An Advanced Relationship Mechanism for Object-Oriented Database Systems

Jürgen Schlegelmilch
Department of Computer Science, University of Rostock, Germany*

May 6, 1996

Abstract

Unlike the entity-relationship model, object-oriented systems lack a notion of "relating" objects to others: the only means to relate objects are object-valued attributes holding references. This article presents a relationship mechanism for an object-oriented database model that extends known relationship approaches with support for derived relationships, separation of connectivity and visibility, and query language integration. We also introduce a persistence model based on this relationship model, and shortly discuss integrity checking issues.

1 Overview

This paper presents a relationship mechanism for object-oriented databases that offers several improvements over existing approaches. The work has been done in the context of the OSCAR[1] project [HFW90] where we develop an object-oriented database management system; OSCAR is based on the database model EXTREM[2] and offers all structural elements of an object-oriented database management system. Its strength are the query languages that are based on relational counterparts, and can handle all of EXTREM's constructs adequately.

The paper is organized as follows: The first part introduces the mechanism to model general relationships in OSCAR. After preparing the grounds in Section 2, we review existing relationship constructs in Sect. 3. Then we define the syntax and semantics of our relationship mechanism in Sect. 4, with the behavioral aspects in Sect. 5. In the second part, we present our enhancements: Sect. 6 introduces derived relationships, and in Sect. 7 we define briefly the persistence model. Finally, Sect. 8 compares different approaches to ensure the consistency of relationships, before we summarize our contributions in Sect. 9.

2 Introduction

In object-oriented systems, the object is the unit of discussion: every aspect of the universe of discourse is described as an object, i.e. as local state and behavior. However, there are properties that naturally belong

*University of Rostock, D-18051 Rostock, Germany; email:schlegel@informatik.uni-rostock.de

[1] **O**bject Management **S**ystem for **C**omplex **A**pplications, Approach: **R**elational

[2] **Ext**ended **Re**lational **M**odel

to more than one object, and are usually model-led by relationships in object-oriented design methodologies [CY90], [RBE+90]. Most object-oriented programming languages offer only an object-centered view on non-local properties, by means of references to other objects. References are attributes that hold a pointer to the related object, with no possibility to associate additional information with them. In contrast, relationships provide the complete picture: unlike references, they can hold additional information about connections, such as attributes, integrity constraints, propagation patterns and operations.

The advantages of general relationship mechanisms have already been discussed in the literature, e.g. in [Rum87], [AGO91] and [GD91]. The usual remedy for object models with references is to use relationship classes, whose objects each represent a connection between other objects, holding the attributes of the connection. In contrast, we use the nested relational model for general relationships, following the idea of representing the Entity-Relationship model [Che76] in the relational model: relations are either sets of entities, or sets of connections with foreign keys[3]. By replacing the foreign keys by object identifiers we can model relationships in object-oriented databases. In subsection 3.1, we motivate our decision in detail.

Our approach offers the following contributions:

- It separates the issues of connection and access: objects can be related without having direct access to each other. With references, both notions are identical, while other relationship constructs often do not provide access between related objects. By separating both, we avoid both littering the name space of objects and tight coupling.
- The relational base allows for derived relationships. In all existing proposals, relationships are static, and connections have to be maintained explicitly. In contrast, derived relationships can connect objects dynamically since their structure and set of tuples are defined by a query.
- We use it as a base for a persistence model that relies on both of the above features, and is more flexible than existing approaches.

3 Related work

As already mentioned, programming languages such as C++ [ES90] or Eiffel [Mey93] offer only references. These are direct pointers from one object to the related one, with no place for additional information such as relationship attributes or cardinality constraints. A reference makes the related object directly accessible, by simply dereferencing the pointer. Due to the unidirectional nature of pointers, the referenced object cannot access the owner of the reference; another reference has to be established, thereby splitting the connection into two completely independent parts. Also, there is no way to relate objects without providing any access, thus cluttering the name space of objects.

Efficient implementation is said to be the major advantage of references, however this argument intermingles the conceptual with the internal layer. General relationships are a concept; possible implementations of them are of course system-maintained references, or tuples in a relation, as shown in [Rum87]. Furthermore, database research has shown that references are not as efficient when it comes to sets of objects: they need to be supported by relation-like structures like path indexes [LLOW91] or access support relations [KM90].

[3] Mixed cases are possible and can be seen as an optimization.

3.1 Relationship Classes

References link one object to another one, and can therefore only model binary, hierarchical relationships with no attributes adequately; to model general relationships, one has to create relationship classes where each object holds references to the objects that are related to each other. This approach can be found e.g. in [HFW90] and [DG90]. In order to access its related objects, an object has to have a reference to the relationship object. So, connection and access are already separated here. Relationship objects can hold the attributes of the connections as well as integrity constraints. However, using ordinary classes for relationships introduces problems:

- A connection should be merely the medium between connected objects, its identity is not recognized by them and therefore not needed. This indicates that relationship instances are values, not objects.
- As [DG90] correctly mentions, changing one role establishes a new connection; this disqualifies connections as objects since the relationship object's identity would have to change with every update. Connections are better identified by the set of objects they relate to each other.
- A connection depends on each of its constituting objects: if one of them is deleted, the whole connection has to be broken up. If the connection is model-led as an object, this object does not have a life on its own: it is a weak object.

To summarize, connections show more characteristics of values than of objects: They do not have an identity, do not evolve over time, and depend strongly on other objects. We therefore argue that relationship classes are not suitable for modeling relationships.

Using classes for relationships allows to build up a specialization hierarchy of relationships. This is a weak form of higher-order relationship that has to be considered; we discuss it in section 5.2.

3.2 Other Relationship Mechanisms

[Rum87] is one of the first papers on relationships. It presents the concept of relationships and already covers separation of conceptual and internal level. Relationships are relations with methods to add and delete elements and query the relation; the query facilities are much weaker than relational algebra, e.g. there is no join operator. Access to relationships is provided by access methods in the involved classes. Neither persistence nor derived relationships are discussed.

In [DG90], relationships are model-led as relationship classes in an object-oriented version of prolog. The mechanism is able to insert derived attributes into the objects in binary relationships; these attributes are restricted to read-only references and cannot offer a restructured view onto the relationship. Visibility of roles and attributes can be restricted to the participating objects. Persistence in a database is determined by reachability both from an ordinary or a relationship object. Prolog would allow for derived relationships but the paper does not discuss this.

[AGO91] presents a relationship mechanism for the object-oriented language Galileo. It essentially uses relations to model relationships, but introduces its own algebra to allow queries over relationships. The presented relationship mechanism allows arbitrary arity and attributes, and offers a persistence model comparable to our proposal. However, it does not provide easy access to related objects, nor derived relationships.

The object database standard ODMG'93 [Cat94] offers a relationship construct to integrate two references in possibly different classes into one binary relationship. No attributes or integrity constraints are allowed.

As with plain references, connection and visibility are coupled. ODMG'93 includes an object-oriented query language, but this cannot be used to declare derived relationships since it cannot extend existing objects with new references. Although [Cat94] discusses persistence, the model is essentially left open for implementors.

[The95] generalizes the approach taken by ODMG'93 by allowing to group more than two references into one relationship, and also to have relationship attributes. It uses nested relations as model, just as we do. However, the paper does not discuss access methods, persistence issues or derived relationships. Although there are mature relational query languages, this approach cannot use them for derived relationships since it requires references in the objects.

4 An Advanced Relationship Mechanism

In this section, we first introduce the EXTREM[4] model as the base of our work, and then describe the syntax and semantics of our relationship mechanism. The following sections present the concept of derived relationships, the persistence model, the behavioral part of relationships, and integrity checking strategies.

4.1 The EXTREM database model

The original EXTREM model [HH91] supports both values and objects. Values are grouped into types, objects are grouped into classes. The classes are divided into abstract and free ones and placed into an inheritance lattice formed by sub-setting of class extents. Each type and class describes the structure and behavior of its elements by typed attributes and methods. Each object belongs to exactly one abstract class and may belong to several free classes; it can move into and out of subclasses, and has a set of values for the attributes of the classes it belongs to. Objects of subclasses may be substituted for those of super-classes; for method calls, dynamic binding picks the most specific implementation regardless of the context. Values cannot be substituted for those of other types, and implementations for their methods can therefore be bound statically.

For OSCAR we use relations to model relationships instead of attributes with class types. Relations are sets of tuples and therefore a specific kind of type. However, ordinary values may not contain objects while connections are designed to hold objects in order to relate them. Also, methods for relationships are not bound statically but dynamically, depending on the set of objects in a connection.

4.2 Syntax and Semantics

Relationships in OSCAR are described with the following syntax:[5]

[4] **Ext**ended **Re**lational **M**odel

[5] `Typewriter` font denotes keywords, non-terminal symbols appear in *italic*. Brackets *[ ]* enclose optional parts, parts in braces {} may be repeated zero or more times, and the bar | separates alternatives. Concatenation has precedence over |; parentheses ( ) can be used to change precedence.

relationship ::=`relationship` *relation-name*
`(` *rel-definition* `)`
{ `;` *key-definition*}
{ ; constraints}
[; persistence]
{ `;` *access-def*} .
rel-definition ::=*schema-definition* | *query-expression*
schema-definition::=*col-definition* { `,` *col-definition*}
col-definition ::=*column-name*: *type[cardinality]*
cardinality ::= `[`*limit[* `:` *limitI][* `,` *limit[* `:` *limitI]]*`]`
limit ::=*IntegerNumber*
limitI ::=*limit*|`*`
key-definition ::=`key` *column-name* { `,` *column-name*}
constraints ::=*reaction* `unless` *condition*
persistence ::=`vital` *role-name* { `,` *role-name*}
access-def ::=`in` *role-name* `as` *feature { , feature}*
feature ::=*[*`readonly`*] feature-name[* `(` *parameters* `)` *][* `:` *result-type]*
parameters ::=*identifier* `:` *type* { `,` *identifier* `:` *type*}

```
relationship managesDept
    (manager:Employee[1,0:*],
     dept:Department[1:*,1]);
   in dept as readonly managed_by:Employee;
   in manager as readonly manages:set(Department).

dept.managed_by is (π[manager](σ[dept=this](managesDept)));
manager.manages is (π[dept](σ[manager=this](managesDept)));
```

Figure 1: Relationship between `Employees` and `Departments`

The semantics of this constructs is explained in the sequel using the example in Fig. 1. The relationship `managesDept` models a binary relation between departments and employees. Each `Department` has a manager of class `Employee`, whereas an `Employee` can manage several `Departments`, or none. A `Department` can see its manager through the method `managed_by`, and an `Employee` gets his set of `Departments` via the method `manages`. Both access methods are implemented by an algebraic query expression. In this paper, we use an algebra for nested relations with the following operators: π for projection, σ for selection, $\bowtie$ for the natural join, β for renaming, μ to nest attributes into a new relation-valued attribute, and ν to unnest such an attribute.

The *rel-definition* defines a relationship between elements to be a relation. We use the usual representation of a relation as a table with columns where each tuple of the relation is a row. Columns have an associated *type*; those with class types are called *roles* and hold the objects that are related to each other, all others are relationship attributes that describe the connection between the roles by values. Any number of roles is allowed; the case of no role at all covers the classical relational model. Using the *query-expression* instead of the explicit *schema-definition* defines a derived relationship and is discussed in Sect. 6.

The *cardinality* clause defines one or two intervals of $\mathbf{N} \cup \{\infty\}$, where ∞ is denoted as `*`; intervals $n:n$ are abbreviated as n. The relationship R satisfies its cardinality constraints, if and only if for each column a with cardinality $[n:m, n':m']$, two conditions hold:

$$\forall t \in \pi[a_i \in R, a_i \neq a](R) : n \leq |\sigma[\textstyle\bigwedge_i a_i = t(a_i)](R)| \leq m \tag{1}$$

The first interval, the inner range, is computed over all tuples of the relationship, therefore its lower bound must not be zero. It specifies how many entities of this column may participate in a complex connection determined by the rest of the columns. For the role `dept` of the example in Fig. 1, (1) becomes

$$\forall e \in \pi[\texttt{manager}](\texttt{managesDept}) : 1 \leq |\sigma[\texttt{manager} = e](\texttt{managesDept})| \leq \infty$$

This means, each `Employee` may manage arbitrarily many `Departments`.

The second interval is called the outer range, and its condition is

$$\forall o \in \mathit{class}(a) : n' \leq |\sigma[a = o](R)| \leq m' \tag{2}$$

computed over all objects of the role's class. It specifies how often an object may play the role *a* in this relationship. This number is meaningless for values: multiple occurrences of a value are unrelated to each other, while multiple occurrences of an object identifier denote the same object. In the example, (2) becomes for role `dept`

$$\forall d \in \texttt{Department} : 1 \leq |\sigma[\texttt{dept} = d](\texttt{managesDept})| \leq 1$$

so each `Department` must appear exactly once in `managesDept`: there may be no `Department` without a manager. The default cardinality is `[1:*,0:*]`, which results in no restriction at all.

Columns with 1 as the upper bound of the inner range are keys for the relation. The *key-definition* allows the definition of additional compound keys. The optional *constraints* is a predicate of first order predicate logic over the columns of the relationship including the attributes of the role classes, together with the reaction on violation. If a connection of the relationship evaluates the *condition* to `false`, *reaction* is performed. Possible reactions are given in Sect. 8, with a discussion of different approaches to check the consistency of a relationship.

The *persistence* clause defines role-based persistence and is discussed in Sect. 7.

4.3 Access methods

The *access-def* clause defines an access method *feature-name* in the class of *role*. Access methods provide the participating objects with their local view on the relationship, and are therefore restricted to use only roles and attributes of that relationship[6]. They are not allowed for relationship attributes since values don't have an identity to look at the relationship. Despite their name, access methods can look like attributes of the object: their implementation then consists of a pair (*get*,*set*) of methods, as e.g. in GOM [KMWZ91].

Access methods offer better functionality than references: being object-preserving views, their result is updatable (cf. [HS91]), plus they can calculate transitive, reflexive or symmetric closures of simple relationships as proposed for the ODMG'95 standard [Cat94], using nested relational query languages[7]. In the example in Fig. 1, two access methods are defined: Each `Department` has a method `managed_by` returning the `Employee` that manages this department, and the `Employees` have a method `manages` giving the set of `Departments` managed by him or her. Both are declared `readonly` so we only need an implementation for retrieval.

Connections sharing some objects in a column form complex connections, and access methods with nested relational algebra can visualize them. The ranges of a role can be seen as restrictions on the cardinality

[6] Methods deriving information from more than one relationship can of course be defined in any class, or as access methods in derived relationships (see Sect. 6).

[7] The transitive closure requires a query language based on logic, e.g. DATALOG, or a special fix-point operator for the relational algebra.

of sub-relations for different nestings. By using flat relations in the declaration, we do not impose a fixed view on the relationship. For example, the access method `manages` in Fig. 1 shows the relationship `managesDept` as if the `dept` attribute had been nested into an attribute `manages` by

$$\mu\texttt{[(dept);manages](managesDept)}$$

i.e. it groups the `Departments` for each `Employee`.

Postgres [SK91] uses a similar approach in a relational environment to provide access between tables that are connected via foreign keys. [DG90] discusses derived attributes for binary relationships; these attributes are restricted to read-only references and cannot offer a restructured view onto the relationship. Visibility of roles and attributes can be restricted to the participants.

5 Operations on Relationships

To retrieve information from relationships, all of OSCAR's query languages are suitable, algebraic as well as rule-based or SQL-like, since they include the corresponding nested relational part. Updates can be performed by generic commands. Automatic deletion may be caused by a constraint violation, e.g. if an object is removed from the class of its role (see Sect. 8). Although the current syntax does not include method definition, relationship methods are allowed. Their semantics differ from normal class methods because their is no receiver object.

5.1 Multiple Dispatch Methods

Class methods in classes are designed to handle single objects. In most object-oriented systems, they are allowed to take other objects as arguments. This introduces the covariance–vs.–contravariance problem (see [Cas94]) of method signatures which has not been solved satisfactorily. Contravariance only allows to generalize method arguments, except for the receiver class, and is type-safe with local type-checking algorithms, while covariance allows to specialize them, but requires a global type-check [Jon92] that inhibits separate compilation of modules. Eiffel [Mey93] and O_2 [Deu91] use covariance, while [CW85] and Tool [GM95] use contravariance.

Recent languages like Cecil [CL95] or Dylan [BH93] overcome the problem using *multi-dispatch*: An implementation is selected based on the actual type of all object-valued arguments. This solves the original problem but introduces others:

1. Multi-methods do not belong to any particular class and therefore cannot access private data, or have to break encapsulation.
2. Multi-dispatch is less efficiently implementable than single-class dispatch.
3. With overriding, the method lookup is more complicated: more than one implementation might qualify.

Problems 1 and 2 only occur in languages without single-dispatched methods. In OSCAR, we have both variants available: relationships are the obvious place for multi-methods since here is where objects meet. Problem 3 is the most serious one. The method lookup has to consider the actual type of all role objects. If there are two or more implementations that are equally appropriate, the conflict has to be resolved. An example demonstrates the problem:

Assume we have to classes A and B with $A \prec B$, and want to define a multi-method m dispatching on two objects of A. The signature of m is then $m : A \times A \to \emptyset$. Now we redefine m twice to $m' : A \times B \to \emptyset$ and $m'' : B \times A \to \emptyset$. If m is called on two B objects, both redefinitions are equally suitable: m', if the first B argument is substituted for an A, or m'', if the same is done for the second.

Dylan orders implementations by constructing a hierarchy of super-classes and would execute m'', while Cecil detects the ambiguity at compile time: a definition of $m'' : B \times B \to \emptyset$ is needed to resolve the conflict. We will adopt the type checking algorithm presented in [CL95].

5.2 Hierarchies of Relationships

In some approaches, relationships can be placed into hierarchies:

1. Subset inclusion of one relationship in another
   In our approach, this can either be enforced by a suitable constraint, or by deriving a relationship as the union of others. The class hierarchy also induces some inclusion dependencies for relationships. Only [RBE+90] considers this kind of hierarchy.

2. Inheritance of structure and implementation
   [Rum87], [DG90] and [AGO91] offer this kind of hierarchy. In our approach, this leads to name clashes for access methods in the related classes.

Since connections are values in our model, there is no substitutability possible. Our model already has means for the inclusion variant, so we do not need a specialization hierarchy on relationships.

6 Derived Relationships

The relationships discussed so far are managed by the user or programmer by inserting or deleting tuples. In contrast, intensional or derived relationships are managed by the system: they are defined by the *query-expression* instead of the *schema-definition*. Thus, the set of attributes, their cardinalities, keys and constraints are determined by the defining query. Derived relationships correspond to the views of relational databases, and allow to integrate derived information that cannot be attributed to any object. The defining query can be formulated in any relational query language, for example relational algebra or the relational calculi. These languages are efficiently implementable, optimizable, and are guaranteed to deliver finite results. In OSCAR, all query languages are supersets of their relational equivalents and can therefore be used here. In contrast to access methods, derived relationships are not restricted to derive information from a single relationship but can use arbitrary many relationships.

As an example of a derived relationship, we can derive a `colleague` relationship from a `works_in` relationship like this:

```
relationship works_in
 (worker:Employee[1:*,1],
  dept:Department[1,0:*]).
relationship colleagues
 (π[worker,worker1](works_in ⋈ β[worker1 ← worker](works_in))).
```

We want two `Employees` to be `colleagues` if they work in the same `Department`. To achieve this, we join the `works_in` relation with itself over the `dept` attribute. `worker` is renamed to `worker1` in one of the instances, so after the projection we get a relation with schema (`worker`,`worker1`) that holds pairs of `Employees`.

Object-oriented query languages like ABRAXAS [HHRW96] or OQL [Cat94] include object-generating clauses to be able to represent new combinations of existing objects. These clauses have relational semantics in the sense, that the result of the query is calculated as a nested relation, and afterwards each top level tuple of that relation gets an object identifier. However, there are problems: if such a clause is evaluated twice, are the resulting objects identical? Or, more generally, if two evaluations of such clauses yield the same nested relation, are the same identifiers used? With our relationship mechanism, the need for object-generating queries is much weaker, as new combinations of objects can be modeled by derived relationships.

[Rum87] and [AGO91] only discuss extensional relationships; they do no offer means for intensional relationships. Their main concern is to offer a replacement for references. [DG90] does not discuss intensional relationships either, but the underlying language (Prolog) would be able to provide derived relationships since those are modeled as sets of facts.

7 Role-based persistence

For a database, the objects in it are called *persistent*, all others are *transient*. Now, the question of persistence is a central one: Which objects are stored in the database, which ones are not? There are in general two possible answers to this question:

1. Objects are persistent if the programmer explicitly told them so. So, persistence is managed manually, with varying degrees of support from the programming language. Most C++-based object-oriented databases follow this policy. Class-based persistence falls in this category, too, as it is usually managed via the constructor, simply reducing the amount of code needed.

2. The programmer describes declaratively, which objects she wants to be persistent. Here, persistence is system-maintained, and the programmer is freed from the responsibility to care for it. To cope with the problem, the system uses a garbage collector that deletes objects from the database that do not match the programmer's description. Systems like O_2 [Deu91] and GemStone [Ser91] work this way.

From both the programmer's and the user's point of view, the second approach is more comfortable. Therefore, it was chosen for the OSCAR project. The declarative description that both mentioned systems use is commonly called *persistence by reachability*, and uses references:

> An object is persistent as long as it is directly or indirectly referenced from either a program or a root set of persistent objects.

This definition makes use of the uni-directional nature of references and has therefore to be modified for general relationships. It is useless, too, in the presence of class extents, since these hold references to all objects of a class and would therefore keep all their objects persistent, effectively disabling garbage collection.

By separating the issues of connection and visibility, we are able to establish relationships without providing access. This allows for class extents again. Additionally, we separate access and persistence: only

objects in roles that are declared to be `vital` are persistent. The *persistence* clause is also allowed for derived relationships, thus full declarative persistence is possible.

In the example of figure 1, to model persistence by reachability for a reference from `Employees` to `Departments` we add the following declaration to the relationship as defined there:

```
vital dept;
```

Now, any `Department` object playing the `dept` role in this relationship will be persistent. If the object in the `manager` role of its tuple is deleted, the whole tuple is removed from the relationship, and the `Department` objects loses the vital role. If it is in no other vital role, the garbage collector will get it.

Role-based persistence is much more flexible than reference-based persistence because of the arbitrary arity of relationships, and derived relationships: While persistence by reachability is more declarative than manually managed persistence, it does not allow to have "weak" references, and it cannot support persistence defined by a predicate, while role-based persistence together with derived relationships can. The declarative nature of role-based persistence stems from the declarative query languages, and the high-level concept of relationships: Any set of objects that can be retrieved by a query can be made persistent. we can even use a degenerated form of relationships with only one role to achieve class-based persistence. A detailed presentation of role-based persistence together with a comparison can be found in [Sch96].

An equivalent to role-based persistence was presented in [AGO91]. However, the relationship mechanism used there lacks derived relationships, and is therefore not able to include, for example, class-based persistence. Furthermore, the semantics of the keywords is given operationally in terms of reactions on insertions and deletions. [DG90] only mentions the dependencies between connections and their objects, but offers no flexible way to modify them.

8 Constraint handling

Keeping relationships consistent is not as easy as keeping objects consistent. An object's consistency is defined by a predicate over its attributes, it is therefore sufficient to check its state after method calls. In contrast, a relationship's consistency depends on the state of the objects in its connections, which may be changed individually and without notification. So, relationships can become inconsistent due to changes in one of the participating objects. There are several approaches to ensure relationship consistency across changes to single objects.

[Gre93] uses a rewriting technique to modify user transactions so that they preserve the database consistency. This requires that all changes are done in one transaction, disallowing nested or design transactions. [MAD92] propose to add pre- and post-conditions to methods that may raise exceptions. The exceptions are either processed immediately or at the end of the transaction, aborting it in case of a constraint violation. This leaves it to the schema designer to account for all relationships. All these approaches allow constraints of first order predicate logic.

In [Saa91], constraints are given in a temporal logic, and converted into labeled transition graphs. The idea is to track updates on objects by calculating markings in the graph in order to detect unrecoverable constraint violations. The relationship is consistent if the marking contains a final node. Constraints are given per class, so each object only has to store the markings. The paper only considers flat transactions.

We note that our object model with its clear distinction between objects and relationships fits perfectly to this approach: The transformation of temporal logic formulae into transition graphs proposed in [Saa91] requires the formula to be *reduct-free*, which means that configurations of objects do not change within

the scope of the formula. This is hard to achieve in the object model used in [Saa91] where relationships are modeled by complex objects. In contrast, our relationship mechanism offers exactly this situation, since connections exist only as long as their roles remain unchanged. Our goal is therefore to integrate this approach in our system.

Besides checking, the database system has to react on violations of the relationship constraints. Four reactions are possible: *abort* the transaction, *break* the inconsistent connection, *remove* one of the objects in its roles from the class of the role, thus also breaking the connection, or *repair* the damage. The first three reactions are save in a sense that the database will eventually reach a stable state, while *repair* is equivalent to triggers and has the same problems. The default reaction on all constraint violations is *abort*.

[DG90] distinguishes two possible reactions, namely to break the connection or to repair it. In [AGO91], relationships only have cardinality constraints and react with either abort, break or remove depending on the formulation of the violated constraint. [Rum87] proposes to break a connection, or abort the offending operation.

9 Summary and Outlook

The relationship mechanism presented here resembles existing approaches like [Rum87] or [AGO91], but integrates also parts of [DG90]. It extends all of them by derived relationships and a powerful persistence model, that gains its power especially from the derived relationships. With this ingredients, relationships are able to replace references in object-oriented databases, and offer flexibility and expressive power beyond them.

There are several directions for future work:

- The integration of an integrity checking mechanism is the next step. Especially the work in [Saa91] seems to fit well into our object model. Further research has to be done on the constraint violation reactions.
- Relationships allow to structure the model into separate objects and their connections. This offers an opportunity for inter-object parallelism, based e.g. on multi-level transactions. So, the influence of relationships on concurrency control and synchronization seems promising.
- The integration of relationships into the query languages left open problems with the combination of classes and relationships into classes, that require additional work.

References

[AGO91] Antonio Albano, Giorgio Ghelli, and Renzo Orsini. A relationship mechanism for a strongly typed object-oriented database programming language. In *Proceedings of the 17th International Conference on Very Large Data Bases*, pages 565–575, September 1991.

[BH93] Tamme D. Bowen and Kelly M. Hall. Towards a Better Understanding of Dylan. Technical report, Laboratory for Applied Logics, University of Idaho, May 1993.

[Cas94] Guiseppe Castagna. Covariance and Contravariance: Conflict without a Cause. Technical report, LIENS(CNRS)-DMI, 1994.

[Cat94] R.G.G. Cattell, editor. *The Object Database Standard: ODMG - 93*. Morgan Kaufmann, San Mateo, CA, 1994.

[Che76] P.P. Chen. The entity-relationship model — towards a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–36, 1976.

[CL95] Craig Chambers and Gary T. Leavens. Type Checking and Modules for Multi-Methods. Technical report, Dept. of Computer Science, Iowa State University, USA, August 1995.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[CY90] P. Coad and E. Yourdon. *Object-Oriented Analysis*. Prentice Hall, Englewood Cliffs, New Jersey, 1990.

[Deu91] O. Deux. The O_2 system. *Communications of the ACM*, 34(10):34–48, October 1991.

[DG90] Oscar Diaz and P. M. D. Gray. Semantic-rich User-defined Relationships as a Main Constructor in Object Oriented Databases. In *Conf. on Object-Oriented Databases*, Windermere, July 1990.

[ES90] M.A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, MA, 1990.

[GD91] A. Geppert and K.R. Dittrich. Objektstrukturen in Datenbanksystemen — oder: Auf der Suche nach voller Objektorientierung. In *Proceedings GI-Fachtagung "Datenbanksysteme für Büro, Technik und Wissenschaft", Kaiserslautern*, pages 421–429. Springer, Informatik-Fachberichte 270, 1991.

[GM95] A. Gawecki and F. Matthes. TooL: A persistent language integrating subtyping, matching and type quantification. FIDE Technical Report Series FIDE/95/135, FIDE Project Coordinator, Department of Computing Sciences, University of Glasgow, Glasgow G128QQ, 1995.

[Gre93] Paul W.P.J. Grefen. Combining Theory and Practice in Integrity Control: A Declarative Approach to the Specification of a Transaction Modification Subsystem. In *Proc. Int. Conf. on Very Large Databases*, pages 581–591, 1993.

[HFW90] A. Heuer, J. Fuchs, and U. Wiebking. OSCAR: An object-oriented database system with a nested relational kernel. In *Proc. of the 9th Int. Conf. on Entity-Relationship Approach, Lausanne*, pages 95–110. Elsevier, October 1990.

[HH91] C. Hörner and A. Heuer. EXTREM — The structural part of an object-oriented database model. Informatik-Bericht 91/5, TU Clausthal, October 1991.

[HHRW96] A. Heuer, C. Hörner, H. Riedel, and U. Wiebking. Syntax, semantics, and evaluation of the EXTREM object algebra. Informatik-Bericht, University of Rostock and University of Konstanz, 1996. In preparation.

[HS91] A. Heuer and M.H. Scholl. Principles of object-oriented query languages. In *Proceedings GI-Fachtagung "Datenbanksysteme für Büro, Technik und Wissenschaft", Kaiserslautern*, pages 178–197. Springer, Informatik-Fachbericht 270, 1991.

[Jon92] Rick Jones. Extended type checking in Eiffel. *Journal of Object-Oriented Programming*, pages 59–62, May 1992.

[KM90] A. Kemper and G. Moerkotte. Advanced query processing in object bases using access support relations. In *Proc. Int. Conf. on Very Large Databases*, pages 290–301, 1990.

[KMWZ91] A. Kemper, G. Moerkotte, H.-D. Walter, and A. Zachmann. GOM — a strongly typed, persistent object model with polymorphism. In *Proceedings GI-Fachtagung "Datenbanksysteme für Büro, Technik und Wissenschaft", Kaiserslautern*, pages 198–217. Springer, Informatik-Fachbericht 270, 1991.

[KNN89] W. Kim, J.-M. Nicolas, and S. Nishio, editors. *Proc. 1st International Conference on Deductive and Object-Oriented Databases, Kyoto*. Elsevier, December 1989.

[LLOW91] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The Objectstore Database System. *Communications of the ACM*, 34(10), October 1991.

[MAD92] Herve Martin, Michel Adiba, and Bruno Defude. Consistency Checking in Object Oriented Databases: a Behavioral Approach. In *Intl.Conf. on Information and Knowledge Management*, November 1992.

[Mey93] Bertrand Meyer. *Eiffel: The Language*. International Series in Computer Science. Prentice-Hall, Englewood Cliffs, 1993.

[RBE+90] J. Rumbaugh, M. Blaha, F. Eddy, W. Lorensen, and W. Premerlani. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, NJ, 1990.

[Rum87] James Rumbaugh. Relations as Semantic Constructs in an Object-Oriented Language. In *Proceedings of the ACM Conference on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA)*, pages 466–481, 1987.

[Saa91] G. Saake. Descriptive specification of database object behaviour. *Data and Knowledge Engineering*, 6:47–73, 1991.

[Sch96] Jürgen Schlegelmilch. Role-based Persistence. In Burkhard Freitag, Clifford B. Jones, Christian Lengauer, and Hans-Jörg Schek, editors, *Object-Orientation with Parallelism and Persistence*. Kluwer Academic Publishers, 1996. (submitted for publication).

[Ser91] Servio Logic Development Corp. *GemStone Product Overview*, 1991.

[SK91] M. Stonebraker and G. Kemnitz. The POSTGRES next generation database management system. *Communications of the ACM*, 34(10):78–92, October 1991.

[The95] Sven Thelemann. Assertion of Consistency Within a Complex Object Database Using a Relationship Construct. In Micheal P. Papazoglou, editor, *Proceedings of the 14th International Conference on Object-Oriented and Entity-Relationship Modeling, Gold Coast, Australia*, volume 1021 of *LNCS*, pages 32–43, Berlin, December 1995. Springer-Verlag.