Interdatabase Existence Dependencies: a Metaclass Approach

Malú Castellanos, Thomas Kudrass*, Fèlix Saltor, Manuel García-Solaco

Dept Llenguatges i Sistemes Informàtics. U.P.Catalonia
{castellanos, mgarcia, saltor}@lsi.upc.es

*Technische Hochschule Darmstadt. F. Informatik
kudrass@dvs1.informatik.th-darmstadt.de

Abstract

We present a novel approach to deal with the modeling and operational aspects of the interdatabase existence dependency problem. By extending our canonical model BLOOM with high level abstractions (Metaclasses) for the different kinds of existence dependencies and their corresponding enforcement policies, inter-dependencies are declaratively specified and their behavior embodied into the model by exploiting the metaclass mechanism of BLOOM.

1. Introduction

In a federated environment, where the component databases (DBs) represent overlapping or related parts of the real world, some data in one DB depend on data in another. These data are said to be "interdependent". If consistency between interdependent data is desired, the consistency requirements must be specified through constraints named “interdatabase dependencies” [1] (*interdependencies* for short). The price paid for the consistency is the loss of some autonomy.

Several kinds of interdependencies exist, as shown in [2] and [3]. In this paper we deal only with *existence dependencies* (EDs), in which the presence of data in one database depends on related data being present in another. As observed in [2], EDs are perhaps the most significant in practice, either alone or together with other kinds of interdependencies. According to [3], EDs can be either *directional*, if the values in one of the DBs are treated as primary, or *nondirectional*, if values are treated in a symmetric fashion between DBs.

Since interdependencies constitute common knowledge about the federation, they should be specified at the federated level in the canonical model. A good canonical model should provide the necessary abstractions to represent interdependencies and the support mechanisms to enforce them. BLOOM [4] is an object model that was developed as a canonical model for federated DBs. Therefore, it provides, among other things, abstractions needed to model interdependencies and enforces them through policies expressed in metaclasses. Alternatively, the relational or a generic object model such as ODMG93 or C++, can be extended through event-condition-action (ECA) rules to specify and enforce the interdependencies [3], [5]. In this paper we concentrate on the use of BLOOM to handle interdependencies by exploiting its metaclass mechanism. Implementation issues are not covered.

The paper presents the modeling (section 2) and enforcement of EDs (section 3), and a comparison with other work (section 4). It concludes with references to future work.

2. Modeling Existence Dependencies

2.1 Inherent Directional EDs.

Two classes C1 and C2 of two export schemas, are integrated in a federated schema by using different abstractions according to their resemblance at the schema level [6] and their relationship at the extension level. One kind of resemblance is equality. Once C1 and C2 have been discovered to be equal at the schema level, their relationship at the extension level is analyzed: equality, inclusion, disjunction and overlapping are the possibilities. Only the inclusion relationship involves a directional ED: for each object O1i in C1 there must be a corresponding object O2j in C2, but there may be objects in C2 without a corresponding object in C1. At the federated schema level, C1 is a subclass of C2 by one of several kinds of specialization supported in BLOOM, the *general specialization*. The behavioral interpretation of this abstraction corresponds to the desired directional ED. This is similar to the sub_OC relationship to support EDs in [2].

To illustrate the idea let us assume that any student who appears in the DB of the CS_department (DB1) must also appear in the general university DB (DB2). This corresponds to an interdependency which states that for an object to be a member of the class 'cs_students' in DB1 it has to exist as a member of the 'univ_students'

class in DB2. In the federated schema, 'cs_student' would be a subclass by *general specialization* of 'univ_student'.

2.2. Explicit Directional EDs.

The powerful specialization abstractions of BLOOM are not enough to handle the full spectrum of existence dependencies. In particular they fail to cover *ad-hoc* cases which require to explicitly specify the ED. In order to meet this requirement we identified different kinds of EDs that result from the combination of different orthogonal criteria (strict vs. relaxed and exclusive vs. non exclusive) with their various enforcement policies, and extended BLOOM with corresponding high level abstractions. In this paper we deal only with *exclusive strict dependencies* (*strict EDs* for short) where the existence of an instance of a class ***(dependent)*** depends on the existence of an instance of another class ***(Dependor)***. By using these abstractions, EDs can be specified declaratively. Since a given ED may apply only to those members of the dependent class that satisfy a given condition ***C***, the ED specification optionally includes *C*. In order to establish correspondences between the dependent and the dependor class members, an identification function ***I*** has to be previously defined. Attributes of the classes may be used for such a function. The classes involved in an ED belong to different databases in the case of interdependencies.

The operations (*violating actions*) that may violate an ED (no corresponding dependor object) are:

i) insertions into dependent: *insert-d*

ii) updates to dependent:

ii.1) on attributes used in the condition *C:update-C(d)*

ii.2) on attributes used in the identification function *I: update-i(d)*

iii) deletions from Dependor: *delete-D*

iv) updates to Dependor on attributes used in the identification *I: update-I(D)*

A possible syntax for this kind of ED is:

d[C] *strict-dep-on* D *with* {insert-d-effect, update-C(d)-ef., update-I(d)-ef., delete-D-ef., update-I(D)-ef.}

where:

- The condition *C* can be an arbitrarily complex query (expressed in BOL (BLOOM Object Language)). At present we restrict to simple logical expressions involving simple comparison operators.
- The enforcement policy enclosed between {}: specifies the effects or reactions that have to take place when there is a violating action:
  - i) *insert-d-effect*: when no corresponding dependor exists for the dependent being inserted, the insertion can be either *propagated* (insert the corresponding dependor) or *blocked* (not allowed)
  - ii.1) *update-C(d)-effect*: if the new value of the updated attribute now satisfies the condition *C*, the object turns into a dependent. If no corresponding dependor exists the effect can be either *'insert'* (insert the corresponding dependor) or *'block'*.
  - ii.2) *update-I(d)-effect*: when an attribute of a dependent used in the identification function *I* is updated, the corresponding dependor changes too. If this dependor doesn't exist then the possible effects are *'propagate'* (update the old corresponding dependor), *'insert'* (insert the new corresp. dependor), or *'block'*.
  - iii) *delete-D-effect*: if there are dependents for the dependor being deleted then either the deletion is *propagated* (delete its dependents) or *blocked*.
  - iv) *update-I(D)-effect*: updating a dependor attribute that participates in the identification function *I* leads to a violation if there is no longer a dependor for the dependents corresponding to the old value. In this case the effect can be *'propagate'* (update the dependents), *'delete'* (delete the dependents) or *'block'*.

The following table summarizes these effects:

Viol.Action / Effect	insert_on_d	update_C(d)	update_I(d)	delete_D	update_I(D)
propagate **p**	X	-	X	X	X
insert **i**	-	X	X	-	-
delete **d**	-	-	-	-	X
block **b**	X	X	X	X	X

('X' means 'applicable' and '-' not applicable)

For the syntax given above, the effects must be specified in the order: i), ii.1), ii.2), iii) and iv). Only the updates on the attributes specified above may result in violation of an ED. Dependencies that may affect the values of other attributes of objects related by an existence dependency are not EDs but 'value dependencies', not covered here. An example borrowed from [3] is: if there is a plant in the class Plants of the DB of Italy whose region is Milano, then that plant must exist in the class Nodes of the DB of the Milano region.

Assume that the desired enforcement policy is: i) *propagate* on insert in Nodes, ii.1) *insert* on an update that sets the attribute 'region' of a plant in Nodes to 'Milano, ii.2) *propagate* on an update of a Milano plant in Nodes on an attribute that participates in the identification function, iii) *propagate* on deletion from Plants, and iv) *block* when there is an update of a plant attribute (in Plants) that participates in the identification function. The corresponding ED is:

Nodes [region = 'Milano' ^ function = 'plant']
strict_depends_on Plants
with {p, i, p, p, b}

2.3 Nondirectional Existence Dependencies

Nondirectional EDs are handled similarly to their directional counterparts (sections 2.1 and 2.2). a) The inherent ones occur when the relationship between the extensions of the classes C1 and C2 is the equality: to each object O1i in C1 there must be a corresponding object O2j in C2, and vice versa. At the federated level they are integrated into a single class. b) The nondirectional explicit EDs correspond to the *bidirectional strict dependency* of BLOOM. A possible syntax is: i-d_1 *bidir-strict-dep-on* i-d_2, where i-d_1 and i-d_2 are *interdependent* classes: the existence of any object member of i-d_1 depends on the existence of a corresponding object member of i-d_2, and vice versa. In contrast to directional EDs, no enforcement policy has to be specified because only *'propagate'* on insert, delete and update of any interdependent object makes sense.

3. Embodying Enforcement Policies into the Model: a Metaclass Approach

3.1 Metaclasses.

In our approach metaclasses do not only specify the behavior of their instance classes, but also that of the instances of the instance classes, similar to [7]. Embodying the inherent behavior of abstractions into metaclasses provides the basis for the extensibility of the model: to extend the model with new abstractions, only the corresponding metaclasses must be defined for them. To extend BLOOM to cope with the interdependency problem: new metaclasses corresponding to the different kinds of explicit EDs are defined.

3.2 Inherent Directional EDs

As explained in 2.1, the behavioral interpretation of the general specialization abstraction of BLOOM corresponds to this kind of ED. For each BLOOM abstraction there exists a metaclass. The metaclass embodies the behavioral interpretation of the abstraction given by its *inherent existence dependencies*. For the general specialization abstraction the inherent ED states that an object can exist as member of a subclass only if it exists as member of the superclass. The enforcement is done along the instantiation dimension in an analogous way as for the explicit EDs explained below.

3.3 Explicit Directional EDs

To explain how existence dependencies are enforced in BLOOM, we will first analyze behaviors corresponding to single effects, then we will proceed with behaviors given by the combination of these effects to obtain the complete enforcement policies. For each single effect there is a metaclass which embodies the effect in the form of a *metamethod*. Along the generalization dimension these metaclasses are subclasses of the root strict_ED metaclass by using the possible violating actions as specialization criteria. As an example we show the metaclasses resulting from specializing strict_ED by the insert-dependent (insert_d) criteria. The specification of the metamethods is given informally in an ECA rule style.

Metaclass p_strict_ED
Metamethod
propagate_on_insert_d
{on insert in d
if inserted[1] not in D
then insert in D}

Metaclass b_strict_ED
Metamethod
block_on_insert_d
{on insert in d
if inserted not in D
then reject insert}

These metaclasses are defined explicitly and constitute the first level of the 'Strict Existence Dependency Specialization Semilattice' (SEDSS). The rest of the levels of the semilattice are automatically generated by applying the BLOOM specialization mechanism [8]. Essentially, the mechanism consists in progressively specializing each one of the metaclasses given above, by the different specialization criteria. Thus, at any level, a metaclass is further specialized by the criteria that have not been used yet along the successions of specializations leading to it. Since the combinations of the different effects on violating actions (specialization criteria) lead to a total of 72 (2x2x3x2x3) enforcement policies, there are 72 kinds of explicit strict EDs whose behavior is embodied in the metaclasses of the leaf level: p_i_p_p_p, p_i_p_p_d, p_i_p_p_b, p_i_p_b_p, etc. One such metaclass is shown next:

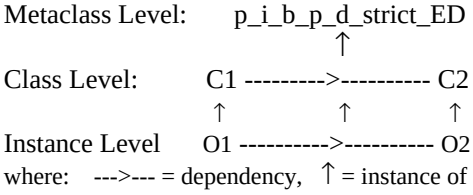
Metaclass p_i_b_p_d_strict_ED
specialization_of
[*by* insert_d] x_i_b_p_d_strict_ED
[*by* update_c(d)] p_x_b_p_d_strict_ED
[*by* update_i(d)] p_i_x_p_d_strict_ED
[*by* delete_D] p_i_b_x_d_strict_ED
[*by* update_i'(D)] p_i_b_p_strict_ED
metamethods
propagate_on_insert_d (inherited)

[1]objects manipulated by an action are kept in transitory classes *inserted, deleted, old-updated* and *new updated*.

```
        insert_on_update_c(d)          (inherited)
        block_on_update_i(d)           (inherited )
        propagate_on_delete_D          (inherited )
       delete_on_update_i'(D)          (inherited)
end_metaclass
```

Notice the modularity of the approach, which makes it easy to extend the model with new abstractions. It is also important to recall that metaclasses are just the vehicle to automatically enforce EDs, therefore the user doesn't have to deal with them.

When a dependency is specified, it is defined by instantiation of the corresponding metaclass as shown in the figure below. The metamethods are instantiated as methods of this instance and insert, delete, and update methods of the classes invoke them. Note that the methods that embody the enforcement policies are not specified for each ED again and again, but just once in the metaclasses.



3.4 Nondirectional Existence Dependencies

This kind of EDs are embodied similarly to their directional counterparts. a) Non directional inherent EDs are automatically translated from the federated schema level to local schemas using corresponding mappings. b) Non directional explicit EDs have specific metaclasses, with metamethods analogous to those explained in section 3.3.

4. Comparison with Other Work

Our approach strikes a balance between the fully automatic approach of [3] and the fully user-defined approach of [1]. Compared to [3] it provides a higher degree of flexibility because we provide a choice of enforcement policies as part of the dependency specification and do not assume a single enforcement policy. In contrast to [1], actions for the enforcement are generated automatically relieving the user from the burden of defining them and eliminating potential incorrectness. It differs from ECA rules and triggers in commercial relational DBMS, because the definition of dependencies is declarative and the metaclasses model the behavior at a higher level of abstraction. ECA rules can be seen as a representation at a lower level providing more capabilities to express operational semantics.

5. Conclusions and Future Work

We have presented an approach where users specify interdatabase EDs declaratively using high level abstractions of BLOOM and the enforcement policies are embodied in corresponding metaclasses. A number of issues still have to be addressed. In particular, value dependencies, correctness of interdependency specifications, and conflict resolution policies. Currently, the enforcement of interdependencies is considered only in a synchronous mode. A prototype to implement the metaclass approach for the support of existence dependencies is being developed in C++ on top of ObjectStore.

Acknowledgments

We thank Alex Buchmann for his helpful advice and the reviewers for their useful comments. This work is partly supported by the Spanish PRONTIC program TIC93-0436 and Spanish-German action 147-B.

References

[1] M.Rusinkiewicz, A.Sheth, G.Karabatis. "Speci-fying Interdatabase Dependencies in a Multidatabase Environment". *IEEE Computer*, Dec. 1991.

[2] Q.Li, D.McLeod. "Managing Interdependencies among Objects in Federated Databases". In D.Hsiao, E.Neuhold & R.Sack-Davis (eds) *Interoperable DB Systems (DS-5)*, North Holland, 1993.

[3] S.Ceri, J.Widom. "Managing Semantic Heteroge-neity with Production Rules and Persistent Queues". *Proc. 19th Int. Conf. VLDB*. Dublin, Aug. 1993.

[4] M.Castellanos, F.Saltor, M.García.-Solaco "The Development of Semantic Concepts in the BLOOM Model". Rep. LSI-91-22, Faculty of Informatics, Polytech. Univ. of Catalonia, Barcelona, Oct. 1991.

[5] H.Branding, A.Buchmann, T.Kudrass, J.Zimmermann. "Rules in an Open System: The REACH Rule System". *Int. Workshop on Rules in Databases*, Edinburgh, 1993.

[6] M.García, M.Castellanos, F.Saltor. "Discovering Interdatabase Resemblance of Classes in Interoperable Databases" *3rd Int Workshop RIDE-IMS93*. Vienna, 1993.

[7] W.Klas, E.Neuhold, M.Schrefl. "Tailoring Object Oriented Data Models Through Metaclasses". Tech.Rep GMD, Darmstadt, Apr. 1989.

[8] F.Saltor, M.Castellanos, M.Garcia, T.Kudrass. "Modeling Specialization as BLOOM Semilatti-ces". *Proc. 5th Euro-Jap. Seminar on Information Modeling & Knowledge Bases*. Kista, June 1994.