CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2001; **13**:1013–1032 (DOI: 10.1002/cpe.610)

Compensation methods to support cooperative applications: A case study in automated verification of schema requirements for an advanced transaction model

David Spelt* and Susan Even

Centre for Telematics and Information Technology, University of Twente, Enschede, The Netherlands

SUMMARY

Compensation plays an important role in advanced transaction models, cooperative work and workflow systems. A schema designer is typically required to supply for each transaction T another transaction T^{-1} to semantically undo the effects of T. Little attention has been paid to the verification of the desirable properties of such operations, however. This paper demonstrates the use of a higher-order logic theorem prover for verifying that compensating transactions return a database to its original state. It is shown how an OODB schema is translated to the language of the theorem prover so that proofs can be performed on the compensating transactions. Copyright © 2001 John Wiley & Sons, Ltd.

KEY WORDS: object-oriented databases; formal methods; theorem proving

1. INTRODUCTION

Transaction models provide a basic mechanism to manage the concurrent access of a database. The traditional read/write model [1] is inadequate for modern applications of database technology, such as cooperative work and workflow [2–4]. For this reason, so-called *advanced transaction models* have been developed in the past decade. These models often rely on a notion of compensation, where for each operation, an 'inverse' operation has to be provided by the schema designer (see, for example, [5,6]). The intention is that a compensation operation semantically undoes the effects of the original

*Correspondence to: David Spelt, University of Twente, PO Box 217, 7500 AE Enschede, The Netherlands.

Contract/grant sponsor: SION, Stichting Informatica Onderzoek Nederland

operation—it does not merely restore a previous state. Although the correctness of compensation operations is often assumed, little attention has been devoted to the actual definition and verification of these operations.

Ten years ago, Korth *et al.* presented a formal approach to recovery by compensating transactions [7]. Their ideas have been incorporated in numerous transaction models since then. They gave three guidelines for the specification of compensating transactions. The first of their guidelines (so-called 'Constraint 1') is of interest to us here. Informally, this constraint asserts that if a transaction T (considered as a function on the database state) is immediately followed by its compensation CT, then the composed function 'T *followed by* CT' should be the same as the identity mapping. The constraint is assumed to hold in the later sections of their paper. As pointed out in [4], all transaction models using higher-level compensating subtransactions resort to the assumption that CT is defined by the schema designer and correctness of such definitions is implicitly assumed.

In this paper, we present a formal technique for the actual verification of Constraint 1, for a given (T, CT) pair. The technique makes use of the Isabelle theorem prover [8,9]. The approach taken is to define a translation from an object-oriented schema definition language (OASIS) to the formal notation of the theorem prover (higher-order logic—HOL). The translation encodes the semantics of the methods in HOL. This allows formal reasoning about the semantics of the methods. Compensation (Constraint 1) is expressed as a theorem to be proved. Verification relies on the mechanics of the theorem prover.

The paper highlights the mapping of some characteristic object-oriented features, namely class definitions, inheritance, late binding and oids. A case study is used to illustrate the verification of compensation requirements. For this, we consider a particular advanced transaction model that uses compensation, namely the CoAct model [10,11].

The rest of this paper is organized as follows. Section 2 introduces the compensation mechanism of the CoAct transaction model. Section 3 introduces the OASIS schema definition language and gives an example schema with compensation methods. Section 4 briefly introduces the Isabelle theorem prover. Section 5 discusses the translation of a database schema to higher-order logic. Section 6 discusses our practical experiments using Isabelle to verify the compensation methods of the example schema. Section 7 discusses related work. Section 8 concludes the paper.

2. COMPENSATION MECHANISM OF THE COACT MODEL

The CoAct transaction model is used to support cooperative applications [10–12]. The transaction model makes use of on an object-oriented data model, where compensation takes place at the method level. The CoAct transaction manager assumes that for each method `M`, a compensation method `UNDO_M` is to be provided *by the schema designer*. The *signature* of the compensation method `UNDO_M` is derived from the signature of `M`: the parameters of the method `UNDO_M` are the same as for method `M`, plus an additional parameter that corresponds to the result returned by `M`. If the effects of `M` are to be compensated, the transaction manager invokes `UNDO_M` with the same parameter values, plus the value returned by `M` [13]. These values are obtained from the history.

Conceptually, the compensation operation is executed immediately after the operation it compensates. If interim operations have been executed after the to-be-compensated-for operation, they must backward commute with (i.e. be independent of) it [14,15]. This means that the interim operations

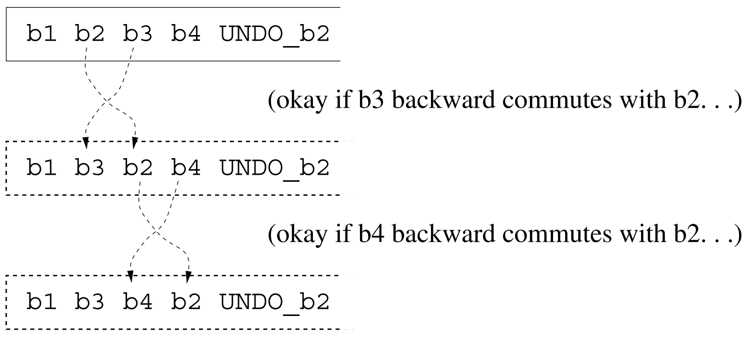
b1 b2 b3 b4 UNDO_b2
(okay if b3 backward commutes with b2. . .)
b1 b3 b2 b4 UNDO_b2
(okay if b4 backward commutes with b2. . .)
b1 b3 b4 b2 UNDO_b2

Figure 1. History-based compensation.

can be moved 'backwards' in the history, ahead of the to-be-compensated operation. This is illustrated in Figure 1. The history reordering is conceptual only. The backward-commutativity requirement guarantees that the reordered history is equivalent to the actual execution history.

The CoAct transaction model assumes that the compensation operations and the commutativity relations are provided by the schema designer. The *correctness* of this information is also the responsibility of the schema designer. The aim of our research is to provide a tool that can be used to help verify the semantics requirements on a database schema. In this paper, we look at verifying compensation. We will not further discuss commutativity.

3. SCHEMA DEFINITION LANGUAGE

We use a language called OASIS [16] to define the database schema and the compensation methods. Our intention in designing this language has been to work with a subset of a real-life OO database language (namely, O2C [17]) that includes 'enough' interesting features to be able to describe interesting examples, yet at the same time can be mapped to higher-order logic—the language of the theorem prover. The design of OASIS has also been based on the ODMG standard [18].

OASIS includes specification facilities for abstract and concrete classes, object creation, generic container types (namely, `set`$\langle\beta\rangle$), heterogeneous collections (of objects, not values), single inheritance of structure and methods and late binding, plus facilities for the specification of integrity constraints, queries (read-only methods) and transactions[†]. Some OO features are not supported in the OASIS language: there is no subtyping on primitive types, and there is no subtyping on records (i.e. no subtyping on '`struct`' types); the subtyping relation is fully induced by the class inheritance hierarchy. We do not consider relationships [18], although some relationships can be expressed as constraints, using quantifiers (see [19] for examples of OASIS constraints).

[†]Linguistically, transactions in OASIS are specified just like methods, but they do not have a receiver object.

Figure 2 gives class declarations for an example database, with operations for editing a generic graph structure. The example is representative of the core functionality of the SEPIA cooperative authoring system [20].

The schema serves to structure a hypermedia document. Atomic nodes (class ANode) contain the hypermedia data of the document; composite nodes (class CNode) contain atomic nodes, links and other composite nodes as elements. The class hierarchy ends at the AC (Atomic Contents) class, which is used for application-specific kinds and formats of media, such as text documents, audio and graphics. Objects of the AC class maintain, for example, a file-system handle to the hypermedia object, along with system commands (represented as strings) for displaying and editing the object. Class Element is abstract and provides a generalization of Node and Link. It declares an abstract method `isConnectedTo`, which has different implementations for classes Node and Link (see Figure 3). Three persistent roots are specified, which serve as extents for classes CNodes, Links, and ANodes.

3.1. Imperative method definitions and constructors

Method bodies are defined using an imperative command language, which includes attribute update, object creation (with constructors), sequential composition, conditional branch, bounded collection iteration [21] and non-recursive update method call. There is no explicit deletion operation in OASIS; *persistence by reachability* is used (as in O2 [17]).

Figures 3 and 4 give definitions for the methods and constructors of the schema. The following method is defined in the CNode class, and will be used as a running example in this paper.

```
boolean removeElement(Element n) {
  if (n != nil) and (n in elements) and (forall x in elements: not x.isConnectedTo(n))
  then { elements -= set(n); return true }
  else return false
};
```

The method removes one of the elements (i.e. a Node or Link) from the elements of a given CNode object, provided that that element is a member of the set 'elements' and that there are no other connections (within the same CNode object) to that element. The method returns *true* if the condition is met, indicating that the Element object has been successfully removed; otherwise *false* is returned.

The complex condition in the if part is expressed using OQL (Object Query Language [18]). Observe that high-level constructs (such as quantifiers) are available for manipulating sets. Also observe that the application of the abstract isConnectedTo method involves *late binding*, because classes Node and Link provide different implementations of this method. The implementation in class Node is given below:

```
boolean isConnectedTo(Element n) {
  return (n in incomingLinks) or (n in outgoingLinks)
}
```

The Boolean test 'n in incomingLinks' relies on the subtyping on objects induced by the inheritance hierarchy: it makes use of the fact that Link (the type of the elements of the incomingLinks collection)

```
class AC {
  attribute string refDir;
  attribute string showStmt;
  attribute string URL;
  . . .
};

abstract class Element {
  attribute string name;
  attribute int position;
  abstract boolean isConnectedTo(Element n);
  . . .
};

abstract class Node extends Element {
  attribute set⟨Link⟩ incomingLinks;
  attribute set⟨Link⟩ outgoingLinks;
  . . .
};

class ANode extends Node {
  attribute set⟨AC⟩ content;
  . . .
};

class CNode extends Node {
  attribute int size;
  attribute set⟨Element⟩ elements;
  boolean removeElement(Element n){ . . .};
  . . .
};

class Link extends Element {
  attribute Node from;
  attribute Node to;
  . . .
};

name set⟨CNode⟩ cnodes;
name set⟨Link⟩ links;
name set⟨ANode⟩ anodes;
```

Figure 2. Class declarations for the SEPIA schema.

```
string Element::changeNameTo(string s) {
 oldName = name; name = s; return oldName
};
boolean Node::addOutgoingLink(Link k) {
 if k == nil or k in outgoingLinks then { return false }
 else { outgoingLinks += set(k) ; return true } };
boolean Node::isConnectedTo(Link n) {
 return (n in incomingLinks) or (n in outgoingLinks)
};
AC ANode::createAC(string path, string stmt, string url) {
 ac = new AC(path, stmt, url); content += set(ac); return (ac)
};
ANode ANode::removeAC(AC ac) {
 if (ac in content) then { content -= set(ac); return this }
 else { return nil }
};
ANode CNode::createANodeIn(string s, int p) {
 ANode n = new ANode(s, p); elements += set(n); anodes += set(n); return n
};
boolean CNode::removeElement(Element n) {
 if (n != nil) and (n in elements) and  (forall x in elements: not(x.isConnectedTo(n)))
 then { elements -= set(n); return true }
 else { return false }
};
Link CNode::createLinkIn(Node na, Node nb, string s, int p) {
 if (na in elements and nb in elements) then {
   Link n = new Link(s, p); n.from = na; n.to = nb; elements += set(n); links += set(n); return n }
 else return nil
};
CNode Link::createCNodeIn(int p, int z, string s) {
 CNode n = new CNode(s, p, z);
 elements += set(n); cnodes += set(n); return n
};
boolean Link::isConnectedTo(Node n) {
 return (from == n) or (to == n)
};
```

Figure 3. Method definitions for the SEPIA schema.

```
AC(string path, string stmt, string url) { refDir = path; showStmt = stmt; URL = url };
ANode(string s, int p) { name = s; position = p };
CNode(string s, int p, int z) { name = s; position = p; size = z };
Link(string s, int p) { name = s; position = p};
```

Figure 4. Constructors for the SEPIA schema.

```
void Element::UNDO_changeNameTo(string s, string rtn) {
  name = rtn
};
void Node::UNDO_addOutgoingLink(Link k, boolean rtn) {
  if (rtn) then outgoingLinks -= set(k)
  else skip
};
void ANode::UNDO_removeAC(AC ac, ANode rtn) {
  if (rtn == this) then content += set(ac)
  else skip
};
void CNode::UNDO_createANodeIn(string s, int p, ANode rtn){
  elements -= set(rtn);  anodes -= set(rtn)
};
void CNode::UNDO_createLinkIn(Node a, Node b, string s, int p, Link rtn) {
 if (rtn!=nil) then { elements -= set(rtn);  links -= set(rtn) }
 else skip
};
void CNode::UNDO_removeElement(Element n, boolean rtn) {
  if (rtn) then elements += set(n)
  else skip
};
void CNode::UNDO_createCNodeIn(int p, int z, string s, CNode rtn) {
  elements -= set(rtn); cnodes -= set(rtn)
};
```

Figure 5. Compensation methods for the SEPIA schema.

and Element (the type of n) are compatible types [18]. In the later sections it is shown how these high-level (polymorphic) language features are dealt with by the analysis technique.

3.2. Compensation operations

Figure 5 gives definitions of compensation methods for the methods of the schema. One of these definitions, namely a method to compensate the effects of the removeElement method could be defined as follows.

```
void UNDO_removeElement(Element n, boolean rtn) {
  if (rtn) then { elements += set(n) }
  else skip
};
```

Observe the use of an extra parameter (rtn) for the return value; often this parameter is needed to define the compensation method. Also observe that, because complex (nested) types are supported, the return value can of course be, for example, a tuple or a set. In this particular example, care was taken when defining the *original* method so as to allow us to define the UNDO method.

Some kinds of operation, such as input and output operations, cannot be compensated, because they involve interaction with an external environment. For the SEPIA application on which our case study is based, this amounts to operations that implement the graphical user interface.

The definition of the compensation method gives rise to a proof obligation. For the above example, we have to prove that for any database state DB, receiver object o and input parameter value n, it is the case that $o.\mathsf{removeElement}(n)$ executed in the state DB, immediately followed by $o.\mathsf{UNDO\_removeElement}(n, r)$ results in a state DB' that is equivalent to DB (where r is the value returned by the removeElement method). To mechanically verify such a theorem, we first need to define a formal representation of the methods in the logic of the theorem prover. In the next section we briefly introduce Isabelle's higher-order logic specification language.

4. HIGHER-ORDER LOGIC (HOL) IN ISABELLE

Isabelle is a generic theorem prover [8,9], which supports reasoning in a variety of logics (the so-called *object logics*). The core of the system is a minimal fragment of higher-order logic (the so-called *meta logic*), which is used to represent the syntax and inference rules of the various object logics. Examples of predefined object logics are first-order logic, Zermelo–Fraenkel set theory and (typed) higher-order logic.

Higher-order logic is a generalization of first-order logic, where variables are allowed to range over functions and predicates. There are various kinds of higher-order logic, which may differ in the type system that is used, but the possibility to quantify over functions and predicates is distinctive. Isabelle's higher-order logic is inspired by the functional programming language ML [22].

Isabelle predefines many of the common datatypes in programming languages. Functional definitions in HOL resemble those of ML. However, HOL is not a programming language; it merely provides a way to reason about logical theories written in an ML-like style.

Below, we give a few details of predefined types and terms that will be used later on in this paper.

4.1. Types

Basic HOL types include Booleans (`bool`) naturals (`nat`) and integers (`int`). Function types are written as $\sigma \Rightarrow \tau$, where curried types $\sigma_1 \Rightarrow \cdots \Rightarrow \sigma_n \Rightarrow \tau$ can be abbreviated as $[\sigma_1, \ldots, \sigma_n] \Rightarrow \tau$. All functions in HOL are total, but partiality can be modelled using optional values (see Section 4.2). Other predefined type constructors include product types written as $\sigma \times \tau$, and parametrized collection types (such as sets and lists). The type system supports complex, nested data structures. For example, $(\mathtt{nat} \times \mathtt{bool})\ \mathtt{set}$ is a set of tuples where the first field is a number and the second field is a Boolean. Such types are important for object-oriented databases, where similar data structures are found [18].

HOL type schemes provide ML-like polymorphism. For example, we can use types such as $[\alpha, \alpha\ \mathtt{set}] \Rightarrow \alpha\ \mathtt{set}$ for the polymorphic function to insert a value of type α into an existing set.

4.2. Terms

HOL terms are made up from variables, constants, functions and function applications. Function applications are written in a curried style (i.e. $f\ a\ b$). Functions definitions are given using lambda (λ) notation.

The predefined constants used in this paper include logical connectives and quantifiers, if-then-else, set operations, let bindings and options. Options are frequently used to model partial functions in HOL. The option data type is defined as: '$\mathtt{datatype}\ \alpha\ \mathtt{option} = \mathtt{None} \mid \mathtt{Some}\ \alpha$.' Partial functions are modelled by the total function type '$\sigma \Rightarrow \tau\ \mathtt{option}$,' where `None` values are used to represent undefined values, and `Some` y is used to represent a defined value y. Options types are used to represent the database state as a partial function from object identifiers to object values—see Section 5). Optional values are also used to keep track of modifications to variables. For datatypes, case statements can be used, as in ML. Examples will be seen later.

4.3. Theorems, axioms and rules

Inference rules take the form $[|\ A_1; \ldots; A_n\ |] \Longrightarrow A$ to represent an implication with assumptions $A_1 \ldots A_n$ and conclusion A (the brackets are omitted if there is only one assumption) [8]. In HOL, the conclusion and the assumptions should both be terms of type `bool`. The following rules are axioms in HOL (we use standard Isabelle notation)

$$\begin{array}{rl}
[|\ P \to Q; P\ |] \Longrightarrow Q & (\to E)\\
(P \Longrightarrow Q) \Longrightarrow P \to Q & (\to I)\\
[|\ P \wedge Q; [|\ P; Q\ |] \Longrightarrow R\ |] \Longrightarrow R & (\wedge E)\\
[|\ P; Q\ |] \Longrightarrow P \wedge Q & (\wedge I).
\end{array}$$

The first rule describes *what* to infer from $(P \to Q)$; the second rule describes *how* to infer $(P \to Q)$. The first rule is called an *elimination rule*, while the second rule is called an *introduction rule*. Normally, elimination rules are used for reasoning forward from the assumptions, while introduction rules are for reasoning backwards from the conclusion. The rule language also supports quantifiers; in this case, the rules can have both bound and free variables. The reader is referred to [8] for details.

4.4. Proof tactics

Proofs are constructed by applying *tactics*. Isabelle predefines a variety of tactics, including basic tactics for interactive proof as well as powerful automatic tactics. The automatic tactics constitute the basis of our analysis technique for verifying compensation. A brief description of these algorithms is given below.

The *Simplifier* package permits rewriting with an arbitrary set of rewrite rules. Rewrite rules are rules with a conclusion of the form $A = B$, where each occurrence of A in the goal is replaced by B. Isabelle installs over a few hundred standard rewriting rules for HOL and new rules can be easily added. Many advanced features in term-rewriting are supported, such as the handling of conditional rewrite rules, the splitting of conditionals (if-then-else), and the use of congruence rules. We highly benefit from these features.

The *Classical Reasoner* automates deductive reasoning, based on a set of introduction and elimination rules. The default configuration of the tool includes machinery to reason about sets, lists, tuples, Booleans, etc. The tool implements a depth-first search strategy; variables introduced by the use of quantifiers can automatically be instantiated, and backtracking is performed between different alternative unifiers. Between deduction steps, the Simplifier can be used to permit rewriting. This provides a powerful tool for automated reasoning in HOL.

5. MODELLING THE SCHEMA IN HOL

An OASIS schema is translated to HOL in order to allow formal reasoning using a theorem prover. The translation implements a *semantic embedding* of the database language in HOL. There are two approaches for doing this: so-called *deep embedding* and *shallow embedding* [23]. A deep embedding means that the syntax of the source language, as well as the semantics functions (similar to, e.g., $\mathcal{E}[\![-]\!]$ in denotational semantics) are encoded *within* HOL to assign meanings to programs. A shallow embedding means that the syntax and semantics functions are not part of the logical representation itself; rather, an external schema translator is used to parse the source syntax directly to semantic structures in HOL. In this case, the schema translator 'computes' the semantics equations.

The level of embedding is important. A deep embedding allows meta-level reasoning about the semantics equations (i.e. abstract properties can be proved about the language constructs). A shallow embedding is restricted to reasoning about specific pieces of code (i.e. at the level of their denotations). We use a shallow embedding for OASIS, since compensation analysis amounts to proving that a given method undoes the effects of another. The translation is analogous to a semantics mapping, where the output is HOL notation. However, because the HOL notation is only intended to be reasoned about by the Isabelle system, some aspects of the mapping are declarative in nature. For example, the 'newness' of new oids only needs to be asserted as an assumption in proofs; new oid values do not need to be computed.

The essential ingredients of the translation (namely, a *generic* model of objects, a *database-specific* object type and methods as functions on the state) are discussed below. Additional details, including the definition of the translation algorithm and its implementation are found in [16].

$$
\begin{array}{l}
\texttt{oids} :: (\texttt{oid} \Rightarrow \beta\ \texttt{option}) \Rightarrow \texttt{oid set} \\
\texttt{eval} :: [\beta\ \texttt{option}, \beta \Rightarrow \texttt{bool}] \Rightarrow \texttt{bool} \\
\texttt{get} :: [\beta\ \texttt{option}, \beta \Rightarrow \alpha] \Rightarrow \alpha \\
\texttt{set} :: [(\texttt{oid} \Rightarrow \beta\ \texttt{option}), \texttt{oid}, \beta \Rightarrow \beta] \Rightarrow (\texttt{oid} \Rightarrow \beta\ \texttt{option}) \\
\texttt{smash} :: [(\texttt{oid} \Rightarrow \beta\ \texttt{option}), (\texttt{oid} \Rightarrow \beta\ \texttt{option})] \Rightarrow (\texttt{oid} \Rightarrow \beta\ \texttt{option}) \\
\texttt{apply\_to\_all} :: [\alpha\ \texttt{set}, [\alpha, \texttt{oid}] \Rightarrow \beta\ \texttt{option}] \Rightarrow (\texttt{oid} \Rightarrow \beta\ \texttt{option}) \\
\texttt{new} :: [\texttt{oid}, \beta] \Rightarrow (\texttt{oid} \Rightarrow \beta\ \texttt{option}) \\
\texttt{skip} :: (\texttt{oid} \Rightarrow \beta\ \texttt{option})
\end{array}
$$

Figure 6. Generic operations for objects.

5.1. The generic object store

The generic object store provides a schema-independent notion of state. In HOL, the store is represented as a (partial) function type that maps object identifiers to storage cells: `oid` $\Rightarrow$ β `option`. The type variable β gets instantiated with a schema-specific object type, which reflects the class hierarchy of a given schema (see Section 5.2). The `option` type in the codomain includes the cases '`None`' (to represent *undefined* results) and '`Some` v' (to represent *defined* results, where the actual value v is supplied as an argument).

Generic operations for retrieval and update are defined on the object store. Figure 6 lists the operations, along with their signatures. The operations `oids`, `get`, and `eval` are used to retrieve information from the state. For example, '`get` $(\sigma\ x)\ f$' applies the function f to the value of the object with identity x in the state σ.

The other operations in the figure are used to update the state; they result in a 'little' object store, which comprises local changes to the state. The operation `set` takes an object store, an oid and a function that is applied to the associated storage cell's content. The result is a little object store with one binding, that of the modified object. The operation `smash` takes two object stores; the result is their merge, with the bindings in the second argument overriding those of the first. Examples that illustrate the use of the operations in Figure 6 to encode method definitions are shown in later sections.

Using the Isabelle system, we have proved a number of theorems about these operations (an overview can be found in the first author's dissertation [16]). Rules are derived mainly for rewriting purposes. For example, the following rule shows how to simplify applications of `get` to a store with a modified value of the object with identity *idb*.

$$
\begin{array}{ll}
\texttt{get}\ ((\texttt{smash}\ os_1\ (\texttt{set}\ os_2\ idb\ f))\ ida)\ g = & \texttt{if}\ (idb = ida) \wedge idb \in \texttt{oids}\ os_2 \\
& \texttt{then get}\ (os_2\ ida)\ (g \circ f) \\
& \texttt{else get}\ (os_1\ ida)\ g
\end{array}
\tag{1}
$$

5.2. Class declarations as a datatype in HOL

The schema translator generates a HOL datatype that captures the structural information about the objects in a given database schema. This datatype is used to instantiate the type variable β that appears

```
[| eval o (λz · case z of AC refDir showStmt URL ⇒ p1
                        | ANode name pos ins outs content ⇒ p2
                        | CNode name pos ins outs size elts ⇒ p3
                        | Link name pos from to ⇒ p4);
   !! refDir showStmt URL · [| o = AC refDir showStmt URL; p1 |] ⇒ R;
   !! name pos ins outs content · [| o = ANode name pos ins outs content; p2 |] ⇒ R;
   !! name pos ins outs size elts · [| o = CNode name pos ins outs size elts; p3 |] ⇒ R;
   !! name pos from to · [| o = Link name pos from to; p4 |] ⇒ R |] ⇒ R
```

Figure 7. Elimination rule for `eval` for the schema.

in the generic object store. To describe the domain of object values, we use a variant type. Cases are introduced for each of the concrete classes in the database schema[‡]. The class names are used as the type constructors. For the example schema, the following `object` type is obtained.

```
datatype object =  AC string string string
                 | ANode string int (oid set) (oid set) (oid set)
                 | CNode string int (oid set) (oid set) int (oid set)
                 | Link string int oid oid
```

The abstract classes of the schema (Element and Node) are not listed in the type, since instances of these classes cannot be created. Structural information for an object (attribute values) is supplied as an argument to its data type constructor. This information includes all attributes inherited from superclasses. Class references in compound objects appear as 'pointer' references in the form of oid-values. This accommodates object sharing and heterogeneous sets: representations of objects from different classes can be grouped in one and the same set, since they all have the same Isabelle type `oid`. The constructors of type `object` provide *run-time* type information. Type decisions are encoded using *case splits* to examine the type tag (for details, see [19]).

As for the generic object store, rules are also derived for the database-specific store, namely for the introduction and elimination of the `eval` predicate. For example, Figure 7 shows the elimination rule for `eval`-elimination for the example schema. Such rules enable Isabelle to automate deductive reasoning using its Classical Reasoner tool. An example of the application of this rule is shown in Section 6.

5.3. Methods as functions on the database state

Methods are represented as functions in HOL. These representations are automatically generated by a schema translator. A method maps an input object store, persistent roots, an oid (the `this` parameter), actual parameter values and any required new oids to a *tuple*. The components of this tuple depend on whether the method is a read-only method, or an update.

[‡]These types do not necessarily correspond to the ones listed as persistent roots (e.g. class AC is concrete, but has no persistent root).

5.3.1. *Read-only methods*

As an example, we look at the representation of the implementation of the isConnectedTo method for class Node.

```
Node_isConnectedTo ≡
λos : OS · λcnodes : oid set · λlinks : oid set · λanodes : oid set ·
  λthis : oid · λn : oid · n ∈ (get (os this) inLinksOf) ∨ n ∈ (get (os this) outLinksOf)
```

The method is defined as a function acting on object store, persistent roots, a receiver oid and parameter value (the oid ‘n’); the output is its Boolean return value.

In the schema translation, the name of a method is prefixed with the name of its defining class. This prevents name clashes due to overloading. The type ‘OS’ abbreviates the database-specific object store type ‘oid ⇒ object option’. In the body of the function, the get operation (see Figure 6) is used to look up the values of the incomingLinks and outgoingLinks attributes of the receiver object. The names *inLinksOf* and *outLinksOf* stand for functions that encode the actual attribute selections, using case-splits. For example, *inLinksOf* expands to the following HOL code.

```
inLinksOf ≡
 λ val · case val of  AC refDir showStmt URL ⇒ arbitrary
                    | ANode name pos ins outs content ⇒ ins
                    | CNode name pos ins outs size elts ⇒  ins
                    | Link name pos from to ⇒ arbitrary
```

The get application returns the value of the incomingLinks attribute if ‘n’ is Some-tagged in the object store, with the right type (ANode or CNode); otherwise an *arbitrary* value is returned. The *arbitrary* constant is available for all types and provides a common way of dealing with undefined function results in HOL. For example, the constant is used to ‘define’ the head of the empty list in the standard Isabelle/HOL library.

5.3.2. *Update methods*

Methods with side effects are more complicated. In this case, the output is a *tuple* that includes a component for the return value of the method and an additional component for updates of the object store. It also includes components for modifications to the persistent roots and the method parameters.

Updates of the object store are encoded using *delta values* [24–26]. A delta value describes tentative changes to the object store, which corresponds to a ‘difference’ between database states. A delta value can include changes to multiple objects (methods in our language can update objects other than the receiver). We use the symbol Δ as an abbreviation in later examples. In our formal model, delta values are constructed using if-then-else and the operations skip, set, new, smash and apply_to_all shown in Figure 6.

Updates to persistent roots and parameters are by value-result: modifications to the roots and parameters are returned as Some-tagged values, where the value represents the updated root or parameter; ‘None’ is returned as a result if no modifications are made. The *return* value of the method application is given in the last position of the tuple. As an example, we look at the HOL definition of the removeElement method discussed in the previous sections.

```
CNode_removeElement ≡
λos : OS · λcnodes : oid set · λlinks : oid set · λanodes : oid set ·
  λthis : oid · λn : oid · if condition
                            then (set os this delElement, None, None, None, None, None, True)
                            else (skip, None, None, None, None, None, False)
```

The output of this function is a 7-tuple. Since no changes occur to roots and variables, their positions are filled with None values. The first component of the tuple is a Δ value. In this example, only the receiver object is modified. The modification is encoded using 'set': the function argument *delElement* encodes the modification to the elements field of the receiver object, by removing oid 'n' from the set. Function *delElement* expands to

```
delElement ≡
 λ val · case val
  of  AC refDir showStmt URL ⇒ AC refDir showStmt URL
    | ANode name pos ins outs content ⇒ ANode name pos ins outs content
    | CNode name pos ins outs size elts ⇒ CNode name pos ins outs size (elts − {n})
    | Link name pos from to ⇒ Link name pos from to
```

Note that the identifier *delElement* is merely an abbreviation for the tuple transformer function. The variable n is bound by the context (e.g. by the λ in the definition of CNode_removeElement).

The *condition* is the translation of if-part in the definition of the removeElement method, which involves *late binding* (see Section 3.1). It is represented using a number of applications of the 'eval' and 'get' operations of the theory of objects.

```
condition ≡
 (eval (os n) isElement) ∧ (n ∈ (get (os this) elementsOf) ∧
 (∀x ∈ (get (os this) elementsOf) |
   ¬ ( if (eval (os x) isLink) then (Link_isConnectedTo os cnodes links anodes x n)
       else ( if (eval (os x) isNode)
              then (Node_isConnectedTo os cnodes links anodes x n)
              else arbitrary)))
```

The first occurrence of eval represents the 'n != nil' comparison. The eval operation is applied to the storage cell associated with n, and the function *isElement*: it returns True if n is Some-tagged in the object store, with the right type. The second subexpression checks that n is in the 'elements' field of the 'this' object (the function *elementsOf* is defined analogously to the *inLinksOf* function shown above). The last subexpression is the HOL representation of the following test in the method definition: forall x in elements : not(x.isConnectedTo(n))). This test involves *late binding*: based on the (run-time) type of x, the appropriate version of the isConnectedTo method is applied; an 'arbitrary' value results for an ill-typed or nil x.

5.3.3. *Compensation methods*

Compensation methods are represented in HOL in the same way as normal methods. The method UNDO_removeElement is represented as follows.

Table I. Experimental results for method compensation.

Class	Method	Compensation	Proof time (s)
`Element`	`changeNameTo`	`UNDO_changeNameTo`	5.25
`ANode`	`createAtomicContentIn`	`UNDO_createAtomicContentIn`	22
`ANode`	`removeAtomicContent`	`UNDO_removeAtomicContent`	15
`Node`	`addIncomingLink`	`UNDO_addIncomingLink`	7
`Node`	`addOutgoingLink`	`UNDO_addOutgoingLink`	8
`CNode`	`createANodeIn`	`UNDO_createANodeIn`	82
`CNode`	`createCNodeIn`	`UNDO_createCNodeIn`	85
`CNode`	`createLinkIn`	`UNDO_createLinkIn`	107
`CNode`	`removeElement`	`UNDO_removeElement`	14

```
CNode_UNDO_removeElement ≡
λos : OS · λcnodes : oid set · λlinks : oid set · λanodes : oid set ·
  λthis : oid · λn : oid · λrtn : bool ·
        if condition then (set os this addElement, None, None, None, None, None, ())
        else (skip, None, None, None, None, None, ())
```

Observe that this method takes the return value of the removeElement method as an additional argument. It returns the unit value (()), which is used for void-typed methods. The function *addElement* is defined similarly to the function *delElement* shown above: the expression (`elts` − {`n`}) is replaced by (`elts` ∪ {`n`}).

The next section shows how the Isabelle theorem prover can be used to verify that the compensation method is correct. It makes use of the HOL definitions in doing this.

6. VERIFICATION OF METHOD COMPENSATION USING ISABELLE

In order to analyse the method definitions in the database schema, the desired compensation requirements are given as proof goals to the Isabelle theorem prover. The verification of these goals relies on the use of standard Isabelle proof tactics, including a simplification tactic and a tactic for classical reasoning [8,9]. Using these tactics we have verified all the methods in the database schema. The proof times varied from a few seconds up to several minutes of proof time on a normal workstation (see Table I).

6.1. Example proof

Let us examine one of the proofs, namely the proof of the (`removeElement`, `UNDO_removeElement`) method pair in more detail below. The goal that needs to be verified is

Level 0

```
(eval (os this) $isCNode$) $\wedge$ (x $\in$ oids os) $\rightarrow$
let ($\Delta$, $cnodes_1$, $links_1$, $anodes_1$, $this_1$, $n_1$, $rtn_1$) =
        CNode_removeElement os cnodes links anodes this n;
    ($\Delta^{-1}$, $cnodes_2$, $links_2$, $anodes_2$, $this_2$, $n_2$, $rtn_2$) =
        CNode_UNDO_removeElement(smash os $\Delta$)
                (case $cnodes_1$ of None $\Rightarrow$ cnodes | Some y $\Rightarrow$ y)
                (case $links_1$ of None $\Rightarrow$ links | Some y $\Rightarrow$ y)
                (case $anodes_1$ of None $\Rightarrow$ anodes | Some y $\Rightarrow$ y) this n $rtn_1$
in get ((smash os (smash $\Delta$ $\Delta^{-1}$)) x) $idf$ = get (os x) $idf$ $\wedge \cdots$
```

Informally, this goal states that the compensation method returns the database to its original state, for an arbitrary input database state, receiver object and parameter values (the variables in the goal are implicitly universally quantified). The equivalence of the stores, before and after the application of the methods, is asserted by the first subformula in the `in` part of the `let` statement. It asserts that any look up of the value of an object with oid '`x`' in the modified store should result in the same value as the value obtained by a look up of that object in the initial store '`os`' (*idf* stands for the identity function). The other subformulae assert the equivalence of the persistent roots, before and after the updates, using case statements.

The proof starts by opening the definitions of the method and its compensation using the Rewriter. In general this results in a huge proof term. Fortunately, Isabelle can simplify this rather complex goal using its Simplifier tool. Isabelle rewrites the above goal in various steps by applying rules such as (1) and other standard rewrite rules. The end result is the following rewritten goal.

Level 1

```
eval (os this) $isCNode$ $\wedge$ x $\in$ oids(os) $\rightarrow$
k $\in$ get (os this) $ElemOf$ $\wedge$ x = this $\rightarrow$
      get (os this) ($addElement \circ delElement$) = get (os this) $idf$
```

The proof continues by applying standard introduction and elimination rules using the classical reasoning tactic. Furthermore, the use of the database specific elimination rule of `eval` (see Figure 7) splits the goal into four subgoals, one for each concrete class in the schema. We obtain the following list of subgoals.

Level 2

```
1. !! refDir showStmt URL.
   [| os this = Some(AC refDir showStmt URL); False;
     x $\in$ oids(os); n $\in$ get (os this) $ElemOf$;
     x = this |] $\Rightarrow$ get (os this) ($addElement \circ delElement$) = get (os this) $idf$

2. !! name pos ins outs content.
   [| os this = Some(ANode name pos ins outs content); False;
     x $\in$ oids(os); n $\in$ get (os this) $ElemOf$;
     x = this |] $\Rightarrow$ get (os this) ($addElement \circ delElement$) = get (os this) $idf$

3. !! name pos ins outs size elts.
   [| os this = Some(CNode name pos ins outs size elts); True;
     x $\in$ oids(os); n $\in$ get (os this) $ElemOf$;
     x = this |] $\Rightarrow$ get (os this) ($addElement \circ delElement$) = get (os this) $idf$
```

```
4. !! name pos from to.
   [| os this = Some(Link name pos from to); False;
     x ∈ oids(os); n ∈ get (os this) ElemOf;
     x = this |] ⇒ get (os this) (addElement ∘ delElement) = get (os this) idf
```

By static typing of the receiver, we know that only the third subgoal is relevant, namely the one for `CNode`. The other subgoals for `AC`, `ANode` and `Link` denote wrongly typed cases (the receiver is known to be of static type CNode); the proof of these subgoals is trivial because *false* is in their list of assumptions. The remaining third subgoal is non-trivial, but can also be solved. Finally, Isabelle returns the following message to indicate that the proof has succeeded.

```
Level 3
No subgoals!
```

7. RELATED WORK AND EXTENSIONS

Our verification framework is inspired by the pioneering work of Sheard and Stemple [27], which applies automated theorem proving techniques to the verification of transaction safety in the context of relational databases. Our work address a number of issues that do not arise in relational databases, such as *object sharing*, *object creation*, *inheritance* and *heterogeneity*. Benzaken *et al.* have studied the problem of method safety for object-oriented databases [28]. They apply a technique based on abstract interpretation. To verify that a method satisfies a particular constraint, a sufficient precondition is derived for which automated verification is attempted using a theorem prover. This limits their approach to safety analysis. In this paper we have shown that our formal framework can also be used to prove compensation requirements of method code.

Ammann *et al.* apply formal methods to the semantic-based decomposition of transactions [29]. In their work, the Z specification language [30] is used to formally define transactions. They focus on decomposing a transaction into steps that preserve certain properties, including database integrity constraints and a compensation property. The analyses and proofs in [29] are done by hand, for the specific example schema used in the paper. The observation is made that for real life applications '*it will be necessary to automate to the extent possible the process of discharging the proof obligations*'. Chkliaev *et al.* use the PVS system [31] to obtain mechanized support for the verification of concurrency control protocols, such as two-phase locking (2PL) [32]. The atomic operations they consider are limited to read and write actions on a database. The traditional notion of R–W and W–W conflict is used. Our approach could possibly be used in combination with theirs, as a building block to support the verification of semantics-based concurrency control protocols, and multilevel transaction management [3].

Support for the verification of compensation requirements provides an important first step to support advanced transaction models at the schema design level. As discussed in Section 2, compensation should be combined with backward commutativity in order to support cooperative work, because intermittent operations might be executed before a compensation operation is attempted. Use of the theorem-prover-based analysis technique to support the verification of commutativity requirements has also been under investigation.

We have demonstrated the feasibility of our approach for one particular case study. The obvious question is whether the approach scales up to different and larger examples. The example methods in our case study cover a diversity of object-oriented language features. However, the analysis of methods consisting of 15 to 20 lines of code obviously increases the size of proof goals manipulated by the theorem prover. The time required to find a proof becomes a limiting factor in the utility of the analysis tool. This requires further investigation.

The OASIS database language has been purposefully designed such that it covers the data types already provided by the standard distribution of the Isabelle/HOL theorem prover. The extensions we made to the theorem prover address mainly the development of a theory of objects in HOL. Our goal has been to demonstrate the feasibility of using a theorem prover for the automated analysis of database methods, not to do research in mathematics. The development of new data type theories is actively addressed by the theorem-prover community and as new theories become available, they can be readily integrated in our tool, because of its orthogonal design (this amounts to extending the parser and schema translator with additional rules). Some important database language features are not yet fully covered by the theorem prover, and for this reason, they are not yet available in OASIS. This includes bags and aggregate operations on collections (e.g. sum, count, average). At present, only set and list collection types are directly supported in HOL. A theory of bags—multisets—is under development, but is still preliminary.

A possible extension of our work is to consider equivalence relations other than state-based equality, which is used in this paper. State-based equality of two states means that the values of the attributes of every object are the same in both states. This is a rather strong requirement, which is often relaxed in the transaction model literature [4,33–35] where bisimulation is used instead. Bisimulation of two states means that the states are 'observably' indistinguishable [36]. Bisimulation is weaker than actual equality of states, because it may leave certain internal differences undetected (e.g. private attributes). In principle, the approach outlined in this paper extends to equality based on bisimulation. However, it is likely that proofs will require more 'intelligence' from the user of the proof system, because proving that two states are bisimilar involves finding a bisimulation relation $\approx$ over states (see [36] for details). For a typical database schema, this is non-trivial.

8. CONCLUSIONS

In this paper, we have discussed the use of a theorem prover to verify compensation requirements for an object-oriented database schema. The analysis technique is based on the *semantics* of the database operations, with respect to a formal model of memory that reflects the type-tagged storage structure of an implementation. Issues such as object creation, inheritance and late binding of methods (all of which are linked to run-time type information) are accommodated by the formal model. The tool is built using the Isabelle theorem prover [8]. Our tool was initially developed to verify consistency requirements. In [19], we have shown that the tool can be used to verify that a method preserves a number of static integrity constraints. The compensation analysis discussed in this paper uses the same automated proof procedure, which is based on standard machinery provided by the Isabelle theorem prover. Our work demonstrates that *different* requirements on the semantics of database operations can be verified within a single formal framework.

We suspect that it is not possible to characterize the kinds of method code for which the analysis can be performed automatically (e.g. 'methods with conditionals work' or 'only updating the receiver object works'). This situation arises because of incompleteness[§]. If the automated proof procedure does not find a proof, it returns the goals it cannot solve. In these cases, human interaction is needed in order to (try to) complete the proof. Unfortunately, interactive proof requires detailed knowledge of both the theorem prover and the semantic embedding of the database language in HOL. An unprovable goal typically corresponds to an error in the database schema, but identifying such an error, and the subsequent correction of the code, requires human intelligence and skill (just as for a pencil and paper proof). A theorem prover works as a proof *assistant*. It can most effectively be used to identify those compensation methods that are correct, and further, to identify those compensation methods that *might not be* correct. For the latter, the specifier must study and revise the method code, and attempt the proof again.

REFERENCES

1. Eswaran K, Gray J, Lorie R, Traiger I. The notions of consistency and predicate locks in a database system. *Communications of the ACM* 1976; **19**(11):624–633.
2. Elmagarmid AK (ed.). *Database Transaction Models for Advanced Applications*. Morgan Kaufmann, 1992.
3. Özsu MT. Transaction models and transaction management in object-oriented database management systems. *Advances in Object-Oriented Database Systems* (*Computer and Systems Sciences*, vol. 130). Springer, 1994; 147–184.
4. Veijalainen J, Wäsch J, Puustjärvi J, Tirri H, Pihlajamaa O. Transaction models in cooperative work—an overview. *Transaction Management Support for Cooperative Applications*, ch. 3. Kluwer Academic, 1997; 27–58.
5. Chen Q, Dayal U. Failure handling for transaction hierarchies. *Proceedings of ICDE*, Birmingham, U.K., April 1997; 245–254.
6. Martin CP, Ramamritham K. Delegation: Efficiently rewriting history. *Proceedings of ICDE*, Birmingham, U.K., April 1997; 266–275.
7. Korth HF, Levy E, Silberschatz A. A formal approach to recovery by compensating transactions. *Proceedings of the 16th VLDB Conference*, Brisbane, Australia, 1990; 95–106.
8. Paulson LC. *Isabelle: A Generic Theorem Prover* (*Lecture Notes in Computer Science*, vol. 828). Springer, 1994.
9. Nipkow T. Tutorial on Isabelle/HOL. Electronic document available at www.cl.cam.ac.uk/Research/HVG/Isabelle [1998].
10. Rusinkiewicz M, Klas W, Tesch T, Wäsch J, Muth P. Towards a cooperative transaction model—the cooperative activity model. *Proceedings of the 21st VLDB Conference*, Zurich, Switzerland, September 1995.
11. Klingemann J, Tesch T, Wäsch J, Klas W. The TransCoop transaction model. *Transaction Management Support for Cooperative Applications*, ch. 7. Kluwer Academic, 1997; 149–172.
12. Wäsch J, Klas W. History merging as a mechanism for concurrency control in cooperative environments. *Proceedings of the IEEE Workshop on Research Issues in Data Engineering: Interoperability of Nontraditional Database Systems*, 1996; 76–85.
13. Even S, Faase F, Kaijanranta H, Klingemann J, Lehtola A, Pihlajamaa O, Tesch T, Wäsch J. Deliverable VII.1: Design of the TransCoop demonstrator system. *Report TC/REP/GMD/D7-1/704*, Esprit Project No. 8012, 1996.
14. Weihl WE. Commutativity-based concurrency control for abstract data types. *IEEE Transactions on Computers* 1988; **37**(12):1488–1505.
15. Weihl WE. The impact of recovery on concurrency control. *Journal of Computer and System Sciences* 1993; **47**:157–184.
16. Spelt D. Verification support for object database design. *PhD Thesis* (*CTIT PhD Series*, No. 99-24), University of Twente, Enschede, The Netherlands, 1999.
17. Delobel BC, Kanellakis P. *Building an Object-oriented Database System: The Story of O2*. Morgan Kaufmann, 1992.
18. Cattell RGG, Barry DK (eds.). *The Object Database Standard: ODMG 2.0*. Morgan Kaufmann: San Francisco, CA, 1997.
19. Spelt D, Even S. A theorem prover-based analysis tool for object-oriented databases. *Proceedings of the 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'99* (*Lecture Notes in Computer Science*, vol. 1579). Springer, 1999; 375–389.

[§]The compensation problem is, in general, undecidable. See, for example, [37] and [38], for analogous foundational results on transaction safety and commutativity analysis even for very simple transaction languages.

20. Klas W, Aberer K, Neuhold EJ. Object-oriented modelling for hypermedia systems using the VODAK modelling language (VML). *Advances in Object-Oriented Database Systems* (*Computer and Systems Sciences*, vol. 130). Springer, 1994; 389–443.
21. Qian X. The expressive power of the bounded-iteration construct. *Acta Informatica* 1991; **28**(7):631–656.
22. Paulson LC. *ML for the Working Programmer* (2nd edn). Cambridge University Press: New York, 1996.
23. Boulton R, Gordon A, Gordon M, Harrison J, Herbert J, van Tassel J. Experience with embedding hardware description languages in HOL. *Proceedings of the International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience* (*IFIP Transactions*, vol. A-10). North-Holland, 1992.
24. Doherty M, Hull R, Derr M, Durand J. On detecting conflict between proposed updates. *Proceedings of the International Workshop on Database Programming Languages (DBPL)*, Gubbio, Italy, September 1995.
25. Doherty M, Hull R, Rupawalla M. Structures for manipulating proposed updates in object-oriented databases. *Proceedings of the ACM SIGMOD Symposium on the Management of Data*, June 1996; 306–317.
26. Doherty M. A multistate service based on deltas and its application to support collaborative work. *PhD Thesis*, University of Colorado, Boulder, Colorado, 1998.
27. Sheard T, Stemple D. Automatic verification of database transaction safety. *ACM Transactions on Database Systems* 1989; **14**(3):322–368.
28. Benzaken V, Schaefer X. Static integrity constraint management in object-oriented database programming languages via predicate transformers. *Proceedings of ECOOP* (*Lecture Notes in Computer Science*, vol. 1241). Springer: Dublin, 1997; 60–85.
29. Ammann P, Jajodia S, Ray I. Applying formal methods to semantic-based decomposition of transactions. *ACM Transactions on Database Systems* 1997; **22**(2):215–254.
30. Spivey JM. *The Z Notation: A Reference Manual* (2nd edn). Prentice Hall, 1992.
31. Owre S, Rajan S, Rushby JM, Shankar N, Srivas MK. PVS: Combining specification, proof checking, and model checking. *Computer-Aided Verification (CAV '96)* (*Lecture Notes in Computer Science*, vol. 1102), July/August 1996.
32. Chkliaev D, Hooman J, van der Stok P. Serializability preserving extensions of concurrency control protocols. *Proceedings of the Andrei Ershov International Conference on Perspectives of Systems Informatics* (*Lecture Notes in Computer Science*), Novosibirsk, 1999. Springer.
33. Herlihy M. Apologizing versus asking permission: Optimistic concurrency control for abstract data types. *ACM Transactions on Database Systems* 1990; **15**(1):96–124.
34. Weikum G. Extending transaction management to capture more consistency with better performance. Invited paper, *Proceedings of the 9th French Database Conference*, Toulouse, 1993.
35. Lynch N, Merrit M, Weihl W, Fekete A. *Atomic Transactions*. Morgan Kaufmann, 1994.
36. Jacobs B. Coalgebraic reasoning about classes in object-oriented languages. *Electronic Notes in Theoretical Computer Science*, **11**, www.elsevier.nl [1998].
37. Benedikt M, Griffin T, Libkin L. Verifiable properties of database transactions. *Proceedings of Principles of Database Systems (PODS)*, 1996; 117–127.
38. Ibarra O, Diniz P, Rinard M. On the complexity of commutativity analysis. *Proceedings of the International Computing and Combinatorics Conference*, 1996.
39. Even S, Spelt D. Compensation methods to support generic graph editing: A case study in automated verification of schema requirements for an advanced transaction model. *Proceedings of the 1st ECOOP Workshop on Object-oriented Databases*, Lisbon, Portugal, 15 June 1999.
40. Spelt D, Balsters H. Automatic verification of transactions on object-oriented databases. *Proceedings of the Workshop on Database Programming Languages (DBPL)*, Estes Park, Colorado, 1997.