# A Theorem Prover-Based Analysis Tool for Object-Oriented Databases

David Spelt and Susan Even*

University of Twente
Centre for Telematics and Information Technology
P.O. Box 217, Enschede, The Netherlands

**Abstract.** We present a theorem-prover based analysis tool for object-oriented database systems with integrity constraints. Object-oriented database specifications are mapped to higher-order logic (HOL). This allows us to reason about the semantics of database operations using a mechanical theorem prover such as Isabelle or PVS. The tool can be used to verify various semantics requirements of the schema (such as transaction safety, compensation, and commutativity) to support the advanced transaction models used in workflow and cooperative work. We give an example of method safety analysis for the generic structure editing operations of a cooperative authoring system.

## 1 Introduction

Object-oriented specification methodologies and object-oriented programming have become increasingly important in the past ten years. Not surprisingly, this has recently led to an interest in object-oriented program verification in the theorem prover community, mainly using higher-order logic (HOL). Several different approaches to modelling object-oriented features in HOL have been presented [13,8]. These approaches emphasise the methods and behaviour of a single object. For an object-oriented database, a different viewpoint is needed: a database typically includes integrity constraints over collections of objects that have a lifetime beyond an application program. Operations on the database transform it from one consistent state to another. In this paper, our point of view is the database state itself, and the persistent collection of objects it contains. We give a formal model for a persistent object store in HOL, which simulates the type-tagged memory structure of an implementation. This model is sufficient to describe the operational semantics of the typical features of an object-oriented database programming language, such as heterogeneous collections, inheritance, late binding, and nil values.

Many recently proposed database systems rely on transaction models that require designers to provide various assertions about the semantics of their schema. Examples include (but are not limited to), consistency requirements (i.e., a method/transaction has to preserve a number of static integrity constraints) [4], the correctness of undo methods (i.e., for each method another method has to be specified which compensates the effects of the method) [9], and commutativity tables (i.e., for each method pair,

* Research supported by SION, Stichting Informatica Onderzoek Nederland.

W.R. Cleaveland (Ed.): TACAS/ETAPS'99, LNCS 1579, pp. 375–389, 1999.

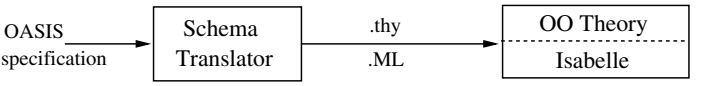
it has to be specified when two methods commute) [17]. Such knowledge about the semantics of a schema is used by so-called advanced transaction models to provide more flexible mechanisms for concurrency control [12]. This is essential for many modern applications of database technology, such as workflow management and cooperative work. It is often assumed that database designers provide the required knowledge about the schema. This, however, is problematical, since people will make mistakes in their specifications: a seemingly trivial line of code, such as a nil-check, is easily forgotten, and it may lead to inconsistency of persistent data.

In this paper, we describe a tool which can assist database designers to verify the correctness of assertions about an object-oriented database schema. We outline a method (transaction) safety analysis framework based on theorem proving in higher-order logic (HOL). We show how to adapt the Isabelle/HOL theorem prover [10] for this task. We first define a general Isabelle theory of object-oriented systems. Using this theory, we show (1) how specific object-oriented schemas can be encoded in HOL, and (2) how proofs about these schemas can be performed using the Isabelle system.

The paper is organised as follows. An overview of our analysis framework, including a brief introduction to the Isabelle system, is given in Section 2. Section 3 introduces a database specification language, called OASIS, and a case study. Section 4 discusses the formal model of the persistent object store, in HOL, which is used to encode the semantics of specific database schemas. This model includes a number of generic (higher-order) operations and theorems about their combination for term-rewriting. The actual representation of database-specific schema information in terms of these operations is discussed in Section 5. We show how typical object-oriented language features, such as heterogeneous collections, methods, late binding, and transactions, can be encoded. Section 6 shows how to extend the Isabelle tools to assist in reasoning about these schema representations. An example proof is discussed, which uses the framework for transaction safety analysis. Section 7 discusses related work on object-oriented analysis that makes use of theorem prover technology. Section 8 gives a summary and discusses future work.

## 2 Architecture of the OASIS tool

Our schema specification language is called OASIS (for *Object AnalySIS*). It includes facilities for constraint and query definition, object manipulation, and transaction definition. The features of the object manipulation language are common to object-oriented database technology (e.g., late binding, inheritance, and heterogeneous collections [1]). The structure of the OASIS tool is similar to that of the LOOP tool [8]. The OASIS specification language is mapped by a *schema translator* to a simple formal model of objects in higher-order logic (HOL). This model resembles the type-tagged memory structure of an implementation and is sufficient to describe the operational semantics of the specification language. The reasoning component of the tool is implemented using the higher-order logic incarnation of the Isabelle theorem prover [10]. The two major components of the OASIS tool are described in more detail below:




*Extended Isabelle Theorem Prover.* Isabelle is an open system, implemented in ML. Its HOL theory provides a formal theory of the standard data types one finds in databases, such as booleans, integers, characters, strings, tuples, lists, and sets. To reason about heterogeneous collections of objects with shared subcomponents, we extend these standard theories with a *Generic OO Theory* that simulates the type-tagged memory of an implementation. The theory defines functions that describe the effects of primitive update and retrieval operations on the object-store (e.g., attribute update and attribute selection). We have derived a number of theorems about the interactions of these operations, which are used for the analysis of database methods and transactions.

*Schema Translator.* The schema translator is directly implemented in ML. It maps a specific object-oriented schema to a low-level representation, defined in terms of the operations of the generic theory of objects. The input is an OASIS schema, in ascii form, which is parsed and converted to an internal abstract syntax tree in ML. The output consists of two Isabelle files: a file with extension '`.thy`' that gives definitions for the database-specific class structures, methods, transactions, and constraints; and an ML file with extension '`.ML`' that contains some standard lemmas about the schema. These files can be loaded into an Isabelle session, and proofs about the schema can be initiated.

## 3 An example OASIS specification

An OASIS database schema consists of a number of class definitions, named persistent roots, integrity constraints, and transactions. Classes (which can also be abstract) contain definitions of methods, written in a simple procedural update language. OASIS supports single inheritance. Persistent roots provide named entry points to the database; they can be used as global variables in methods, transactions, and integrity constraints. OASIS provides facilities for associating constraints with a schema. These constraints are boolean-valued query expressions over the database state. For queries, we use OQL (Object Query Language) [5].

Figure 1 shows part of a generic graph schema, which includes some basic structure editing operations. This example is based on the implementation of the SEPIA document authoring system [16]. The schema defines abstract classes for Elements and Nodes. Atomic nodes (class ANode) and composite nodes (class CNode) are concrete classes, which are extensions of class Node. Link is another concrete class, which extends the abstract class Element. If a class does not extend any other class, it implicitly extends the abstract class Object. The schema defines three named persistent roots: cnodes, links, and anodes. These are analogous to the attributes of the `main` class in object-oriented programming languages.

Figure 1 also gives some example integrity constraints over the contents of the persistent roots. Constraints $c_1$, $c_2$, and $c_3$ assert non-nil requirements. Constraint $c_4$ asserts

```
abstract class Element {
   attribute string name;
   attribute int position;
   abstract boolean isConnectedTo(Element n);
};
abstract class Node extends Element {
   attribute set<Link> inLinks;
   attribute set<Link> outLinks;
};
class ANode extends Node {
   attribute set<string> content;
};
class CNode extends Node {
   attribute int size;
   attribute set<Element> elements;
   CNode(string s, int p, int z);
   boolean removeNodeOrLink(Element n);
   CNode createCNodeIn(int p, int z, string s);
};
class Link extends Element {
   attribute Node from;
   attribute Node to;
};
name set<CNode> cnodes;
name set<Link> links;
name set<ANode> anodes;
constraints {
  c1 : forall n in cnodes : n!=nil and (forall e in n.elements : e!=nil);
  c2 : forall n in links : n!=nil;
  c3 : forall n in anodes : n!=nil;
  c4 : forall cn in cnodes :
          forall e in cn.elements :
               ((e instanceof Link) implies
                       (((Link)(e).from in cn.elements) and
                        ((Link)(e).to in cn.elements)));
  c5 : forall n1 in cnodes : forall n2 in cnodes :
         n1 == n2 or (forall n in n1.elements: not(n in n2.elements));
};
```

**Fig. 1.** Classes, Persistent Roots, and Constraints of the SEPIA Schema

that all links in a composite node should link nodes within that same composite node. Constraint $c_5$ asserts that elements are nested within at most one "parent" CNode object.

The command language we use consists of a small number of commonly used constructs. Atomic updates are object creation, and variable and attribute update. There is no object deletion, because *persistence by reachability* is used (as in Java and the O2 database system [1]): that is, an object is in the database as long as it is directly or indirectly reachable from one of the roots. Compound commands are formed using sequential composition, bounded iteration, conditional branch, collection iteration, and (at present) non-recursive update method call.

Method bodies are defined using a command statement. A method can apply updates to the receiving (i.e., `this`) object, as well as to the objects referenced by `this`, the persistent roots, and the attributes of objects passed in as actual parameters. An OASIS schema also declares a number of named transactions. Transactions are similar to methods, but there is no receiver object. A transaction typically executes a sequence of method applications. Traditionally, the notion of database integrity is tied to database transactions, but our system also allows one to verify integrity at the method level (which is often preferred). In this paper, we focus on methods rather than transactions.

Figure 2 gives some example method definitions for the schema. Method removeNodeOrLink on composite nodes will be used as an example in later sections. This method removes an Element from the elements component of the receiver CNode object, provided that it is not connected to any other Element (within the same CNode). This condition is tested by applying the abstract method isConnectedTo of class Element, which has different concrete implementations in classes Node and Link. Late binding selects the appropriate implementation of the method, based on the run-time type of the receiver object. Method removeNodeOrLink respects the integrity constraints on the schema. In Section 6, we show how the OASIS system proves this automatically. Constraint $c_4$ is non-trivial with respect to this method; both address the elements attribute of a CNode.

## 4 A generic Isabelle theory of objects

Isabelle specifications are called *theories*. A theory consists of a collection of definitions and axioms. Our system extends the default collection of Isabelle/HOL data type theories that are available. In this section, we define a *generic theory of objects*, which describes schema-independent knowledge about object-oriented databases. Database-specific knowledge can be expressed in terms of this theory (this is the subject of Section 5). Isabelle/HOL syntax is similar to ML syntax and is for the most part self-explanatory. We give annotations to clarify its more cryptic symbols.

The database state (object store) is modelled as a partial function from object identifiers to values. In Isabelle, we represent such functions using the predefined '`option`' data type, as '`oid => 'b option`.' Isabelle/HOL function types (`=>`) are total; partial function types can be modelled using options. The option data type includes the constructors `None` (to represent *undefined* function results) and `Some` (to represent *defined* function results—the actual value is supplied as an argument). The type variable

```
CNode CNode::createCNodeIn(int p, int z, string s) {
   var n:CNode {
      n = new CNode(s, p, z); elements += set(n); cnodes += set(n)
   } returns (n) };
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) };
boolean Node::isConnectedTo(Element n) {
   (n in inLinks) or (n in outLinks) };
boolean Link::isConnectedTo(Element n) { (from == n) or (to == n) };
```

**Fig. 2.** Example Method Definitions for the SEPIA Schema

$\beta$ (written `'b`) in the co-domain type of '`oid => 'b option`' will be instantiated with a concrete type that describes the schema-specific class structures (see Section 5). The type of object identifiers (`oid`) is defined as a datatype, which we omit here.

On this abstract notion of database state, we define several higher-order functions for database retrieval and update. Figure 3 lists these operations with their signatures. These functions are modelled as schema-independent operations, which take (functions

```
oids :: (oid ⇒ β option) ⇒ oid set
eval :: [β option, β ⇒ bool] ⇒ bool
get :: [β option, β ⇒ α] ⇒ α
set :: [(oid ⇒ β option), oid, β ⇒ β] ⇒ (oid ⇒ β option)
smash :: [(oid ⇒ β option), (oid ⇒ β option)] ⇒ (oid ⇒ β option)
apply :: [α set, [α, oid] ⇒ β option] ⇒ (oid ⇒ β option)
new :: [oid, β] ⇒ (oid ⇒ β option)
skip :: (oid ⇒ β option)
```

**Fig. 3.** Generic Operations on Objects

as) parameters to make them specific. The operations `oids`, `get`, and `eval` are used to retrieve information from the state. For example, the operation `get` is used for the translation of attribute selection. The other operations in the figure are used to update the state; they result in a "little" object store (called a *delta value* [6]), which comprises local changes to the state. For example, the operation `set` is used for the translation of attribute assignment. The `smash` operation is used to encode sequential compositions ('`;`') of commands. It is defined as a functional override, where the bindings in the

second argument take precedence over those in the first. The `smash` operation is also used to apply method changes to the object store.

Isabelle can be used to prove abstract properties (theorems) about the operations in Figure 3, based on their definitions in HOL. At present, the generic theory of objects includes 49 theorems. First-order rules are derived for the associativity and reflexivity of `smash`. Second-order rewrite rules (with functions in arguments) are derived for applications of `eval` and `get` to modified object store values. Below, we give an example of one of these theorems (rule $r_1$):

```
get ((smash os1 (set os2 idb f)) ida) g =
       (if idb=ida & idb:oids os2 then get (os2 ida) (g ∘ f)
        else get (os1 ida) g)
```

This rule illustrates how a `get` operation is "pushed through" an updated object store. Such theorems are used as rewrite rules during proofs, in a left-to-right manner.

## 5 Modelling database-specific knowledge

The OASIS schema translator supplements the generic theory discussed in the previous section with database-specific information. For an input database schema, the schema translator generates an Isabelle '`.thy`' file that contains the database-specific HOL definitions of class structures, methods, transactions, and integrity constraints. In effect, the schema translator implements a *semantics mapping*, where the output is HOL notation. The schema translation has been defined and implemented for all of the OASIS constructs we show in this paper (as well as a few others, such as foreach, which we do not discuss here).

The previous section introduced an abstract notion of database state as a partial function from oids to values of generic type `'b`. For a specific database schema, the type variable `'b` should be instantiated with type information that reflects the database-specific class hierarchy. This is done using a data type definition:

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

The above data type is a disjoint union type, with a case for each of the concrete classes in the schema; the abstract classes `Element` and `Node` are *not* included, because they do not have concrete instantiations. Structural information of objects (i.e., attribute values) is supplied as an argument to the data type constructors. 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 for the required *run-time* type information. In object-oriented systems with inheritance, this information is needed to model run-time type-based decisions, such as late-binding. Using our Isabelle representation, these

decisions can be conveniently encoded using case-splits to examine the type tag. The following sections show how to encode OASIS features in terms of the generic theory of objects, enhanced with schema-specific information.

*Queries and constraints.* The schema translator maps OASIS query expressions to functions in Isabelle/HOL. These functions take the input object store as an argument. The Isabelle predefined data types support most commonly used OQL query language constructs [5]. For example, set expressions in OQL (e.g., union, select-from-where, except, and intersect) are available in the Isabelle syntax. The translation of most OQL expressions is straightforward. However, the translation of operations on objects (e.g., attribute selection and nil comparisons) is complicated by the introduction of object identifiers. For these constructs, explicit lookups on the object store are needed. We encode these using the generic retrieval operations get and eval of the theory of objects.

To represent nil comparisons in Isabelle, we make use of the function eval. For example, the expression 'n!=nil', where n is of type Node, amounts to a check that n is in the object store, with the right type. The following Isabelle code accomplishes this:

```
eval (os n) (%val. case val
  of ANode name position inLinks outLinks content => True
   | CNode name position inLinks outLinks size elements => True
   | Link name from to => False)
```

The expression (os n) looks up the object-typed value associated with oid n. The second argument to eval is a boolean-valued function (the symbol % is HOL syntax for $\lambda$-abstraction). This function returns True if the type tag on the value is ANode or CNode; otherwise, if n does not have a binding in os, or is bound to a Link value, then False is returned. In the examples, we abbreviate the case-split function with a name, such as *isNode* for the above.

Attribute selections are coded using the get operation. For example, the OASIS expression 'n.elements', where n is of type CNode, is represented as follows:

```
get (os n) (%val. case val
  of ANode name position inLinks outLinks content => arbitrary
   | CNode name position inLinks outLinks size elements => elements
   | Link name from to => arbitrary)
```

Observe that an arbitrary value is returned for the wrongly typed cases; this is a common way of dealing with undefined function results in HOL [7].

Constraints are boolean-valued queries. Constraint $c_4$ of the Sepia schema is represented in Isabelle as follows:

```
c4 os cnodes links anodes ==
  ! cn:cnodes.  ! e:(get (os cn) elementsOf).
       (eval (os e) isLink) -->
           ((get (os e) fromOf):(get (os cn) elementsOf)) &
           ((get (os e) toOf):(get (os cn) elementsOf))
```

In Isabelle syntax, the forall quantifier is written as '!'. The type cast in the original constraint falls away in the translation to HOL.

*Update methods, late binding, and transactions.* Update methods are represented as named functions in HOL. Such functions map an input object store, persistent roots, an oid this, actual parameter values and any required new oids to a *tuple*. The tuple includes the modifications to the object store, persistent roots, and method parameters; the *return* value of the method is given in the last position of the tuple. The removeNodeOrLink method of class CNode has the following HOL representation:

```
CNode_removeNodeOrLink os cnodes links anodes this n ==
  if (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 Node_isConnectedTo os cnodes links anodes x n))
  then (set os this f, True)
  else (skip, False)
```

The right-hand side is a conditional expression that reflects the structure of the original method body. Within the conditional, the application of the isConnectedTo method to element object 'x' in the `if`-clause involves *late binding*: based on the actual run-time type of 'x', the correct implementation of the method is applied. In our framework, such a run-time type-based decision is easily expressed using an `if-then-else` clause, and the `eval` predicate. The inner conditional expression yields a boolean value, which is negated with the operator '~'. It is important to realise that nothing is computed by a conditional expression; it is only used as an assumption in the `then` and `else` branches of the proof.

The first component of the tuple returned by the `then` branch is a `set` expression, which describes the effects of the assignment to the elements attribute of the this object, in an algebraic manner. The function *f* abbreviates a case-split for the actual update:

```
(%val. case val of ANode name position inLinks outLinks content =>
             ANode name position inLinks outLinks content
         | CNode name position inLinks outLinks size elements =>
             CNode name position inLinks outLinks size (elements - {n})
         | Link name position from to => Link name position from to)
```

The second component of the tuple is the return value of the method, which is a boolean value. We omit changes to the persistent roots and parameters in the above example.

Our schema translator generates less "efficient" code than that shown above; this is inherent in automatic code generation. However, we easily obtain the above simplified form, using term rewriting (see Section 6).

A transaction is not the same as a method: a transaction is a sequence of updates, whose changes are not propagated to the database until the transaction commits. A transaction is further distinguished by not having a receiver object. Transaction semantics is provided by applying an additional `smash` to the input object store and the delta value that represents the transaction body's updates. A method can be "lifted" to the transaction level by putting code to lookup the receiver object in the transaction, and

then applying the method. The next section uses an example in which we give transaction semantics to the removeNodeOrLink method.

## 6 Using the system

The OASIS tool currently provides support for automated transaction safety analysis. The tool implements an *automated proof strategy*, which is comprised of the following four successive steps: (i) specification of an initial proof goal; (ii) normalisation of the goal using rewriting; (iii) safe natural deduction inference steps; and (iv) exhaustive depth-first search. This strategy can verify many non-trivial combinations of transactions and constraints, although the search is inherently incomplete [2]. The automated proof procedure returns any goals that it cannot solve. We now explain in detail each of these steps.

*Starting a transaction safety proof.* To start a transaction (or method) safety proof, an Isabelle *proof goal* should first be constructed. Our schema translator defines the ML functions `start_proof` and `method_safety_goal`, which automate this process for a given method and constraint. For example, to verify that method removeNodeOrLink, defined in class CNode, is safe with respect to constraint $c_4$, we type the following:

```
- start_proof(method_safety_goal("removeNodeOrLink","CNode","c4",["c1"]));
```

Verification of a method or transaction with respect to an individual constraint predicate may depend on additional constraints on the schema. In this example, constraint $c_1$ is necessarily assumed, since in order to extract the elements attribute from a CNode object, that object must be non-nil. Additional assumptions are given as parameters to the `start_proof` command. Isabelle now responds with the following initial proof goal:

```
Level 0
...
(eval (os this) isCNode) &
c4 os cnodes links anodes & c1 os cnodes links anodes -->
(let (delta,result) =
       CNode_removeNodeOrLink os cnodes links anodes this n
 in c4 (smash os delta) cnodes links anodes)
```

The goal is in the form of an implication, where the constraints are assumed to hold in the initial state `os` (as seen in the premise); the conclusion is in the form of a `let` expression, which substitutes the modifications resulting from the method application into the constraint expression. Recall that our running example ignores modifications to the persistent roots. Observe that the new database state in which the constraint is evaluated takes the form `(smash os delta)`. The `smash` "implements" the transaction-level commit of the changes in the little object store `delta` to the input object store `os`, as mentioned in Section 5.

*Normalisation of the proof goal.* The actual proof starts by unfolding the database-specific definitions (of methods, constraints, and transactions) in the initial goal. This is done using the Isabelle *Simplifier*. The Simplifier performs term-rewriting with a set of theorems of the following form: `[|`$H_1$ ; $\cdots$ ; $H_n$`|] ==>` $LHS$ `=` $RHS$. Such theorems are read as conditional rewrite rules: a term unifying with the expression on the left-hand side of the equality sign $(LHS)$ is rewritten to the term that appears on the right-hand side $(RHS)$, provided that the hypotheses $(H_1, \ldots, H_n)$ hold. The default Isabelle Simplifier installs a large collection of standard reduction rules for HOL; new rules are easily added to customise the Simplifier to particular tasks. We have extended the Simplifier by adding a number of rewrite rules for simplifying expressions involving the constructs of the generic theory of objects. In addition to these, the '`.ML`' file that is generated by the schema translator asserts all database-specific definitions as rewrite rules. Thus definitions are automatically unfolded by the normalisation step.

Unfolding the database-specific definitions rewrites the initial goal into a more complex form, in which every occurrence of the input object store `os` in the goal's conclusion is replaced by an expression that reflects the modifications to `os`. During normalisation, one of the subterms for the example is:

```
(get ((smash os (set os this f)) e) fromOf) :
    (get ((smash os (set os this f)) cn) elementsOf)
```

This subterm represents the condition '`e.from in cn.elements`' (in constraint $c_4$), in the context of the *updated object store*. At this point, patterns such as the above can be reduced using the rewrite rules of the generic theory of objects. The above term is rewritten (in several steps) to:

```
(get (os e) fromOf):(if na=this then (get (os cn) elementsOf)-{n}
                     else (get (os cn) elementsOf)
```

The rewriting "pushes" the attribute selection through the algebraic update operations (`smash`, `set`). For example, the update of the elements attribute is irrelevant with respect to the selection of the from field. This is identified by the Simplifier by application of rule $r_1$ from Section 4. Observe that, in the result term, all attribute selections are expressed directly in terms of the input object store.

During the normalisation phase, constraints that are irrelevant with respect to a part of the proof goal can be detected. (For example, straightforward term rewriting can already prove that method removeNodeOrLink does not interact with constraint $c_2$.) The example proof above requires more analysis, because updates are applied to the same parts of the database (i.e., the elements attribute).

*Safe natural deduction inference steps.* In addition to term rewriting with the Simplifier, Isabelle also uses natural deduction. Its *Classical Reasoner* uses a set of introduction and elimination rules (i.e., theorems) for higher-order logic to automate natural deduction inferences. 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 be automatically instantiated, and backtracking is performed between different alternative unifiers. The tool requires

a distinction to be made between so-called *safe* and *unsafe* rules. Safe rules can be applied deterministically; they do not introduce or instantiate variables, so there is no need to undo any of these steps at later stages in the proof. For example, introduction of universal quantification is safe, whereas its elimination is unsafe. Safe steps get rid of trivial cases. The Classical Reasoner interleaves these steps with further simplification.

As we did for the Simplifier tool, some extensions have to be made to the Classical Reasoner. The extensions include a database-specific rule for the introduction (and its converse rule for elimination) of the predicate `eval`. These rules (and their proof scripts) are generated automatically by the OASIS schema translator and reside in the '`.ML`' file; they provide a mechanism for case-based reasoning for the database-specific `object` type. For example, for an expression of type Node, cases are generated for types ANode and CNode; simplification immediately discards the other cases, which are irrelevant. Applying safe inference steps to our example goal generates a list of 12 subgoals. These goals require more in-depth analysis.

*Exhaustive depth-first search.* Once the safe steps have been performed, any remaining goals are subject to an exhaustive depth-first analysis [7]. Safe inference steps are now interleaved with unsafe steps. This may involve backtracking, and undoing of unification steps. Isabelle allows a limit to be imposed on the search depth. This guarantees termination of the search tactics. In our practical experiments, a depth of 2 was sufficient for most cases.

Steps (ii) to (iv) of the automated proof strategy are packaged as a single Isabelle *tactic* (`oasis_tac`, which is a customization of Isabelle's `auto_tac`). A tactic is a proof procedure (i.e., proof instructions for the system) that may implement a heuristic. The `oasis_tac` tactic takes as a parameter a limit on the search depth. Calling this tactic with a depth of 2 on the example's initial goal produces the following output:

```
> by (oasis_tac (claset()) (simpset()) 2);
Applying simplification steps...
Applying safe inference steps...
Now trying : 12...Done!
Now trying : 11...Done!
...
No subgoals!
```

The `oasis_tac` tactic automatically finds the required proof, using exhaustive depth-first search. Isabelle prints the just-proved theorem (omitted from the output), and the message "`No subgoals!`" □

*Practical results.* The OASIS schema translator consists of approximately 2029 lines of ML code. At present, the generic OO theory is 632 lines of Isabelle/HOL code, and 49 theorems. The input SEPIA schema currently includes 6 class definitions, 18 method definitions, and 5 constraints.[1] The Isabelle/HOL theory and ML files generated for this schema comprise 162 lines of code.

Table 1 shows experimental results for verifying the safety of two methods of class

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

| METHOD | CONSTRAINT | PROOF TIME |
|---|---|---|
| CNode::removeNodeOrLink | $c_1$ | 3.35s. |
| CNode::removeNodeOrLink | $c_2$ | 1.04s. |
| CNode::removeNodeOrLink | $c_3$ | 1.06s. |
| CNode::removeNodeOrLink | $c_4$ | 161.77s. |
| CNode::removeNodeOrLink | $c_5$ | 109.63s. |
| CNode::createCNodeIn | $c_1$ | 10.93s. |
| CNode::createCNodeIn | $c_2$ | 2.89s. |
| CNode::createCNodeIn | $c_3$ | 2.88s. |
| CNode::createCNodeIn | $c_4$ | 222.46s. |
| CNode::createCNodeIn | $c_5$ | 551.49s. |

**Table 1.** Some Experimental Results for Method Safety

CNode, with respect to the constraints in Figure 1. All proof times are in seconds, with Isabelle running on a SUN 296 MHz Ultra-SPARC-II, under Solaris. The times given are only a rough guide of the efficiency of the automated method safety proofs. The times indicate that the trivial proofs are immediately solved by the theorem prover. For example, the combination of constraint $c_2$ and method removeNodeOrLink operate on different attributes. As discussed in the previous section, the proof is trivial and is done using straightforward term rewriting, by the Simplifier. The real power of the theorem prover reveals itself in the cases where the constraint and method operate on the same attributes and/or persistent roots. For example, the combination of constraint $c_4$ and method removeNodeOrLink (illustrated in the previous sections) takes 161.77 seconds. In this case, the proof involves many tedious steps.

## 7 Related work

Theorem prover techniques have been applied in the context of *relational databases* using formalisms such as Boyer-Moore logic [14] and Hoare logic [11], for the verification ([14]) and deductive synthesis ([11]) of transactions that respect a number of static integrity constraints. Our work shares similarities with these approaches, but it is based on an object-oriented framework and uses a modern theorem prover. At the time the above authors published their work, theorem prover technology was still in an early stage of development. For example, in [14], higher-order extensions are made to a first order theorem prover, and standard data types such as natural numbers and sets are defined from scratch. Nowadays, these modelling capabilities are available "off the shelf," using a standard HOL theorem prover.

Within an *object-oriented database* framework, Benzaken *et al* [3] study the problem of method verification with respect to static integrity constraints, using *abstract interpretation*. A tableaux reasoner is used to analyse some properties of application code using first-order logic. However, important issues such as transactions, type information, and object sharing are not addressed.

Theorem prover techniques that use higher-order logic are applied in the context of *object-oriented programming* in [13,8]. Santen [13] uses Isabelle/HOL to reason about

class specifications in Object-Z. A trace semantics is encoded to support reasoning about behavioural relations between classes. Jacobs *et al* study the verification of Java code, using the PVS theorem prover [8]. A tool called LOOP (Logic of Object-Oriented Programming) translates Java classes into the higher-order logic of the PVS system. The semantics of their approach is based on coalgebras, in particular to support proofs about refinement relations. Jacobs *et al* address a number of issues that we do not, such as exceptions, termination, and recursion. In contrast to the work on object-oriented programming, we study database transactions on a persistent object store, rather than the behaviour of individual objects.

The work in this paper extends our previous work ([15]) by considering additional topics such as inheritance and heterogeneity. Here, emphasis is placed on modelling an object-oriented database schema in HOL, and on the extensions to the Isabelle system to provide automated reasoning for such a database schema. We build on the ideas of Doherty and Hull [6] in which database state changes are encoded as *delta values* (a difference between database states). In their work, delta values are used to describe proposed updates in the context of cooperative work; whereas in our work, delta values are used to cope with intra-transaction parallelism due to set-oriented updates.

## 8 Conclusions and future work

We have shown how to represent the constructs of an object-oriented database specification language in the higher-order logic of the Isabelle theorem prover. To achieve this, we defined an Isabelle theory of objects, which resembles the type-tagged memory of a persistent object store. The constructs of the specification language are defined as generic higher-order operations in this theory. Higher-order logic allows us to achieve schema-independent reasoning: we have proved theorems about the generic operations that are used in reasoning about specific database operations.

We presented some of our experimental results on the static analysis of database integrity. The example proof shown in Section 6 involves a combination of typical object-oriented features (namely, heterogeneous collections, abstract methods, down-casting, late binding, and nil references). This example is representative of the interaction of language features encountered in many object-oriented applications. The example schema we are working with is based on the generic graph editing functionality of a real system (the SEPIA system [16]). All 90 method safety requirements in the case study could be verified automatically, using the Isabelle tool. It is worth mentioning that our initial specification contained a few bugs, such as forgotten nil-checks. These kinds of errors in the schema are easily overlooked by the specifier, but immediately spotted by the theorem prover.

Our tool is not limited to transaction safety analysis. Because the theory used by the tool is based on very general semantics properties of the update language, we expect our experimental results to be extendible to the kinds of proof requirements encountered in other application areas, where reasoning about the semantics of database operations is needed. We are currently looking at applications of the OASIS reasoning tool in the areas of workflow and cooperative work, for the verification of e.g., compensation requirements (that is, proofs that one method compensates the results of another).

## References


1. F. Bancilhon, C. Delobel, and P. Kanellakis, editors. *Building an Object-oriented Database System: The Story of O2*. Morgan Kaufmann, 1992.
2. M. Benedikt, T. Griffin, and L. Libkin. Verifiable properties of database transactions. In *Proceedings of Principles of Database Systems (PODS)*, pages 117–127, 1996.
3. V. Benzaken and X. Schaefer. Static management of integrity in object-oriented databases: Design and implementation. In *Extending Database Technology (EDBT)*, March 1998.
4. A. J. Bernstein, D. S. Gerstl, W.-H. Leung, and P. M. Lewis. Design and performance of an assertional concurrency control system. In *Proceedings of ICDE*, pages 436–445, Orlando, Florida, February 1998.
5. R. G. G. Cattell and Douglas K. Barry, editors. *The Object Database Standard: ODMG 2.0*. Morgan Kaufmann Publishers, San Francisco, California, 1997.
6. M. Doherty, R. Hull, M. Derr, and J. Durand. On detecting conflict between proposed updates. In *International Workshop on Database Programming Languages (DBPL)*, Gubbio, Italy, September 1995.
7. Isabelle. `http://www.cl.cam.ac.uk/Research/HVG/isabelle.html`.
8. B. Jacobs, J. van den Berg, M. Huisman, M. van Berkum, U. Hensel, and H. Tews. Reasoning about Java Classes (Preliminary Report). In *Proceedings of OOPSLA*, 1998. To appear.
9. Cris Pedregal Martin and Krithi Ramamritham. Delegation: Efficiently rewriting history. In *Proceedings of ICDE*, pages 266–275, Birmingham, U.K., April 1997.
10. Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *LNCS*. Springer-Verlag, 1994.
11. Xiaolei Qian. The deductive synthesis of database transactions. *ACM Transactions on Database Systems*, 18(4):626–677, December 1993.
12. 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.
13. Thomas Santen. A theory of structured model-based specifications in Isabelle/HOL. In *Proc. of the 1997 International Conference on Theorem Proving in Higher Order Logics (TPHOLs97)*, Lecture Notes in Computer Science. Springer-Verlag, 1997.
14. Tim Sheard and David Stemple. Automatic verification of database transaction safety. *ACM Transactions on Database Systems*, 14(3):322–368, September 1989.
15. David Spelt and Herman Balsters. Automatic verification of transactions on object-oriented databases. In *Proceedings of the Workshop on Database Programming Languages (DBPL)*, Estes Park, Colorado, 1997.
16. N. Streitz, J. Haake, J. Hannemann, A. Lemke, W. Schuler, H. Schuett, and M. Thuering. SEPIA: A cooperative hypermedia authoring environment. In *ACM Conference on Hypertext (ECHT)*, pages 11–22, Milano, Italy, 1992.
17. 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.



10.1007/b107031130026