In: The VLDB Journal 8:1, 1999, pp. 25-43.

The Intrinsic Problems of Structural Heterogeneity and an Approach to their Solution

Theo Härder	Günter Sauter	Joachim Thomas
University of Kaiserslautern	Daimler-Benz AG, Research & Technology	UBS AG
Dept. of Computer Science	Dept. CAE-Research (FT3/EK)	Hochstr. 16
67653 Kaiserslautern, Germany	89013 Ulm, Germany	4002 Basel - Switzerland
haerder@informatik.uni-kl.de	guenter.sauter@dbag.ulm.DaimlerBenz.com	joachim.thomas@ubs.com

Abstract

This paper focuses on the problems that arise when integrating data from heterogeneous sources in a single, unified database view. At first, we give a detailed analysis of the kinds of structural heterogeneity that occur when unified views are derived from different database systems. We present the results in a multiple tier architecture which distinguishes different levels of heterogeneity and relates them to their underlying causes as well as to the mapping conflicts resulting from the view derivation process. As the second essential contribution, the paper presents our approach to a mapping language solving the identified conflicts. The main characteristics of the language are its descriptiveness, its capability to map between schemas written in the relational, object-oriented, ER, or EXPRESS data model, and its facilities for specifying user-defined update operations on the view that are to be propagated to the data sources. Finally, we briefly discuss how this mapping information is employed to convert queries formulated w. r. t. the integrated view, into database operations over the heterogeneous data sources.

Keywords: heterogeneity, legacy systems, mapping language, schema integration, schema mapping, updatable views

1. Introduction

The approach discussed in this paper resulted from research on a uniform product data model, which has been carried out at Daimler-Benz Research in Ulm/Germany in the past few years. For Daimler-Benz, just like for many other manufacturing companies, the increasing demand for flexibility and variety of product palettes calls for a uniform system environment permitting global consistency of product data as well as interoperability among all participating data processing subsystems. For example, geometrical product data, supported by CAD systems, and their corresponding logical bill-of-material structure, administered by data management systems, must be maintained in a single environment in order to provide a quick overview of as well as fast access to all relevant data associated with certain product lines. Even the introduction of new systems requires the interoperation with or, at least, the access to so-called legacy systems. Together with cross-platform exchange of information via the World Wide Web, the demand for integrating the data of multiple databases (DBs) is strongly increasing.

Data integration, often applied in reverse engineering processes for legacy data sources, must consider organizational as well as technical requirements. Data modeling should be up to today's needs which primarily means that the definition of integrated views should be unbiased by the intricacies and peculiarities of legacy data sources. Often, inter-organizational data definition standards govern the composition of views in accordance to given design principles. Furthermore, data sources to be integrated are described by heterogeneous schemas and frequently come from multiple organizations (cross-organizational data access). As a consequence, integrated view definitions are preexisting (e. g., by inter-organizational agreements), that is, the definition of views must be decoupled from their mapping and execution. This is one reason why, for example, SQL is no viable solution. Another reason is the potential heterogeneity between source schemas and view schema. View definition languages based on a single data model cannot bridge heterogeneity. Hence, a language is needed to map between the integrated view definition and data source schemas. In this paper, we propose such a **mapping language** supporting separately the definition of **view mapping** and **view execution** mechanisms.

In Sect. 2, we will compare the related work to our subject and focus on the goal of our paper. The first contribution of this paper is to present a detailed analysis of heterogeneity based on our exploration of data representations in the automotive industry (cf., Sect. 3). Sect. 4 surveys the properties of existing approaches to view languages and reveals their shortcomings which serve to justify our own approach. Our proposal to a schema mapping language is presented in Sect. 5 and is called BRIITY (mapping language bridging heterogeneity). Finally, a short description of the execution model of our mapping language is given in Sect. 6, before we conclude our work in Sect. 7.

2. Related Work and our Goal

There are two general types of problems that impede interoperability which is defined as the capability of *heterogeneous* systems to cooperate. Firstly, the schemas of the DBs to be integrated might strongly differ (**structural heterogeneity** including incomplete coverage of data types and possibly different data models) and cannot be replaced by homogeneous alternatives. One of the main characteristics of legacy systems is either the absence of a conceptual schema or its strong similarity to the internal schema. Most application programs require high-speed access to data which often calls for unnormalized schemas that are tuned for specific access profiles. As a consequence, the structures of schemas differ with the applications and their access characteristics. However, migrating to a new system generation that would allow to reimplement applications in a more uniform way and that would abstract as far as possible from details of the physical data representation, is often highly uneconomical. Legacy systems are usually intertwined into the information-processing infrastructure being queried via hand-coded interfaces by application programs and related systems. An atomic switch to powerful successors is therefore an expensive and delicate undertaking. Another argument additionally opting against this strategy is the relatively low frequency of accesses to those systems.

The second problem type impeding to couple systems in a straightforward way is **heterogeneity of semantics**. The DB design is influenced by the needs of a particular application to optimize run-time performance. Analogously, integrity constraints are often embedded, distributed, and replicated within application programs thereby preventing a uniform, system-enforced control of the data semantics. As a result, at the level of the DB schema only a partial description of the application semantics is conceivable. Hence, capturing all aspects of semantics can't necessarily be conducted in an automatic way.

Many papers with varying technical depth have been published presenting a taxonomy of heterogeneity [BCN91, BHP92, BLN86, CS91, DKMRS93, Du94, GSC96, KCGS95, KS91, KS96, MR93, PBE95][1] and discussing approaches how an integrated view of heterogeneous databases can be provided for an application. Besides referring to practical applications, the current paper goes beyond these publications by establishing a classification of structural heterogeneity according to its causes and the mapping complexity resulting from it.

In general, aspects of **structural heterogeneity** and issues of schema integration are reflected

1. See [Sa96b] for a comprehensive listing of papers on heterogeneity.

in the architecture of federated database systems (FDBSs, [SL90]). The key idea is the translation of schemas captured in heterogeneous data models into so-called component schemas written in a common data model. The latter schemas, resp. views on these schemas, are then integrated into the federated schema.

Early approaches to FDBSs are the so-called DB gateways, mostly provided by commercial software vendors (cf., Sect. 2.1). The arrival of the object-oriented methodologies brought about the design of wrapper-based approaches using encapsulated access to source data (cf., Sect. 2.2). In contrast to wrappers, schema mapping languages serve to explicitly define the inter-relationship between source and target information (cf., Sect. 2.3).

There are several approaches dealing with **heterogeneity of semantics**. They are either based on automatic analysis of semantics [BHP94, Du94, GLN92, KFMRN96, SGN93, SPD92, ZHKF95] or on human expertise. The latter proposals comprise reverse engineering methodologies [An94, CMS94, HTJC93, PB94, VA95] and enrichment of the DB itself [SSR94]. However, to our experience the basic assumption underlying any kind of automatism, i. e., having a schema with expressive names and a low degree of inter-relationships among entities, is wishful thinking in most practical environments. Furthermore, the enriched schemas that are being generated are most often not formally related to the corresponding original DBs. Proposals to enrich the databases themselves might be promising, but tend to make large DBs even larger.

We have chosen the international (ISO) standard for the exchange of product model data (STEP, [IS94a]) as the basis for common schema structures and reference terms. This standard defines the object-oriented data model EXPRESS [IS94b], the access interface SDAI [IS96a], and a set of standardized schemas representing various application domains. For example, the schema described in [IS98a] represents bill-of-material structures in the automotive industry. The main advantage of this approach is to have not only a common data model as a basis for schema integration, but also a common schema structuring with semantics defined in ISO documents. This is particularly helpful when integrating DBs containing complex bill-of-material structures of different companies.

The goal of our approach is to overcome structural heterogeneity using standards and well-defined concepts as far as possible. Semantic heterogeneity is only considered as far as STEP is concerned. The main contribution of this paper is the analysis of structural heterogeneity and its solution. We outline existing approaches to this subject in more detail in the following subsections[2] and will show that mapping definition languages are most appropriate w. r. t. our application scenario. Thus, the discussion of the various approaches to such languages will be refined (cf., Sect. 4) according to our requirements, i. e., according to the types of conflicts to be solved. We conclude this section by a brief description of our goal (cf., Sect. 2.4).

2.1 Database Gateways

Early approaches to interoperability among heterogeneous DBs were based on so-called DB gateways. They were designed to provide access from a database application developed for a specific DBS to a DB managed by a different DBS on the same or perhaps another platform. Such gateways offer little or no transparency concerning gateway interfaces and locations. Furthermore, applications that need to refer to data from different sources cannot rely on system

2. We exclude so-called “universal storage” approaches, because they require to convert and store all data in an integrated system that is tailored to provide a unified view of data of different types [Bl97, RS97].

support for establishing a homogenized view to the data and are burdened by all processing steps to fetch, unify, and relate the heterogeneous data, e. g., by join operations. DB gateways are mainly used to migrate legacy systems (hierarchical, network, or relational DBs) to new (relational) environments or for interfacing relational DBs of different vendors.

The approaches sketched in the following are more advanced in the sense that they provide unified data views and DB operations (a uniform API) as though all data resided in a single local DB when, in fact, some or all of the involved data are distributed over remote heterogeneous data sources. However, the approaches differ considerably concerning the variety of data sources to be included, the complexity of view definition, and run-time optimizations of view derivation.

2.2 Approaches Using Wrappers

As the spectrum of data systems increases and the data types to be integrated become more and more complex, it becomes inappropriate for applications to have direct access to data and to their representation. Note there is a growing demand to have integrated access to a variety of data repositories ranging from relational over non-relational DBs up to non-database sources including spread sheets, text-processing documents, electronic mail, images, etc.

For such a wide spectrum of data sources with even higher degrees of heterogeneity, special middleware systems embody an alternative to an integrated view of heterogeneous legacy data without changing how or where the data is stored. [Bl97] characterizes such systems as "universal access" systems. While providing a common interface to new applications, such middleware systems achieve powerful query services for heterogeneous data by means of so-called wrappers [BE96] or data providers [Bl97]. These mechanisms encapsulate the underlying data and mediate between the data sources and the middleware. Since the native query support of these sources is so different in expressive power (varying from simple file scans to join operations on complex objects or media-specific search facilities), it is impossible to perform accesses through a standard interface. Hence, the middleware system approach to homogenized views on heterogeneous sources (heterogeneous views for short) necessarily has to exploit the query and access capabilities of the participating legacy systems. Encapsulation, however, has to be achieved by writing wrappers for every type of data source to be included (existing wrappers could be made available in a special library). Since the middleware system is offering tools for the creation of wrappers and for their integration, some kind of standardization is enforced thereby enabling extensibility.

OLE DB [Bl97] aims at providing a "universal access" facility for business applications. It is based on an infrastructure for encapsulated access to the original data sources, the so-called data providers. Its unifying abstraction for data access is the *rowset* representing a stream of values from a data provider. For simple "flat" data providers with no querying capability, data is easily exposed in a tabular form. More powerful data providers are instructed to accept commands and derive rowsets as result of their "inner" query processing. Although OLE DB does not include a generic, middleware-based query processing facility to be dynamically used for the combination of lower-level result rowsets, a protocol can be defined by which a middleware component (service provider) and data providers can interact to do the job.

The use of an object-oriented interface (based on the ODMG-93 data model including some extensions [Ca96]) for accessing heterogeneous data in a variety of existing data repositories including databases, files, as well as multimedia objects is explored in the Garlic Project

[Ca+95]. Since multimedia types do not lend themselves to schema mapping, the unified access to the different data sources relies on the use of wrappers, too. Compared to OLE DB, the Garlic middleware is taking a much more generic approach to query planning where “the wrapper and the middleware dynamically negotiate the wrapper’s role in answering a query” [RS97].

In contrast to mapping definition languages, the use of wrappers tries to encapsulate the various heterogeneous data and therefore masks many problems of structural and semantic heterogeneity (or shifts them to the wrapper writers). In this approach, it is not distinguished between a description model and an execution model. That is, the mapping specification as well as the mapping execution are hard-coded in the wrappers which disallow for the use of optimizations. Moreover, updates of data sources, although possible in principle, are usually not considered. However, the clear separation of data sources accessed via wrapper interfaces and the role of the middleware as an intermediary makes this approach easily extensible.

2.3 Advanced Approaches to Schema Mapping

Schema mapping approaches permit a separate view definition “independent“ of the source schemas. As a consequence, they require an explicit mapping specification, that is, rules describing the derivation of target data from source data (and vice versa). Obviously, such an approach can provide more flexibility in terms of renaming, source assignment, multi-source correspondences, etc. Moreover, by delegating the mapping task to DB experts the complexities of the source schemas[3] can be kept transparent for the user. As motivated above, these languages have to bridge between potentially poorly defined schemas and, as a consequence, have to make explicit and to document hidden semantics. Such an explicit specification allows to better understand and control the interrelationship between the target and source schemas as well as the data correspondences among the source schemas.

Although many papers have been published on this subject [AB91, AR90, Ba95, CL92, Ha95, KC95, KCGS95, KDN90, Ke91, KLK91, PGU96, SPD92, SST92, TC94], a satisfactory approach was not proposed so far. Hence, a language which offers solutions to all problems of structural heterogeneity (Sect. 3) in a formal and understandable way, maps/integrates schemas of heterogeneous data models, and supports the update of views, is still urgently needed.

Such a language realizes a more generic kind of middleware technology by distinguishing between the description model, i. e., the mapping definition language, and its execution. However, current systems allow only limited variability of the data sources, e. g., they can access relational DBs of different vendors as well as non-relational data sources such as formatted files. The views made available to the application are based on traditional view definition languages and are more or less relational with SQL as an API. For example, the product DataJoiner [IBM95] which currently supports single-source updates is indicative for these approaches.

2.4 Our Goal

The environments we have in mind possess a known number of DB source types and do not strongly rely on access to encapsulated data and extensibility of source types. However, they frequently need “manual” mapping support which includes type conversion, conditional mapping, as well as checking and resolution of conflicting values when view attributes are combined from various data sources. Moreover, view update is an important option. These were our major reasons for the development of a mapping definition language.

3. The source schemas we referenced have about 200 entities. An ‘item‘ was represented by about 20 relations, each of them having 50 attributes on the average.

Using the ISO standard STEP, we have a common data model for view definition. However, STEP does not include a mechanism to map EXPRESS views to heterogeneous source schemas. Therefore, a mapping language bridging between pre-existing DB schemas and the (integrated) view schema formulated in STEP is still needed. Thus, the goal of our work is to develop such a language that satisfies the following main requirements:

- Integration of multiple schemas written in potentially heterogeneous data models, i. e., mapping of data between heterogeneous schema structures
- Descriptiveness of the language, i. e., declarative mapping specifications.
- Immunity from technological changes, i. e., independence of the mapping specifications from DBMS and platform characteristics.
- Support of user-defined update statements having a similar expressiveness as retrieval statements.

Since heterogeneous views can be composed in many ways, we give some examples to characterize the possible spectrum that should be supported by an industrial-strength mapping language. As indicated in Fig. 1, source data (denoted by S) typically comes from multiple sources S_i (i=1, ..., n) from which a view (the target data T) has to be derived. Alternatively, data can be transformed from one schema (say T) to another (T'), e. g., to replace a legacy system. For the description of the S_i, we may have a variety of schemas from different data models, typically relational, object-oriented, ER, and EXPRESS schemas. In our approach, we have chosen EXPRESS to define T (T').

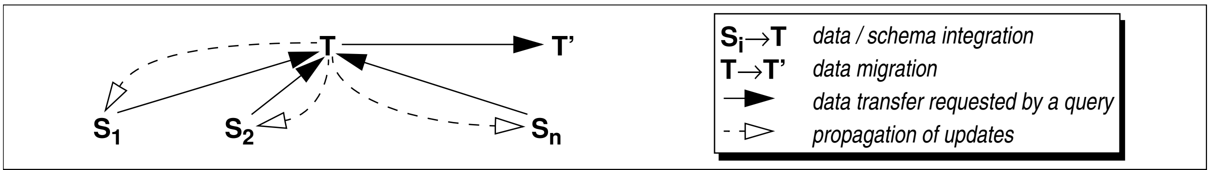


Fig. 1: Scope of heterogeneous views

The set of S_i may be homogeneous or heterogeneous depending on whether or not all sources are described by a uniform data model.

Another aspect directly determining the mapping complexity is the coverage of data types describing the way in which real-world entities are reflected in the sources to be integrated. It is primarily influenced by two criteria, the contents of the source schemas and their respective modelings. Both criteria can be subdivided into three categories: identical, partially overlapping, and disjoint. For example, a real-world entity might be modeled in an identical fashion in two source schemas, the modelings might be similar ("overlapping"), or they might be totally different, i. e., disjoint. If all participating source schemas contain semantically identical object types and properties (attributes), the coverage is said to be *congruent*, otherwise *incongruent*. The most complex case of incongruence and the only one we will refer to subsequently is the one in which both the schema contents and the modelings employed are partially overlapping. Congruence may not only occur among data sources, but also between a source schema and its target schema. In order to properly distinguish those cases, we term congruence among sources *horizontal congruence*, while that between a source and a target is called *vertical congruence* (cf., Sect. 3.1).

We exemplify the possible mapping bandwidth referring to mapping type S → T, i. e., the heterogeneous view construction. The following list indicates mappings of increasing complexity:

- homogeneous and congruent S_i are mapped to T described in the same (a different) data model
- homogeneous and incongruent S_i are mapped to T described in the same (a different) data model
- heterogeneous, but congruent S_i are mapped to T described in given data model
- heterogeneous and incongruent S_i are mapped to T described in given data model.

Another mapping aspect causing additional complexity is the mapping cardinality. It characterizes uni-/bidirectional mapping, i. e., whether the mapping is specified only in one direction, like in traditional relational views from S to T, or in both directions, which will be enabled by our mapping language (see Sect. 5.4). Finally, the mapping T → T' supports data migration or special forms of data propagation via DB import schemas.

In order to show how heterogeneous views can be provided and which kind of mapping problems must be solved, we have to identify the mapping conflicts first. For this reason, we investigate the intrinsic problems caused by the construction of heterogeneous views.

3. The Intrinsic Problems of Structural Heterogeneity

Based on our experience on the mapping between heterogeneous schemas, primarily for bill-of-material structures, we have identified different kinds of structural heterogeneity. In this section, we present a detailed classification of structural heterogeneity according to the mapping complexity they cause and arrange them in a multiple tier representation (cf., Fig. 2)[4]. This figure can be interpreted as a sequence of stairs characterizing the increasing mapping complexity when heterogeneous views are derived. On the horizontal axis, we have listed the features to be dealt with during the mapping process and the sections in which they are further described. Using the stairs in vertical direction from the left to the right, we express the increasing complexity of the features. The most simple case in our classification starts with the bottom level embodying a couple of features which can already be encountered in the relational model. Advanced data models provide additional concepts and issues, e. g., aggregation, abstraction concepts, and object identity in case of object-oriented data models, which require more complex mapping steps. Some of the concepts hide their semantics or are defined procedurally (code of methods or functions), thus diminishing the applicability of automatic mappings. In the most complex situations, for example, ADTs (top right corner), manual interaction may be needed to reach a semantically equivalent mapping result.

In the following we describe the different aspects involved in the mapping of heterogeneous views starting with the simplest situation. That is,

- source data is *structured* and described by a single schema,
- only *one source to one target* schema has to be mapped,

4. Please find the detailed discussion of the multiple tier representation in [Sa96b].

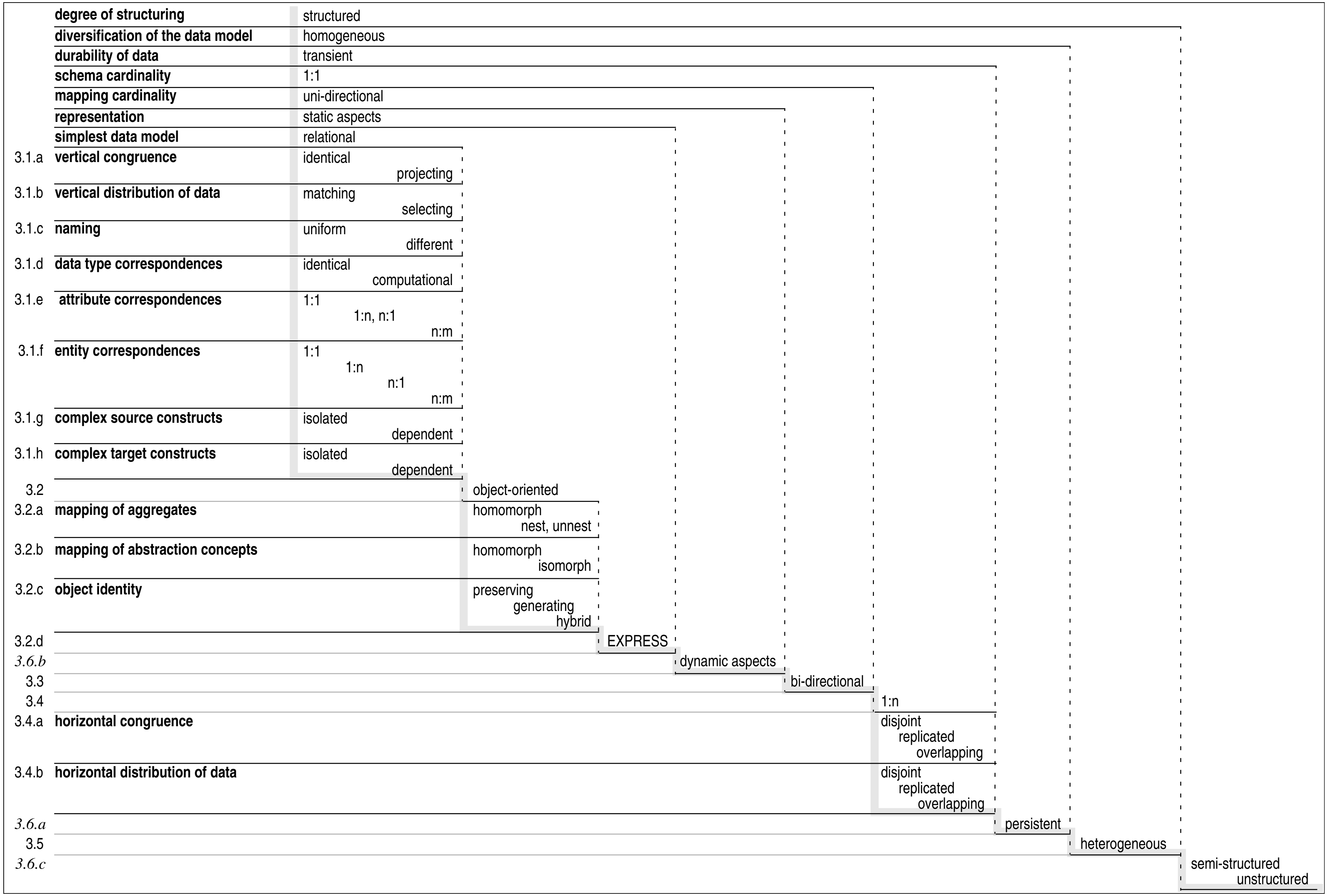


Fig. 2: Multiple tier architecture of heterogeneity

- both schemas are written in a *homogeneous relational*[5] data model, and the mapping is only *uni-directional,* i. e., the mapping is specified from source to target (the so-called read-only views).
- only *static aspects* are considered (i. e., no methods, no specification of behavior, no triggers, etc.), and
- target data is *transient* (i. e., data is only accessible within a single transaction and maintained on volatile storage).

The use of a more powerful data model, e. g. an object-oriented one, introduces more advanced concepts or features burdening the mapping process. This is illustrated in Fig. 2 by the relevant concepts to be discussed in the sequel at the various levels of the tier representation.

Subsequently, we refer to the situation involving data or schema information from source and target as *vertical distribution*, whereas *horizontal distribution* denotes the participation of multiple sources. *Complex constructs* are defined to be composed of elements of multiple sources.

3.1 Basic Mapping Problems

The features which we describe first exist independently of the expressiveness of the data model (relational, object-oriented, etc.), the schema cardinality (1:1 or 1:n), the directions of mapping (uni-directional vs. bi-directional), and the power of representation (static/dynamic aspects).

a) Vertical congruence

The trivial case of heterogeneity is the one in which an *identical* application domain is represented in S and T. Additionally, schema elements might be *projected* from S when deriving T. In this case, the projected information must be added during the generation/insertion of data according to S. Fig. 3 illustrates such a scenario; it essentially leads to the view-update problem.

In Fig. 3, the name of the person referenced by foreign key `ITEM.created_by` and represented as `PERS.name`, is projected to T. As a consequence, in order to exploit the primary key/foreign key relationship between `ITEM.created_by` and `PERS.key`, e. g., when propagating operations on T to operations on S, `item.created_by` can be used to identify the correct tuple in the source relation `PERS`, which, in turn, hosts the requested attribute `PERS.key`.

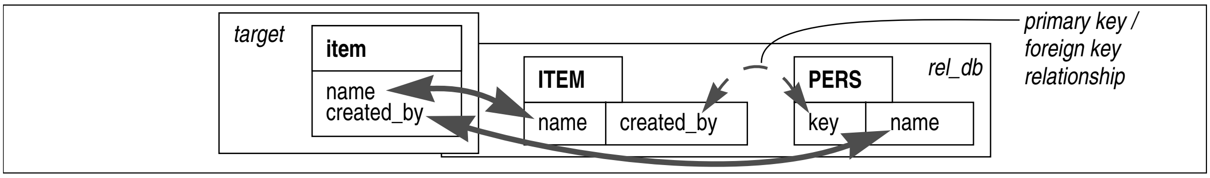


Fig. 3: Projection of information

b) Vertical distribution of data

It is slightly more difficult to map between S and T if data is *selected* instead of having a *matching* between both. Then, predicates representing the selection must be defined.

5. We use the object-oriented data model EXPRESS as the common data model and consequently focus in our general architecture on the mapping between object-oriented schemas and schemas written in the relational, an object-oriented or the EXPRESS data model. Nevertheless, we want to abstract from these assumptions when discussing the various kinds of heterogeneity.

c) Naming

Different naming (homonyms, synonyms) in S and T (i. e., data type, entity[6], and attribute names) as opposed to *uniform* naming requires the use of renaming functions or human interaction, e. g., `item.name` and `ITEM.name` are synonyms (cf., Fig. 3), whereas `item.created_by` and `ITEM.created_by` are homonyms.

d) Data type correspondences

If the data types are not *identical* but only *computationally* equivalent, transformation functions have to be defined, e. g., functions to convert DM to US$. In related work, this criterion is often further sub-classified into conflicts of units, scaling, granularity, etc.

e) Attribute correspondences

If attributes correspond on a *1:n* or an *n:1* basis instead of *1:1*, attributes must be concatenated or split, respectively. Often, this mapping is defined to be conditional, i. e., one attribute corresponds to a specific attribute under a certain condition and to (an)other attribute(s) under different conditions. If attributes have to be mapped on an *n:m* basis, the conversion functions might become even more complicated. For example, if the geometry of a rim is represented in T by three points of a circular line and in S by the centre and the radius of the rim, complex mathematical operations must be applied. In addition, integrity constraints have to be specified to guarantee the same mapping, i. e., one single arrangement of the three points in T from the same source information.

f) Entity correspondences

Permitting an *n:1* correspondence between entities[6] in S and T instead of *1:1* relationships requires the identification of different source entities in the view definition of the target entity. Another problem arises by *1:n* correspondences between entities in S and T. In this case, many target instances might be mapped to one single source instance, and the criterion to distinguish between the target instances is only available in another target instance referencing them. This problem is called instance-dependent mapping and is illustrated in Fig. 4[7].

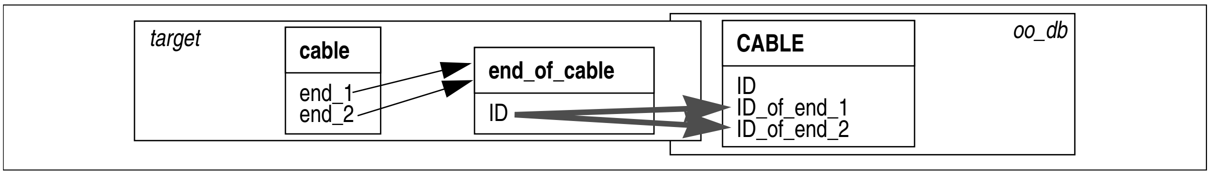


Fig. 4: Instance-dependent mapping

Mapping `end_of_cable` to `CABLE` requires to distinguish between mapping `end_of_cable.ID` to `CABLE.ID_of_end_1` or `CABLE.ID_of_end_2`. Thus, this mapping is dependent on the path along which `end_of_cable` is referenced by `cable`. This problem becomes even worse if all instances of `end_of_cable` are generated first, and the complex instances of `cable` are generated afterwards.

If an *n:m* correspondence at the entity level is allowed, all the problems of the *1:n* and *n:1* cases have to be dealt with.

6. In this sentence, entity stands for a class, an entity, or a relation.
7. In this example, we use object-oriented schemas. Obviously, the same can be modeled in the relational data model having the same effects.

g) Complex source constructs

Allowing for *dependent* source entities/instances instead of *isolated* source entities means that an entity of T must be collected from multiple (heterogeneous) entities of S. That is, the specification of joins in the relational data model and path expressions in object-oriented data models is required.

h) Complex target constructs

Two problems arise when *dependent* target entities[6] must be generated instead of *isolated* target entities. The relationships to be built might deviate from the corresponding relationships in the source, e. g., the direction of relationships might be inverted. Furthermore, the connection between two target entities referencing each other has to be established by specifying the link between the two entity identifiers (IDs) and the referencing attribute resp. the index position of a set-valued attribute. The identifier of the referenced entity might have to be specified in terms of source constructs (IDs, path expressions, primary key/foreign key relationships, etc.). In particular, such an approach is required, if the instantiation of target entities has to be more flexible. For example, consider Fig. 5. If all persons according to the target schema are retrieved first and then all items are instantiated, each instance of `person` to be connected to some item has to be identified. Thus, information about the connection between a specific item and its approving person, which is available in the source DB, must be evaluated, i. e., the path between `ITEM` and `PERSON` in the example below.

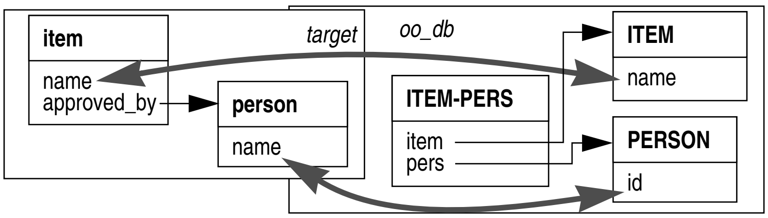


Fig. 5: Network mapping[7]

In Fig. 5, three inter-connected entities in S (resp. two relations having primary key/foreign key relationships in a relational schema) have to be mapped to a complex object in T. In object-oriented source schemas, the entity `ITEM-PERS` represents a relationship object and in the relational data model a relationship table. This mapping example is typical for the kind of specifications encountered during our explorations.

3.2 Enhanced Mapping Problems

The *relational* data model has less expressive power than *object-oriented* data models and, consequently, causes less mapping problems. Object-oriented data models have the following additional concepts [At+89]: OID, complex objects (i. e., use of references instead of values to ‘connect’ related objects and aggregates), abstraction concepts (generalization, overriding, overloading), as well as concepts defining behavior (e. g., methods). We discuss problems according to these aspects in the following subsections.

a) Mapping of aggregates

Aggregates might be mapped in a *homomorphic* way, i. e., target and source set-valued attributes have the same element type (e. g., char) and the same nesting of constructor types (e. g., LIST OF LIST OF char). If the set-valued attributes have different constructor types, this might cause some loss of semantics in the target schema: the ordering of elements specified in a LIST cannot be defined in a SET. If the corresponding attributes do not have the same depth of nesting, the *nest* resp. *unnest* operation must be applied. This situation is illustrated in Fig. 6.

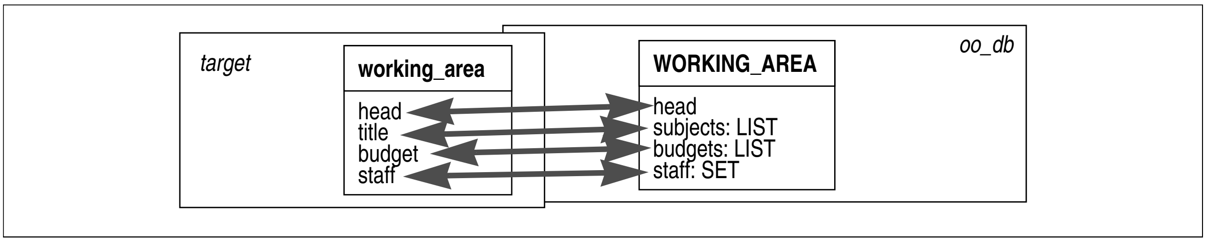


Fig. 6: Nest and unnest

We assume our source schema to contain an entity `WORKING_AREA` having the single-valued attribute `head`, the set-valued attributes `title` and `budget` whose elements are related according to their index positions, and the set-valued attribute `staff` representing those employees that work on all areas, i. e., on all elements of the attribute `title`. The target schema contains the corresponding entity `working_area` having the same attributes but all single-valued. Thus, this scenario might be represented by the source instance (“Sauter“,(“heterogeneity“,“mapping“),(5,10),(“Schäfer“)) and by the target instances (“Sauter“,“heterogeneity“,5,“Schäfer“) and (“Sauter“,“mapping“,10,“Schäfer“).

b) Mapping of abstraction concepts

If abstraction concepts are not mapped in a *homomorphic* but an *isomorphic* way (due to, e. g., mismatch of expressiveness in source and target data models), the precise abstraction semantics usually cannot be preserved during the translation process. For example, the distinction between customers and employees might be represented by two subtypes of supertype `person`, or, alternatively, it might be represented by different attribute values (i. e., “customer“ or “employee“) in another schema. Entity vs. attribute or attribute vs. value conflicts also belong to this category.

c) Object identity

At first sight, the case in which object identities (OIDs) or primary keys of S are to be *preserved* in T seems straightforward. However, if an n:m (n<m) instance correspondence between S and T is encountered, it is not possible to establish a mapping based on this premise[8]. In contrast, *generating* target identities whenever an instance is created allows more powerful mappings, but requires to document the correspondence between target and source identities. A *hybrid* mechanism would allow for both techniques.

c) Advanced type system

The EXPRESS data model introduces advanced concepts w. r. t. the type system such as so-called abstract supertypes (which cannot be instantiated), additional types (e. g., so-called SELECT types), advanced subtype constraints, etc. [IS94b]. As a consequence, more powerful mapping mechanisms are needed to resolve those conflicts.

3.3 Directions of Mapping

In the preceding discussions we assumed that the mapping is specified only *uni-directionally*,

8. Source IDs cannot be used in a preserving way when n instances in S must be mapped to m>n instances in T, since the number of IDs to be generated (m) is greater than the IDs available (n). Hence, a n:m (n<m) correspondence between entities in S and T is not a necessary condition for this type of conflict but one potential cause.

i. e., from S to T. If both directions are specified in one mapping statement, i. e., retrieve operations as well as update operations (*bi-directionally*), the well-known view-update problems come into play. Consider Fig. 3, for example. If the key of the person is projected, an update on the name of the person according to T cannot be automatically transformed into a corresponding operation according to S. The modification operation in T could mean that a creator of a set of items changed his/her name (in which case the name of the person (*PERS.name*) should be changed in S), or it might indicate that a different person was named creator of those items (implying that the references to the person who created the items should be changed (*ITEM.created_by*)).

Another kind of problem arises if the mapping corresponding to Fig. 4 is specified in the opposite way, i. e., from T to S. This type of conflict resembles the so-called check-in dependencies. Let us assume that one instance of *end_of_cable* is transient, so that data must be saved to non-volatile storage media. This check-in procedure is a legal operation according to the target schema, because values are assigned to all attributes of all entities to be flushed out. It is, however, not a legal operation according to the source schema, because the criterion to decide how to map *end_of_cable.ID* (to *CABLE.ID_of_end_1* or *CABLE.ID_of_end_2*) is missing.

3.4 Schema Cardinality

So far, we considered the mapping of *one* target *to one* source schema. Constructing *one* target schema from *many* source schemas, called *schema integration*, requires to combine schema information as well as data that are both distributed over the sources. In this scenario, the following cases can be distinguished.

a) Horizontal congruence

As outlined in Sect. 2.4 where we defined the notion of congruence, source schemas and their application domains might be disjoint, identical, or overlapping. If schema information is *identical* in the sources, semantically equivalent entities might possess different identities (the so-called entity-identification problem) or they might be represented differently in the sources. For both cases, consistency among the sources has to be guaranteed. In general, the probability of heterogeneous structures in the schemas rises with the number of schemas to be connected. If information is *overlapping* among the sources, the complete information in the target schema is not available in some sources. The probability of encountering different identities for one and the same source entity is even higher, because information is projected among the sources.

b) Horizontal distribution of data

Data might be distributed in a replicated, disjoint, or overlapping way. If the data is *replicated*, the same problems occur as if schema information is replicated (see above). If the data is distributed in a *disjoint* way over the sources, predicates must be defined how to propagate the data associated to a target instance to the different sources. If the data is *overlapping* among the sources, data has to be assembled during the integration process to generate target instances.

3.5 Diversification of the data model

The main problem of *heterogeneous* data models is to preserve the semantics while transforming the data to a semantically poorer data model. For example, the abstraction concepts aggregation and association provided by some object-oriented data models are not supported by the relational data model. As a consequence, there is a shift from one level of data model constructs towards a lower level of abstraction, namely towards schema elements such as attribute values.

Obviously, the latter kind of representation is dependent on the DB designer resp. design methodology. As a consequence, it is less semantically clear defined.

3.6 Further Problems

In the following, we briefly discuss problems not addressed specifically by our mapping language. Nevertheless, as will become clear from the subsequent discussions, those problems have to be kept in mind when defining a schema mapping.

a) Durability of data

If the target data is maintained *persistently*, change detection and consistency among source and target data must be supported. Furthermore, guaranteeing inter-transactional correspondence of instance IDs between T and S might become a difficult task, in particular, if some data of S is extracted from an archive. In Sect. 5, we will show that this criterion does not influence the constructs of our mapping language. The research area on persistently maintained target data is addressed by the work on materialized views, data warehousing, and data migration.

b) Dynamic aspects

Up to now, we concentrated on static aspects of the data model. The mapping becomes more complicated if *dynamic aspects* are to be considered as well. In this case, the correspondence between side effects of functions, return values of methods, time, programming languages, etc. has to be documented. On one hand, the behavior of arbitrary programs cannot be derived automatically. On the other hand, despite the wide range of methodologies to specify dynamics, such as Petri-nets, ECA rules, methods, etc., it is impossible to find a single representation that allows to capture general aspects of dynamics in both a precise and sufficiently abstract way. For these reasons, dynamic aspects are not considered by mapping languages yet.

c) Degree of structuring

In the following discussions, we focus on *completely structured* data, i. e., data whose representation can be fully handled by a "DBS-style" data model. As a consequence of this property, generic DB operations suffice to access those data. However, depending on the applications that serve as sources for the mapping, the associated data may be *semi-structured* (e. g., in the form of application-specific data types (ADTs) or HTML files) [Wi95] or even *unstructured*, i. e., basically stored in what in DB terms is called a LOB.

While it is no problem to provide the mere data in T, supporting adequate access functionality becomes more complex the less structured the source data are. To illustrate this, just consider completely unstructured source data. Without any knowledge of the source application, access to those data can only be provided based on LOBs and their generic access facilities (essentially string operations). Offering more sophisticated access routines requires to encapsulate the sources by "wrappers" that serve as interfaces to the data.

In summary, the degree of structuring of source data indicates how the overall complexity of mapping is distributed on the tasks of mapping data or mapping functionality. With decreasing structure of source data, the effort is shifted from data mapping to operational mapping (encapsulation). In the extreme case of completely unstructured data, nothing can be done about data mapping, and encapsulation becomes indispensable.

4. Related Approaches to Schema Mapping and View Definition Languages

So far, we have discussed and classified the mapping conflicts resulting from the construction of heterogeneous views. Before we present our own solution to cope with these mapping conflicts, the language BRIITY, we briefly sketch view definition languages proposed in the literature. These approaches to schema integration resp. view definition can be divided into the following categories: logic-based views, procedural languages, and declarative language approaches.

Although logic-based views as proposed in [KLK91] are specified in a declarative way, it is controversial whether or not a mapping specification based on (first-order) logic is intuitively understandable. Furthermore, only minor conflicts are solved, such as different naming, heterogeneous attribute correspondences, integration of multiple relational schemas, and update capabilities. A fundamental drawback to be found in all logic-based approaches is the fact that they do not consider the propagation of update operations on views. Although such operational constructs are not part of "pure" first-order logic, they are essential to obtain a practically relevant as well as commercial-strength language for view definitions.

Procedural languages [KFMRN96, Ke91] are very powerful and may include explicit data type mappings. However, the larger the schemas to be mapped, the more unreadable and unclear the complete mapping specification will be. Furthermore, the procedural description restricts the execution of the mapping, in particular the creation of target instances which is mostly determined in a specific ordering. Obviously, this strategy might cause problems in applications having large schemas.

Declarative language approaches are easier to understand, more flexible to use, and more suitable to be optimized at run-time. Some approaches are logic-oriented and solve only minor problems [KLK91, PGU96]. The so-called assertion-based approaches [SPD92, KC95] provide powerful and data-model independent mappings. There are many languages which are based on traditional relational or object-oriented views and which are extended for schema integration [AB91, KCGS95, Ba95, Ha95, SST92, CL92, TC94, KDN90, AR90]. However, all these approaches do not support the explicit specification of update operations, i. e., the well-known view-update problems cannot be solved by those languages. Consequently, updating the essential parts of integrated views, which are defined by joining source elements, is restricted. In [BSKW91, SST92], mechanisms are proposed which allow for the specification of updatable views under the assumption of object-preserving operations which is too restrictive w. r. t. our requirements (heterogeneous entity correspondences). Most of these language approaches do not support nest/unnest operations, nor target object inter-relationships. To the best of our knowledge, integrity constraints to be evaluated during the mapping, e. g., to address the problem of conflicting/missing source data, are not within the scope of those languages.

5. The Mapping Language BRIITY

Our approach is designed to avoid the deficiencies described before and to permit the flexible specification of bi-directional mappings. The key characteristics of BRIITY are its power w. r. t. the number of mapping conflicts solved, its descriptiveness, its immunity from technological changes, and its support of user-defined update statements having a similar expressiveness as retrieval statements. In this section, we highlight the general structure of our language by referring to the mapping specification of Ex. 1. It defines the mapping for the example depicted in Fig. 3.

```
 1:  BEGIN
 2:    MAPPED_SCHEMAS
 3:       ts := target_schema <- rel_db:= rel_db@rel_dbs@localhost;
 4:    END_MAPPED_SCHEMAS;
 5:    INCLUDE
 6:       LIB /usr/users/sauter/libstring.a;
 7:       INC string.h;
 8:    END_INCLUDE;
 9:    TYPE_MAPPING
10:       MAP  ts.DM <- rel_db.US$;
11:            ts.DM <- 0.67 * rel_db.US$;
12:            rel_db.US$ <- 1.5 * ts.DM;
13:       END_MAP;
14:    END_TYPE_MAPPING;
15:    ENTITY_MAPPING
16:     MAP item <-  _item := rel_db.ITEM, _pers:= rel_db.PERS;
17:       ON_RETRIEVE
18:          name <- _item.name;
19:          created_by <- _pers.name;
20:          IDENTIFIED_BY(_item.name,_pers.key);
21:          WHERE (_item.created_by = _pers.key);
22:       ON_UPDATE ...
23:       ON_INSERT ...
24:       ON_DELETE ...
25:     END_MAP;
26:    END_ENTITY_MAPPING;
27:  END.
```

Example 1: General structure of a mapping specification[9]

5.1 Overall Structure of a Mapping Specification

A mapping specification starts with basic definitions that lay the foundations for the subsequent mapping rules, i. e., the names of the source(s) and the target schema involved, as well as type-specific mappings between S and T. The essential part of such a specification is the `ENTITY_MAPPING` section which relates target entities[6] and their attributes on one side to source constructs on the other side. The last section of a mapping specification (omitted in Example 1) provides means to declare additional integrity constraints such as check-in dependencies (cf., Sect. 3). Before discussing the `ENTITY_MAPPING` section in detail, we will give a brief overview of the overall constituents of a mapping specification.

Overall declarations and mapping of data types

In the `MAPPED_SCHEMAS` section (line 2-4), one target and one or more source schemas are identified by the name of the corresponding DB, DBS, and the host ID. If the target data is transient instead of persistently stored in a DB, the name of the target schema can also be a file name.

Data types and functions defined in some programming language that are related to the mapping process itself and that, for this reason, cannot be attributed to S or T (to avoid introducing dependencies in those schemas or to violate their autonomy), can be imported with the help of the `INCLUDE` section. For example, string manipulation functions may be defined in a library called *`libstring.a`* and included via the statement in line 6.

For a better structuring of our mapping specifications, we separate the mappings of type-level constructs from those of the entity level. Types (built-in or user-defined ones) are handled in the `TYPE_MAPPING` section. Simple mappings without conversion functions consist only of the type mapping header as, e. g., line 10, which maps a *varchar* in S to a *string* in T. Apart from

9. Without loss of generality, we will disregard giving syntax definitions of our language because of space limitations. Interested readers may refer to [Sa96a].

a type mapping header, more complicated mappings also possess a type mapping body (line 11-13) that permits to refer to arbitrary mathematical expressions and/or (included) functions. Thus, bi-directional conversions between data types can be defined, i. e., from S to T (line 11), or vice versa (line 12).

ENTITY_MAPPING section

Like the type mapping section, the one responsible for entity mapping consists of a header (line 16), relating one target entity[6] to one or more source entities, and a body (line 17-25) with the detailed definition of the mapping itself. The body is further subdivided into an `ON_RETRIEVE`, an `ON_UPDATE`, an `ON_INSERT`, and an `ON_DELETE` clause. In the `ON_RETRIEVE` clause, the user can define how retrieve operations on target attributes of the corresponding entity should be translated to DB accesses. In analogy to Sect. 3, these operations can be basic ones to be applied when an (integrated) view - specified in an object-oriented, relational, or EXPRESS schema - is established on top of relational DBS (cf., Sect. 5.2). Alternatively, they can resemble advanced mapping rules for the integration of object-oriented or EXPRESS schemas. The propagation of update operations according to the modifications of target attributes can be specified in the `ON_UPDATE` clause. Operations to be executed in S after the creation/deletion of a target instance can be defined in the `ON_INSERT` resp. `ON_DELETE` clause.

Integrity constraints

In this section, conflict resolution support is provided for the instance level, which may be applied when contradictory, incorrect, missing, or obsolete data occur. Furthermore, mapping-specific constraints (cf., check-in dependencies, Sect. 3.3) are introduced. These integrity constraints serve to extend native DB constraints to be checked when retrieving source data. This is particularly true for the view integration across legacy systems because they usually don't have explicitly specified integrity rules. Even in current DBSs referenced as data sources, such integrity constraints typically disregard data dependencies to other DBSs. Therefore, the "middleware" integrating data from those isolated systems must incorporate mechanisms to define "global" integrity constraints or to make existing ones explicit.

We now turn to the `ENTITY_MAPPING` section and its facilities for defining rules for propagating retrieval as well as modification operations on T to the source schemas (Sect. 5.2 to Sect. 5.4). In Sect. 5.5, we will describe those elements of our language which address the problem of heterogeneous entity correspondences. Finally, Sect. 5.6 will exemplify the constraint mechanism of BRIITY that allows to solve the problem of check-in dependencies described in Sect. 3.

5.2 Basic Set-Oriented Mapping Rules

A key idea of our language is to combine an object-oriented data model with descriptive and set-oriented (i. e., relational-style) retrieval operations. This is an important objective, since most of our applications require object-oriented views on relational DBSs. Suitable support of such scenarios implies to represent the elements of the target schema in an object-oriented way whereas the corresponding retrieve operations on the source(s) have to be specified in a relational-like style, as illustrated in Fig. 7.

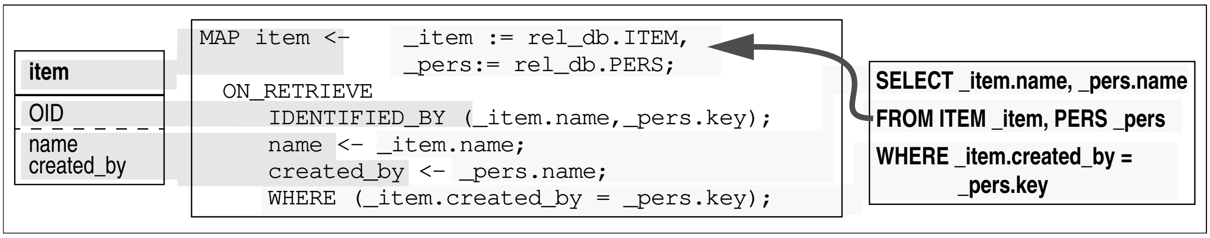


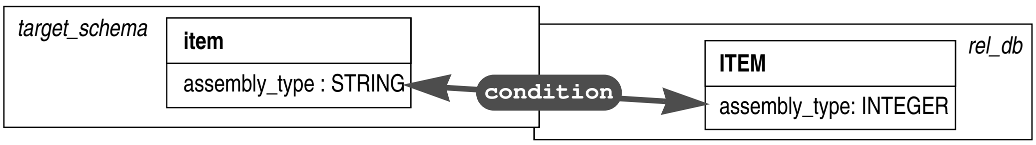
Fig. 7: Diverse styles of the mapping language (cf., Fig. 3 and Example 1)

The overall purpose of the entity mapping is to establish a relationship between instances in T and the corresponding instances in S. It is thus possible to check whether source data is already materialized in T and accessible by queries according to the target schema. To this end, object identifiers are assigned to all target instances. If the target schema is an object-oriented or an EXPRESS schema like in Fig. 7, these object identifiers (OIDs) might be made available in the target view, whereas these target OIDs are used only internally in relational schemas. The target OIDs to be generated or preserved depend on the correspondence of target and source instances. This correspondence is specified in the `IDENTIFIED_BY` clause as shown in Fig. 7.

The right hand side of an entity mapping header corresponds to the FROM clause of an SQL statement. The names of the relations are prefixed by the name of the related DB. This is necessary to distinguish multiple DBs. The right hand side of an attribute mapping rule (like '`name <- _pers.name`') can be compared to the projection of an SQL statement. It even has the same expressive power as this SQL clause if T is specified as a relational schema. In this case, target attributes are single-valued and do not contain pointers (REF) to entities of T. The `WHERE` part of the entity mapping corresponds exactly to the WHERE clause of an SQL statement.

In summary, straightforward mappings between target and source entities[6], i. e., those without conversions or predicates, consist only of the header of the `ENTITY_MAPPING` section. Thus, simple problems can be solved in a simple way by our mapping language. Moreover, as shown so far, we support projection / selection of information, joining of source constructs (cf., Fig. 2: dependent complex source constructs), renaming, and data type conversions by our mapping language as well.

Another problem discussed in Sect. 3 are conditional mappings, i. e., mappings that are to be performed under certain conditions. The language BRIITY allows to specify such conditional mappings based on `IF .. THEN .. ELSE` constructs, where the `THEN` and the `ELSE` clauses have the same power as the right hand side of attribute mapping rules. Conditional mappings might occur in the case of heterogeneous attribute correspondences that require to map many attributes in S to one attribute in T. Another situation is illustrated by Example 2 where, depending on the integer value of an assembly type in S, a string representation in T is produced.




```
MAP item <- _item:=rel_db.ITEM;
    RETRIEVE
        assembly_type <- IF (_item.assembly_type = 512) THEN "manufacturing"
                         ELSE IF (_item.assembly_type = 918) THEN "configurable"
                         ELSE ...;
```

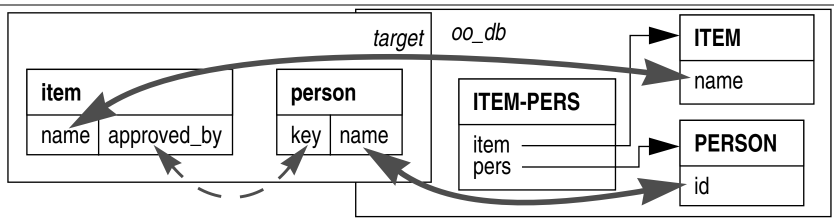
Example 2: Conditional mapping

Up to now, we discussed the fundamental structure of the retrieve section and the basic operations needed to map from a relational data model to a relational or a simple object-oriented one. In the following, we extend our considerations to the relational model including referential integrity constraints (cf., Fig. 2: dependent target constructs), as well as to general object-oriented data models and to EXPRESS. By doing so, we climb the next step in the tier architecture and discuss concepts of BRIITY to solve those problems. We defer the discussion on how our language addresses heterogeneous entity correspondences to Sect. 5.5.

5.3 Complex Mapping Rules

The mapping of referential-integrity dependencies in relational schemas as well as of object networks in object-oriented schemas is supported in BRIITY by the `CASCADED_MAP` operator. As for the description of objects, we strongly adhere to the methodology of object-oriented concepts [At^{+}89] in that the (mapping) definition of a complex object is split into separate (mapping) definitions of the root object itself and all referenced objects. The advantage of this approach is that referenced entities have their own (mapping) definitions being independent of referencing entities. Referential-integrity dependencies in relational target schemas are treated analogously. Starting from the "root" relation, all direct integrity dependencies and, subsequently, the transitive ones have to be defined. Thus, the mapping of complete complex entities in general requires at first the mapping of the root entity and subsequently, in a cascaded way, the mapping of the referenced entities and their links. These connections are represented by the `CASCADED_MAP` operator.

If the target schema is relational, the `CASCADED_MAP` operator can be used to specify referential integrity constraints over T. In Example 3, instead of naming the referenced entity (*person* in Example 4), the `CASCADED_MAP` operator refers to the corresponding primary key according to the target schema (*person.key*).




```
MAP item <- _item:=oo_db.ITEM, _ip:=oo_db.ITEM-PERS, _pers:=oo_db.PERSON;
    ON_RETRIEVE
        name <- _item.name;
        approved_by <- CASCADED_MAP person.key
                          WITH_ID _item.INV(_ip:item).pers.id
        IDENTIFIED_BY (_item.name);
END_MAP;
```

Example 3: Referential integrity in relational target schemas

Example 4 deals with an object-oriented target schema. The mapping definition of *person* is not influenced by the fact that this entity is referenced by entity *item*. On the other hand, in the mapping definition of entity *item*, the `CASCADED_MAP` operator followed by the name of the referenced target entity (*person*) specifies the link between the two entity mappings of *item* and *person*. This expresses the fact that an item is related to the person that created it. The corresponding path from the source instance of *ITEM* along the relationship entity *ITEM-PERS* to *PERSON* is specified by means of the `WITH_ID` clause of the `CASCADED_MAP` operator.

```
MAP person <- oo_db.PERSON;
    ON_RETRIEVE
        name <- oo_db.PERSON.id;
        IDENTIFIED_BY(oo_db.PERSON.id);
END_MAP;
MAP item <- _item:=oo_db.ITEM, _ip:=oo_db.ITEM-PERS, _pers:=oo_db.PERSON;
    ON_RETRIEVE
        name <- _item.name;
        approved_by <- CASCADED_MAP person
                         WITH_ID  _item.INV(_ip:item).pers.id
        IDENTIFIED_BY  (_item.name);
END_MAP;
```

Example 4: Network mapping (cf., Fig. 5)

The `INV` built-in function, which is also employed in the above example, is used to specify an inverse relationship. Hence, the two target instances of *`item`* and *`person`*, which are mapped to two source instances referencing each other, can be linked together.

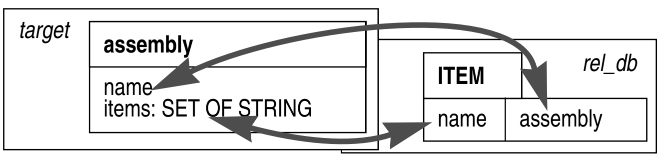
In the header of the mapping specification of subtypes, the supertype is referenced such that the (attribute-) mapping definitions of the supertypes can be inherited and their `WHERE` clauses can be combined. We follow this approach until a proper mechanism for overriding/overloading has been implemented.

The mapping between set-valued source and target attributes as well as the nest operation are supported in BRIITY by the `SET`, `LIST`, and `ARRAY` operators having the following syntax.

```
<set valued attribute mapping> ::=
  [NEST] (LIST|SET|ARRAY) '('<right hand side of attr mapping or nested set valued mapping>')'
  [WHERE <DNF expression>]
  [ORDER_BY <sort expression>]
  [GROUPED_BY <source attr identification>].
```

Definition 1: Mapping of aggregates

Each operator (`SET`, `LIST`, and `ARRAY`) possesses a `WHERE` clause. In contrast to the `WHERE` clause of the `ENTITY_MAPPING` (cf., Sect. 5.2) which selects instances to be mapped to all attributes of the entity, the `WHERE` clause of the set-valued attribute selects attribute values for a specific target attribute. The `ORDER_BY` clause - only applicable for the `LIST` and `ARRAY` operators - has the same semantics as the ORDER BY clause in SQL statements. Grouping of attribute elements (`GROUPED_BY` clause) is needed if the nest operation has to be applied subsequently. This is shown in the following example in which all items (*`ITEM`*) of the same assembly are associated to one instance of *`assembly`*. Therefore, all items must be grouped according to the assembly they belong to.




```
MAP assembly <- _item:= rel_db.ITEM;
    ON_RETRIEVE
        name <- _item.assembly;
        items <-  NEST (_item.name)
                  GROUPED_BY(_item.assembly);
```

Example 5: Nest operation

Opposed to nest, the unnest operation maps set-valued source attributes to multiple target instances with single-valued attributes.

```
MAP working_area <- _wa:=oo_db.W_AREA, oo_db.PERS;
  ON_RETRIEVE
      FOR_ALL (_unnest_subjects := UNNEST(_wa.subjects);
          title <- _unnest_subjects;
```

Example 6: Unnest operation (cf., Fig. 6)

In Example 6, a `FOR_ALL` operator is employed to iterate over the `working_area` subjects of a single entity in S and to assign each element to a different `title` entity in T. Moreover, the `FOR_ALL` operator permits to simultaneously address several attributes (that are related to each other in a parallel way). If attributes correspond to each other in a nested way (cf., Sect. 3), the `FOR_ALL` operators will also have to be nested.

5.4 Rules for the Propagation of Updates

One of the major characteristics of BRIITY is its capability to support update operations over T. To this end, the user can declaratively specify mapping rules that describe source manipulations resulting from target updates[10]. For this purpose, BRIITY allows to define update rules (cf., Definition 2: <update_statement>) within the `ON_UPDATE` clause which can be related to single target attributes. These rules are further subdivided into the `NEW`, `MODIFIED`, and `DELETED` sections (cf., Definition 2: <update_statement_body>) according to the kinds of modifications on the target attributes. The DB update operations listed in each section (<assign_statement>) specify the final state the source DBs have to reach after the modifications on the target data have been propagated[11]. The advantages of using state orientation in the update rules are the independence of update operations from the current state of the DB, the declarative form of the specification which permits to apply optimizations similar to SQL, and the independence of any implementation, e. g., in a specific programming language realizing the query translation. In particular, before propagating updates to the sources, each source DB is checked for validity of the desired final state. If this state has not yet been reached, e. g., an instance supposed to exist does not yet exist, the appropriate operations are performed on the source DB (i. e., the instance is created). The following definition gives an overview of the power of the update rules.

```
<update_clause> ::=
    'ON_UPDATE'       (<update_statement> {',' <update_statement>}
                  |   'INVERSE_TO_RETRIEVE' ';').

<update_statement> ::=
      <target_attribute_id> 'OF' <update_statement_body_list>
   |  <update_statement_for_set_valued_target_attributes>.

<update_statement_body> ::=
  ('NEW'|'MODIFIED'|'DELETED') ':' ('RESTRICTED'|'INVERSE_TO_RETRIEVE'|<assign_stmnt_list>)';'.

<assign_statement> ::=
  'ASSIGN'  <conjunction_of_assgn_statement_expr>
            ['WHERE' <where_clause_containing_bool_expr_of_assgn_statement>].

<assign_statement_expr> ::=
  ['NOT_']'IS_INSTANCE''('<source_entity_id>[<with_instance_id>]['.'<attr_value_expr_list>]')'
   |  <is_element_expr>
   |  <has_value_expr>.
```

Definition 2: Update rules[12]

Update rules are made up of `ASSIGN` statements consisting of an assignment part (following the

10. Alternatively to user-defined update operations, system-generated update operations can be specified using the keyword 'INVERSE_TO_RETRIEVE'.
11. This kind of operation is used in a very similar way in the knowledge-based management system KRISYS [Ma91]

keyword 'ASSIGN') and qualification rules introduced by the keyword 'WHERE'. These rules delimit the sphere of processing that the update statement will relate to. If no qualification rules are specified, the entire DB represents the initial state of the update rule, i. e., the statements of the assignment part are executed on the entire DB.

One fundamental construct of update rules is the (NOT_) IS_INSTANCE statement. Its first argument is the name of the entity[6] required to exist in the identified DB. The other arguments are assignments of attribute values. In addition to the IS_INSTANCE statement, the IS_ELEMENT and HAS_VALUE statements can be used to specify operations on set-valued attributes or on query variables, another key concept of the language.

ASSIGN statements can also be used in the ON_INSERT and ON_DELETE clauses to specify operations to be propagated after the creation or deletion of target instances. Interested readers are referred to [Sa96a].

Coming back to the example shown in Fig. 3, we assume the following intention for the propagation of update operations. After the target attribute *item.created_by* is initialized, the DB should contain two instances (resp. tuples) of *ITEM* and *PERS* that possess the corresponding target attribute values (i. e., ITEM.name ⟵ item.name and PERS.name ⟵ item.created_by), and that are linked together (i. e., ITEM.created_by = PERS.key). The modification of the attribute *item.created_by* should be translated into DB operations so that the person who created the item will change. An instance of *PERS* must be created if the person does not yet exist. Removing the attribute value of *item.created_by* requires the deletion of the corresponding instance in *PERS*. This is defined in BRIITY as shown in Example 7.

```
1:  ON_UPDATE created_by OF
2:    NEW:        ASSIGN (IS_INSTANCE(_item: name = ts.item.name, created_by = ?1) AND
3:                        IS_INSTANCE(_pers: key = ?1, name = ts.item.created_by));
4:    MODIFIED:   ASSIGN (IS_INSTANCE(_pers: name = ts.item.created_by));
5:                ASSIGN (IS_INSTANCE(_item: name = ts.item.name, created_by = ?1))
6:                WHERE  (IS_INSTANCE(_pers: key = ?1, name = ts.item.created_by));
7:    DELETED:    ASSIGN (NOT_IS_INSTANCE(_pers: key = ?1))
8:                WHERE  (HAS_VALUE(?1 = ASSIGNED_ID_VALUE(2)));
```

Example 7: Update operations (cf., Fig. 3 and Example 1)

The initialization of the target attribute is specified using two IS_INSTANCE assignments. One of them is responsible to ensure the existence of a tuple of *ITEM*, the other for guaranteeing that a *PERS* tuple exists (cf., Example 7, line 2 and 3). Query variables are used to specify links between two instances that are established under certain conditions. In our example, the query variable ?1 of line 2 is bound in line 3 to primary key values of the relation *PERS*. Thus, query variables can be realized to specify joins.

For the case of a modification on the target attribute, assignments specified for the initialization cannot be used in general. For this reason, the ON_UPDATE statement possesses the MODIFIED section. Its WHERE clauses (cf., line 4 to line 6) serve to bind query variables. For example, assume that a source DB contains two tuples of *PERS*, namely ("007","Bond") and ("008","Jones"), and one tuple of *item*, namely ("engine","007"). If the instance ("engine","Bond") in T is modified to ("engine","Jones") the statements of line 2 and 3 do not

12. <update_statement_body_list> is a list containing a number of statements of type <update_statement_body>. The same holds for <assgn_stmnt_list> as well as <assign_statement>. <conjunction_of_assgn_statement_expr> is a Boolean conjunction of <assgn_statement_expr>.

indicate whether the primary key value (of *PERS*) or the foreign key value (of *ITEM*) should be changed, since the query variable *?1* can be bound to two “007” (corresponding to line 2) or “008” (corresponding to line 3). For this reason, the WHERE clause (cf., line 6) is used to clearly bind the query variable. In particular, if the first assignment (cf., line 4) is omitted, the WHERE clause returns a false condition if there does not pre-exist a corresponding tuple of *PERS* in S which can be connected to the item.

To identify the corresponding source instance to be deleted after removing the target attribute value, we use the built-in function ASSIGNED_ID_VALUE which returns the value specified in the IDENTIFIED_BY statement of the same target entity. In the example above, the value of the attribute *PERS.key* is needed, which is specified as the second parameter of the corresponding IDENTIFIED_BY statement (cf., Example 1, line 20).

BRIITY supports the propagation of modifications on set-valued target attributes by the following statement.

```
<update_statement_for_set_valued_target_attributes> ::=
 'FOR_EACH' <query_variable> 'WITH' <condition_to_assign_value_to_query_var>
                              'OF' <update_statement_body_list>.
```

Definition 3: Update rules for set-valued target attributes

The query variable of the FOR_EACH construct is used to iterate on a set of elements (of the corresponding set-valued attribute) which is defined by the condition following the key word WITH.

So far, we discussed only update statements. The same syntax (<assgn_stmnt_list>) is used to define operations propagated after the insertion (ON_INSERT) or the deletion (ON_DELETE) of instances in T. Due to space limitations, these language constructs are not discussed in further detail.

From the above considerations it should become clear that by means of user-defined update rules some well-known view-update problems can be solved as, for example, join views [Sa98], e. g., modifications of the target attribute *item.created_by* can be propagated. As described in Sect. 3, traditional view mechanisms do not provide capabilities for the propagation of this modification.[13]

5.5 Multiple Instantiation

In this section, we introduce the concept of partitions supporting instance-dependent mappings and DB partitions addressing the integration of multiple DBs. The problem of instance-dependent mappings, explained in Sect. 3 (heterogeneous entity correspondence), stems from the need to discriminate instances of one entity (type) based only on information of single instances. In Fig. 4, the criterion to distinguish between the ends of a cable is not available in either entity representing an end of a cable, but only in the entity representing the cable. This problem is solved by splitting the mapping of the ends of a cable into multiple partitions each identified with an internal ID, as shown in Example 8.

13. Due to space limitations, a detailed comparison between view-update problems and solutions provided by the language is not performed. Interested readers on updatable views are referred to [Sa98, BCL89, La90, LS91, KLK91, Da83, RMKDP95].

```
MAP end_of_cable <- PARTITION _part_end1: _c:=rel_db.CABLE,
                    PARTITION _part_end2: _c:=rel_db.CABLE;
  PARTITION _part_end1:
     ON_RETRIEVE
        ID <- _c.ID_of_end_1;
        IDENTIFIED_BY (_c.ID, _c.ID_of_end_1);
  PARTITION _part_end2:
     ON_RETRIEVE
        ID <- _c.ID_of_end_2;
        IDENTIFIED_BY (_c.ID, _c.ID_of_end_2);
END_MAP;
MAP cable <- _c:=rel_db.CABLE;
   ON_RETRIEVE
      end_1 <-CASCADED_MAP end_of_cable
               PARTITION _part_end1
               WITH_ID (_c.ID, _c.ID_of_end_1);
      end_2 <-CASCADED_MAP end_of_cable
               PARTITION _part_end2
               WITH_ID (_c.ID, _c.ID_of_end_2);
      IDENTIFIED_BY(_c.ID);
END_MAP;
```

Example 8: Partitions (cf., Fig. 4)

The mapping is partitioned, so that the mapping process can be carried out separately for each of them. That is, the complete MAP statement is the union of all the partitions of that statement. Thus, an instance is created for each partition and assigned with the name of the partition to an internal ID. In the example above, two instances of *end_of_cable* are created, one having *_c.ID_of_end_1* and the other having *_c.ID_of_end_1* as attribute values. The two instances can be distinguished by the different names of their partitions. The `CASCADED_MAP` operator is extended so that it contains the name of the partition in order to specify the correct relationship between *cable* and *end_of_cable*.

Up to now, we discussed in detail the mapping between one target and one source schema, both written as relational, object-oriented, or EXPRESS schemas. The concept of mapping partitions can also be used to integrate multiple DBs into one federated schema. A DB partition (having the same syntax as partitions) must be specified for all entities of the same source schema to be mapped to one target entity. The mapping of overlapping information among the source DBs can be associated to additional partitions identified by the names of the corresponding DBs, e. g., '`PARTITION db1 AND db2: ...`'. These partitions consist of a `KEY_EQUIVALENCE` statement which relates the potentially heterogeneous primary keys of the various source DBs.

5.6 Additional Integrity Constraints

The language BRIITY allows the specification of constraints to address the problem of check-in dependencies described in Sect. 3. Dependent entities can be specified by documenting the links (cf., Example 9, line 4) between them.

```
1:  INTEGRITY_CONSTRAINTS
2:     DEPENDENCIES
3:        GROUP  cable, end_of_cable
4:        WHERE  (cable.end_1 = OID(end_of_cable)) AND (cable.end_2 = OID(end_of_cable))
5:     END_DEPENDENCIES;
6:  END_INTEGRITY_CONSTRAINTS;
```

Example 9: Check-in dependencies

ECA rules are supported to have more powerful means for the specification of integrity constraints. Actions will be documented as `ASSIGN` statements, and it will be distinguished between the events `ON_RETRIEVE`, `ON_UPDATE`, etc. described above.

6. Brief Overview of the Execution Model

Due to space limitations, we only briefly introduce the concepts of the execution model realizing an integrated view over heterogeneous DBSs. The following explanations refer to Fig. 8.

Data establishing heterogeneous views is cached in an object buffer which can be accessed by applications using a navigational interface (C++). In order to make such a view available, the data has to be fetched from the sources and mapped according to the given specifications. For this purpose, a mapping layer provides a set-oriented interface to the object buffer (operations similar to SQL3) and currently an SQL interface to the DBs (sources). The translation of user queries into DB accesses is performed in the mapping layer.

The power of the mapping language resp. its individual elements as well as the expressiveness of the target interface (schema together with the query language) determine the power of the algebra needed to suitably implement the mapping operations. We have chosen an algebra similar to NF^2.

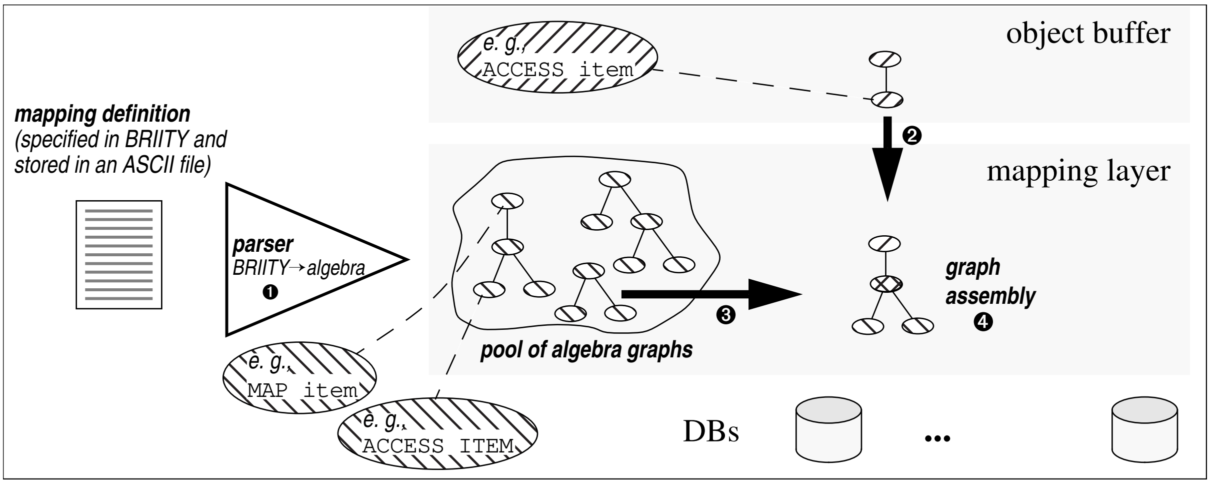


Fig. 8: Execution model

To derive a particular heterogeneous view, the corresponding mapping specification is translated by a parser into a directed algebra graph, the so-called mapping graph (cf., Fig. 8 ❶)[14]. The leaves of this operator graph embody DB access operations for the integrated DBSs (e. g., `ACCESS ITEM` in Fig. 8), its internal nodes are formed by algebra operators, whereas its root represents an operator to create target instances (e. g., `MAP item` in Fig. 8).

Loading data into the object buffer is requested by sending a set-oriented query to the mapping layer. There the query is also transformed into an algebra graph, the so-called query graph (cf., Fig. 8 ❷). The corresponding mapping graph is selected from the pool of algebra graphs (cf., Fig. 8 ❸), i. e., the graph with the resp. root operator. Both the query and the mapping graphs are then assembled by removing the root operator of the mapping graph and the access operator of the query graph (cf., Fig. 8 ❹). This is necessary in order to make the pool graphs independent of the application scenarios in which they are to be employed. The resulting graph builds the basis for further optimizations and for the generation of executable code to be sent to the DBs. The retrieved data is kept in (nested) relations within the mapping layer and processed according to the operators of the algebra graph.

14. Currently, the complete mapping specification is parsed at compile time, translated, and stored in a pool of algebra graphs.

Data from different DBs are integrated by join operators. In addition to relating data items, those operators also check the sources for inconsistent and conflicting data. For example, instances of different sources having the same logical ID and different attribute values are rejected. As described in Sect. 5, we are extending the integrity section of the language to provide more powerful constraint mechanisms.

7. Conclusions

In this paper, we have primarily focussed on two issues - the classification of mapping conflicts occurring in real-world applications as well as the concepts and specification of a mapping language coping with these conflicts. We have identified the types of structural heterogeneity appearing in our industrial-strength environment where heterogeneous views must be established on top of bill-of-material structures stored in different DBs. We ordered the conflict categories into a tier architecture which illustrates their particular influence on the mapping complexity when deriving heterogeneous views. One of the major results of this analysis is that all the identified structural problems may occur when a mapping between two object-oriented resp. EXPRESS schemas is specified. The additional difficulty of mapping schemas written in heterogeneous data models is to define the correspondence in a semantics-preserving way.

Based on these explorations, a language was needed to bridge the identified types of heterogeneity and to provide operational flexibility over the target schema that conforms to the semantics of the underlying sources. Related work concerning the development of such languages was found to be insufficient (cf., Sect. 4). We presented the mapping language BRIITY which is more powerful w. r. t. the number of problems being solved than the previous approaches. It was shown that the integration of multiple schemas written in heterogeneous data models can be supported. Further characteristics of the language are its descriptiveness, its technological independence, and its applicability to define transient views as well as mappings between DB schemas, i. e., to support data migration. Due to space limitations, the concatenation of mapping specifications (e. g., view$\leftrightarrow$DB$_1$ and DB$_1$$\leftrightarrow$DB$_2$), which permits to migrate applications to other DBs, was not considered in this paper. Another key feature of the language is its support of user-defined update statements having the same power and descriptiveness as retrieve statements. We only mentioned that the language can also be used to define updatable views solving the well-known view-update problems. However, for reasons of brevity, we refrained from comparing view-update problems and solutions provided by BRIITY. Finally, it is worth mentioning that our work is one of the major contributions to the ongoing standardization effort of EXPRESS-X [IS98b] which is a language to support the mapping between EXPRESS schemas.

Acknowledgements

We thank the referees for their helpful comments which led to essential clarifications and improvements of this paper.

8. References

AB91 S. Abiteboul, A.J. Bonner: Objects and Views. In: Clifford J, King R (eds) Proc. ACM SIGMOD, Denver, Col., SIGMOD Record 20(2), 1991, pp 238-247

An94 M. Andersson: Extracting an Entity-Relationship Schema from a Relational Database Through Reverse Engineering. In: Loucopoulos P (ed) Proc. 13th Int. Conf. on the Entity Relationship Approach, Manchester, UK, in: LNCS 881, Springer, Heidelberg, 1994, pp 403-419

At+89 M. Atkinson, et al.: The Object-Oriented Database System Manifesto. In: Kim W, Nicolas JM, Nishio S (eds) Proc. 1st Int. Conf. on Deductive and Object-Oriented Databases, Kyoto, Japan, 1989, North-Holland/Elsevier Science Publishers 1990, pp 223-240

AR90 R. Ahmed, A. Rafii: Relational Schema Mapping and Query Translation in Pegasus. Technical Report HPL-DTD-90-12, Hewlett-Packard Laboratories, 1990, Palo Alto, Calif.

Ba95 I. Bailey: EXPRESS-M Reference Manual. ISO Document of TC184/SC4/WG5 N243, 1995

BE96 O.A. Bukhres, A.K. Elmagarmid: Object-Oriented Multidatabase Systems. Prentice Hall, Englewood Cliffs, NJ, 1996

BCL89 J.A. Blakeley, N. Coburn, P.-Å. Larson: Updating Derived Relations: Detecting Irrelevant and Autonomously Computable Updates. ACM Trans Database Syst 14(3), 1989, pp 369-400

BCN91 C. Batini, S. Ceri, S. Navathe: Conceptual Database Design: An Entity-Relationship Approach. Benjamin Cummings. New York. 1991

BHP92 M.W. Bright, A.R. Hurson, S.H. Pakzad: A Taxonomy and Current Issues in Multidatabase Systems. IEEE Computer 25(3), 1992, pp 50-60

BHP94 M.W. Bright, A.R. Hurson, S. Pakzad: Automated Resolution of Semantic Heterogeneity in Multidatabases. ACM Trans Database Syst 19(2), 1994, pp 212-253

Bl97 J.A. Blakeley: Universal Data Access with OLE DB. Proc. IEEE Compcon'97, San Jose, IEEE Computer Society Press, 1997, Los Alamitos, Calif., pp 2-7

BLN86 C. Batini, M. Lenzerini, S.B. Navathe: A Comparative Analysis of Methodologies for Database Schema Integration. ACM Computing Surveys 18(4), 1986, pp 323-364

BSKW91 T. Barsalou, N. Siambela, A.M. Keller, G. Wiederhold: Updating Relational Databases through Object-based Views. In: Clifford J, King R (eds) Proc. ACM SIGMOD, Denver, Col., 1991, SIGMOD Record 20(2), pp 248-257

Ca96 R. Cattell (Ed.): The Object Database Standard: ODMG-93 (Release 1.2). Morgan Kaufmann, San Mateo, Ca., 1996

Ca+95 M. Carey, et al.: Towards Heterogeneous Multimedia Information Systems: The Garlic Approach. In: Buhkhres OA, Özsu MT, Shan MC (eds) Proc. 5th Int. Workshop on Research Issues in Data Engineering: Distributed Object Management, Taipei, Taiwan, 1995, IEEE Computer Society Press, Los Alamitos, Calif., pp 124-131

CL92 J. Chomicki, W. Litwin: Declarative Definition of Object-Oriented Multidatabase Mappings. in: Distributed Object Management, M.T. Özsu, U. Dayal, P. Valduriez (Eds.), Morgan Kaufmann, San Mateo, Ca., 1992, pp 375-392

CMS94 R.H.L. Chiang, T.M. Barron, V.C. Storey: Reverse Engineering of Relational Databases: Extraction of an EER Model form a Relational Database. Data & Knowledge Engineering 10(12), 1994, pp 107-142

CS91 A. Chatterjee, A. Segev: Data Manipulation in Heterogeneous Databases. ACM SIGMOD Record 20(4), 1991, pp 64-68

DKMRS93 P. Drew, R. King, D. McLeod, M. Rusinkiewicz, A. Silberschatz: Report of the Workshop on Semantic Heterogeneity and Interoperation in Multidatabase Systems. ACM SIGMOD Record 22(3), 1993, pp 47-56

Da83 U. Dayal: Processing Queries Over Generalization Hierarchies in a Multidatabase System. In: Schkolnick M, Thanos C (eds) Proc. 9th VLDB Conf., Florence, Italy, 1983, Morgan Kaufmann, San Mateo, Calif., pp 342-353

Du94 Y. Dupont: Resolving Fragmentation Conflicts in Schema Integration. In: Loucopoulos P (ed) Proc.13th Int. Conf. on the Entity-Relationship Approach, Manchester, UK, in: LNCS 881, Springer, 1994, pp 513-532

GLN92 W. Gotthard, P.C. Lockemann, A. Neufeld: System-guided View Integration for Object-oriented Databases. IEEE Transactions on Knowledge and Data Engineering 4(1), 1992, pp 1-22

GSC96 M. Garcia-Solaco, F. Saltor, M. Castellanos: Semantic Heterogeneity in Multidatabase Systems. in: O.A. Bukhres, A.K. Elmagarmid (Eds.): Object-Oriented Multidatabase Systems - A Solution for Advanced Applications, Prentice Hall, Englewood Cliffs, NJ, 1996, pp 129-202

Ha95 M. Hardwick: EXPRESS-V Language. Technical Report, Rensselaer Polytechnic Institute, Laboratory for Industrial Information Infrastructure, Troy, NY, 1995

HTJC93 J.-L. Hainaut, C. Tonneau, M. Joris, M. Chandelon: Transformation-based Database Reverse Engineering. In: Elmasri R, Kouramajian V, Thalheim B (eds) Proc. 12th Int. Conf. on Entity-Relationship Approach, Dallas, Tx., 1993, LNCS 823, Springer, Berlin Heidelberg New York, pp 364-374

IBM95 "DataJoiner: A Multidatabase Server - Version 1. White Paper, IBM, 1995, http://www.software.ibm.com/data/pubs/papers/

IS94a ISO 10303 - Industrial automation systems and integration - Product data representation and exchange - Part 1: Overview and fundamental principles. Int. Standard, 1st edition, 1994

IS94b ISO 10303 - Industrial automation systems and integration - Product data representation and exchange - Part 11: Description methods: The EXPRESS language reference manual. Int. Standard, 1st edition, 1994

IS96a ISO 10303 - Industrial automation systems and integration - Product data representation and exchange - Part 22: Standard Data Access Interface. ISO Document TC184/SC4/WG7, 1996

IS98a ISO 10303 - Industrial automation systems and integration - Product data representation and exchange - Part 214: Application Protocol: Core Data for Automotive Mechanical Design Processes, ISO Document TC184/SC4/WG3, 1998

IS98b ISO Document TC184/SC4/WG11 N052: EXPRESS-X Language Reference Manual. http://www.nist.gov/sc4/wg_qc/wg11/, 1998

KC95 J.-L. Koh, A.L.P. Chen: A Mapping Strategy for Querying Multiple Object Databases with a Global Object Schema. In: Buhkhres OA, Özsu MT, Shan MC (eds) Proc. IEEE Int. Workshop on Research Issues on Data Engineering - Distributed Object Management, 1995, IEEE Computer Society Press, Los Alamitos, Calif., pp 50-57

KCGS95 W. Kim, I. Choi, S. Gala, M. Scheevel: On Resolving Schematic Heterogeneity in Multidatabase Systems. in: W. Kim (Ed.): Modern Database Systems - The Object Model, Interoperability, and Beyond, Addison Wesley, Reading, Mass., 1995, pp 521-550

KDN90 M. Kaul, K. Drostern, E. Neuhold: ViewSystem: Integrating Heterogeneous Information Bases by Object-Oriented Views. Proc. 6th Int. Conf. on Data Engineering Los Angeles, Ca., 1990, IEEE Computer Society Press, Los Alamitos, Calif., pp 2-10

Ke91 W. Kent: Solving Domain Mismatch and Schema Mismatch Problems with an Object-Oriented Database Programming Language. In: Lohman GM, Sernadas A, Camps R (eds) Proc. 17th VLDB Conf. , Barcelona, Spain, 1991, Morgan Kaufmann, San Mateo, Calif., pp 147-158

KFMRN96 W. Klas, P. Fankhauser, P. Muth, T.C. Rakow, E.J. Neuhold: Database Integration Using the Open Object-Oriented Database System VODAK. in: O. Bukhres, A.K. Elmagarmid (Eds.): Object-oriented Multidatabase Systems: A Solution for Advanced Applications. Prentice Hall, Englewood Cliffs, NJ, 1996, Chapter 14

KLK91 R. Krishnamurhy, W. Litwin, W. Kent: Language Features for Interoperability of Databases with Schematic Discrepancies. In: Clifford J, King R (eds) Proc. ACM SIGMOD, Denver, Col., 1991, SIGMOD Record 20(2), pp 40-49

KS91 W. Kim, J. Seo: Classifying Schematic and Data Heterogeneity in Multidatabase Systems. IEEE Computer 24(12), 1991, pp 12-18

KS96 V. Kashyap, A.P. Sheth: Semantic and Schematic Similarities Between Database Objects: A Context-Based Approach. VLDB Journal 5(4), 1996, pp 276-304

La90 R. Langerak: View Updates in Relational Databases with an Independent Scheme. ACM Trans Database Syst 15(1), 1990, pp 40-66

LS91 J.A. Larson, A.P. Sheth: Updating relational views using knowledge at view definition and view update time. Information Systems 16(2), 1991, pp 145-168

Ma91 N.M. Mattos: An Approach to Knowledge Base Management. LNAI 513, Springer, Heidelberg, 1991

MR93 P. Missier, M. Rusinkiewicz: Extending a Multidatabase Manipulation Language to Resolve Schema and Data Conflicts. Technical Report UH-CS-93-10, Univ. Houston, Tx., 1993

PB94 W. J. Premerlani, M.R. Blaha: An Approach for Reverse Engineering of Relational Databases. Commun ACM 37(5), 1994, pp 42-49

PBE95 E. Pitoura, O. Bukhres, A. Elmagarmid: Object Orientation in Multidatabase Systems. ACM Computing Surveys 27(2), 1995, pp 141-195

PGU96 Y. Papakonstantinou, H. Garcia-Molina, J. Ullman: MedMaker: A Mediation System Based on Declarative Specifications. In: Su SYW (ed) Proc. 12th Int. Conf. on Data Engineering, New Orleans, Lou., 1996, IEEE Computer Society Press, Los Alamitos, Calif., pp 132-141

RS97 M. T. Roth, P. Schwarz: Don't Scrap It, Wrap It! A Wrapper Architecture for Legacy Data Sources. In: Jarke M, Carey MJ et al (eds) Proc. 23rd VLDB Conf. Athens, Greece, 1997, Morgan Kaufmann, San Mateo, Calif., pp 266-275

RMKDP95 N. Roussopoulos, C. Melvin Chen, S. Kelley, A. Delis, Y. Papakonstantinou: The ADMS Project: View R Us. Data Engineering Bulletin 18(2), 1995, pp 19-28

Sa96a G. Sauter: „The Mapping Language BRIITY - Reference Manual“, Technical Report F3-96-007, Daimler-Benz AG, Research & Technology, 1996

Sa96b G. Sauter: „Impacts of Heterogeneity on Interoperability“, Technical Report F3-96-021, Daimler-Benz AG, Research & Technology, 1996

Sa98 G. Sauter: Interoperability of Database Systems with Structural Heterogeneity (in German), Doctoral Thesis, Computer Science Dept., University of Kaiserslautern, Infix, St. Augustin, 1998

SGN93 A.P. Sheth, S.K. Gala, S.B. Navathe: On Automatic Reasoning for Schema Integration. Int. Journal of Intelligent and Cooperative Information Systems 2(1), 1993, pp 23-50

SL90 A. Sheth, J.A. Larson: Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. ACM Computing Surveys 22(3), 1990, pp 183-236

SPD92 S. Spaccapietra, C. Parent, Y. Dupont: Model Independent Assertions for Integration of Heterogeneous Schemas. VLDB Journal 1, 1992, pp 81-126

SSR94 E. Sciore, M. Siegel, A. Rosenthal: Using Semantic Values to Facilitate Interoperability Among Heterogeneous Information Systems. ACM Trans Database Syst 19(2),1994, pp 254-290

SST92 M.H. Scholl, H.J. Schek, M. Tresch: Object Algebra and Views for Multi-Objectbases. in: Distributed Object Management, M.T. Özsu, U. Dayal, P. Valduriez (Eds.), Morgan Kaufmann, San Mateo, Ca., 1992, pp 353-374

TC94 P.S.M. Tsai, A.L.P. Chen: Concept Hierarchies for Database Integration in a Multidatabase System. Proc. 6th Int. COMAD, Bangalore, India, 1994

VA95 M.W.W. Vermeer, P.A.G. Apers: Reverse Engineering of Relational Database Applications. In: Papazoglou MP (ed) Proc. Int. 14th Int. Conf. Object-Oriented and Entity-Relationship Modelling, Gold Coast, Australia, 1995, LNCS 1021, Springer, Berlin Heidelberg New York, pp 89-100

Wi95 J. Widom: Research Problems in Data Warehousing. Proc. 4th Int. Conf. on Information and Knowledge Management, Baltimore, MD, 1995, pp 25-30

ZHKF95 G. Zhou, R. Hull, R. King, and J.-C. Franchitti: Using Object Matching and Materialization to Integrate Heterogenous Databases. In: Laufmann S, Spaccapietra S, Yokoi T (eds) Proc. 3rd Int. Conf. on Cooperative Information Systems, Vienna, Austria, 1995, pp 4-18