with such systems.

At the Intermediate layer, ETL jobs are still represented in a product-specific manner, but are now captured in models that reflect the ETL processing aspects relevant to Orchid. For example, a DataStage ETL job can be seen as a graph that consists of a number of connected stages (e.g., Transform, Filter, Lookup, Funnel, etc.) with specific operational semantics for processing data. This information is captured by using DataStage specific stages at the Intermediate layer. A separate Intermediate model must be implemented for each data processing platform supported by Orchid, although this is often trivial (see Section V). For mapping information, the Intermediate layer makes use of Clio mappings, which are described in Sections V and VI.

The Abstract layer supports our Operator Hub Model (OHM) to represent the operational semantics of ETL processing steps in a product-independent manner. OHM can be characterized as an extension of relational algebra with extra operators and meta-data annotations that characterize the data being processed. OHM is discussed in Section IV. Introducing OHM at the Abstract layer has several advantages:

First, Orchid represents and manipulates the semantics of ETL jobs in a platform-independent manner. This facilitates transformations to and from declarative mappings and makes the model product-independent.

Second, Orchid is extensible with respect to data processing platforms and mapping tools. New ETL import/export and compilation/deployment components, and new mapping functionality, can be added to the system without impacting any of the functionality of the OHM layer. Likewise, additional operators can be added at this layer without impacting existing ETL or mapping components.

Third, by being close to relational algebra, OHM lends itself to the same optimization techniques as relational DBMS. That is, we can leverage the vast amount of knowledge and techniques from the area of relational query rewriting and optimization and adapt these to the data processing model found at this level. For example, the deployment step (generating a data processing definition for one or more target platforms, such as DataStage jobs or IBM DB2 SQL queries) can be better implemented based on OHM (vs. logical mappings), because OHM is already close to the data processing semantics of many target deployment platforms. This is especially useful if a combination of target platforms is considered. For instance, a DataStage job can be imported, optimized and redeployed to a combination of DataStage and DB2, thereby increasing performance. In a similar way, optimization capabilities available at the OHM level can be used to optimize an existing ETL job on a given platform by importing it into

Orchid, performing optimizations at the OHM level, and then deploying back to the original platform. This makes query optimization applicable to ETL systems, which usually do not support such techniques natively.

In the next section, we describe OHM, our Abstract Layer model. In Section V, we use an example to illustrate how Orchid converts an ETL job into an OHM instance and how Orchid converts that OHM instance into a mapping. Section VI then uses the same example to show the process in the reverse direction: starting from a mapping, create an OHM instance and then deploy that OHM instance as an ETL job.

IV. The Operator Hub Model

The main goal for introducing our Operator Hub Model (OHM) in the Orchid architecture is to provide a model for representing data transformation operations independently of specific ETL platforms. Such platforms frequently support a repertoire of operators that take sets of rows as input data and produce one or more sets of rows as output. We wanted OHM to stay as close as possible to the operator-based approach found in ETL platforms, because this would allow us to reflect the common processing capabilities and reduce our efforts to translate between OHM and (multiple) ETL systems. On the other hand, OHM must also be capable of representing the transformations inherent in schema mapping specifications, which are dominated by declarative constructs that can be interpreted in a query-like manner. To achieve both goals, we chose relational algebra as the starting point for our operator model. Relational algebra operators and semantics are well-known within the database community [9] and capture the common intersection of mappings and ETL transformation capabilities. ETL is heavily rooted in a record-oriented data world, and (extended) relational algebra is commonly accepted as a foundation for record-oriented data transformations by the relational database community, where it serves as the foundation of query processing. Moreover, numerous extensions support nested structures (e.g., NF2 nest/unnest) [10], which we can leverage in OHM. Furthermore, we can take advantage of the vast array of processing and optimization techniques based on relational algebra developed over the last decades.

Formally, an OHM instance is a directed graph of abstract operator nodes. The graph represents a dataflow with data flowing in the direction of the edges. Each node in the graph represents a data transformation operation and is annotated with the information needed to capture the transformation semantics of the ETL operation it represents. Each edge in the graph is annotated with the schema of the data flowing along it. OHM operator names are written in UPPERCASE to distinguish them from similarly named stages at the Intermediate level. Figure 5 depicts an example OHM graph.

Orchid uses a special nested-relational schema representation to capture the schemas of data. This representation is rich enough to capture both relational and XML schemas. However, the initial implementation of Orchid deals only with flat transformations and thus does not use the nesting capabilities of our schema representation.

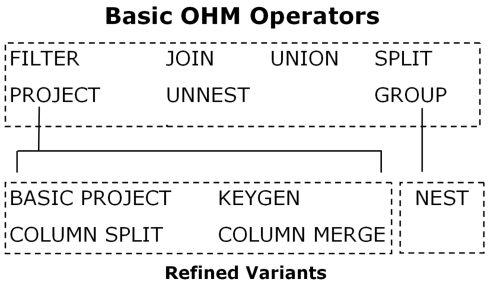

Fig. 2. Current OHM Operators

OHM operators are defined by identifying input and output data parameters and operational properties that represent the details of the operator behavior. Some OHM operators have properties whose values are expressions, such as boolean expressions for defining logical conditions, or scalar expressions for describing how the values of new columns are derived from existing ones. For example, a PROJECT operation has a single input, a single output, and a set of column derivation expressions that define how each output column is constructed from the columns of the input data.

OHM uses a generalized notion of projection that includes the generation of new output columns based on potentially complex expressions, similar to the expressions supported in the select-list of a SQL select statement. OHM borrows from SQL in that regard, using a subset of the respective SQL syntax clauses to represent expressions of any kind. However, the set of functions available in such expressions is extensible in order to capture any functional capabilities not directly supported by built-in SQL functions.

The set of operators currently defined in OHM includes well-known generalizations of the traditional relational algebra operators [9] such as selection (FILTER), PROJECT, JOIN, UNION, and GROUP (for performing aggregation and duplicate elimination), but also supports nested data structures through the NEST and UNNEST operators, similar to operators defined in the NF2 data model [10]. A detailed discussion and formal definitions of such operators is beyond the scope of this paper and can be found in the literature referenced above. Because the same data in a complex data flow may need to be processed by multiple subsequent operators, OHM includes a SPLIT operator, whose only task is to copy the input data to one or more outputs. The operators currently supported in Orchid are depicted in Figure 2.

OHM supports refined variants of the basic operators through a notion of operator subtyping. An operator subtype may introduce additional semantics by defining how new properties are reflected into inherited properties and by providing a set of constraints for property values. For example, the following operators are refinements of PROJECT:

- BASIC PROJECT permits only renaming and dropping columns, and does not support complex transformations or data type changes.
- KEYGEN introduces and populates a new surrogate key column in the output dataset.

- COLUMN SPLIT and COLUMN MERGE are a pair of operators that split the content of a single column into multiple output columns, or vice versa.

Note that a refined operator must be a specialization of its more generic base operator. That is, its behavior must be realizable by the base operator. Consequently, rewrite rules that apply to a base operator also apply to any refined variant. However, a refined operator may be easier to use when modeling an ETL job and may be closer to the operational behavior found in a number of ETL-specific scenarios and products. Refined variants are also useful for deploying an OHM graph to a specific product platform (see Section VI-B for more details).

Handling unknown stages. As we mentioned in Section I, not all ETL operations can be translated as mappings. Some complex ETL operations, like data cleansing, data compression, data encoding, pivoting of columns into rows, and operations that merge data into existing tables are generally not supported by mapping systems. Our initial implementation of OHM mainly covers operations that can be expressed by mapping systems and, thus, cannot capture these complex ETL operations. Furthermore, ETL systems allow users to plug-in their own "custom" stages or operators which are frequently written in a separate host language and executed as an external procedure call when the ETL flow is executed. We currently treat complex or custom ETL operators as black-boxes in our OHM graph; we may not know the transformation semantics of the operator but we at least know what are the input and output types. We introduce a catch-all OHM operator, named UNKNOWN, for these cases. UNKNOWN operators will appear when translating from ETL into mappings. We discuss how to handle this special operator in Section V.

Relational schema mapping systems allow users to specify operations whose semantics can be expressed as relational algebra operators (or simple extensions of RA). In Section VI we discuss how to use the graph of OHM operators in Figure 9 to capture mapping transformation semantics. Here we note that most (relational) schema mapping systems allow users to enter column-to-column transformations, filtering and join predicates, grouping conditions, aggregate functions, unions, and, in certain cases, conditions over the resulting target instance and logic that splits the computation into more that one target table or column. Detailed examples of mappings and their transformation semantics can be found in [11]. Because OHM is designed to capture the transformation semantics of mappings, UNKNOWN will not appear in OHM instances generated from mappings.

V. Transforming ETL into Mappings

In this section we discuss how to convert ETL jobs into mappings via the OHM. We first describe how ETL jobs are compiled into an instance of the OHM and then describe how OHM instances are converted into mapping specifications. Section VI discusses the opposite direction, from a mapping to an ETL job.

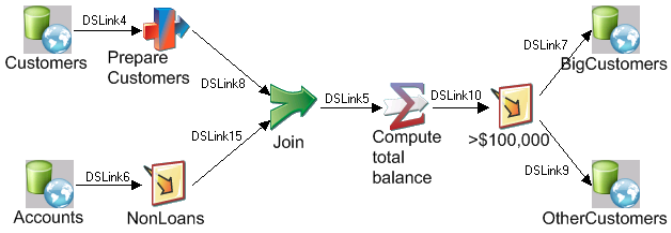

Fig. 3. Example ETL job

Figure 3 shows the example we will use in our discussions. This simple IBM WebSphere DataStage job takes as input two relational tables, Customers and Accounts and separates the Customers information into two output tables, BigCustomers and OtherCustomers, depending on the total balance of each person's accounts. Figure 4 shows the schemas involved. Similar jobs (albeit with more operations and inputs) are routinely executed to, for example, tailor credit card or loan offers to customers.

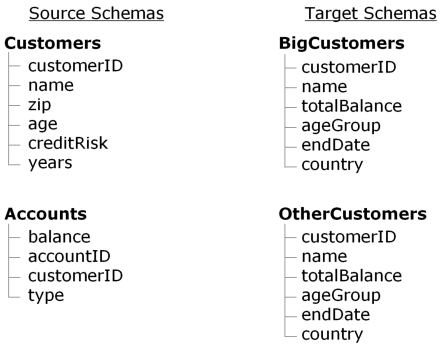

Fig. 4. Source and Target schemas

The example ETL job starts by applying transformations to some of the columns in Customers. This occurs in the Transformer stage labeled *Prepare Customers* in Figure 3. Figure 8 shows the transformation functions within the resulting mappings. The Filter stage labeled *NonLoans* applies the filter "Accounts.type $\neq$ 'L'" to incoming Accounts tuples. That is, only tuples for non-loan accounts pass the filter. Then, the processed Customers tuples are joined with the non-loan Accounts tuples in the *Join* stage, which uses the join predicate "Customers.customerID = Accounts.customerID". The *Compute Total Balance* stage groups all incoming tuples by customerID and applies a *sum* aggregate function to balance. The final filter stage, labeled *>$100,000*, first applies a predicate that checks if the computed total balance is greater than $100,000; if it is, the tuple is routed into the BigCustomers output table. Otherwise, the tuple is routed into the OtherCustomers table.

A. Compiling ETL jobs into OHM

Converting ETL jobs into OHM instances involves compiling each vendor-specific ETL stage into one or more OHM operators. The result of this compilation is a sequence of OHM subgraphs which are connected together to form the OHM representation of the job.

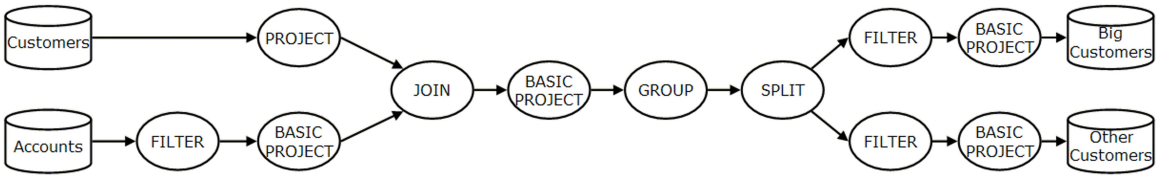

Fig. 5. OHM instance

Orchid compiles ETL jobs in two steps. In the first step, the vendor-specific ETL representation is read by our Intermediate layer interface and is converted into a simple directed graph whose nodes wrap each vendor-specific stage. The Intermediate layer is most useful when an ETL system does not have a programmable API that allows direct access to its internal representation. For example, this was the case with DataStage 7.5x2 and earlier. The only way to access these DataStage jobs is by serializing them into an XML format and then compiling that serialization into an Intermediate layer graph. In other words, the Intermediate layer graph often serves as a stand-in object model when no model is provided by an ETL system. Newer versions of DataStage (such as the version in IBM Information Server) do provide an object model and hence Orchid simply wraps each stage with a node in the Intermediate layer graph.

The second step is to compile the vendor-specific operation wrapped by each node of the Intermediate layer graph into a graph of one or more OHM operators. Orchid traverses the Intermediate layer graph and, for each node, invokes a specific compiler for the stage wrapped by the node. For example, the Intermediate layer graph for our example in Figure 3 is structurally isomorphic to the ETL job graph. When Orchid visits the node representing a Filter stage, it looks for a vendor-specific compiler for this stage. This compiler then creates the necessary OHM operator graph to capture the stage semantics. (In this case, a FILTER followed by a BASIC PROJECT). The compiler also computes the output schema of the data at each output edge of the operator. Compilation proceeds by connecting together the OHM subgraphs created by compiling each stage visited during the traversal of the Intermediate layer graph.

Figure 5 shows the OHM instance that is produced by Orchid for our example job. The *NonLoans* Filter stage is compiled into a FILTER operation followed by a BASIC PROJECT operation. Similarly, the *Join* stage is compiled into a JOIN operator followed by a BASIC PROJECT. Here, the JOIN operator only captures the semantics of the traditional relational algebra join, while the BASIC PROJECT removes any source column that is not needed anymore (for instance, only one customerID column is needed from this point on in the OHM graph).

Of particular interest is the result of compiling the final Filter stage. Unlike an OHM FILTER operator, a Filter stage can produce multiple output datasets, with separate predicates for each output. An input row may therefore potentially be copied to zero, one, or multiple outputs. Alternatively, the

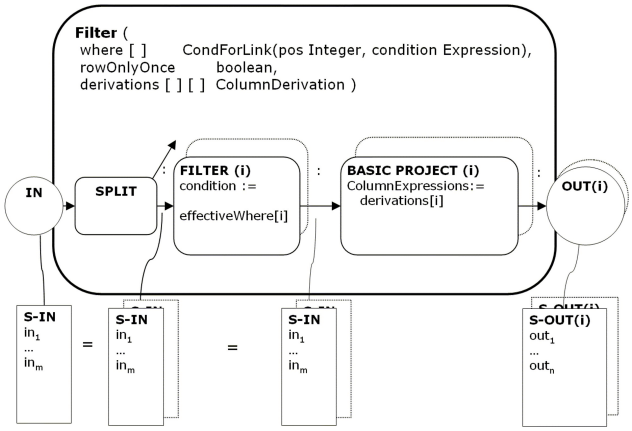

Fig. 6. Filter Stage Representation in OHM

Filter stage can operate in a so-called "row-only-once" mode, which causes the evaluation of the output predicates in the order that the corresponding output datasets are specified, and does not reconsider a row for further processing once the row meets one of the conditions. In addition to the essential filtering capabilities, the Filter stage supports simple projection for each output dataset.

Figure 6 describes the processing behavior of the Filter stage using a corresponding OHM representation. The properties of the Filter stage hold descriptions of filter predicates (i.e., filtering conditions), one for each output dataset, and column derivation expressions describing the simple projections that can occur for each output dataset. An equivalent OHM graph contains a SPLIT operator and a FILTER - BASIC PROJECT operator sequence for each output dataset, with operator properties corresponding to the predicates and derivations defined for the original Filter stage. If the Filter stage is in row-only-once mode, the predicates for each output dataset need to be combined with the (negated) predicates of previous output stages. Note that the SPLIT and BASIC PROJECT operators are optional and do not necessarily need to be generated. SPLIT is not needed if the Filter stage only has a single output dataset. BASIC PROJECT is not needed if no output column is projected from an output dataset. (In order to simplify the logic of the stage compilers, we allow the compilers to generate redundant (i.e., empty) operators which will later be eliminated by a generic rewrite step before pursuing further operations on the graph.)

In our example, the compiler detects that the input tuples of the final Filter stage will be split between two or more output links and adds a SPLIT operator to the OHM graph.

Then a FILTER operation is placed on each outgoing path. The FILTER on the path to BigCustomers contains the original Filter stage predicate, namely, "totalBalance > 100000". The FILTER on the path to OtherCustomers receives the opposite predicate, "not(totalBalance > 100000)", because the semantics of the stage requires all tuples not satisfying the stage predicate to flow into OtherCustomers.

To recap, to enable Orchid to compile a vendor-specific ETL job into an OHM instance, a programmer must provide an importer that converts ETL stages into nodes in the Intermediate graph. Then, the programmer writes compilers that transform each supported stage into an OHM graph. Orchid uses a plug-in architecture and each compiler is a dynamically detected plug-in that follows an established interface. In our initial implementation, we support 15 DataStage processing stages. Understanding the semantics of the stages and writing the 15 compilers was a 4 person-month effort. Note that because there is often an overlap in the semantics of the stages, compilers can be designed to form a hierarchy of compiler classes; more specific stages use compilers that are subclasses of compilers for more general stages.

B. Deploying OHM as mappings

This section discusses how to convert an OHM instance into one or more mappings. Each operator node in the OHM instance is converted into a simple mapping expression that relates the schema(s) in its input edge(s) to the schema(s) in its output edge(s). Orchid then composes neighboring mappings into larger mappings until no further composition is possible.

We leverage Clio's mapping language and technology [2] to represent and manipulate mappings in Orchid. Clio expresses mappings using declarative logical expressions that capture constraints about the source and target data instances. Clio mappings are formulas of the form $\forall\phi(X) \rightarrow \exists Y\psi(X,Y)$. These mapping expressions can be easily translated into many other mapping specifications. An important property of this class of mapping expression is that we understand how and when we can compose two mapping formulas [12][13]. In other words, given two mappings $A \rightarrow B$ and $B \rightarrow C$, Clio (and hence Orchid) can compute $A \rightarrow C$ (if possible) in a way that preserves the semantics of the two original mappings.

More formally, given a directed acyclic graph (DAG) of OHM operators, Orchid creates a similar DAG of mappings. Orchid then performs an ordered traversal of the nodes in the mapping DAG, starting with the source-side nodes. As nodes are visited in the direction of the edges, Orchid attempts to compose the mapping in the current node with all the mappings targeting any of the node's incoming edges. If this composition is possible, the composed mapping replaces all participating mappings in the mapping DAG. A visited node in the graph which does not admit composition in this way has at least one edge that serves as a *materialization point*. Materialization points identify boundaries between OHM graph sections inside of which mappings are completely composed. The result is a set of mappings that touch only at the materialization points (i.e., the target side of one mapping is part of the source side of the next mapping).

Materialization points occur for two reasons. First, some OHM operators always have edges that serve as materialization points, e.g. SPLIT. (Although it is theoretically possible to compose mappings over a SPLIT operator, a SPLIT represents a fork in the job that was placed there by an ETL programmer and as such is a natural place to break between generated mappings.). As we mention at the end of Section IV, some complex or custom ETL stages will appear as UNKNOWN operators OHM instances. The end-points of a continuous sequence of UNKNOWN operators are also marked as materialization points. Second, by composing neighboring mappings, we are in effect performing view unfolding [14] (i.e., you can think of each operator as a relational view that uses as input one or more views. By composing these views we are, essentially, unfolding the views). There are semantic restrictions that limit how many of these views we can unfold: for instance, we cannot compose two mappings that involve grouping and aggregation [7]. In general, any operation that eliminates duplicates cannot be composed with an operation that uses the cleansed list for further processing [15]. For example, if we compute an aggregate function like *sum* after we remove duplicates, we cannot compose those two operations and have *sum* operate over the sources that contain duplicates.

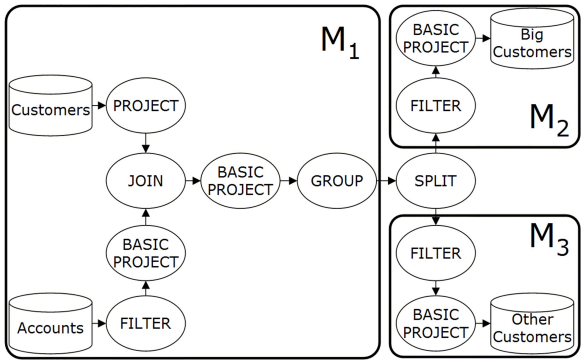

Fig. 7. Extracted mappings

In the case of our example OHM instance, the above process identifies one materialization point: the edge after the GROUP operator and before the SPLIT operator. By chance, this is a materialization point for both of the above reasons. The result is three mappings that touch at the materialization point edge. The composed mapping boundaries are shown in Figure 7[5].

Figure 8 shows the three computed mappings expressed using a query-like notation, with variables bound to set-type elements (e.g., Customers). Notice that M_1 computes a target relation named "DSLink10" and that this is the source relation for mappings M_2 and M_3. This intermediate relation is defined by the schema of the data flowing by the materialization point

[5]To simplify the diagram, we draw M_2 and M_3 starting at the output edges of the SPLIT operator. We can do this because, internally, all output schemas of SPLIT are equivalent to its the input schema. SPLIT ties the input edge to the two output edges but does no transformation.

M_1: $\forall\ s_0 \in$ **Customers**, $s_1 \in$ **Accounts**
s.t. s_1.customerID = s_0.customerID $\wedge$ s_1.type $\neq$ "L"
$\exists\ t_0 \in$ **DSLink10**
s.t. t_0.customerID = s_0.customerID $\wedge$
t_0.accountID = s_1.accountID $\wedge$
t_0.name = s_0.name $\wedge$
t_0.ageGroup = (*IF* (s_0.age $\leq$ 18) *THEN* 1
ELSE IF (s_0.age > 18 $\wedge$ s_0.age $\leq$ 35) *THEN* 2
ELSE IF (s_0.age > 35 $\wedge$ s_0.age $\leq$ 65) *THEN* 3
ELSE 4) $\wedge$
t_0.totalBalance = Sum (s_1.balance) $\wedge$
t_0.endDate = (*IF* (s_0.years < 0) *THEN* SetNull()
ELSE s_0.years) $\wedge$
t_0.country = (*IF* (IsNotNull (s_0.ZIP)) *THEN* "US"
ELSE "Unknown") $\wedge$
t_0 = *GROUP* [t_0.accountID, t_0.customerID, t_0.name,
t_0.ageGroup, t_0.endDate, t_0.country]

M_2: $\forall\ s_0 \in$ **DSLink10**
s.t. s_0.totalBalance > 100000
$\exists\ t_0 \in$ **BigCustomers**
s.t. t_0.customerID = s_0.customerID $\wedge$
t_0.name = s_0.name $\wedge$
t_0.totalBalance = s_0.totalBalance $\wedge$
t_0.ageGroup = s_0.ageGroup $\wedge$
t_0.endDate = s_0.endDate $\wedge$
t_0.country = s_0.country

M_3: $\forall\ s_0 \in$ **DSLink10**
s.t. s_0.totalBalance $\leq$ 100000
$\exists\ t_0 \in$ **OtherCustomers**
s.t. t_0.customerID = s_0.customerID $\wedge$
t_0.name = s_0.name $\wedge$
t_0.totalBalance = s_0.totalBalance $\wedge$
t_0.ageGroup = s_0.ageGroup $\wedge$
t_0.endDate = s_0.endDate $\wedge$
t_0.country = s_0.country

Fig. 8. Generated Mappings

edge in the OHM instance (the edge after the GROUP operator). The long expressions on the body of M_1 are the transformation functions used to compute the values of ageGroup, endDate, and years.

Finally, consider what happens when there is an UNKNOWN operator in the OHM instance. For example, suppose there is a custom operator just after the *Join* stage in our example (see Figure 3). This custom operator appears as an UNKNOWN operator directly between the BASIC PROJECT and the GROUP operators in Figure 2. If we assume the input relation to this UNKNOWN operator is "DSLink5" (see Figure 3) and the output relation is now called "customOut", then both these edges will be marked as materialization points. Orchid computes the following five mappings: M_1 now maps from the source tables into "DSLink5" and does not contain the grouping condition. Then, a new and "empty" mapping M_4, maps "DSLink5" to "customOut", and stands in place of the custom operator. This "empty" mapping only records the source and target relations and a reference (e.g., the name) of the custom operator that created this mapping. M_4 does not contain any column-to-column mapping, or any filtering predicates. Another new mapping M_5 now maps "customOut" into "DSLink10" and captures the grouping condition that was in M_1. M_2 and M_3 are the same as before, connecting "DSLink10" to the target tables.

VI. Transforming Mappings into ETL

We now describe how mapping specifications are transformed into ETL jobs. We first describe how to compile mapping specifications into an OHM instance and then describe how OHM instances are deployed as ETL jobs. Details of the procedures described in this section can be found in the extended version of this paper [16].

A. Compiling Mappings into OHM

We begin by assuming that the user starts from the mappings in Figure 8. These mappings can be entered using a mapping tool like Clio. Although users might want to enter two mappings, one that goes from the sources into BigCustomers and another that goes into OtherCustomers, this is not currently possible in Clio (and many other mappings tools). The reason is that the last filter predicate ranges over the result of the sum of all balances. Instead, users of Clio must create the three mappings in Figure 8: M_1 computes the total balance for each Customers and M_2 and M_3 then route the tuples into BigCustomers or OtherCustomers.

To enter a mapping like M_1, a user loads Customers and Accounts as a source and defines an intermediate table, called DSLink10 and whose schema is similar to BigCustomers, as a target (see Figure 4). The user draws lines connecting the relevant columns and adds any transformation functions needed on the lines. This includes the conditional expressions that determine the target values for ageGroup, endDate, and country. Then, the user adds any table-level predicate needed for the transformation. In the case of M_1, this includes the filter condition on Accounts, the join condition, and the grouping condition. The details of how Clio compiles these lines into mappings are detailed in [2].

Given a set of mappings, Orchid first creates an OHM graph that captures the data dependency between the mappings. In our example, the output of M_1 flows into both M_2 and M_3, and thus Orchid creates a SPLIT operator that connects the generated OHM graphs for each mapping. If two or more mappings share a common target relation (which is not the case in our example) Orchid creates a UNION operator to combine the flows.

To compile each individual mapping into a graph of OHM operators, Orchid creates a skeleton OHM graph from the template shown in Figure 9. This template captures the transformation semantics expressible in many relational schema mapping systems. Orchid then identifies the operators in this template graph that are actually required to capture the semantics of the mapping. The unnecessary operators are removed from the template graph instance, resulting in an OHM graph that represents the mapping.

For example, consider M_2 in Figure 8. Because this mapping only uses one input table and one output table, the JOIN and SPLIT operators are removed from the template graph. The left-most FILTER operator in the graph receives the filtering predicate in M_2. The BASIC PROJECT after that FILTER captures the simple column mappings in M_2. Orchid then removes all other operators in the template graph resulting in the simple

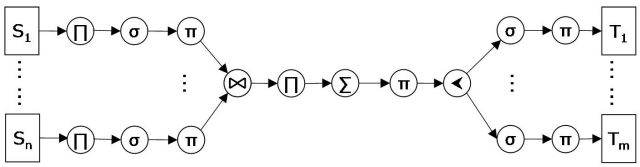

Fig. 9. Operator Template

DSLink10 → FILTER → BASIC PROJECT → BigCustomers flow. M_1 and M_3 are compiled into OHM instances using the same procedure. In the case of M_1, the JOIN operator receives the join condition between Customers and Accounts. The SPLIT operator is removed because there is only one target relation. The complex transformation functions appear in the left-most PROJECT operator that is connected to the Customers table. The resulting OHM for this simple example has (not surprisingly) the same shape as the one created from the ETL job (i.e., the OHM graph in Figure 5).

B. Deploying OHM instances as ETL

Orchid can deploy an OHM graph into multiple runtime environments (e.g., a combined ETL and database runtime). We first describe how Orchid deploys into a homogenous environment with a single runtime platform (RP) and then describe how Orchid deals with heterogenous runtime platforms.

Creating a deployment plan involves a number of steps, which we illustrate using our running example. We start with the OHM graph in Figure 5, which may have been generated from a declarative mapping. Orchid first assigns each operator to a RP; for the purposes of the initial discussion, we assume that DataStage is the only runtime platform available. In Figure 10, we see the OHM graph of Figure 5, with OHM operators enclosed by one or more "RP operator boxes". The chosen runtime platform is indicated at the bottom of each RP operator box (e.g., DS for DataStage). In our example, each OHM node is annotated as supported by DS.

When a runtime platform is registered in Orchid, it must declare a number of available runtime operators. For instance, the DataStage RP registers stages like Transformer, Join, and Filter. Every such runtime operator specifies which OHM operator(s) it can fully implement. Some RPs, such as DataStage, offer multiple alternatives for implementing each OHM operator. For example, all DataStage stages can perform simple projections. Thus, the DataStage RP marks all its operators as capable of handling OHM's BASIC PROJECT. The Filter and Transform DataStage stages can implement OHM's FILTER operator. Similarly, the OHM SPLIT operator can be implemented by DataStage's Copy, Switch, Filter, and Transform stages. Notice that it is possible that some OHM operators cannot be implemented with a single RP operator. For example, a complex PROJECT operation may require Transform and SurrogateKey stages in DataStage. When this happens, Orchid attempts to split the OHM operator into multiple (and simpler) OHM operators.

At the end of this initial step, all OHM operators in the graph (except, of course, UNKNOWN operators) are annotated as supported by one or more RP operators. The next step is to merge neighboring RP operator boxes to capture more complex processing tasks that span multiple OHM operators. In general, reducing the number of RP operators by exploiting such capabilities results in better performance characteristics for the operator graph (i.e., we are trying to reduce the number of RP operators).

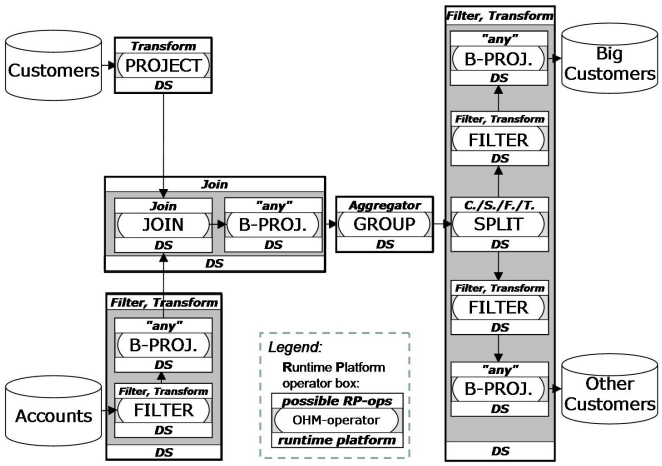

Fig. 10. Deployment Planning

For each operator box in the graph, Orchid checks for adjacent operator boxes (following the direction of the edges) that are tagged as supported by the same RP operator. For example, based on the RP operators assigned to the FILTER and BASIC PROJECT sequence at the bottom left of Figure 10, we can group these boxes into a bigger RP operator box. This merged box can be implemented with either a single Filter or Transform stage.

Notice, however, that even if neighboring boxes are tagged with the same RP operator, this does not necessarily mean Orchid can merge them into larger operator boxes. Each RP operator registers a template OHM subgraph that represents its transformation semantics. For example, the center region of Figure 6 depicts this template for the DataStage Filter operator. Notice how this template matches the subgraph of OHM operators that starts at the SPLIT operator in Figure 10. This is why all those operators are merged into one RP operator box that can be implemented with a Filter stage (and, as it turns out, a Transform stage as well).

To illustrate a case where we cannot merge two neighboring RP operator boxes, consider the BASIC PROJECT and GROUP operators in the middle of Figure 10. Technically, both can be implemented by an Aggregator DataStage stage. But we cannot merge them into one Aggregator RP operator box because the Aggregator template starts with a GROUP operator and cannot match a subgraph that starts with BASIC PROJECT.

We do make two simplifying assumptions when merging neighboring boxes to find a deployment strategy. First, we merge RP operator boxes as much as possible, thus preferring solutions that have less RP operators. Reliable runtime cost information for each RP operator is needed if we want to

compare solutions that use less merging. We currently lack such a cost model for DataStage ETL operators. Second, to guide our search for a deployment strategy, we use a "greedy" strategy for combining boxes, starting with the operators closest to the data sources and attempting to combine them with adjacent operators until this is no longer possible. Although this is a simple strategy, it works well for the platforms and example scenarios we have worked with. More sophisticated strategies considering alternative overlays are left for future research.

Finally, Orchid chooses the RP operator for boxes that contain multiple alternatives. This choice should be dependent on the processing costs of the operators, if such information is available, or on the intended semantics of the RP operators. In our example, we have two boxes where we can use a Filter or a Transform stage. In both cases, a Filter stage would be the natural choice, because FILTER operators are contained in the RP operator box, and no complex projection operations (which would demand a Transform stage) are required.

When there are multiple RP available to deploy the OHM graph, the merging of neighboring RP operators is only done for operators marked for the same RP. An interesting case occurs when one of the RP is the DBMS managing the source data. Orchid can use the deployment algorithm to do a pushdown analysis, allowing the left-most part of the operator graph to be deployed as an SQL query that retrieves the filtered and joined data. Currently, Orchid pushes as much processing as possible to the DBMS by identifying maximal OHM operator subgraphs that process data originating from the same source and assigning the operators to the DBMS platform, if the operator is supported by the DBMS. In our example scenario (this is not illustrated in Figure 10), Orchid identifies the operators up to and including the GROUP) operator as operators to be pushed into the DBMS. Each one of these operators is marked for deployment using a SQL Select statement. Merging these SQL RP operators into larger boxes is done using the same analysis described before: The SQL RP registers an OHM template graph that describes the semantics of the supported SQL statement. Then, the OHM operators and matched to this template and the corresponding RP operator boxes merged as needed. In effect, the SQL statement is slowly built as the OHM graph is visited from left-to-right in Figure 10.

VII. Conclusion and Outlook

In this paper we have described Orchid, a prototype system developed at IBM Almaden that has been integrated into *FastTrack*, a component of IBM Information Server. Orchid is currently capable of converting IBM WebSphere DataStage ETL jobs into mappings that mapping tools like Clio or Rational Data Architect understand and display. Orchid can also perform the reverse transformation: given a Clio or RDA-like mapping, it can convert the declarative specification into a DataStage ETL job that captures the same transformation semantics. Although our implementation currently only connects three systems (DataStage, Clio, and RDA) and, thus, it could be argued that an ad-hoc implementation between each system might be a more practical approach, we think we have a more scalable and long-term solution for converting between mappings and ETL jobs. Based on an abstract ETL operator model, the operator hub model (OHM), Orchid can be extended to support additional systems by implementing compiler and deployment components for their external representation. Once an external system is registered, arbitrary conversions between registered systems are possible.

Orchid enables many interesting opportunities regarding rewrite operations, optimization, and deployment of OHM graphs. Currently, Orchid only supports basic rewrite heuristics (e.g., selection push-down), and additional optimization techniques still need to be applied. Most importantly, the deployment of OHM graphs needs further investigation and thorough validation for the full complexity of the intended deployment platforms, the range of supported platforms, and the desirable and applicable strategies in the presence of complex, multi-platform environments.

Acknowledgements

This work was funded in part by the U.S. Air Force Office for Scientific Research under contract FA9550-07-1-0223. We thank our IBM colleagues Lucian Popa, Martin Klumpp, and Mary Roth for the discussions regarding this paper.

References

[1] L. M. Haas, M. A. Hernández, H. Ho, L. Popa, and M. Roth, "Clio Grows Up: From Research Prototype to Industrial Tool," in *SIGMOD*, 2005, pp. 805–810.

[2] L. Popa, Y. Velegrakis, R. J. Miller, M. A. Hernández, and R. Fagin, "Translating Web Data," in *VLDB*, 2002, pp. 598–609.

[3] M. Roth, M. A. Hernández, P. Coulthard, L. Yan, L. Popa, H. Ho, and C. C. Salter, "XML Mapping Technology: Making Connections in an XML-centric World," *IBM Systems Journal*, vol. 45, no. 2, pp. 389–410, 2006.

[4] R. Kimball and J. Caserta, *The Data Warehouse ETL Toolkit*. Wiley Publishing, 2004.

[5] A. Simitsis, "Modeling and managing ETL processes." in *VLDB PhD Workshop*, 2003.

[6] A. Simitsis, P. Vassiliadis, and T. K. Sellis, "Optimizing ETL Processes in Data Warehouses." in *ICDE*, 2005, pp. 564–575.

[7] H. Pirahesh, J. M. Hellerstein, and W. Hasan, "Extensible/Rule Based Query Rewrite Optimization in Starburst." in *SIGMOD*, 1992, pp. 39–48.

[8] S. Melnik, E. Rahm, and P. A. Bernstein, "Rondo: A Programming Platform for Generic Model Management." in *SIGMOD*, 2003, pp. 193–204.

[9] H. Garcia-Molina, J. D. Ullman, and J. D. Widom, *Database Systems: The Complete Book*. Prentice Hall, 2001.

[10] H.-J. Schek and M. H. Scholl, "The relational model with relation-valued attributes," *Inf. Syst.*, vol. 11, no. 2, pp. 137–147, 1986.

[11] A. Raffio, D. Braga, S. Ceri, P. Papotti, and M. A. Hernández, "Clip: a Visual Language for Explicit Schema Mappings," in *ICDE*, 2008.

[12] J. Madhavan and A. Y. Halevy, "Composing Mappings Among Data Sources," in *VLDB*, 2003, pp. 572–583.

[13] R. Fagin, P. Kolaitis, L. Popa, and W.-C. Tan, "Composing Schema Mappings: Second-Order Dependencies to the Rescue," in *PODS*, 2004, pp. 83–94.

[14] M. Stonebraker, "Implementation of integrity constraints and views by query modification," in *SIGMOD*, 1975, pp. 65–78.

[15] S. Chaudhuri, "An overview of query optimization in relational systems," in *PODS*, 1998, pp. 34–43.

[16] M. A. Hernández, S. Dessloch, R. Wisnesky, A. Radwan, and J. Zhou, "Orchid: Integrating Schema Mapping and ETL," 2008, Technical Report: http://wwwlgis.informatik.uni-kl.de/cms/index.php?id=156.