Utilising the MISM Model Independent Schema Management Platform for Query Evaluation*

Cornelia Hedeler and Norman W. Paton

School of Computer Science, The University of Manchester
Oxford Road, Manchester M13 9PL, UK
{chedeler,norm}@cs.manchester.ac.uk

Abstract. Model Management, and its associated operators, provides generic means for dealing with multiple schemas and the mappings between them, for example, in the context of multiple heterogeneous data sources that need to be integrated. One example of a Model Management framework is the 'Model Independent Schema Management'(MISM) platform. In the context of MISM, algorithms and implementations of various operators have been proposed that act on a source-model independent metamodel. However, although the results on MISM indicate how to import and manipulate data from heterogeneous source types, to date no approach has been proposed to utilise MISM for querying across the multiple data sources. This paper presents SMql, a query language over the source-model independent supermodel, presents an algebra into which the query is translated and presents an approach for rewriting SMql queries into source-model-specific queries posed over the corresponding relational or XSD models of the data source to be queried. Thus this paper helps to complete the collection of problems that need to be addressed to allow source model-independent model management using universal models in the context of MISM.

Keywords: Model Management, Query Rewriting, Query Language.

1 Introduction

The vision of model management [5,6] was proposed to address the recurring issues that arise when dealing with data sources, whether multiple heterogeneous data sources that need to be integrated or a single data source with a continually evolving schema. In the case of multiple heterogeneous data sources, these could also be represented using different data models, e.g., relational or XSD. Model Management aims to provide generic operators that make it easier to manipulate schemas, that may be associated with, e.g., relational, object-relational or XML data sources, and the relationships between the schemas. With a view to obtaining data model independence, an approach has been proposed to represent models expressed in various different data models, within the same universal

* The work reported in this paper was supported by a grant from the EPSRC.

Table 1. Model-generic and model-specific constructs

Metaconstructs	Relational	XSD
Abstract		Root Element
Aggregation	Table	
StructOfAttributes		ComplexElement
Lexical	Column	Simple Element
Foreign Key	Foreign Key	Foreign Key

model, referred to as a supermodel. Utilising the benefits of data model independence, it has been suggested recently that dataspace management systems could utilise model management systems, e.g., for the integration of heterogeneous schemas or for dealing with evolving schemas [9]. This, however, requires model management systems to provide support for query evaluation across the various heterogeneous data sources.

A prominent example of a source-model independent approach is the 'Model Independent Schema Management'(MISM) [1,4] framework. In MISM, implementations of various of the model management operators have been proposed, e.g., an extended version of ModelGen [3], which in addition to translating schemas from a representation in one model into an equivalent representation in another model is also able to translate the corresponding data. This, however, required the whole database to be loaded, making it only suitable as an off-line approach. This was addressed for relational data sources in a later proposal [2] in which the data translation rules presented previously are translated into executable SQL statements. The framework was later extended further by proposing definitions and implementations for Diff and Merge [1] over the supermodel.

However, even though MISM now provides support for managing and integrating schemas represented in various models, it still does not provide support for querying across the various data sources associated with the integrated schemas, whereas other model management platforms have been extended to provide support for query answering (e.g., [17,13]). This paper addresses this gap by presenting an approach for processing queries in the MISM framework. To do this we define a query language (*SMql*) and an algebra over the MISM supermodel and present an approach to evaluating *SMql* queries over relational and XML sources.

The remainder of the paper is organised as follows. Section 2 introduces the relevant components of the supermodel of the MISM platform, and Section 3 introduces the query language over the supermodel and the algebra. Section 4 describes the approach for query rewriting. Related work is presented in Section 5 and Section 6 concludes the paper.

2 Background

This section introduces the two levels of schema descriptions of MISM as proposed by Atzeni *et al.* [1,4]. The two levels are the model-specific description, which contains all the constructs required to represent schemas in a particular model (see the two UML diagrams in the bottom half of Figure 1 for relational

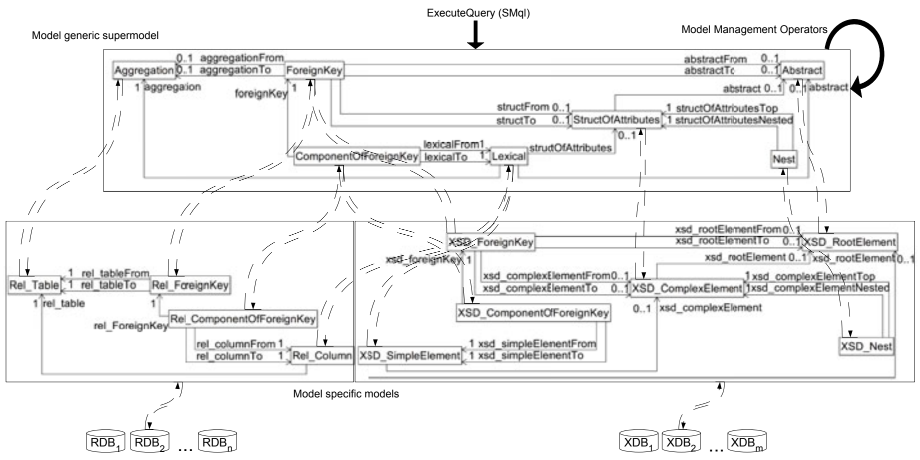
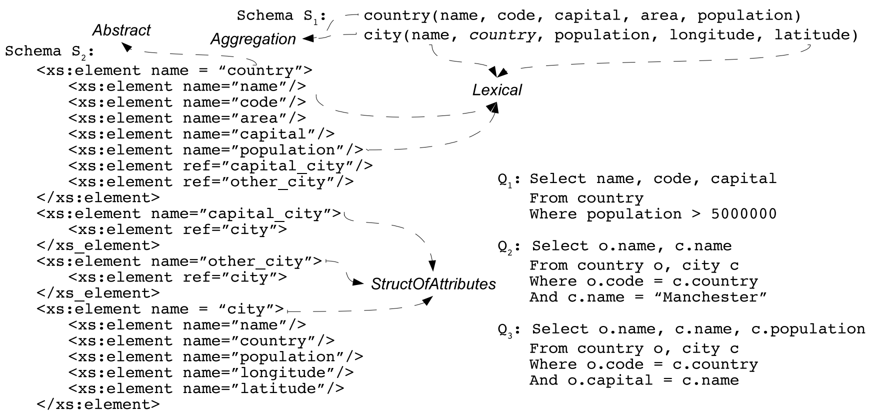


Fig. 1. Constructs in source-model independent supermodel and model specific models as well as the correspondences between them

and XSD, respectively) and the source-model independent supermodel, which uses a small set of model-generic constructs, so called *metaconstructs* [1] to represent model-specific constructs by aggregating over their similarities (see the UML diagram in the top half of Figure 1). Model management operators are defined over the constructs in the supermodel, which is depicted by the arrow in the top right corner of Figure 1. The UML diagrams in Figure 1 also include additional constructs that are required to represent all the information present in the models, e.g., `ComponentOfForeignKey` and `Nest`. By capturing both, the model-specific constructs and the model-generic representations of a schema, this approach is both model-independent and model-aware. Table 1 lists the model-generic metaconstructs and their corresponding model-specific constructs for relational and for XSD models. The dashed lines in Figure 1 from the model specific constructs to the model generic metaconstructs depict the corresponding constructs in the different models. The correspondences between the model specific and the model generic constructs are utilised during import of models, information that we later utilise in the opposite direction for query translation (depicted in Figure 1 by the dashed lines from the model generic constructs to the model specific constructs). As this paper focusses on relational and XSD models only, the remaining models that can be represented by the universal model have been omitted here, but are described in [4].

3 Query Language

This section introduces the query language *SMql*, a declarative query language inspired by SQL but defined over the constructs of the supermodel. A *SMql* query is of the form `SELECT` $l_1, ..., l_n$ `FROM` $c_1, ..., c_m$ `WHERE` p, where $l_1, ..., l_n$ is a project list of Lexicals, $c_1, ..., c_m \in \{Abstract|Aggregation|StructOfAttributes\}$



```
Schema S1:  country(name, code, capital, area, population)
            city(name, country, population, longitude, latitude)

Schema S2:
   <xs:element name = "country">
      <xs:element name="name"/>
      <xs:element name="code"/>
      <xs:element name="area"/>
      <xs:element name="capital"/>
      <xs:element name="population"/>
      <xs:element ref="capital_city"/>
      <xs:element ref="other_city"/>
   </xs:element>
   <xs:element name="capital_city">
      <xs:element ref="city">
   </xs_element>
   <xs:element name="other_city">
      <xs:element ref="city">
   </xs_element>
   <xs:element name = "city">
      <xs:element name="name"/>
      <xs:element name="country"/>
      <xs:element name="population"/>
      <xs:element name="longitude"/>
      <xs:element name="latitude"/>
   </xs:element>

Q1: Select name, code, capital
    From country
    Where population > 5000000

Q2: Select o.name, c.name
    From country o, city c
    Where o.code = c.country
    And c.name = "Manchester"

Q3: Select o.name, c.name, c.population
    From country o, city c
    Where o.code = c.country
    And o.capital = c.name
```

Fig. 2. Example schemas and queries in SMql

Table 2. SMql algebra

Operator
SCAN(Abstract|Aggregation|StructOfAttributes) → Collection
REDUCE(Collection, {Lexical}) → Collection
FILTER(Collection, Predicate) → Collection
JOIN(Left_Collection, Right_Collection, Predicate) → Collection
UNION(Left_Collection, Right_Collection) → Collection
EvaluateSQL(SQLqueryString, Predicate, {tuple})→ {tuple}
EvaluateXQuery(XqueryString, Predicate, {tuple})→ {tuple}

and p is a conjunctive predicate. Figure 2 shows two simplified schemas, S_1 of a relational data source and S_2 of an XSD data source and some example queries in *SMql*. The figure also shows the corresponding constructs in the supermodel for the two relational tables in S_1, as well as the root element (`country`) and the complex elements (`capital_city`, `other_city`, `city`) in S_2. The columns in S_1 and the simple elements in S_2 all correspond to *Lexical*, but for clarity not all those correspondences are shown in Figure 2.

SMql queries are translated into the algebra from Table 2 following standard translation schemes [10]. For example, query Q_1 in Figure 2 is translated into REDUCE(FILTER(SCAN(country), population > 5000000), {name, code, capital}) and query Q_2 is translated into REDUCE(JOIN(SCAN(country o), FILTER(SCAN(city c), c.name = 'Manchester'), o.code = c.country), {o.name, c.name}). The UNION operator will be used later in the context of query unfolding whereas EvaluateSQL and EvaluateXQuery will be used later in the context of evaluation of the rewritten source-specific subqueries (see Section 4 for an example).

4 Query Rewriting

This section introduces the approach to rewriting a *SMql* query posed over constructs of the supermodel and expressed in the algebra shown in Table 2 into potentially multiple SQL or XQuery queries, respectively, depending on the constructs which are queried and their respective sources. For example, if a *SMql* query is posed over a model that was generated using the model management operator Merge on two models, the query is expanded into a *SMql* query over constructs from potentially multiple source models using query unfolding [11] and associations between the source models and the merged model. The expanded *SMql* query is compiled into the algebra and optimised, and subqueries of the logically optimised plan that are associated with specific sources are then translated to the source specific query languages as described in Sections 4.1 and 4.2, whereby the left hand side input and the right hand side input of the UNION operator are treated as separate subqueries that are processed separately even if they are to be evaluated over the same source. The translated subqueries are passed to the operators EvaluateSQL and EvaluateXQuery, respectively, which can be parameterised with tuples and a Predicate, e.g., in the case of joins between different sources. Assume, for simplicity, that query Q_2 is posed over the merged schema of S_1 and S_2 (not shown) and that constructs with the same names in the three schemas are associated. The expanded algebra with the translated subqueries of Q_2 resulting from the use of these associations, query unfolding and the translation algorithms introduced in Sections 4.1 and 4.2 is shown in Figure 3.

Query rewriting is a two step process applied both for translating (portions of) a query into SQL and into XQuery. The first step is a recursive algorithm that traverses all operators in the query posed over the supermodel, and gathers all the information required for query rewriting in the corresponding data structures appropriate for the type of target query. In the second step, this information is utilised to generate a string representation of the target query. The process is described in more detail in the following.

The approach presented here only deals with syntax; dealing with semantics is beyond the scope of this paper. Once access has been provided to a source, the other model management operators provide techniques for manipulating the resulting integration model in ways that reflect semantic issues.

4.1 SMql Query over Supermodel into XQuery over XML

Based on the parts of an XQuery, namely the *Let*, *For*, *Where* and *Return* clauses, the following data structure is introduced to gather the information that is needed for each of the clauses.

An XQuery is a quadruple <let, for, where, return>, where *let* is a map of variable names and references to source documents, *for* is a list of abstract/root element or structOfAttributes/complex elements, *where* is a list of conjunctive predicates and *return* is a list of fully qualified lexical/simple element names, either qualified with the name of the abstract|structOfAttributes / root|complex element the lexical belongs to or with the corresponding variable name. Both lists and maps support the operators *add* and *contains*. We assume here that the data source, or the source document of each instance *i* of a construct in the supermodel can be obtained by *i.source*.

We follow the two step process described briefly above, which consists of gathering the information for the various clauses of the XQuery by recursively

```
UNION(
      UNION(
            EvaluateXQuery(
                  'for $o in $s/country
                  for $c in $o/capital_city/city
                  where $o/code = $c/country
                  and $c/name = "Manchester"',
                  null, null),
            EvaluateXQuery(
                  'for $o in $s/country
                  for $c in $o/other_city/city
                  where $o/code = $c/country
                  and $c/name = "Manchester"',
                  null, null)),
       EvaluateSQL(
            'Select o.name, c.name
            from country o, city c
            where o.code = c.country
            and c.name = "Manchester"',
            null, null))
```

Fig. 3. Expanded and rewritten query Q_2

```
Require: s = query (fragment) to be translated, expressed in algebra introduced above
 1: if s instance of SCAN(abstract c) | s instance of SCAN(structOfAttributes c) then
 2:     if !s.let.contains(c.source) then
 3:         s.let.add(c, c.source)
 4:     end if
 5:     s.for.add(c)
 6: else if s instance of REDUCE(Collection c, {Lexical}) then
 7:     for all Lexical l ∈ {Lexical} do
 8:         s.return.add(l)
 9:     end for
10:     Translate2XQuery(c)
11: else if s instance of FILTER(Collection c, Predicate) then
12:     s.where.add(p)
13:     Translate2XQuery(c)
14: else if s instance of JOIN(Left_Collection lc, Right_Collection rc, Predicate) then
15:     s.where.add(p)
16:     Translate2XQuery(lc)
17:     Translate2XQuery(rc)
18: end if
```

Fig. 4. Translate2XQuery(s)

traversing the *SMql*-algebra (Algorithm shown in Figure 4) followed by the generation of a string representation of the XQuery utilising the gathered information (Algorithm shown in Figure 5). The algorithms presented are not the only way to organise the information and generate the corresponding XQuery, as there are several ways of expressing the same XQuery. For example, an equivalent XQuery to query Q_1 in Figure 2 posed over S_2 could be written as one of the two versions v1 or v2:

```
v1:for $c in doc("...")//country[population>"5000000"]
v2:let $s := doc("...") for $c in $s//country where $c/population > 5000000
```

We have decided to follow the structuring of the information and consequently of the XQuery that is somewhat related to the structure of a SQL query, i.e., `for` corresponds to `from`, `return` corresponds to `select`, `where` corresponds to `where`, and to capture only the reference to the source document in `let`, which results in queries structured as exemplified in v2.

To generate an XQuery that reflects the relationship between the structure of the supermodel and the structure of the document, we order all the (root and complex) elements that are gathered in `s.for` according to their hierarchical structure in the source document (line 5 of the algorithm shown in Figure 5). We differentiate between rootElements and complexElements, which correspond to different constructs in the supermodel (lines 7-12) when generating the path expression in the for clause. We assume in Figure 5 that the (variable) name of a construct c queried can be obtained through $c.name$.

Using the algorithms presented, the XQueries posed over S_2 that correspond to the example queries Q_1, Q_2 and Q_3 are shown in Figure 6. The *let* and *return* clauses are omitted for XQueries Q_2 and Q_3. As there are two alternatives for mapping `city` in the integration schema to `city` in S_2, namely, `capital_city/city` or `other_city/city`, two subqueries are to be evaluated over the respective data source with schema S_2 and their results unioned. Both subqueries are shown in Figure 6.

4.2 SMql Query over Supermodel into SQL Query over Relational Model

Based on the three parts of an SQL query, namely the *Select*, *From* and *Where* clauses, the following data structure is introduced to gather the information that is needed for each of the three clauses.

A SQL query is a triple <select, from, where>, where *select* is a list of fully qualified lexical/column names, either qualified with the name of the aggregation/table the lexical belongs to or with the corresponding variable name, *from* is a list of aggregations/tables, and *where* is a list of conjunctive predicates. Lists support the operator *add*. As the correspondences between model specific constructs and model generic constructs are trivial for relational data sources, the rewriting algorithm is straightforward.

In the first step, a recursive algorithm (omitted here due to space constraints; for the corresponding algorithm for XQuery see Algorithm 4) traverses the *SMql* algebra and places the appropriate information into the data structures corresponding to each part of the SQL query, e.g., predicates of FILTER and JOIN operators are added to s.where, and all lexicals in REDUCE are added to s.select.

In the second step of the query rewriting, the gathered information is translated into a string for evaluation. As this step is straightforward for SQL, the algorithm is omitted here.

5 Related Work

Various contributions on query rewriting for data integration have been made over the years (e.g., [11,14]). In addition, some of the model management platforms have been extended to support query evaluation (e.g., Automed [7] and GeRoMe [12]).

GeRoMe, utilises a role based metamodel in which multiple model-independent roles can be attached to each model-specific schema element thereby specifying

```
Require: s = query (fragment) to be translated with all information gathered in s.let, s.for, s.where,
    and s.return
 1: String qs = new String("<result>")
 2: for all ⟨variableName v, document d⟩ ∈ s.let do
 3:     qs += "let $" + v + ":=doc(" + d + ")
 4: end for
 5: order all c's in s.for according to their hierarchical structure in the source document utilising
    the information captured in the supermodel
 6: for all element c ∈ ordered s.for starting from top do
 7:     if c instance of root element then
 8:         qs += " for $" + c.name + " in $" + v + "/" + c.name
 9:     else if s.c instance of complex element then
10:         qs += " for $" + c.name + " in $" + p.name + "/" + c.name
11:         where p = parent complex element of c
12:     end if
13: end for
14: if !s.where.isEmpty() then
15:     qs += " where "
16:     for all predicate p in s.where do
17:         if p of kind ⟨simple element l1⟩ ⟨op⟩ ⟨simple element l2⟩ then
18:             qs += "$" + c1.name + "/" + l1.name + op + "$" + c2.name + "/" + l2.name
19:             where c1 and c2 are the corresponding parent complex elements of simple elements l1
                and l2, respectively
20:         else if p of kind ⟨simple element l⟩ ⟨op⟩ ⟨constant⟩ then
21:             qs += "$" + c.name + "/" + l.name + op + constant
22:             where c is the corresponding parent complex element of simple element l
23:         end if
24:         if s.where.hasNext() then
25:             qs += " AND "
26:         end if
27:     end for
28: end if
29: qs += "return <tuple>"
30: for all simple element l in s.return do
31:     qs += "<" + c.name + "." + l.name + ">"
32:     qs += " fn:data($" + c.name + "/" + l.name + ")"
33:     qs += "< /" + c.name + "." + l.name + ">"
34:     where c is the corresponding parent complex element of simple element l
35: end for
36: qs += "</tuple>"
37: qs +="</result>"
38: return  queryString
```

Fig. 5. toXQueryString(s)

```
XQuery Q1:
<result>
let $s := doc("...")
for $o in $s/country
where $o/population > 5000000
return
  <tuple>
    <o.name>{fn:data($o/name)}</o.name>
    <o.code>{fn:data($o/code)}</o.code>
    <o.capital>{fn:data($o/capital)}</o.capital>
  </tuple>
</result>
```

```
XQuery Q2:
for $o in $s/country
for $c in $o/capital_city/city
where $o/code = $c/country
and $c/name = "Manchester"

for $o in $s/country
for $c in $o/other_city/city
where $o/code = $c/country
and $c/name = "Manchester"
```

```
XQuery Q3:
for $o in $s/country
for $c in $o/capital_city/city
where $o/code = $c/country
and $c/capital = $c/name

for $o in $s/country
for $c in $o/other_city/city
where $o/code = $c/country
and $c/capital = $c/name
```

Fig. 6. (Partial) XQueries corresponding to Q_1, Q_2 and Q_3

its properties in detail [12]. At the language level, GeRoMe uses source-to-target extensional mappings which are based on second-order tuple generating dependencies (SO tgs) [8] that are specified over the schema elements and their roles [13] to express the relationships between heterogeneous schemas represented using different data models. A conjunctive query posed over an integration schema expressed using the same formalism as the extensional mappings, is rewritten into a query over the sources using composition of the conjunction of source-to-target mappings between the source schemas and the integration schema and the query itself. The predicates of the resulting query, which is expressed over all the source schemas, are partitioned according to the corresponding sources and are then translated into the corresponding source-specific query language (SQL or XQuery) to be evaluated [13]. In contrast, rather than using composition we have illustrated an approach for expansion of a query posed over an integration schema using query unfolding [11] and presented in detail the rewriting of *SMql* (sub-) queries that are associated with specific sources into the source-specific query languages (SQL and XQuery).

In contrast, Automed utilises a lower-level hypergraph data model consisting of edges, nodes and constraints to represent schemas expressed in heterogeneous data models including XML [15]. Relationships between different schemas are expressed by a number of low level transformations between them, e.g., removing a node or an edge, that can be combined to form more complex transformations. The approach is called both as view (BAV) and the transformations are specified in such a way that they are reversible and that both local as view (LAV) and global as view (GAV) mappings can be derived between an integration schema and source schemas from the BAV transformations [16,7]. A query over an integration schema or any of the source schemas can be expressed in Automed's IQL query language, a comprehension-based functional query language, that is reformulated into a query over the (other) sources schemas using a combination of LAV and GAV query processing techniques over the BAV transformations [17]. However, no detail is provided on how the IQL query posed over the source schemas is rewritten into the source-specific query languages, which is the main contribution of our approach presented here.

6 Conclusions

Complementing the model management platform MISM we have presented *SMql*, a query language and its algebra over the MISM supermodel. We have illustrated an approach for expanding queries over multiple sources and presented an approach for rewriting *SMql* queries into the corresponding source specific queries posed over the sources to be queried. To add query rewriting capabilities for other data models for which MISM already provides support, such as, the object-relational model, the same approach as presented here for XSD and the relational model can be followed, i.e., gather the information according to the structure of the corresponding query language and use the information on the correspondences between the model-specific model and the source-model independent supermodel to create the specific target query. To include other models that are not yet supported by MISM, e.g. RDF, the MISM model will have to be extended first and then the approach described here followed.

References

1. Atzeni, P., Bellomarini, L., Bugiotti, F., Gianforme, G.: Mism: A platform for model-independent solutions to model management problems. J. Data Semantics 14, 133–161 (2009)
2. Atzeni, P., Bellomarini, L., Bugiotti, F., Gianforme, G.: A runtime approach to model-independent schema and data translation. In: EDBT, pp. 275–286 (2009)
3. Atzeni, P., Cappellari, P., Bernstein, P.A.: Model-independent schema and data translation. In: Ioannidis, Y., Scholl, M.H., Schmidt, J.W., Matthes, F., Hatzopoulos, M., Böhm, K., Kemper, A., Grust, T., Böhm, C. (eds.) EDBT 2006. LNCS, vol. 3896, pp. 368–385. Springer, Heidelberg (2006)
4. Atzeni, P., Gianforme, G., Cappellari, P.: A universal metamodel and its dictionary. T. Large-Scale Data- and Knowledge-Centered Systems 1, 38–62 (2009)
5. Bernstein, P.A., Halevy, A.Y., Pottinger, R.A.: A vision for management of complex models. SIGMOD Record 29(4), 55–63 (2000)
6. Bernstein, P.A., Melnik, S.: Model management 2.0: manipulating richer mappings. In: SIGMOD Conference, pp. 1–12 (2007)
7. Boyd, M., Kittivoravitkul, S., Lazanitis, C., McBrien, P., Rizopoulos, N.: Automed: A bav data integration system for heterogeneous data sources. In: Persson, A., Stirna, J. (eds.) CAiSE 2004. LNCS, vol. 3084, pp. 82–97. Springer, Heidelberg (2004)
8. Fagin, R., Kolaitis, P.G., Popa, L., Tan, W.C.: Composing schema mappings: Second-order dependencies to the rescue. ACM Trans. Database Syst. 30(4), 994–1055 (2005)
9. Franklin, M.J., Halevy, A.Y., Maier, D.: From databases to dataspaces: a new abstraction for information management. SIGMOD Record 34(4), 27–33 (2005)
10. Garcia-Molina, H., Ullman, J.D., Widom, J.: Database Systems The Complete Book, 2nd edn. Pearson International Edition, London (2009)
11. Halevy, A.Y.: Answering queries using views: A survey. The VLDB Journal 10(4), 270–294 (2001)
12. Kensche, D., Quix, C., Chatti, M.A., Jarke, M.: Gerome: A generic role based metamodel for model management. Journal on Data Semantics 8, 82–117 (2007)
13. Kensche, D., Quix, C., Li, X., Li, Y., Jarke, M.: Generic schema mappings for composition and query answering. Data & Knowledge Engineering (DKE) 68(7), 599–621 (2009)
14. Lenzerini, M.: Data integration: A theoretical perspective. In: PODS, pp. 233–246 (2002)
15. McBrien, P., Poulovassilis, A.: A semantic approach to integrating xml and structured data sources. In: Dittrich, K.R., Geppert, A., Norrie, M.C. (eds.) CAiSE 2001. LNCS, vol. 2068, pp. 330–345. Springer, Heidelberg (2001)
16. McBrien, P., Poulovassilis, A.: Data integration by bi-directional schema transformation rules. In: ICDE, pp. 227–238 (2003)
17. McBrien, P., Poulovassilis, A.: P2p query reformulation over both-as-view data transformation rules. In: Moro, G., Bergamaschi, S., Joseph, S., Morin, J.-H., Ouksel, A.M. (eds.) DBISP2P 2005 and DBISP2P 2006. LNCS, vol. 4125, pp. 310–322. Springer, Heidelberg (2007)