ON AUTOMATIC REASONING FOR SCHEMA INTEGRATION*

AMIT P. SHETH
Bellcore, BRC-1221A, 444 Bros Ave., Holmdel, NJ 07733, USA
amit@bellcore.com

SUNIT K. GALA
UnixSQL, Inc., 2521 Research Blvd., Austin, TX 78733, USA
sunit@unixsql.com

SHANKHANT B. NAVATHE
College of Computing, Georgia Institute of Technology, Atlanta, GA 30332-0280, USA
shank@ict.gatech.edu

Received 12 December 1992
Revised 8 April 1993

ABSTRACT

Success in database schema integration depends on the ability to capture real-world semantics of the schema objects, and to reason about the semantics. Earlier schema integration approaches mainly rely on heuristics and human reasoning. In this paper, we discuss an approach to automate a significant part of the schema integration process.

Our approach consists of three phases. An attribute hierarchy is generated in the first phase. This involves identifying relationships (equality, disjointness and inclusion) among attributes. We discuss a strategy based on user-specified semantic clustering. In the second phase, a classification algorithm based on the semantics of class antecedents is applied to the class definitions and the attribute hierarchy to automatically generate a class taxonomy. This class taxonomy represents a partially integrated schema. In the third phase, the user may employ a set of well-defined comparison operators in conjunction with a set of restructuring operators, to further modify the schema. These operators, as well as the automatic reasoning during the second phase are based on subsumption.

The formal semantics and automatic reasoning utilized in the second phase are based on a terminological logic as adopted in the CANDIDE data model. Classes are completely defined in terms of attributes and constraints. Our observation is that the inability to completely define attributes and class completely capture their real-world semantics imposes a fundamental limitation on the possibility of automatically reasoning about attribute definitions. This necessitates human reasoning during the first phase of the integration approach.

Key-words: Schema integration, attribute relationship, class taxonomy, formal semantics, human and automatic reasoning, classification, subsumption, CANDIDE, terminological logic.

*This research was supported by Bell Communications Research, Inc.
1. Introduction

Many organizations need to develop and maintain databases (either central or distributed) that provide shared and consistent access to the data by multi-applications and users. The process of schema or view integration is critical to such organizations because it can support: (1) the development of a large data-intensive application, (2) the development of multiple applications that share data, or (3) the integration of several existing (possibly distributed) heterogeneous and autonomous databases.

There is a body of work in schema integration (5, 9, 43) which views schema integration as a well-defined task of synthesizing dependencies (e.g., functional, class, and exclusion) from underlying domains. We believe that the integration process requires information about dependencies that go well beyond such dependencies. Some of the reasons why a more complete understanding of dependencies are important is [38]: (1) semantic (or other) data models are needed to sufficiently capture the semantics of real world objects; (2) their meaning and use; (2) it is necessary to capture more meta-data (directory/database) information about the modeled objects that is typically captured in schema, as well as information about intended use of the integrated schema; and (3) there can be multiple views or interpretations of a given application domain which may change with time. This schema integration involves a subjective task which relies heavily on the knowledge and intuition of the user about the application domain (domain of discourse), intended use of the integrated schema, and a system that manages data modeled by the schema. Thus, the results of a schema integration activity are not unique and cannot be generated totally automatically.

An important goal of our work is to automate the schema integration process as much as possible. In this paper, we discuss our views with respect to the use of the schema integration process that can be automated and the tasks that need to be performed primarily upon human guidance. It is to be noted that substantial human work is needed before automatic tasks can be performed. The automatic tasks are facilitated by a formal semantics and the human tasks are to be specified and monitored by the user. We also discuss appropriate values for performing the bisection. Specifically, the main contributions of this paper are:

(1) identification of the portion of the schema integration process that can be meaningfully automated,
(2) use of the formal model CANIDE [4] which provides subsumption and classification as a vehicle for automated class integration, and
(3) the development of the overall framework based on a three-step approach that involves human intervention (that is, heuristic reasoning) where necessary at the use of automatic reasoning when possible.

*The first two sections are often called view integration and the third one is often called database integration. They are jointly referred to as schema integration in [4].

*By user, we mean the person who is integrating the database schemas.
This paper is organized as follows. Section 2 provides the background — it discusses relevant previous work on schema integration, the role and choice of a canonical data model for schema representation, and introduces the problem of schema analysis. In our approach, the schema integration process consists of three phases, namely, schema analysis, class integration, and schema restructuring. In Section 3, we discuss schema analysis which results in the specification of attribute relationships (equivalent, containing/contained-in, and disjoint). An attribute hierarchy can then be automatically created using attribute relationships. We comment on the inadequacy of the attribute equivalence theory proposed in [20] and introduce the notion of semantic clustering of attributes. The semantic clustering is a heuristic process that is largely driven by the user. Yu et al. [22] and Sciore et al. [27] investigated the generation of a concept hierarchy that can aid in determining attribute relationships.

Using the attribute hierarchy thus generated, an automated reasoning process (called classification) automatically performs class integration. This constitutes the second phase, called class integration, and is discussed in Sec. 4. The third phase called schema restructuring phase is discussed in Sec. 5, in which we define operators to modify the integrated schema to make up for the limitations of the earlier phases. We thus derive a final, integrated schema. Section 6 gives an example and we give our conclusions in Sec. 7.

2. Background

In this section, we briefly review related work, comment on the common model in which automatic class integration is performed, and introduce our approach.

2.1. Previous Approaches to Schema Integration

The survey paper by Bainu et al. [2] discusses and compares 12 methodologies for schema integration work. These methodologies are oriented towards integration that involve only human reasoning (i.e., all relationships among schema objects are identified by the user). The portion of the integration process that can be automatically (and meaningfully) performed in most methodologies purely on the basis of schema definitions is severely limited. Two distinct types of operations are performed during integration:

(i) specifying relationships between classes based on the user's understanding of the domain; and

(ii) specifying mappings between the integrated schema and the component schemas (i.e., the schemas that are integrated).

We assume that all schemas to be integrated are represented in the same data model, which in our case happens to be CANAIDE. The approach has been extended to integrate E-R schemas by developing mappings between Extended E-R models and CANAIDE models [23].
The second type of operations are not strictly necessary to generate an integrated schema, but are very useful if the integrated schema is to be used for multi-database access. Such mappings are not discussed in this paper.

Bason et al. [2] divide schema integration into five steps: preintegration, comparison, conforming, merging, and restructuring. The preintegration step involves translation of schemas into a common data model so that they can be compared. Additionally, it may also include specification of global constraints and defining a catalog of naming correspondences (e.g., specifying a Measure that may be used for identifying naming conflicts, homonyms, synonyms, etc.).

The comparison activity involves analyzing and comparing various aspects of the component schemas, including identification of naming conflicts (e.g., homonyms and synonym detection), domain (i.e., value type) conflicts, differences in structure and constraints. Conforming activities are closely tied to the comparing activities since the comparing activities lead to an identification of what needs to be conformed. Conformity is not meaningful unless the same information or concept is represented in a similar way in different schemas. Naming conflicts are usually resolved by renaming. Structural differences are dealt with by the conforming activities in which some structures are transformed into others so that the same structure is equivalently represented (i.e., having the same semantics). For example, [4] and [32] show how to translate one E-R structure into another. Spector et al. [41] also allow specification of assertions between the objects that are structurally different.

One significant result of schema analysis is an attribute equivalence specification. In the previous methodologies [2,13,20] as well as tools [11,16,37], attribute equivalence has been used as a heuristic in the process of specifying low classes are related in the component schemas. User specification of class relationships may lead to the merging activity, which can be automated (e.g., lattice generation in [37]).

We recognize attribute relationship specification and class relationship specification (called assertion specification in [13]) to be the most critical tasks in the integration of schemas. Attribute relationships include attribute equivalences as well as attribute inclusions (also called containment). Most earlier work was limited to using only attribute equivalences. Attribute equivalence is an attempt to identify different attributes that represent the same property across related classes. All previous efforts that use attribute equivalence, to our knowledge, require the user to identify the equivalent attributes, or propose heuristics (e.g., [11,16]).

Tsamak et al. [13] discuss using attribute equivalence as a heuristic to order pairs of classes that may be related. The user then specifies all relationships between each pair of related classes. These relationships are used as an input to a lattice generation or merging activity to derive a single integrated schema. Some of the efforts, particularly those that do not use class relationships for automatic merging, provide a set of operators for the user to correlate or integrate the classes in the component schemas to generate an integrated schema [6,21].

Footnote: Later work by [19] and [22] consider the equal and containment relationships among domains of the attributes.
2.2. Choice of a Canonical Data Model

The data model into which all schemes are translated is called a common (or unified or canonical) data model (CDM). If a schema is not in the CDM, it should be translated from its native data model into the CDM. This process is called schema translation. For example, to integrate a CODASYL database schema and a relational database schema, the corresponding CODASYL and relational schemas should be translated into the equivalent CDM representation to facilitate their analysis and merging [18]. Typically, schema translation is treated as an entirely separate process from that of schema integration. Due to the lack of formal semantics of the data models usually involved, equivalence can only be informally justified in terms of the knowledge of a person doing the translation, and cannot be formally proved.

The choice of the CDM is crucial in the process of schema integration. The CDM should be semantically rich, in that it should be able to provide constructs, abstractions, and constraints relevant to schema integration. Semantic (or conceptual) data models are much preferred over a traditional data model such as relational, network, or hierarchical. However, choosing the semantic data model as the CDM from among many known models (e.g., those surveyed in [35,31]) is difficult. We believe that a model that has too many non-orthogonal constructs or abstractions may have benefits when defining or designing schemas (by providing constructs to the schema designer that she is familiar with), but is not well suited for schema integration because it makes schema analysis and modification very difficult. Automating any part of the schema integration process would be facilitated by a model that is formal, provides orthogonality of constructs leading to a basis for automatic reasoning.

2.3. Proposed Approach

Here we give an overview of three phases in our schema integration approach: schema analysis, class integration, and schema restructuring. While these phases can be found in many previous approaches, our framework significantly differs in details. All schemes are assumed to be represented in a canonical data model (to be defined in Sec. 4). Each component schema contains a class taxonomy and an attribute hierarchy (see Sec. 5). The integration process involves generating an attribute hierarchy common to all component schemes and subsequently a class taxonomy involving all classes of the component schemas. This approach has been implemented in a tool developed at Bellcore [33,30].

23.1. Schema analysis

The main result of the schema analysis phase is the generation of a global attribute hierarchy. This global attribute hierarchy is generated by integrating in-

[Excerpts have been formally defined in the case of schema restructuring within a well defined mode [35]].
individual attribute hierarchies corresponding to each component schema. Sections 3.2 discussed a methodology based on a clustering technique to generate the atribute hierarchy. In [12,23,37], schema analysis results in the specification of a of pairwise attribute equivalences. In our approach, instead, an attribute hierarchy is produced which, in addition to equivalence, also represents disjointness as an attribute relationships between attributes. The conforming phase involves the merging of the relations [6] which is said to occur when the same concept is represented using different model constructs (e.g., a city and an attribute represent the same thing). This is a critical issue as when necessary, we resort to the proposals for transformations among different constructs [e.g., 1,36]. As mentioned earlier, the task of conforming is very strong related to that of comparing because comparing is difficult when the representation [constructs] do not conform. Hence, we regard both of them as belonging to the schema analysis phase.

2.3.2. Class integration

In previous efforts, all relationships among classes were supplied by the user, although algorithms for simple graph manipulation to combine class hierarchies have been given (e.g., lattice generation in [12,37]). On the other hand, in our approach, identification of class relationships and the subsequent class integration process is completely automated. This is done by classifying every class from each component schema into a simple class taxonomy. The global attribute hierarchy generated in the schema analysis phase is used to perform the classification. The resulting class taxonomy represents a partially integrated schema. The approach in [13] relies on a classification of attributes into equivalence classes followed by a specification of pairwise relationships between related objects. These relationships are equal, contains, contains, overlap and disjoint-but-integrate. In our approach the last three relationship types are deduced (and the schema duty restructuring) by the automatic reasoner while the last two are deferred to the latter phase of schema restructuring (even though the system can declare overlap and disjoint — see Sec. 5). It is important to note that while use of attribute equivalent in [13,37] is only as a heuristic to facilitate the object class assertion specification in our approach, the information contained in the attribute hierarchy is fully exploited for merging various schemas (see Sec. 4.3).

2.3.3. Schema restructuring

In some of the previous efforts, merging and restructuring have not been treated separately [see Sec. 2.8 in [2]]. However, use of an automatic reasoner necessarily makes these tasks distinct. The case of disjoint-but-integrate mentioned above can be safely specified only by the user in the schema restructuring phase. Alternatively, this relationship may be considered during the schema analysis phase while creating the attribute hierarchy. Since we believe that semantics cannot be
completely represented and automatically reasoned about [23,36], we provide a set of well-defined operations to allow the user to make any changes to the partially integrated schema. The class integration phase merges the component schemas into a single schema. The schema restructuring phase involves user-driven modification of the integrated schema with a well-defined set of operations defined in Sec. 5.

3. On Attribute Relationships

Here we show that the attribute descriptors as defined in the schema integration literature are only a partial list of characteristics and therefore inadequate to establish equivalence. We next describe our methodology to generate an attribute hierarchy.

3.1. Comments on Attribute Equivalence in Previous Works

Use of attribute equivalence for schema integration has been discussed in [22,29]. In [29], an attribute is defined in terms of the following descriptors (i.e. meta-data information): Uniqueness, Lower Bound, Upper Bound, Domain, Static Semantic Integrity Constraints, Dynamic Semantic Integrity Constraints, Security Constraints, Allowable Operations, and Scale. Furthermore, they tie the attribute to a class (entity type or relationship type) definition. The attribute equivalence theory proposed by them essentially states that if there exists a certain mapping function between the domains (i.e. the value set) of two attributes, then the two attributes are said to be equivalent. They further define different types of equivalence based on whether the mapping is an isomorphism and whether the mappings hold at the current time or for all values of time. However, these details are inconsequential to our discussion. We now observe that this attribute equivalence definition is incomplete and inadequate. First, the particular set of descriptors used to define an attribute are arbitrary. That is, we can always add some more descriptors or choose a different set of descriptors. These descriptors are merely relevant to the information that is traditionally stored in a schema. Second, the descriptors fail to capture the semantics of an attribute. For example, consider two attributes person.name (i.e. attribute name belonging to an entity type person) and department.name. We may be able to define a mapping (possibly isomorphic) between the domains of these two attributes, and thereby, according to [29], declare them equivalent. Unfortunately, this is contrary to the intended semantics of the two attributes. Furthermore, according to [29], the user has to define the mappings. We do not dispute the utility of defining the domain mappings (they must be defined in order to allow multibase database access from an integrated schema), but the existence of a mapping does not imply attribute equivalence in the semantic sense. Furthermore, we do not know of a particular set of descriptors with which to compose attribute relationships. At best, given a set of descriptors, one can define heuristics that may help in discovering attribute equivalences. However, such heuristics may either make incorrect equivalences or fail to detect all meaningful equivalences. The issue of semantic and semantic relationships has been discussed in more detail in [36].
3.2. Attribute Hierarchy Generation

The purpose of data modeling is to create an isomorphic mapping between the intended real world semantics of a given universe and its symbolic representation on a machine. It is this isomorphic mapping which allows us to manipulate real world concepts (or objects) on a machine in a meaningful manner. As mentioned earlier, within the framework of schema integration, we are interested in identifying relationships among attributes and among classes that lead to the creation of an integrated schema consisting of a taxonomy of classes. In our approach, we first establish attribute relationships, which then help us in automatically computing class relationships. Note that this step is optional, that is, the class integration algorithm (to be described in Sec. 4) does not require an attribute hierarchy as mandatory input. The attribute relationships of interest are equivalences and inclusions. That is, for two attributes, \( a \) and \( b \), one of the following hold:

1. \( a \) and \( b \) are equivalent, or \( a \equiv b \).
2. \( a \) contains (is-contained-in) \( b \), or \( a \subset (\subset) b \).
3. If neither of the above, then \( a \) and \( b \) are disjoint.

Our attribute hierarchy is a strict partial ordering of inclusion relationships. Given the relationship between every pair of attributes, an attribute hierarchy can easily be generated [22]. We now propose heuristics for reasoning about attributes as follows.

We limit our universe to:

1. assertions about attribute definitions, and their relationships, and
2. assertions about object definitions, and their relationships.

Let \( RWUS(U) \) denote the intended world semantics for some universe of discourse \( U \). Further, let \( RWUS(a) \) denote the real world semantics of an attribute \( a \) and \( NWUS(c) \) that of a class \( c \) in \( U \). We can now define a semantic space for the \( U \) in which schemes are being integrated as follows:

\[
RWUS(U) = (NWUS(c_1)) \cup (RWUS(a_i)), 1 \leq i \leq n, 1 \leq j \leq n
\]

where \( n \) is the total number of classes and attributes in \( U \), respectively. We assume that class definitions are denoted by single-place predicates \( P(x) \) and attributes are denoted by two-place predicates \( U(x,y) \). Here, we define a framework:

\[\text{This is a restriction imposed by our canonical data model. It further implies that it is not possible to model overlapping attributes.}\]

\[\text{The semantic space is seen as an abstraction of the semantics in the user's mind.}\]

\[\text{This is a reasonable assumption since one can always translate object definitions in most structured representation schemes (such as semantic data models, semantic networks and frame description languages) into equivalent logic statements which consist of only single- and two-place predicates. For example, Parents0(Man) = Parents0 in the sense as (Parents0(Men) \subseteq (Parents0), and means that for all values of \( x \) in the universe satisfying the predicate Parents0(Men), those same values must also satisfy the predicate Parents0. A similar argument is made for attributes, an example of which is Sex(x, y) = Chick(x, y).}\]
within which we can formalize a theory for attribute relationships. A given universe is dichotomized into two subspaces in this manner so that we can first reason about attributes and then about classes. As argued in Sec. 3.3, a complete specification of the real world semantics for attributes is not possible. This imposes a fundamental limitation and therefore, we describe some heuristics below to define an attribute hierarchy. On the other hand, it is possible to define a limited model for classes (or entity types) in which we automatically determine class relationships (see Sec. 4).

The problem of attribute equivalence can be broken into two subproblems:

(1) mapping from the space of attribute labels and their characteristics to its corresponding semantic space \( \mathbb{R} \) (i.e. establishing a correspondence), and

(2) generating an attribute hierarchy in the semantic space.

By looking at Fig. 1, we can see that the links between the attribute definitions space \( \mathcal{D} \{ \mathcal{Q}(x, y) \} \) and the semantic space \( \mathbb{R} \) \( \{ \mathcal{Q}(x, y) \} \) denote the mappings

\[
\begin{align*}
\mathcal{f} : & \mathcal{Q}(x, y) \rightarrow \mathbb{R} \cdot \mathcal{Q}(x, y), \\
\mathcal{g} : & \mathbb{R} \cdot \mathcal{Q}(x, y) \rightarrow \mathcal{Q}(x, y)
\end{align*}
\]

It can be easily seen that the mapping \( \mathcal{f} \) is 1-1 whereas the inverse link \( \mathcal{g} \) is 1-n. Consider a point \( \mathcal{Q}(x, y) \) in the semantic space. Corresponding to this point we can have many points \( \mathcal{Q}(x, y) \) in the attribute definition space. For example, we see that the attribute labels \( \text{emp.name} \) and \( \text{work.name} \) and \( \text{name} \) map onto the same point in the semantic space. This necessarily means that the three attributes in question are equivalent in \( \mathcal{V} \). Additionally, there are other attributes \( \text{person.name} \) and \( \text{student.name} \) which map onto two other distinct points in the semantic space. Similarly, \( \text{emp.name} \) and \( \text{student.name} \) map onto two distinct semantic points which \( \text{dept.name} \) maps onto yet another semantic point.

We now make the following observations:

- Since \( \text{emp.name} \), \( \text{work.name} \), and \( \text{name} \) map onto the same semantic point \( \mathcal{Q}_1 \), they are semantically equivalent.
- It is also the case in our universe that \( \text{person.name} \) and \( \text{student.name} \) which map onto \( \mathcal{Q}_2 \) and \( \mathcal{Q}_3 \) respectively, are related to \( \mathcal{Q}_1 \) and are therefore located close to each other and to \( \mathcal{Q}_1 \) in the semantic space.
- \( \mathcal{Q}_2 \) and \( \mathcal{Q}_3 \) map onto \( \mathcal{Q}_2 \) and \( \mathcal{Q}_3 \) respectively, and their distance from \( \mathcal{Q}_1 \) is greater than that between \( \mathcal{Q}_2 \) and \( \mathcal{Q}_3 \).
- On the other hand, \( \text{dept.name} \) maps onto the semantic point \( \mathcal{Q}_9 \) which is disjoint from \( \mathcal{Q}_1 \).

The above suggests a certain classification pattern based upon equivalence and inclusion relationships between attributes as well as their relative meaning. Thus, we can define the following clusters in our semantic space, called semantic clusters.
Fig. 1. Mapping attributes into semantic space.

1. \( S_1 = \{G_1, G_2, G_3\} \)
   The semantic cluster denoted by \( S_1 \), labeled as people_names for convenience, includes \( \text{emp.name, person.name, and student.name} \) (the set is non-exhaustive and mutually exclusive). However, it may also be the case that \( G_1 \subset G_2 \) and \( G_2 \subset G_3 \). This is equivalent to saying that \( \text{RWAS(emp.name)} \subset \text{RWAS(person.name)} \) and \( \text{RWAS(student.name)} \subset \text{RWAS(person.name)} \). Thus, there may be a strict ordering within a given semantic cluster. However, any two semantic clusters are necessarily disjoint. This hierarchy is shown in Fig. 2.

2. \( S_2 = \{G_1, G_2\} \)
   The cluster \( S_2 \) labeled people_numbers contains, among other semantic points, \( \text{emp.num} \) and \( \text{stu.num} \).

3. \( S_3 = \{G_2\} \)

4. \( S_4 = S_1 \cup S_2 \)
   This is a semantic cluster labeled people_identifiers, which includes, among other things, the clusters people_names and people_numbers. Again, this cluster is a non-exhaustive and mutually exclusive union of two semantic subsets.
We now propose the following methodology to deal with attribute equivalence and inclusion:

1. Make a list of all attribute names appearing in all the schemas to be integrated.
2. The user (a domain expert or DBA) must now identify various appropriate semantic clusters (this can be done in advance or dynamically when considering a new attribute not belonging to a previously defined semantic cluster).
3. All sets of equivalent attributes should be determined and each set must be denoted by a unique NVAS.
4. All remaining attributes are then associated with their respective unique tables.
5. Each NVAS is placed in exactly one semantic cluster based on the judgements and intuitions of the user.
6. Now that each semantic cluster contains a set of unique NVASs, the user can specify piece-wise information about inclusion between the NVASs within the same semantic cluster (recall that any two semantic clusters are necessarily disjoint), which is then assimilated into a hierarchically ordered hierarchy.

The global attribute hierarchy thus generated can be used to merge the different schemas.

3.3. An Underlying Assumption

We identify two different kinds of attribute relationships. Namely, those that hold true throughout the universe of discourse, and those that hold true only within a given class. For example, consider the class of persons with attributes son and business-partner. We observe two facts about this universe:

- It makes sense to say that any value in our universe that satisfies the attribute son (regardless of what class it is attached to) also satisfies the attribute child (regardless of the class to which it is attached), i.e., son ⊆ child.
A similar relationship cannot hold between nond and business-partners because their relationship is merely a constraint imposed on them in order to define the class in question. The relationship nonC business-partners does not hold outside the class parent-child-and-are-business-partners.

Such a distinction is not made in previous works on attribute equivalences. Thus, the attribute relationships (in terms of equivalence or inclusion) should be determined such that they are used throughout the domain of discourse. This is necessary since the classes of the component schemes are merged against a single global attribute hierarchy. Therefore, attribute relationships within this hierarchy must hold throughout the universe of discourse. While an attribute's association in the class in question in the component scheme may provide part of the (and in some cases significant) semantics required to determine a relationship with another attribute it is neither the only nor the overriding factor. For example, when considering attribute age in class Employee and attribute age in class Engineer, we may wish to refer to the semantics of age in persons, humans, or living organisms (regardless of whether corresponding classes appear in the original scheme) and limit the semantics of the attributes to the classes that they help define in the component schema. Attribute relationships that hold only within a single class definition (e.g., car make and car model option) have no place in the global attribute hierarchy (and should be dealt with in the class integration phase).2

4. Class Integration

This is the second phase of our approach to schema integration, in which various class relationships are automatically deduced. We now describe CANDIDE, our canonical data model, in Sec. 4.1. Then we give an example of class definitions, and show the role played by the attribute hierarchy on computing subsumption in Sec. 4.2. In Sec. 4.3, we discuss why CANDIDE was chosen as the canonical data model.

4.1. Overview of the CANDIDE Data Model

CANDIDE belongs to the KL-ONE [7] family of systems, also known as terminological logic or term subsumption language. It is essentially an extension of the KANDOR knowledge representation system [30]. This extension involved using a notation that conforms to the more common data modeling terms and adding constructs to simplify representation of standard data domains. For example, we have added the domain constructors range, set, and composite domain, and allowed for additional built-in domains such as real. These do not really increase the power of the model or affect computational complexity, but they improve the specifiability of the model. CANDIDE is a correct and complete model of computation and

2Such constraints can be expressed in a system as expressive as KL-ONE [7] (in which computing subsumption is undecidable), but not in the CANDIDE system (in which computing subsumption is decidable and at least NP-hard).
the problem of computing subsumption in CANIDE is at least co-NP-hard [25].
CANIDE can be viewed as an orthogonal, minimal, and relativistic model [25]. A
complete description of CANIDE can be found in [3, 4]. It was employed in the
Linked Interfaced Information Base project as the canonical model [25]. Here we provide
a description of some of the concepts, particularly those that are relevant to class
integration.

4.1.1. Structural aspects of the data model

The CANIDE model explicitly supports the abstractions of aggregation and
generalization. A relationship or abstraction between two or more classes is modeled
as a class which is the aggregation of attributes (attaching each member class of the
relationship), in addition to other possible attributes which the relationship
may have.

The database schema consists of two partial orders one for the attribute hierarchy
and the other for the class taxonomy. In the attribute hierarchy, which has
a root called “Top,” each attribute can have at most one parent attribute along
with an associated domain. This domain is either an instance, or a set of instances
specified by a class defined in the class taxonomy. The domain of an attribute must
be an improper subclass of the domain of its parent attribute. The root of the class
taxonomy is a universal class called “Thing”.

An attribute appearing in a class description can be qualified by additional
value constraints on its domain specified using the class description. Further,
these attribute constraints in a class must logically imply the constraints on each
corresponding attribute of each of its superclasses. Such constraints also specify
validity requirements for instances to be members of the class. A class can
have many superclasses, subclasses, and instances. A disjoint class means that the
named subclasses of a given class cannot have any common instance.

4.1.2. Constraint specification

A class description comprises its superclasses, subclasses, instances, and at-
tributes. One can specify additional constraints on these attributes. There are four
types of constraints: max, some, exactly, and all. The max n constraint means
that an attribute can have at most n values. The some n constraint means that
there exist at least n values, each value belonging to a certain domain qualified by
value constraints. The exactly n constraint means that exactly a specified number
of attributes must satisfy a value constraint; it is the combination of some n and
max n. The all constraint specifies that all values of an attribute must belong to
a given domain qualified by value constraints. Note the similarity of the all and
some constraints to the universal and existential quantifiers of first-order predicate
calculus [19].

Value constraints on attributes specify domains. Domains may be specified by
ranging the class as in class (classname, string, integer or real, or by using the
domain constructors range, set, setdif, and composite. Range specifies a range of values between some upper and lower bounds which may be inclusive, exclusive, or nil (± infinity), and is typed over real and integer. The set construct allows a set of domains to be specified such that the attribute values must belong to the union of these domains. A set may recursively include other value constraints. setdif allows a special form of negation (set difference) to be handled in a safe way. For example, "setdif f g", where f is a class and g is its subclass, denotes a set consisting of only those instances belonging to f and not to g. The composite domain is the aggregation of other (possibly complex) domains, in which each component domain is labeled by an attribute name along with its constraints. For example, a composite domain for an attribute called Date would have component attributes of Month, Day, and Year.

4.1.3. Classification and Subsumption

A class f subsumes a class g if and only if every instance of g is also an instance of f, i.e., f is a superclass of g. This subsumption relationship is computed on the basis of whether the attribute constraints for class g logically imply the corresponding attribute constraints for class f. Computing subsumption can be automated because a class definition provides necessary and sufficient conditions for deciding class membership. The classification operation can recompute missing class-subclass relationships that is, those left unspecified by the user when building the class taxonomy, by a controlled application of the subsumption function, and can completely specify the class taxonomy.

Classifications can be viewed as the process of correctly locating a new class in an existing taxonomy. The correct location is immediately below the most specific superclass which subsumes the new class and immediately above the most general class subsumed by this new class. One simple way of generating a class taxonomy is to take the transitive reduction over a Boolean matrix generated by applying subsumption between all possible pairs of classes in the schema. Since there are \(n^2\) such pairs, classification is an \(O(n^2)\) algorithm where the fundamental unit of computation is the subsumption operation.

4.2 Role of the Attribute Hierarchy in Computing Subsumption

To see how subsumption can be computed in CANDIDE, consider an example of two classes, where \(c_1\) is defined to have an attribute sex with the domain man, and \(c_2\) is defined to have an attribute sex with the domain person. Suppose that we are also given that men is beneath persons in the class taxonomy and sex is beneath child in the attribute hierarchy. These classes are defined in CANDIDE as follows:

- class c1 defined
- superclass thing
- attributes
  - sex: all class men
class \( c \) defined

superclass thing

attributes

\[
\text{son, all class person}
\]

Therefore, \( c \) is above \( c_1 \) in the class taxonomy when subclass \((c_2, c) = \text{true}\). This happens if and only if the logical constraints on \( c_2 \) — logical constraints on \( c_1 \), i.e.

\[
\text{child, all class men} \rightarrow \text{son, all class person}
\]

This can be derived as follows from a set of inference rules that define the composition of subsumption. From these rules, we know that

\[
\text{if attribute } x \text{ is above } x \text{ then } x; \text{ all } f \rightarrow x; \text{ all } f.
\]

Since child is above \( \text{son} \) in the attribute hierarchy, applying this rule to our example tells us that

\[
\text{child, all class men} \rightarrow \text{son, all class men}
\]

There is another inference rule which says that

\[
\text{if subclass}(c, f) = \text{true} \text{ then } x; \text{ all } f \rightarrow x; \text{ all } f.
\]

Since person is above \( \text{man} \) in the class taxonomy, applying this rule to our example tells us that

\[
\text{son, all class man} \rightarrow \text{son, all class person}
\]

Therefore, from these two deductions, we have by transitivity of implication,

\[
\text{child, all class man} \rightarrow \text{son, all class person}
\]

This without explicitly having stated anything about the precise definition of \( \text{son} \), but just based on information about the child-men and person-men inclusion dependencies, we were able to determine that \( c_2 \) subclass \( c_1 \). The formalization of CADEIDE is derived from KAVENOR, which appeared in [50], along with a decision procedure for computing subsumption. It is important to note that this inference could not have been made if we did not have information about the inclusion of \( \text{son} \) within \( \text{child} \) in the attribute hierarchy. Thus, the class taxonomy contains information about the inclusion relationship between single place predicates whereas the attribute hierarchy deals with two-place predicates. The decision procedure for computing subsumption allows us to automatically generate the class taxonomy based on object definitions given an attribute hierarchy. If the attribute hierarchy is a flat structure (i.e., a tree of depth 1), then the lack of depth will be reflected in the class taxonomy. For example, given a flat attribute hierarchy, we cannot say things like every son is necessarily a child. In such a situation, the system could never have deduced that \( c_2 \) is a subclass of \( c_1 \). Further, note that a flat attribute hierarchy structure is the same as having no attribute hierarchy at all, or rather, just a list of attribute names. This means that the previous step of building an
attribute hierarchy is entirely optional with a caveat that attributes with the same name will be assumed to have the same meaning.

4.3. Choice of CANDIDE as the Canonical Data Model

There are many reasons for choosing CANDIDE as the canonical data model:

(1) It has a well defined semantics which allows us to automatically compute the class-subclass (subsumption) relationship. That is, the system can reason about class definitions in a correct and complete manner.

(2) The same subsumption function can be used to decide whether
   (a) a given class is incoherent,
   (b) two given classes are disjoint,
   (c) two given classes overlap, and
   (d) two given classes are equivalent.

(3) CANDIDE incorporates an attribute hierarchy as an integral part of the database schema definition; i.e., it is assumed that an attribute hierarchy is given to the system. This allows us to distinguish between heuristic reasoning (to determine the attribute hierarchy) and automatic reasoning (to determine class relationships). This means that the question of whether an attribute a is above b in a given hierarchy is treated like an oracle. Thus, the system does not care how the attribute hierarchy is generated, but given one, it will correctly reason about class definitions within the schema. In other words, information about attribute inclusion is exploited in the inference rules for computing subsumption in a sound fashion.

(4) It is a minimal model as it gives only the generalization and aggregation abstractions. [40] The schema graph (or class taxonomy) can be generated by using only the two abstractions provided. A class (or node in the schema graph) can be defined by specifying cardinality and domain constraints for each attribute of the class. These two kinds of constraints are orthogonal to each other. Domains can be atomic, simple, or complex (generated by using the domain constructors).

(5) It is a relational model since the system semantics allow us to automatically determine that two different forms representing the same concept are equivalent. This is done by reducing such class equivalence to computing bidirectional implication of the constraints on each corresponding attribute defining the classes in question (see Sec. 2.1).

(6) The CANDIDE classifier computes the minimal superclasses and maximal subclasses of a given class by employing the subsumption function as described above. Thus merging a schema essentially reduces to classifying each class in each schema to generate a new, or global schema. This schema is generated by modifying the class-subclass links in each class as dictated by the classification function and is independent of the order in which schemas or classes are chosen for merging. Note that the aggregation links are left unchanged. Also, no new classes are created that did not already exist in at least one of the n schemas. We shall see more about this in Sec. 5.
The first two items imply that there is no need for the user to specify the class assertions of the type equal and inclusion (contains/contained-in) as in [13,27] since the classifier automatically deduces such assertions. Furthermore, subsumption can be used to deduce "overlap" and "disjoint" relationships between two classes. However, the classifier cannot generate a new class and cannot decide whether two classes that overlap or are disjoint should be merged (see elaborate this in Sec. 5). These two cases are handled in the third phase by the use of appropriate operators by the user.

5. Schema Reorganization

After the s-component schemas have been merged, the user can employ a set of well-defined operators to reorganize the schema. We first define a set of comparison operators which help in determining class relationships, and then a set of restructuring operators. We then discuss some guidelines for applying these operators to restructure schemas.

5.1 Operators to Determine Class Relationships

As mentioned above, it is important to automatically compute various class relationships such as equivalent, includes, is-included-in, overlap, and disjoint. The compare function compares includes and is-included-in as an obvious manner. We now give a few definitions which show how these relationships can be computed with the help of the subsume function. Let f, g be two classes and E[f], E[g] represent their respective extensions, i.e., set of instances.

(1) Subsume

\[ \text{subsume}(f, g) = \text{true} \iff E[f] \supseteq E[g] \]

However, as explained in Sec. 4.1.3, subsumption is computed on the basis of class definitions and not the actual extensions. This means that the subsume function returns true if and only if the constraints on each attribute of g logically imply the constraints on the corresponding attribute of f. Therefore, subsume(f, g) = f includes g = is-included-in f.

(2) Equivalence

\[ \text{equivalent}(f, g) = \text{true} \iff E[f] = E[g] \]

This function can be computed using subsumption as follows:

\[ \text{equivalent}(f, g) = \text{subsume}(f, g) \land \text{subsume}(g, f) \]

(3) Overlap

Overlap is defined as follows:

\[ \text{overlap}(f, g) = \text{true} \iff E[f] \cap E[g] \neq \emptyset \]
Overlap can be computed as follows:
\[ \text{overlap} \{ f, g \} = \text{true} \iff \text{subsume} \{ f, g \} \land \neg \text{subsume} \{ g, f \} \land \neg \text{disjoint} \{ f, g \} \]

That is, two classes overlap if they do not subsume each other, and are not disjoint. As an example, consider the two classes Instructor and Student. They can overlap because an instance of Instructor may also be an instance of Student, such as a Teaching Assistant.

(4) Disjoint

Two classes are defined as disjoint if the intersection of all possible extensions of the two classes is empty.
\[ \text{disjoint} \{ f, g \} = \text{true} \iff E[f] \cap E[g] = \emptyset \]

Disjoint can be computed as follows:
\[ \text{disjoint} \{ f, g \} = \text{true} \iff \text{incoherent} \{ \text{conjunction} \{ f, g \} \} \]

Here conjunction is a function which returns a new class from two given classes such that the extension of the new class is the intersection of the extensions of the given classes:
\[ E[\text{conjunction} \{ f, g \}] = E[f] \cap E[g] \]

Incoherent is a boolean function which tests for logical inconsistency in constraints on the attributes of a given class. For example, the maximum cardinality cannot be less than the minimum cardinality on any attribute (for more details see [30]). This means that there cannot be any instances of an incoherent class.

For example, the class Part-time-student from schemas \( s_1 \) with the constraint that a part-time student can register for a maximum of nine credit hours, and the class Full-time-student from schemas \( s_2 \) with the constraint that a full time student must register for at least 12 credit hours, clearly show that there cannot be a student who is both part-time and full-time. As another example, if a class Student-attending-class has the constraint that it must have at least two part-time students among its members and the class Student-attending-class has the constraint that only one full-time student can be its member, then these two classes do not have a common instance. A conjunction of these two classes will lead to an incoherence.

Therefore, the classes are identified as disjoint.

The functions defined above are based on the ability to compute subsumption, i.e. they are based on the semantics of class definitions. However, it is possible to define boolean functions which return true or false based on more syntactic criteria such as attribute relationships. Therefore, we define two additional operators. These are merely alternate definitions of the notions of overlap and disjoint. They are defined here because the user may wish to use these definitions rather than the ones defined above.
(5) \text{in class overlap}(f, g).

This function returns true if and only if there exists at least one pair of attributes \(a_1\) and \(a_2\) such that \(a_1 \equiv a_2\) or \(a_1 \sqsubseteq a_2\) or \(a_2 \sqsubseteq a_1\) where \(a_1\) is any attribute of \(f\) and \(a_2\) is any attribute of \(g\).

(6) \text{disjoint}(f, g).

This function returns true if and only if there does not exist any pair of attributes \(a_1\) and \(a_2\) such that \(a_1 \equiv a_2\) or \(a_1 \sqsubseteq a_2\) or \(a_2 \sqsubseteq a_1\) where \(a_1\) is any attribute of \(f\) and \(a_2\) is any attribute of \(g\).

All these functions can be computed automatically. The user now restructures the global schema based on his or her perspective of the universe of discourse with the help of these functions.

5.2. Schema Restructuring Operators

This second set of operators can be used to create new classes from existing ones and/or delete existing classes. The schema integrator uses these operators to restructure the schema.

- delete(\(f, g\)). The function will delete the class \(f\) from the integrated schema and check for consistency. If the resulting schema is found inconsistent then the user is alerted with an appropriate message. For example, it might be the case that there is a class "student" which has an attribute "belong to", the domain of which is the class "department" which was just deleted. Such deletion will be disallowed.

- c = \text{generalize}(f, g): The two classes \(f\) and \(g\) are generalized to create a new class \(c\) (the label for class \(c\) is provided by the user, e.g. teacher = \text{generalize}(\text{teacher, instructor}). Only the common attributes are chosen for \(c\) and a union for logical conjunctions) of the constraints on these attributes as defined. The resulting class is checked for reachability, and is well-defined if the following relationships are satisfied: \(E[f] \supseteq E[c]\) and \(E[g] \supseteq E[c]\).

- c = \text{generalize}(f, g). The same as \text{generalize} except that \(f\) and \(g\) are now deleted from the schema. This is useful when there is no need for definitions of the classes in the component schemas from which a class in the integrated schema is derived. An example is when a new database is created based on the integration and potential users have their own view of what the schema should look like (e.g. since the two classes schema employer and schema employee are generalized to create final employee, there may be no need to keep the original two versions of employee in the integrated schema).

- c = \text{specialize}(f, g). The two classes \(f\) and \(g\) are specialized to create a new class \(c\) (the label for class \(c\) is provided by the user). For the common attributes of \(f\) and \(g\) chosen for \(c\), the logical conjunction of the constraints on these attributes is defined. The remaining attributes are merely constrained to the definition of \(c\). The resulting class is then checked for consistency, and is well-defined if the following relationships are satisfied: \(E[c] \supseteq E[f]\) and \(E[c] \supseteq E[g]\).
\[ r = \text{specialize}(f, g) \]. The same as specialize except that \( f \) and \( g \) are then deleted from the schema. This function is needed for similar reasons as generalize.

5.3. Applying Functions to Integrate Schemas

To integrate two schemas, \( \text{src} \) and \( \text{src}2 \), we apply the above class comparison functions between each pair of classes \( f \) and \( g \) such that \( f \in \text{src} \) and \( g \in \text{src}2 \). (It is assumed that each individual schema is already "well-defined".) By well-defined, we mean that there are no inconsistencies in the individual schemas.) Whenever two classes are identified as equivalent, they are merged into a single class. Similarly, when \( \text{src} \) class subsumes another class, the integrated schema is appropriately restructured. The user then applies the schema restructuring operators to get a final, integrated schema based on his knowledge of the universe of discourse. Disjoint or overlapping classes may or may not be merged, depending on the situation. For example, birds and aircraft may be disjoint classes, but can be merged under a common superclass called flying objects by executing the following operator: flying objects = generalize(flying, aircraft). Similarly, instructors and students may overlap, and can be possibly merged to form a common subclass called teaching assistants. On the other hand, even though courses and instructors are related in the sense that instructors teach courses, and regardless of whether they overlap or are disjoint, they need not be merged into a common class. Such decisions are and can be made solely by the user.

Shown below are two types of class relationships: mergeable and non-mergeable. If two classes satisfy the semantics for equivalence or inclusion relationship based on substitution, then they are obviously mergeable. However, as explained above, if two classes overlap or are disjoint, then user intervention is required to decide whether they are mergeable or not.

The following rules provide guidelines to decide mergeability which the user can exploit to restructure the schemas:

1. If \( \text{class1} = \text{class2} \) or \( \text{class1} \subseteq \text{class2} \) then merge in obvious manner.
2. If \( \text{class1} \cap \text{class2} \) is disjoint or overlap and the two classes have a common ancestor other than null then the two classes can be generalized.
(2) if \( \text{class-rel} = \text{overlap} \) and the two classes have a common subclass, then the two classes can be specialized.

(4) if \( \text{class-rel} = \text{disjoint} \) or \( \text{overlap} \) and the two classes do not have a common ancestor, then the user makes a decision.

(5) if \( \text{class-rel} \) = overlap and the two classes do not have a common subclass, then the user makes a decision.

The first rule is always taken care of in the schema merging phase described earlier since it is the result of the classification function. The second and third rules can also be incorporated as part of the automatic reasoner. However, user intervention becomes a must when one of the last two rules is triggered.

6. An Example of Schema Integration

We now show how our approach will apply to a simple example used in [13]. Figure 3 shows two ER schemas, \( s_1 \) and \( s_2 \), to be integrated. Figure 4 shows the equivalent CANDIDU schema consisting of the respective attribute hierarchies and class taxonomies. Figure 5 shows the attribute hierarchy against which these two schemas are to be integrated.

The attribute hierarchy essentially represents the various semantic clusters and the attribute relationships within a given cluster. For example, \( \text{Name of Department} \) in schema \( s_1 \) is equivalent to \( \text{Name of Department} \) in schema \( s_2 \) and \( \text{Grad. Student Name} \). The distinction between points in the semantic space and the attribute definition space can be dropped once the attribute relationships have been decided by the user. Any set of equivalent attributes must be replaced by a single label since they all map onto the same semantic point (and also given the fact that the attribute hierarchy actually represents the relationships between semantic points). Further, ambiguity may be resolved by appending the attribute by a schema identifier as in \( s_1.\text{Department} \). Entities and attributes can have the same labels — this causes no ambiguity in the CANDIDU classifier.

All attribute relationships need not be known in advance. The integration process can be iterative in the sense that if the user is not satisfied with the results obtained by classifying each class in all schemes being integrated, who can modify the attribute hierarchy and start again. Thus the user can start with a flat structure (the same as having no attribute hierarchy) and successively modify and refine this structure until a satisfactory schema is achieved. The classification process essentially reorganizes the entities in each schema by discovering the class/subclass relationship among entities regardless of which schema they occur in. This phase is called class integration and is followed by the schema restructuring phase. Figure 6 shows the result of class integration in our example. Figure 7 shows the fully integrated schema including the effects of schema restructuring operators. Note that the mere application of the classification function will not result in the generation of new classes (entities) such as \( \text{E.Student.Faculty} \). Furthermore, the names of newly generated classes are given by the user.
Fig. 3. Index selection x1 and x2.

Fig. 4. (i) CANDEK class hierarchy for x1. (ii) CANDEK attribute hierarchy for x2.
Fig. 4(b). (i) CANDIDE class hierarchy for s2. (ii) CANDIDE attribute hierarchy for s2.

Fig. 5. Results of Phase 1: Global attribute hierarchy.
Fig. 6. Result of Phase II + III: Integrated CANDIDE scheme

\[
\begin{align*}
D.\text{Student} & \equiv \text{generalized} (\text{Student, Faculty}) \\
E.\text{StudentMajor} & \equiv \text{generalized} (\text{Student, Major})
\end{align*}
\]

The integration phase is independent of the order in which schemas are integrated, and can be described by the following steps:

1. Translate each E-R schema into an equivalent CANDIDE schema (see [23] for more details).
2. Optionally, define an attribute hierarchy using the guidelines presented in Sec. 3. This hierarchy will be used for integrating all schemas in a given domain of discourse.
3. Start building the integrated schema by beginning with some arbitrary schema and classify each entity in that schema with respect to the integrated schema.
4. For all remaining schemas, classify each entity in each schema.
5. Apply schema restructuring operators to make further modifications to the integrated schema.
6. If the integrated schema is unsatisfactory, refine the attribute hierarchy and reclassify all classes in the integrated schema. Repeat Step 6 until the integrated schema can be made satisfactory by applying the schema restructuring operators.
Steps 5 is driven by the user, and was discussed in Sec. 5. As an example, the faculty and instructor entity can be generalized into the teacher entity, and teacher and student can be further generalized into person. The number of iterations (that is, Step 6) is also decided by the user.

7. Conclusions

We have discussed certain limitations of previous approaches to schema integration, especially with respect to automatically discovering attribute relationships. We then proposed a methodology with which the user can identify attribute relationships. This information is used to generate a global attribute hierarchy. The second phase of our approach involves automatically integrating classes from component schemas based on a classification algorithm. The fundamental unit of comparison in this algorithm is subsumption.

Based on the semantics of the CANDIDE data model, we have been able to decentralize the schema integration activity into first reasoning about attributes and then classes. Reasoning about attributes involves heuristic reasoning (performed by a human) while reasoning about classes utilizes automatic and formal reasoning. Reasoning about the relationships between attributes is by no means an easy task and we have discussed that this task cannot be automated. It can however be made somewhat manageable and structured using the methodology discussed in the paper. We also discussed why the attribute reasoning should not be limited by their associations with the classes that they belong to in the component schemas.

Once the attribute hierarchy is generated most relationships between the classes can be automatically determined and an integrated schema can be generated automatically by merging all associated related classes using the process of classification. The inheritance rules to compute subsumption exploit the attribute relationship information contained in the attribute hierarchy in a second manner. They also adhere to the constraints that are captured in the class definitions. The order in which a given class is classified is irrelevant. The subsumption operator is further used to define a set of comparison operators. The user can exploit these comparison operators in the final phase of schema reorganization for which a well-defined set of schema restructuring operators are given.

The main contributions of this paper are:

(1) identifying portions of the schema integration process that can be meaningfully automated and those portions that are necessarily user-driven,
(2) exploiting the notions of classification and subsumption to provide a vehicle for automatic class integration,
(3) defining a set of schema comparison and restructuring operators, and
(4) developing an overall schema integration methodology with three phases.

A schema integration toolkit called BERDI based on the approach discussed in this paper has been implemented and tested at Bellcore [35-39]. The user interface of BERDI is based on an extended E-R model. It supports generation of attribute
hierarchy as described in this paper, which can then be used for automatic class integration as discussed here, or alternatively used as constraints when the user asserts class relationships. For the automatic class integration option, after generating an attribute hierarchy, the component E-R schemas are semi-automatically translated into equivalent CANDIDE schemes, as described in this paper, and translated back into an extended E-R schema.

An area for further research is to investigate possible learning algorithms which can help in discerning attribute relationships. It may also be desirable to decide, perhaps through an extensive empirical study involving real schemas, how complete and how fixed the integrated schema is at the end of the second phase, and how much work is needed in the third phase. We hope that the use of automated reasoning, in our case, one based on terminological logic, and our study on where automatic reasoning is possible and where it is not possible (although limited to the context of schema integration), will be a step towards developing future intelligent and cooperative information systems.

Acknowledgements

We would like to thank Padma Chann, Howard Marcus, and Ashoke Sanyalere for implementing the scheme integration tool at Bellcore. Many interesting discussions with Ashoke Sanyalere and Howard Marcus are also acknowledged. Ranee Huddart and James Lamm provided helpful comments on the early draft of the paper.

References

[26] M. Manville and W. Egelhof, Matching techniques in global scheme design, in Proc. 12th Int. Conf. on Data Engineering, Los Angeles, April 1994.