# Compensation methods to support generic graph editing: A case study in automated verification of schema requirements for an advanced transaction model

Susan Even* and David Spelt

Centre for Telematics and Information Technology
University of Twente, P.O. Box 217, 7500 AE Enschede, The Netherlands
{seven,spelt}@cs.utwente.nl

Compensation plays an important role in advanced transaction models, cooperative work, and workflow systems. However, compensation operations are often simply written as $a^{-1}$ in transaction model literature. This notation ignores any operation parameters, results, and side effects. A schema designer intending to use an advanced transaction model is expected (required) to write correct method code. However, in the days of cut-and-paste, this is much easier said than done. In this paper, we demonstrate the feasibility of using an off-the-shelf theorem prover (also called a proof assistant) to perform *automated* verification of compensation requirements for an OODB schema. We report on the results of a case study in verification for a particular advanced transaction model that supports cooperative applications. The case study is based on an OODB schema that provides generic graph editing functionality for the creation, insertion, and manipulation of nodes and links.

## 1 Introduction

Traditional transaction models provide a basic mechanism to manage the concurrent access of a database. However, traditional models are inadequate for modern applications of database technology, such as cooperative work and workflow [Elm92, Özsu94, VWP+97]. 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 (see, for example, [CD97, MR97]). The intension is that a compensation operation semantically undoes the effects of the original 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.

Nine years ago, Korth *et al* presented a formal approach to recovery by compensating transactions [KLS90]. Their ideas have been incorporated in numerous transaction models since then, including the one we consider in this paper ([KTWK97]). 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 a compensation $CT$, then the composed function '*$T$ followed by $CT$*' should be an identity mapping. After this constraint is introduced in [KLS90], it is assumed to hold for all compensating transactions in the later sections of their paper. Our work provides tool support for the verification of such a constraint, at the method level, where a transaction is seen as a sequence of method invocations.

We demonstrate the feasibility of using an off-the-shelf theorem prover to *automate* the verification of compensation requirements. The analysis is based on the *semantics* of OODB methods with respect to a formal model of a persistent object store. We translate imperative method

*Research supported by SION, Stichting Informatica Onderzoek Nederland.

code to purely functional definitions in higher-order logic (HOL), which can be reasoned about using a theorem prover. In [SE99], we have shown how the Isabelle theorem prover ([Isa]) can be used to automatically verify that a given method preserves the integrity constraints specified for a schema. In this paper, we investigate the verification of compensation in the same formal framework. We consider compensation in the context of a particular advanced transaction model, namely the CoAct model [RKT+95, KTWK97]. A case study is used to illustrate the verification of compensation requirements for this transaction model.

**Compensation and commutativity requirements of CoAct.** The CoAct model is used to support cooperative applications [RKT+95, KTWK97]. Each user has a private workspace, with an associated workspace history. Transaction management support encompasses the private workspaces as well as a common workspace. The model makes use of three semantics-based transaction management ideas: *backward commutativity* [Wei88], *compensation* [KLS90], and *forward commutativity* [Wei88, LMWF94]. Each of these places requirements on the semantics of the operations of the database schema.

*Backward commutativity* (failure to backward commute) is used to identify operations that depend on each other, for the calculation of consistent units of work (closed subhistories); it concerns operations in the same workspace history [RKT+95, WK96]. *Compensation* also addresses a single workspace 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, these operations must backward commute with (i.e., be independent of) it ([Wei88]). This means that the interim operations can be moved "backwards" in the history, ahead of the to-be-compensated operation, with the result that the undo operation is executed immediately after it. This is illustrated in Figure 1. The history reordering is conceptual only.

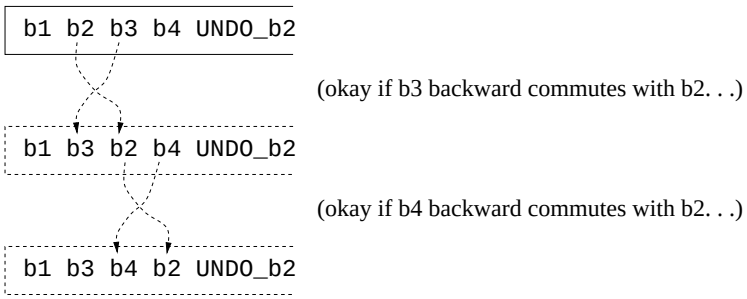


**Figure 1. History-based Compensation**

The backward commutativity requirement guarantees that the reordered history is equivalent to the actual execution history.

*Forward commutativity* (failure to forward commute) is used to identify conflicts between operations in different histories when work is exchanged between users. A consistent unit of work from one workspace history (a subhistory of interdependent operations) can be exported to another workspace history by first conceptually "moving" the to-be-exported operations backward in the history. The operations (which can successfully be applied to the initial workspace state) are then conceptually "moved forward" in the destination workspace history, provided the operations of the two histories forward commute. The operations in the exported subhistory are put at the end of the destination workspace history by the transaction manager. Details of the CoAct merge algorithm are found in [WK96].

The CoAct transaction model assumes that all semantics requirements on the operations (namely, the backward commutativity relations, compensation operations, and forward commutativity relations) are provided by the database schema designer. The *correctness* of this information

is also the responsibility of the schema designer. The aim of our research is to provide a reasoning tool that can be used to help verify the semantics requirements on a database schema. In this paper, we look at compensation.

**Advanced transaction models are not supported by ODMG.** In the ODMG Object Database Standard, methods are applied to persistent objects *within* a transaction [CB97]. The changes are made persistent either at transaction commit, or at a checkpoint within the transaction. A `checkpoint` is said to be equivalent to a `commit`, followed immediately by a `begin`; a subsequent `abort` of the transaction can not rollback the checkpoint. This corresponds to the traditional ACID transaction properties recognised in the literature. The notion of transaction offered in the ODMG language bindings contrasts sharply with the needs of transaction models for cooperative applications, for which transaction management support is needed at the data operation (method) level. This mismatch is best illustrated by the explanation of the relation between threads and transactions in the C++ language binding: "*if threads use separate transactions, the database system maintains ACID transaction properties just as if the threads were in separate address spaces. Programmers must not pass objects from one thread to another running under a different transaction; ODMG does not define the results of doing this.*" In short, in the case of advanced applications, the standard asserts that concurrency control is the responsibility of the application code. This is at odds with the view that concurrency control is something provided by a database system, and it restricts the granularity of transaction management to the level of `begin`, `commit`, `abort` primitives.

## 2 An example database schema

A characteristic of the CoAct transaction model is that high-level transactions are constructed *dynamically*, by selecting and executing atomic operations. It is these atomic operations that must be described in the database schema, as methods. We use a language called OASIS for this. Our intention in designing this language has been to work with a subset of real-life OO database language (namely, O2C [BDK92]) 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 [CB97].

OASIS includes specification facilities for abstract and concrete classes, object creation, generic container types (namely, `set<`$\beta$`>`), 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.[1] Object equivalence is by object identity; this amounts to an equality test on `oid`s. 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 subtype relation is that induced by the class inheritance hierarchy. We do not consider relationships ([CB97]), although some relationships can be expressed as constraints, using quantifiers (see [SE99] for examples of OASIS constraints).

We now look at an example OASIS schema specification, with operations for editing a generic graph structure. The example is representative of the core functionality of the SEPIA cooperative authoring system [KAN94]. Class declarations for the schema are given below:

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

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

```
abstract class Node extends Element {
   attribute set<Link> incomingLinks;
   attribute set<Link> outgoingLinks;
   boolean addIncomingLink(Link k);
};
class ANode extends Node {
   attribute set<AtomicContents> content;
};
class CNode extends Node {
   attribute int size;
   attribute set<Element> elements;
   Link createLinkIn(Node a, Node b, string s, int p);
   boolean removeNodeOrLink(Element n);
};
class Link extends Element {
   attribute Node from;
   attribute Node to;
   Link(string s, int p, Node a, Node b) { name = s; position = p; from = a; to = b };
};
class AtomicContents {
   attribute string referenceDirectory;
   attribute string showStatement;
   attribute string URL;
};
name set<CNode> cnodes;
name set<Link> links;
name set<ANode> anodes;
```

Class Element is abstract. It declares an abstract method `isConnectedTo`, which has different implementations for classes Node and Link (see Figure 2). 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 schema serves to *structure* a hypermedia document. For this reason, the class hierarchy ends at the AtomicContents class, which is used for application-specific kinds and formats of media, such as text documents, audio, and graphics. Objects of the AtomicContents class maintain e.g., a file-system handle to the hypermedia object, along with system commands (represented as strings) for displaying and editing the object. Three persistent roots are specified, which serve as extents for classes CNodes, Links, and ANodes.

**Methods and compensation operations.** Method bodies are defined using an imperative command language, which includes attribute update, object creation (with constructors), sequential composition, conditional branch, bounded collection iteration [Qia90, Qia91], and non-recursive update method call. Database methods are not designed to perform complex computations; these are done in the application programs that invoke them. This has advantages for verification, since we avoid some general purpose programming constructs that are difficult to reason about mechanically, such as recursion and unbounded iteration (albeit with a tradeoff of expressive power).

There is no explicit deletion operation in OASIS; persistence by reachability is used (as in O2 [BDK92]), where references are broken using `nil` values. The language of expressions is side-effect-free OQL [CB97]. Methods can update (objects reachable via) parameters and persistent roots, as well as (via) the receiver object's attributes. Object parameters are passed by `oid`. Objects are not encapsulated.

Figure 2 gives definitions for the methods declared in the schema, plus definitions for four

compensation methods. The abstract method `isConnectedTo` has different implementations in Link and Node. The boolean tests in the method bodies rely on the subtyping on objects induced by the inheritance hierarchy. For example, the test '`n in incomingLinks`' relies on the fact that `set<Link>` is a subtype of `set<Element>`. Method `isConnectedTo` is invoked by method `removeNodeOrLink`, within an iteration over Elements; *late binding* determines the actual implementation that is used.

The compensation methods are applied by the CoAct transaction management software to undo the effects of an operation with the same name (i.e., `changeNameTo`, `addIncomingLink`, `createLinkIn`, or `removeNodeOrLink`) in a workspace history. In general, the transaction model assumes that for each method `M`, a compensation method `UNDO_M` is to be specified 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, which corresponds to the result returned by `M`. If the effects of `M` are to be compensated, `UNDO_M` is invoked with the same parameter values, plus the value returned by `M` [EFK+96]. Observe that care was taken when defining the original methods so as to allow us to define the UNDO methods. For example, the `addIncomingLink` method calculates whether the method will succeed in doing anything, using the condition '`k == nil or k in incomingLinks`'. Although it is true that if the new link '`k`' is already in the set of `incomingLinks`, its insertion into the set will not cause the set contents to change, the test becomes important when the `UNDO_addIncomingLink` is executed: a previously existing link in the set should not be deleted by mistake (actually the same link, inserted by a earlier invocation of the `addIncomingLink` method). It is subtle issues such as this that make a proof tool a valuable aid to the specifier.

Some kinds of operation, such as input and output operations, can not 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.

To prove that one method compensates another, we make use of the fact that the UNDO method is applied (by the transaction manager) immediately after the original method. Proof goals are discussed in Section 4. The next section discusses details of our formal model.

## 3 Modelling the schema in higher-order logic

To be able to reason about methods, we translate the imperative code to functions in higher-order logic (HOL), in a format used by the Isabelle system. For the translation, we make use of data types that are predefined by Isabelle. These meet the demands of object-oriented database programming to a large extent; complex values are already supported by the system (e.g., a set of lists of integer values). The only thing that is missing is a model of a persistent object store (i.e., with object identifiers).

There are two essential ingredients in the formal model we use for a database schema: a *generic* model of objects, and a *database-specific* object type, which follows the inheritance relation established by the schema. These are explained below.

**The generic object store.** The object store maps object identifiers to storage cells. In HOL, it is represented as a (partial) function type that is parameterised with a type variable. This type variable gets instantiated with a schema-specific object type. In HOL, the object store type is written as: `oid` $\Rightarrow \beta$ `option`. The '`option`' type constructor 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). For our example schema, the type variable $\beta$ gets instantiated with the type `object` shown below. Generic operations for retrieval and update are defined on the object store. Using Isabelle, we have proved a number of theorems about these operations. The theorems are used as rewrite rules during proofs.

**Type-tagged object values.** The type variable $\beta$ that appears in the generic object store type is instantiated with database-specific information obtained from the schema. To describe

```
string Element::changeNameTo(string s) {
   var oldName:string { oldName = name; name = s } returns (oldName) };

void Element::UNDO_changeNameTo(string s, string rtn) { name = rtn };

boolean Link::isConnectedTo(Element n) { (from == n) or (to == n) };

boolean Node::isConnectedTo(Element n) {
   (n in incomingLinks) or (n in outgoingLinks) };

boolean Node::addIncomingLink(Link k) {
   if k == nil or k in incomingLinks then { skip } returns (false)
   else { incomingLinks += set(k) } returns (true) };

void Node::UNDO_addIncomingLink(Link k, boolean rtn) {
   if (rtn) then incomingLinks -= set(k) else skip };

Link CNode::createLinkIn(Node a, Node b, string s, int p) {
   var n:Link {
      if (a in elements and b in elements) then {
          n = new Link(s,p,a,b); elements += set(n); links += set(n) } returns (n)
      else { skip } returns (nil) } };

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 };

boolean CNode::removeNodeOrLink(Element n) {
   if (n != nil) and (n in elements) and
       (forall x in elements : not(x.isConnectedTo(n))) then {
          elements -= set(n) } returns (true)
   else { skip } returns (false) };

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

**Figure 2. Example (UNDO) Operations for the SEPIA Schema**

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 = ANode string int (oid set) (oid set) (oid set)
                | CNode string int (oid set) (oid set) int (oid set)
                | Link string int oid oid
                | AtomicContents string string string
```

The abstract classes of the schema (Element and Node) are not listed in the type, since instances of these classes can not 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 [SE99]).

**Methods as functions on the database state.** Methods are represented as functions in HOL. These representations are automatically generated by a schema translator, which has been implemented in ML. 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*. This tuple includes components for the modifications to the object store, the persistent roots, and the method parameters. The modifications to the object store are described by a *delta value*, given as the first component of the tuple. A delta value describes tentative changes to the object store [DHR96], and 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 $\Delta$ as an abbreviation in later examples. The `smash` operator of the generic object theory is used to "commit" the delta value changes of a method to the object store. For example, the expression '`(smash` *os* $\Delta$`)`' commits the changes in $\Delta$.

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. (Recall that the HOL '`option` $\beta$' type is generic.) The *return* value of the method application is given in the last position of the tuple.

The schema 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 schema translator has been implemented in ML to *automatically* generate the database-specific Isabelle theory file for an input schema.

## 4 Compensation analysis using the Isabelle theorem prover

The definition of a compensation method gives rise to a *proof obligation*, which can be verified using Isabelle. Consider a method `M` with $n$ input parameters. Informally, we have to prove that for an arbitrary database state $DB$, receiver object $o$, and input parameter values $v_1 \cdots v_n$, it is the case that $o.\texttt{M}(v_1, \ldots, v_n)$ executed in state $DB$, immediately followed by $o.\texttt{UNDO\_M}(v_1, \ldots, v_n, r)$ results in the same old state $DB$, where $r$ is the value returned by `M`. Notice that the proof obligation involves a universal quantification over all possible database states, receiver object, and parameter values. Based on the representation of methods in HOL, such a requirement can be entered as a proof goal in Isabelle. Our tool uses Isabelle's automated proof procedures to verify such goals.

**Compensation in terms of the formal model.** To prove that one operation compensates another, we must show that any changes to the object store posed by the first are undone when the operations are composed. We construct proof goals for the database schema by specifying boolean-valued expressions about method applications with respect to the input object store, the persistent roots, and the parameter values. To express what we mean by a compensation requirement for method `UNDO_M` with respect to method `M`, consider the following applications (and results) of the two methods:

$$\begin{array}{l}\text{let } (\Delta, v'_1, \ldots, v'_n, r) = \texttt{M}\ os\ v_1\ \cdots\ v_n \\ \text{let } (\Delta', v''_1, \ldots, v''_n, r') = \texttt{UNDO\_M}\ (\texttt{smash}\ os\ \Delta)\ v_1\ \cdots\ v_n\ r\end{array}$$

The UNDO method is applied to the same arguments as the original method, with the exception of the object store argument; it is also applied to the return value of `M`. The delta value $\Delta$ contains the (tentative) changes to the database state made by method `M`. These changes are combined with the original object store in order to evaluate method `UNDO_M`. The operator `smash` is used to hide (override) any bindings for the same objects in its first argument (i.e., it applies the changes to the database state). The above notation omits changes to the persistent roots. For our example schema, values $v_1$, $v_2$, and $v_3$ are actually case expressions that check for changes to the persistent roots; they pass on either the initial value or the modified value.

As can be seen above, method `UNDO_M` is applied to an `M`-modified object store; it returns a delta value $\Delta'$ that contains its (tentative) changes to the database state. For compensation, we want to prove that the original object store *os* is equivalent in some sense to the object store obtained after the changes proposed by both methods are applied (i.e., we want to show $os \equiv \texttt{smash}\ (\texttt{smash}\ os\ \Delta)\ \Delta'$).

A compensation proof goal is universally quantified over the *defined* objects in the initial object store. For these objects, Isabelle reasons about the equivalence of each object's value in the original and updated object stores. The equivalence is not equality. The initial proof goals that must be entered to the Isabelle system are quite large. We have defined an ML function to automate their generation for a particular schema. To do this, the user types in a command such as the following:

```
- start_proof(undo_method_goal("CNode","removeNodeOrLink"));
```

The generated goal corresponds to the let-expression form shown above, but includes additional parameters for the persistent roots. It also includes an assumption about the "definedness" of nested object identifiers in the initial object store, and assumptions about the "freshness" of any new oids needed for object creation.

**Automated proof procedure.** The analysis tool implements an *automated proof strategy.* This is essentially the same strategy we have previously used for verifying consistency requirements (i.e., that a method preserves a number of static integrity constraints), and on which we have reported elsewhere [SE99]. The proof strategy is based on the standard machinery provided by the Isabelle theorem prover, which uses a combination of term rewriting and natural deduction [Pau94]. The standard machinery of the theorem prover supports the common data types in programming languages (e.g., int, bool, list, set, tuple). As shown in Section 3, these data types are used to represent similar data types of our database language. Thus, we get most of the automated reasoning for free. For example, the set membership operation in the database language is mapped to the predefined set membership operation in HOL; Isabelle already installs a number of theorems to reason about set membership.

The extensions we have made primarily deal with our theory of objects, which supplements the standard data types. This demonstrates the advantage of using an extensible theorem prover, such as Isabelle; the new theory can be easily integrated. At present, the generic object theory is 632 lines of Isabelle/HOL code; it contains 49 theorems. Most of these theorems are equivalence theorems that are used by Isabelle for term-rewriting purposes.

Automated theorem proving is inherently limited. The proof search may fail because of undecidability. Any subgoals that cannot be solved automatically are returned by the proof procedure and can be examined interactively in Isabelle.

**Practical results.** Our example database schema for the case study currently includes 6 class definitions, 27 method definitions (9 of which are UNDO methods), and 5 integrity constraints.[2] The Isabelle theory and ML files generated for this schema comprise 162 lines of HOL-code. All compensation requirements of our case study could be proved automatically by Isabelle. Table 1 gives experimental results for verifying the compensation methods of the example schema. The

| Class | Method | Compensation | Proof Time |
|---|---|---|---|
| Element | changeNameTo | UNDO_changeNameTo | 5.25s. |
| ANode | createAtomicContentIn | UNDO_createAtomicContentIn | 22.39s. |
| ANode | removeAtomicContent | UNDO_removeAtomicContent | 15.29s. |
| Node | addIncomingLink | UNDO_addIncomingLink | 7.42s. |
| Node | addOutgoingLink | UNDO_addOutgoingLink | 7.55s. |
| CNode | createANodeIn | UNDO_createANodeIn | 81.72s. |
| CNode | createCNodeIn | UNDO_createCNodeIn | 85.34s. |
| CNode | createLinkIn | UNDO_createLinkIn | 107.25s. |
| CNode | removeNodeOrLink | UNDO_removeNodeOrLink | 14.17s. |

**Table 1. Experimental Results for Method Compensation**

proof times are in seconds, with Isabelle running on a SUN 296 MHz Ultra-SPARC-II, under Solaris. The times indicate that the proof strategy is reasonably efficient (although Isabelle performs up to a few hundred proof steps per second).

## 5 Related work and extensions

Our verification framework is inspired by the pioneering work of Sheard and Stemple ([SS89]), 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 [BS97]. 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 [AJR97]. In their work, the Z specification language ([Spi92]) 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 [AJR97] 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 ([PVS]) to obtain mechanised support for the verification of concurrency control protocols, such as two-phase locking (2PL) [CHvdS99]. 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 [Özsu94].

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 the introduction, compensation should be combined with backward commutativity in order to support cooperative

[2] Only parts of the SEPIA schema are shown in this paper.

work, because intermittent operations might be executed before a compensation operation is attempted. Since the verification of consistency and compensation requirements has proven feasible in our framework, we are currently working to generalise the techniques to support the verification of commutativity requirements, both backward and forward.

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 [Isa], but it is still preliminary.

## 6 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 automated theorem prover [Pau94]. Our tool was initially developed to verify consistency requirements. In [SE99], 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 largely based on standard machinery provided by the Isabelle theorem prover. Our work demonstrates that *different* requirements on the semantics of update operations can be verified within a single formal framework.

We suspect that it is not possible to characterise 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.[3] If the automated proof procedure does not find a proof, it returns the goals it can not 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). The 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.

[3]The compensation problem is, in general, undecidable. See, for example, [BGL96] and [IDR96], for analogous foundational results on transaction safety and commutativity analysis.

## References


[AJR97] Paul Ammann, Sushil Jajodia, and Indrakshi Ray. Applying formal methods to semantic-based decomposition of transactions. *ACM Transactions on Database Systems*, 22(2):215–254, June 1997.

[BDK92] F. Bancilhon, C. Delobel, and P. Kanellakis, editors. *Building an Object-oriented Database System: The Story of O2.* Morgan Kaufmann, 1992.

[BGL96] M. Benedikt, T. Griffin, and L. Libkin. Verifiable properties of database transactions. In *Proceedings of Principles of Database Systems (PODS)*, pages 117–127, 1996.

[BS97] V. Benzaken and X. Schaefer. Static integrity constraint management in object-oriented database programming languages via predicate transformers. In *Proceedings of ECOOP*, volume 1241 of *LNCS*, pages 60–85. Springer-Verlag, 1997.

[CB97] R. G. G. Cattell and Douglas K. Barry, editors. *The Object Database Standard: ODMG 2.0.* Morgan Kaufmann Publishers, San Francisco, California, 1997.

[CD97] Q. Chen and U. Dayal. Failure handling for transaction hierarchies. In *Proceedings of ICDE*, pages 245–254, Birmingham, U.K., April 1997.

[CHvdS99] Dmitri Chkliaev, Jozef Hooman, and Peter van der Stok. Serializability preserving extensions of concurrency control protocols. In *Andrei Ershov International Conference on Perspectives of Systems Informatics*, Lecture Notes in Computer Science, Novosibirsk, 1999. Springer-Verlag. To appear.

[DHR96] M. Doherty, R. Hull, and M. Rupawalla. Structures for manipulating proposed updates in object-oriented databases. In *ACM SIGMOD Symposium on the Management of Data*, pages 306–317, June 1996.

[EFK+96] S. Even, F. Faase, H. Kaijanranta, J. Klingemann, A. Lehtola, O. Pihlajamaa, T. Tesch, and J. Wäsch. Deliverable VII.1: Design of the TransCoop Demonstrator System. Report TC/REP/GMD/D7-1/704, Esprit Project No. 8012, 1996.

[Elm92] Ahmed K. Elmagarmid, editor. *Database Transaction Models for Advanced Applications.* Morgan Kaufmann Publishers, 1992.

[IDR96] O. Ibarra, P. Diniz, and M. Rinard. On the complexity of commutativity analysis. In *International Computing and Combinatorics Conference*, 1996.

[Isa] Isabelle. `http://www.cl.cam.ac.uk/Research/HVG/isabelle.html`.

[KAN94] W. Klas, K. Aberer, and E. J. Neuhold. Object-Oriented Modelling for Hypermedia Systems Using the VODAK Modelling Language (VML). In *Advances in Object-Oriented Database Systems*, volume 130 of *Computer and Systems Sciences*, pages 389–443. Springer-Verlag, 1994.

[KLS90] Henry F. Korth, Eliezer Levy, and Abraham Silberschatz. A formal approach to recovery by compensating transactions. In *Proceedings of the 16th VLDB Conference*, pages 95–106, Brisbane, Australia, 1990.

[KTWK97] Justus Klingemann, Thomas Tesch, Jürgen Wäsch, and Wolfgang Klas. The TransCoop Transaction Model. In *Transaction Management Support for Cooperative Applications*, chapter 7, pages 149–172. Kluwer Academic Publishers, 1997.

[LMWF94] Nancy Lynch, Michael Merrit, William Weihl, and Alan Fekete. *Atomic Transactions.* Morgan Kaufmann Publishers, 1994.

[MR97] Cris Pedregal Martin and Krithi Ramamritham. Delegation: Efficiently rewriting history. In *Proceedings of ICDE*, pages 266–275, Birmingham, U.K., April 1997.

[Özsu94] M. Tamer Özsu. Transaction models and transaction management in object-oriented database management systems. In *Advances in Object-Oriented Database Systems*, volume 130 of *Computer and Systems Sciences*, pages 147–184. Springer-Verlag, 1994.

[Pau94] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *LNCS*. Springer-Verlag, 1994.

[PVS] PVS World Wide Web page: `http://www.csl.sri.com/pvs/overview.html`.

[Qia90] Xiaolei Qian. An axiom system for database transactions. *Information Processing Letters*, 36(4):183–189, November 1990.

[Qia91] Xiaolei Qian. The expressive power of the bounded-iteration construct. *Acta Informatica*, 28(7):631–656, October 1991.

[RKT+95] Marek Rusinkiewicz, Wolfgang Klas, Thomas Tesch, Jürgen Wäsch, and Peter Muth. Towards a cooperative transaction model—The Cooperative Activity Model. In *Proceedings of the 21st VLDB Conference*, Zurich, Switzerland, September 1995.

[SE99] David Spelt and Susan Even. A theorem prover-based analysis tool for object-oriented databases. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99)*, number 1579 in LNCS, pages 375–389. Springer-Verlag, 1999.

[Spi92] J. M. Spivey. *The Z Notation: A Reference Manual, Second edition.* Prentice Hall International, 1992.

[SS89] Tim Sheard and David Stemple. Automatic verification of database transaction safety. *ACM Transactions on Database Systems*, 14(3):322–368, September 1989.

[VWP+97] Jari Veijalainen, Jürgen Wäsch, Juha Puustjärvi, Henry Tirri, and Olli Pihlajamaa. Transaction Models in Cooperative Work—An Overview. In *Transaction Management Support for Cooperative Applications*, chapter 3, pages 27–58. Kluwer Academic Publishers, 1997.

[Wei88] William E. Weihl. Commutativity-based concurrency control for abstract data types. *IEEE Transactions on Computers*, 37(12):1488–1505, December 1988.

[WK96] Jürgen Wäsch and Wolfgang Klas. History merging as a mechanism for concurrency control in cooperative environments. In *IEEE Workshop on Research Issues in Data Engineering: Interoperability of Nontraditional Database Systems*, pages 76–85, 1996.