Automatic Verification of Transactions on an Object-Oriented Database*

David Spelt[1], Herman Balsters[2]

University of Twente, Enschede, The Netherlands

Abstract. In the context of the object-oriented data model, a compile-time approach is given that provides for a significant reduction of the amount of run-time transaction overhead due to integrity constraint checking. The higher-order logic Isabelle theorem prover is used to automatically prove which constraints might, or might not be violated by a given transaction in a manner analogous to the one used by Sheard and Stemple (1989) for the relational data model. A prototype transaction verification tool has been implemented, which automates the semantic mappings and generates proof goals for Isabelle. Test results are discussed to illustrate the effectiveness of our approach.
Keywords: object-oriented databases, transaction semantics, transaction verification

1 Introduction

Static integrity constraints are essential in mission-critical application domains, where one wants to offer integrity preserving update operations to clients. One way to enforce database integrity is by testing at run-time those constraints that are possibly violated by a transaction before allowing the transaction to commit. Various techniques, surveyed in [1], have been proposed to optimize such a test for a limited class of simple constraints, such as key and referential integrity. But commiting complex transactions on large amounts of data becomes increasingly difficult if the constraint language is extended to include full first-order logic formula with bounded quantifications over arbitrary collection types.

A second approach towards integrity maintenance aims at a compile-time reduction of the amount of run-time transaction overhead due to integrity constraint checking. This approach was first introduced by Sheard & Stemple ([2]) for the relational data model. It uses a theorem prover to verify that a transaction will never raise an integrity conflict, provided that the database was in a consistent state before the transaction was executed. Transactions are complex updates involving multiple relations, whereas the constraint language includes quantifications and aggregate constructs. These are related to expressions in higher-order logic for automatic proof assistance.

Another related compile-time approach is proposed in [3,4]. It exploits several techniques of *abstract interpretation* for the task of compile-time transaction verification in an O2 database system extended with a notion of declarative integrity

* Our e-mail addresses are: [spelt,balsters]@cs.utwente.nl

constraints. Their analysis starts with a simple compilation technique. It identifies those constraints that will certainly not be affected by a transaction because different attributes or even different class extents are accessed. For instance, a transaction that only changes the age of a person can never violate a constraint that does not access this field. Recently, in [5], this approach was augmented with a second, more powerful analysis. For each combination of transaction and constraint that could not be proved safe in the first step, it applies Dijkstra's concept of *predicate transformer*. This yields a simple first-order logic formula which is automatically verified by a theorem prover. A major difference between this approach and the one presented in [2] is that the latter fully exploits the (denotational) semantics of a database specification, whereas the former only takes some global abstract properties of the semantics into account.

The above compile-time strategies supplement the existing run-time techniques: they can filter out the set of relevant constraint checks, thus allowing the run-time optimizer to focus on a restricted set of constraint predicates that could not be proved safe at compile-time.

Our work extends the work of [2]. Rather than a relational model, it uses a powerful specification and verification environment for object-oriented databases with transactions and integrity constraints. The specification framework uses TM [6,7], a typed formal specification language based on the well-known ideas of Cardelli [8], extended with logic formalism and sets [9]. The declarative flavour of this language permits compile-time transaction verification using a theorem prover, while retaining ODMG compliancy [10].

The verification framework uses higher-order logic. Specifications in TM are automatically mapped to expressions in higher-order logic (HOL), such that the consistency requirements can be given as input to the higher-order logic Isabelle theorem prover [11] for automatic proof assistance.

The rest of this paper is organized as follows. Section 2 introduces the TM data model and gives an example of a job agency service. This example was also studied by [2]. In Section 3, the Isabelle theorem prover is introduced, along with a motivation for why the HOL theory is used. We then proceed with a sketch of how TM database schemas are represented in HOL. Section 5, discusses transaction verification using the Isabelle theorem prover and shows how standard Isabelle tactics can be used to implement a transaction verifier in HOL. In Section 6, we then compare our work to several related compile-time approaches. We finish by stating our conclusions and give directions for future work.

2 Database Definition in TM

This section introduces the TM data model, using the job agency service example [2], which is re-engineered to the object-oriented data model. Objects in the database include people who apply for, and are placed with certain jobs. A job can be shared by multiple employees. Skills are required to execute jobs, and people should have abilities that satisfy these requirements. TM extends the ODMG interface definitions with formal specifications of methods, transactions

and integrity constraints. The non-procedural declarative characteristic of these additional features enable compile-time verification by a theorem prover.

```
interface Skill
(  extent SKILL)
{
   attribute String description;
};
interface Job
(  extent JOB)
{
   attribute String description;
   attribute Set<Skill> req_skills;
   attribute Boolean placed;
};
interface Person
(  extent PERS
   key ssn)
{
   attribute String ssn;
   attribute Int age;
   attribute Set<Job> applications;
   attribute Set<Struct(job:Job,sal:Int)> placements;
   attribute Set<Skill> abilities;
   Person allocate(in Job j, in Int salary) =
         self except (placements = placements + set struct(job : j, sal : salary),
                      applications = applications − set j)
};
```

2.1 Methods

Methods are specified using OQL, augmented with an additional **except**-construct that enables modification of an object (a mutable and shared value) or record (a non-mutable and non-shared value). The `allocate` method in the Person interface allocates a job to a person. It does so by "moving" the job argument from the `applications` set to the `placements` set, where it is paired with a salary value. The method returns the modified Person-object (**self**).

2.2 Transactions

Transactions in our framework are specified using a high-level declarative update language. An **update**-construct provides for the declarative update of arbitrary collections of objects. It takes a sequence of OQL query blocks and commits to the database all (possibly) modified and newly created mutable objects as indicated by these blocks. Below, the reader finds several examples of complex

transactions using this construct. Actually, these are the same — but reegineered — ones that also appear in the Sheard & Stemple paper.

The following transaction subscribes a new person to the job agency service. The transaction takes as input parameters a person-`ssn` and an initial set of skills; the `applications` and the `placements`-attributes are initialized with the empty set value.

Transaction *Subscribe*(**in String** *pssn*, **in Set** < Skill > *pskills*)
Preconditions
  forall *p* **in** PERS : $p \cdot \texttt{ssn} \neq pssn$
Begin
  update Person(`ssn` : *pssn*, `applications` : **set**(), `placements` : **set**(),
    `abilities` : *pskills*)
End;

Observe that it is necessary to add the precondition, for otherwise the key-constraint might be violated. Our transaction verification system would actually report a potential conflict on this constraint, if the condition was omitted.

Transaction *Hire*(**in** String *assn*, **in** Job *j*, **in Int** *salary*)
Preconditions
  exists *p* **in** PERS : $p \cdot \text{ssn} = assn$
Begin
  update (*j* **except** (placed = **true**), **select** $p \cdot \text{allocate}(j, salary)$
    from PERS *p*
    where $p \cdot \text{ssn} = assn$)
End;

The *Hire* transaction places an applicant on a particular job: observe that two query blocks are supplied to the update construct. The first block sets the `placed` field of the job object to true, while the second block applies the `allocate` method to the Person-object that matches the `ssn` supplied as an input parameter to the transaction. There is no ordering imposed on the execution of these blocks and parallelisation is allowed provided that there are no conflicts of multiple incompatible parallel updates applied to the same object.

Finally, the *Fire* transaction removes a job from the `placements`-attribute. It also sets the `placed`-field in the Job-object to **false** if there is no other person placed in that same job:

Transaction *Fire*(**in** String *assn*, **in** Job *j*)
Preconditions
  exists *p* **in** PERS : $p \cdot \texttt{ssn} = assn$
Begin
  update (*j* **except** (placed = **exists** *x* **in** PERS : **exists** *y* **in** $x \cdot$ `placements` :
    $y \cdot \texttt{job} = j$ **and** $x \cdot \texttt{ssn} \neq assn$),
    select *p* **except** (`placements` = **select** *x* **from** $p \cdot$ `placements` *x*
      where $x \cdot \texttt{job} \neq j$)
    from PERS *p*
    where $p \cdot \text{ssn} = assn$)
End;

2.3 Integrity Constraints

TM extends the ODMG data model with integrity constraints, which can be arbitrary well-typed boolean-valued OQL expressions, ranging over the extents of the database. This generalizes the notion of key constraints in the ODMG data model, which in fact are simple first-order formulae on a single class extent. Although the Object Database Management Group has the provision to also include a more general notion of integrity constraints in a future language release [10], constraint specification remains limited at present. The use of OQL as a constraint definition language is fairly straightforward, as illustrated by the examples below. In addition to the key constraint that was already part of the schema definition, we add the following constraints to our job agency service specification; these constraints will be the subject of transaction verification in Section 5

Example 1. All persons applying for a job should have the required skills to execute those jobs:

$$C_1 : \textbf{forall}\ x\ \textbf{in}\ \texttt{PERS}\ :\ \textbf{forall}\ j\ \textbf{in}\ x \cdot \texttt{applications}\ :\ j \cdot \texttt{req\_skills} \leq x \cdot \texttt{abilities}$$

Example 2. The `placed`-field in the Job-class is a redundant field.

$$C_2 : \textbf{forall}\ x\ \textbf{in}\ \texttt{JOB}\ :\ x \cdot \texttt{placed} = \textbf{exists}\ y\ \textbf{in}\ \texttt{PERS}\ :\ \textbf{exists}\ z\ \textbf{in}\ y \cdot \texttt{placements}\ :\ z \cdot \texttt{job} = x$$

Example 3. A person can never simultanously apply for and be placed in one and the same job:

$$C_3 : \textbf{forall}\ x\ \textbf{in}\ \texttt{PERS}\ :\ \textbf{forall}\ y\ \textbf{in}\ x \cdot \texttt{placements}\ :\ \textbf{not}(y \cdot \texttt{job}\ \textbf{in}\ x \cdot \texttt{applications})$$

Example 4. All persons in the database are younger than 65:

$$C_4 : \textbf{forall}\ x\ \textbf{in}\ \texttt{PERS}\ :\ x \cdot \texttt{age} \leq 65$$

3 Introduction to Isabelle/HOL

Isabelle/HOL is a general-purpose higher-order logic-based theorem proof system. Using the system's built-in deductive system, mechanical reasoning is supported for the most commonly used data types in programming languages, such as booleans, integers, characters, strings, tuples, lists and sets. Isabelle provides an OQL-like functional language interface, supporting complex values nested up to arbitrary depth. From a database perspective, the Isabelle/HOL specification language relates to the NF2 data model, extended in the sense that attributes may be arbitrary collections and tuples, rather than relations. This makes the HOL-language particularly suitable for representing object-oriented database schemas.

Isabelle specifications are called *theories*. A theory consists of a collection of axioms and definitions. Our system generates an Isabelle theory file from a TM database specification. The definitions of this newly added theory being the definitions of methods, transactions, and constraints. Properties can be asserted and proved about these definitions by calling *tactics*, which are implementations of individual proof steps. The Isabelle/HOL package provides powerful tactics that can automate seemingly highly complex proofs. Predefined automatic tactics are available for simplification — term-rewriting with an arbitrary set of (conditional) term-rewriting lemmas is supported — and a natural deduction solver. The *Simplifier* performs term-rewriting with an arbitrary set of theorems of the form

$$H \Rightarrow LHS = RHS$$

Such rules read in the obvious straightforward manner: a term unifying with the expression on the left-hand side of the equation (LHS) is rewritten to the term that appears on the right-hand side (RHS) provided that the hypothesis (H) holds. The default Isabelle/HOL simplifier already installs a large collection of standard reduction rules for HOL, but new rules can be easily added to customize the Simplifier to a particular domain.

The *Natural Deduction Solver* uses a set of introduction and elimination properties for higher-order logic to automate natural deduction inferences. The tool implements a depth-first search strategy. It systematically breaks up the goals that are left after simplification in a number of smaller sub-goals. Variables, introduced by the use of quantifiers, can be automatically instantiated, allowing backtracking between different alternative unifiers. Before each inference step, the solver will call the Simplifier to allow further syntactic reductions to take place. Usually, this amounts to a highly complex proof structure and even seemingly simple proofs may take hundreds (but small, easy to automate) steps. It is not necessary, however, to understand the full details of the algorithms that are used, and the interested reader is further referred to [11].

In Section 5 we discuss how these tools can be used for the task of verifying transaction safety. First we do the representation of OO database schema's in HOL and show how (parts of) the example specification are translated. For the target language, a simply typed lambda calculus is used with OQL, rather than specific Isabelle syntax style, to slightly simplify the presentation. Thus we abstract from certain pecularities of the Isabelle/HOL system. For instance, the Isabelle system uses non-labeled tuples instead of labeled records, but a standard encoding can be used where (1) the order in which the labels occur is fixed and (2) projections are replaced by the typical operations `fst` and `snd`.

4 Mapping OO Language Features to Isabelle/HOL

We first define a structural mapping of the class structures of the ODMG data model to HOL records as a means of implementing these structures. An additional `id`-field of type integer is used to represent an object's identity. At the

same time, class references in compound object types are replaced by pointer (oid) references in the form of integer-values, instead of copies of the objects themselves. The database itself is also represented as a record structure, called object store (OS), which holds entries for each extent of the database. The associated object store of our example specification becomes a record structure:

$$\begin{aligned} OS = \textbf{struct}(&\texttt{SKILL} : \textbf{Set} < \textbf{struct}(\texttt{id} : \textbf{Int}, \texttt{description} : \textbf{String}) > \\ &\texttt{JOB} : \textbf{Set} < \textbf{struct}(\texttt{id} : \textbf{Int}, \texttt{description} : \textbf{String}, \\ &\qquad \texttt{req\_skills} : \textbf{Set} < \textbf{Int} >, \\ &\qquad \texttt{placed} : \textbf{Boolean}) >, \\ &\texttt{PERS} : \textbf{Set} < \textbf{struct}(\texttt{id} : \textbf{Int}, \texttt{ssn} : \textbf{String}, \texttt{age} : \textbf{Int}, \\ &\qquad \texttt{applications} : \textbf{Set} < \textbf{Int} >, \\ &\qquad \texttt{placements} : \textbf{Set} < \textbf{struct}(\texttt{job} : \textbf{Int}, \texttt{sal} : \textbf{Int}) > \\ &\qquad \texttt{abilities} : \textbf{Set} < \textbf{Int} >) >, \end{aligned}$$

Integrity constraints are represented as functions of type $OS \rightarrow Bool$ in the HOL framework. By the introduction of object identifiers, however, we have created some form of indirection which slightly complicates such a translation. For instance, a constraint expression of the form

$$\begin{aligned} &\lambda\, os : OS \bullet \textbf{forall}\, x\, \textbf{in}\, os \cdot \texttt{PERS} : \\ &\qquad \textbf{forall}\, j\, \textbf{in}\, x \cdot \texttt{applications} : j \cdot \texttt{req\_skills} \leq x \cdot \texttt{abilities} \end{aligned}$$

can no longer be maintained in a context where the variable j is an object reference of type **Int**. To select the `req_skills`-attribute of j we now first need to query the Job-extent. This form of indirection is provided for in the translation; i.e., functions like

$$\begin{aligned} get\_Job \equiv \lambda\, os : OS \bullet o : \textbf{Int} \bullet \textbf{elmt}(&\textbf{select}\, x \\ &\textbf{from}\, os \cdot \texttt{JOB}\, x \\ &\textbf{where}\, x \cdot \texttt{id} = o) \end{aligned}$$

will be automatically inserted at appropriate places. Note that the above function is generated for each class C. The function takes an object reference o and retrieves the corresponding full object representation from the associated class extent.

Example 5. The following Isabelle/HOL function representation is generated for the constraint C_1

$$\begin{aligned} C_1 \equiv \lambda\, os : OS \bullet \textbf{forall}\, x\, \textbf{in}\, os \cdot \texttt{PERS} : \textbf{forall}\, j\, \textbf{in}\, x \cdot \texttt{applications} : \\ (get\_Job\ os\ j) \cdot \texttt{req\_skills} \leq x \cdot \texttt{abilities} \end{aligned}$$

Aside from the user-defined *explicit constraints*, the schema also has a number of *implicit constraints*. Implicit schema constraints include constraints for referential integrity and object identity. These will be automatically generated during the translation to HOL.

Example 6. The id-field acts as a key to the PERS-extent.

$$C_5 \equiv \lambda\, os : OS \bullet \textbf{forall}\, x\, \textbf{in}\, os \cdot \texttt{PERS} : \textbf{forall}\, y\, \textbf{in}\, \texttt{PERS} : x \cdot \texttt{id} = y \cdot \texttt{id}\, \textbf{implies}\, x = y$$

Example 7. The oid's in the applications-field refer to items in the JOB-table.

$$C_6 \equiv \lambda\, os : OS \bullet \textbf{forall}\, x\, \textbf{in}\, os \cdot \texttt{PERS} : \textbf{forall}\, y\, \textbf{in}\, x \cdot \texttt{applications} : : \textbf{exists}\, z\, \textbf{in}\, os \cdot \texttt{JOB} : y = z \cdot \texttt{id}$$

The semantics of transactions is functional: transactions are formally represented as functions of type $OS \rightarrow t_1 \rightarrow \cdots \rightarrow t_k \rightarrow OS$ in the Isabelle framework, where the types $t_1 \cdots t_k$ represent the types of optional input parameters. At the semantical level, the **update** primitive constructs a new object store value, where all possibly modified object representations resulting from the functional evaluation of the OQL sub-expressions are unioned with the unmodified objects for each extension. The collection of objects that are not modified is easily obtained by inspecting the id-field. Furthermore, method calls are replaced by substituting the TM-OQL expressions defining their functionality. At present, our prototype does not support recursive method calls.

Example 8. The following Isabelle representation function is generated for the *Hire* transaction:

$$\begin{array}{l} \mathit{Hire} \equiv \lambda\, os : OS \bullet \lambda\, assn : \textbf{String} \bullet \lambda\, j : \textbf{Int} \bullet \lambda\, salary : \textbf{Int} \bullet \\ \quad \textbf{struct}(\texttt{SKILL} : os \cdot \texttt{SKILL}, \\ \qquad \texttt{JOB} : \{(\mathit{get\_Job}\ os\ j)\, \textbf{except}\, (\texttt{placed} = \textbf{true})\} + \textbf{select}\, x \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{from}\, os \cdot \texttt{JOB}\, x \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{where}\, x \cdot \texttt{id} \notin \{j\} \\ \qquad \texttt{PERS} : (\textbf{select}\, p\, \textbf{except}\, (\texttt{placements} = p \cdot \texttt{placements} + \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{set}(\textbf{struct}(\texttt{job} : j, \texttt{sal} : salary)), \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{applications} = \texttt{applications} - \textbf{set}\, \texttt{j}) \\ \qquad\qquad\qquad\qquad \textbf{from}\, os \cdot \texttt{PERS}\, p \\ \qquad\qquad\qquad\qquad \textbf{where}\, p \cdot \texttt{ssn} = assn) + (\textbf{select}\, p \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{from}\, os \cdot \texttt{PERS}\, p \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{where}\, p \cdot \texttt{ssn} \neq assn) \end{array}$$

The above function generates modifications to the JOB as well as the PERS-extent, while the SKILL-extent is not modified. Note that the job-object j is expanded to allow the placed field to be changed. The expression on the left-hand side of the union (+) denotes the collection of modified objects, while the collection of objects that are not updated appears on the right-hand side. The precondition of the transaction is stored in a seperate definition and can be treated as an ordinary constraint.

Example 9. The pre-condition of the *Hire*-transaction is represented as a function:

$$\begin{array}{l} \mathit{Pre\_Hire} \equiv \lambda\, os : OS \bullet \lambda\, assn : \textbf{String} \bullet \lambda\, j : \textbf{Int} \bullet \lambda\, salary : \textbf{Int} \bullet \\ \qquad \textbf{exists}\, p\, \textbf{in}\, \texttt{PERS} : p \cdot \texttt{ssn} = assn \end{array}$$

5 Automatic Transaction Verification in Isabelle/HOL

Once the schema has been translated to Isabelle, its automatic proof tactics as mentioned in Section 3 can be used to statically identify the integrity constraints that are possibly violated by a transaction and the ones that are not. Transaction verification starts by asserting as a *proof goal* the fact that a constraint will never be violated by the execution of a transaction. Given an Isabelle transaction representation function T, an associated pre-condition representation Pre_T, and a constraint representation function C, the following goal needs to be verified:

$$C\,(os) \wedge (Pre\_T\ os\ p_1 \cdots p_k) \Rightarrow C\,(T\ os\ p_1 \cdots p_k)$$

for arbitrary object store os and input parameters $p_1 \cdots p_k$. With slight syntactic modifications — into ASCII — theorems of the above form can be given as input and mechanically verified by the Isabelle theorem prover. In our analysis, we use both the Simplifier and the Natural Deduction Solver — the basic tools (tactics) for automatic proof in Isabelle, as introduced in Section 3.

The rest of this section discusses in some more detail how these tools can be used for the task of compile-time transaction verification. In the next paragraph, we demonstrate how the term-rewriting tool applies to implement a simple, fairly rough analysis, analogous to the *path analysis* presented in [3–5]. The harder cases are then further processed by the natural deduction solver for a more detailed analysis, which is the subject of Section 5.2.

5.1 A Simple Analysis using the Simplifier

When starting an automatic proof, Isabelle first tries to simplify the initial proof goal as much as possible. This is done by term-rewriting with the Simplifier tool. The default Isabelle/HOL Simplifier, however, is not directly suitable to enable verification of a robust class of transactions over arbitrary database schemas, thus requiring some extensions. Extensions to the Simplifier will be made by adding some new rewrite-rules, such that at least the trivial cases — of a transaction and constraint operating on different parts of the database — can be identified. The following example illustrates how the Isabelle Simplifier can be used for verifying transaction safety, and which extensions have been made.

$$\mathtt{C_4(Hire}\ os\ ssn\ j) \qquad (1)$$

$$
\begin{array}{l}
= \textbf{forall}\, x\, \textbf{in} \\
\quad \texttt{struct}(\texttt{SKILL} : os \cdot \texttt{SKILL}, \\
\qquad \texttt{JOB} : \{(get\_Job\ os\ j)\, \textbf{except}\, (\texttt{placed} = \textbf{true})\} + \textbf{select}\, x \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{from}\, os \cdot \texttt{JOB}\, x \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{where}\, x \cdot \texttt{id} \notin \{j\} \\
\qquad \texttt{PERS} : (\textbf{select} p\, \textbf{except}\, (\texttt{placements} = p \cdot \texttt{placements} + \\
\qquad\qquad\qquad\qquad\qquad\qquad \texttt{set}(\texttt{struct}(\texttt{job} : j, \texttt{sal} : salary)), \\
\qquad\qquad\qquad\qquad \texttt{applications} = \texttt{applications} - \textbf{set}\, \mathrm{j}) \\
\qquad\qquad \textbf{from}\, os \cdot \texttt{PERS}\, p \\
\qquad\qquad \textbf{where}\, p \cdot \texttt{ssn} = assn) + \textbf{select}\, p \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{from}\, os \cdot \texttt{PERS}\, p \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{where}\, p \cdot \texttt{ssn} \neq assn \\
\qquad) \cdot \texttt{PERS}\ :\ x \cdot \texttt{age} \leq 65
\end{array} \tag{2}
$$

$$
\begin{array}{l}
= \textbf{forall}\, x\, \textbf{in}\, ((\textbf{select}\, p\, \textbf{except}\, (\texttt{placements} = p \cdot \texttt{placements} + \\
\qquad\qquad\qquad\qquad\qquad\qquad \texttt{set}(\texttt{struct}(\texttt{job} : j, \texttt{sal} : salary)), \\
\qquad\qquad\qquad\qquad \texttt{applications} = \texttt{applications} - \textbf{set}\, \mathrm{j}) \\
\qquad\quad \textbf{from}\, os \cdot \texttt{PERS}\, p \\
\qquad\quad \textbf{where}\, p \cdot \texttt{ssn} = assn) + \textbf{select}\, p \\
\qquad\qquad\qquad\qquad\qquad \textbf{from}\, os \cdot \texttt{PERS}\, p \\
\qquad\qquad\qquad\qquad\qquad \textbf{where}\, p \cdot \texttt{ssn} \neq assn)\ :\ x \cdot \texttt{age} \leq 65
\end{array} \tag{3}
$$

$$
\begin{array}{l}
= (\textbf{forall}\, x\, \textbf{in}\, (\textbf{select}\, p\, \textbf{except}\, (\texttt{placements} = p \cdot \texttt{placements} + \\
\qquad\qquad\qquad\qquad\qquad\qquad \texttt{set}(\texttt{struct}(\texttt{job} : j, \texttt{sal} : salary)), \\
\qquad\qquad\qquad\qquad \texttt{applications} = \texttt{applications} - \textbf{set}\, \mathrm{j}) \\
\qquad\quad \textbf{from}\, os \cdot \texttt{PERS}\, p \\
\qquad\quad \textbf{where}\, p \cdot \texttt{ssn} = assn)\ :\ x \cdot \texttt{age} \leq 65)\, \textbf{and} \\
\quad (\textbf{forall}\, x\, \textbf{in}\, (\textbf{select}\, p \\
\qquad\quad \textbf{from}\, os \cdot \texttt{PERS}\, p \\
\qquad\quad \textbf{where}\, p \cdot \texttt{ssn} \neq assn)\ :\ x \cdot \texttt{age} \leq 65)
\end{array} \tag{4}
$$

$$
\begin{array}{l}
= (\textbf{forall}\, p\, \textbf{in}\, os \cdot \texttt{PERS}\ :\ (p \cdot \texttt{ssn} = assn)\, \textbf{implies} \\
\qquad p\, \textbf{except}\, (\texttt{placements} = p \cdot \texttt{placements} + \\
\qquad\qquad\qquad\qquad \texttt{set}(\texttt{struct}(\texttt{job} : j, \texttt{sal} : salary)), \\
\qquad\qquad \texttt{applications} = p \cdot \texttt{applications} - \textbf{set}\, j) \cdot \texttt{age} \leq 65)\, \textbf{and} \\
\quad (\textbf{forall}\, p\, \textbf{in}\, os \cdot \texttt{PERS}\ :\ (p \cdot \texttt{ssn} \neq assn)\, \textbf{implies}\, (p \cdot \texttt{age} \leq 65)
\end{array} \tag{5}
$$

$$
\begin{array}{l}
= (\textbf{forall}\, p\, \textbf{in}\, os \cdot \texttt{PERS}\ :\ (p \cdot \texttt{ssn} = assn)\, \textbf{implies}\, (p \cdot \texttt{age} \leq 65)\, \textbf{and} \\
\quad (\textbf{forall}\, p\, \textbf{in}\, os \cdot \texttt{PERS}\ :\ (p \cdot \texttt{ssn} \neq assn)\, \textbf{implies}\, (p \cdot \texttt{age} \leq 65)
\end{array} \tag{6}
$$

The above example traces the systematic reduction of the consequent of the goal that is generated for verifying that the *Hire*-transaction preserves integrity of constraint C_4. The proof starts by substituting the transaction in the constraint predicate (1) and unfolding the database specific definitions of the transaction and constraint (2). In general, this results in a highly complex proof term. Fortunately, as already suggested by [2], many of the complex terms can be easily reduced using standard[2] reduction rules for the tuple datatype:

---

[2] In Isabelle/HOL syntax these rules are actually encoded at a much lower level. As was already mentioned in Section 3, Isabelle uses non-labeled tuples instead of labeled records, and the reductions are realized by using standard rules involving the typical operations `fst` and `snd`.

$$\text{[REC1]} \qquad \mathbf{struct}(a_1 : e_1, \cdots, a_n : e_n) \cdot a_i = e_i$$

$$\text{[REC2]} \qquad i \in n \Rightarrow e\,\mathbf{except}(a_1 : e_1, \cdots, a_n : e_n) \cdot a_i = e_i$$

$$\text{[REC3]} \qquad i \notin n \Rightarrow e\,\mathbf{except}(a_1 : e_1, \cdots, a_n : e_n) \cdot a_i = e \cdot a_i$$

The above rules allow the Simplifier to identify those cases of a transaction and constraint operating on different class extents such that integrity is trivially preserved. For instance, application of the first rule [REC1] to (2), discards the update operation on the Job-extent. Note that such an update is 'irrelevant' in the presence of the current constraint predicate, since the constraint only takes the Person-extent into account. At this point, simplification with the default Simplifier stops : none of the standard rewrite-rules matches with the remaining proof term (3) and additional knowledge about the general structure of the proof goals that are generated is needed, to proceed with simplification.

By studying the cases where the Simplifier got stuck during a transaction safety proof, several recurring patterns could be identified. For instance, in Section 4, we defined the contents of the extent of a class after an update operation occurs as the union (+) of the set of objects that got changed and the set of objects that did not change. Combining this with the assumption that many constraint predicates quantify over class extents, we will be frequently left with terms that match with one of the following rules

$$\text{[UN\_ALL]} \qquad (\mathbf{forall}\, x\, \mathbf{in}\, (A + B) \,:\, \phi(x)) = (\mathbf{forall}\, x\, \mathbf{in}\, A \,:\, \phi(x))\, \mathbf{and}\, (\mathbf{forall}\, x\, \mathbf{in}\, B \,:\, \phi(x))$$

$$\text{[UN\_EX]} \qquad (\mathbf{exists}\, x\, \mathbf{in}\, (A + B) \,:\, \phi(x)) = (\mathbf{exists}\, x\, \mathbf{in}\, A \,:\, \phi(x))\, \mathbf{or}\, (\mathbf{exists}\, x\, \mathbf{in}\, B \,:\, \phi(x))$$

The above rules will split universal and existential quantifications so that is discriminated between the 'modified' and the 'unmodified' case. For instance, the first rule [UN_ALL] matches with term (3) of the example proof, and the Simplifier splits the quantification resulting in (4). Note that the proposition on the left-hand side of the conjunction quantifies over the collection of modified objects, while the quantification over the collection of objects that is not modified is on the right-hand side.

At this point, the general structure of the goal gradually seems to disappear. Transactions and constraints can be expressed in many ways, and general patterns can hardly be identified. Transaction definitions, however, frequently use a select-from-where clause, making it useful to add the following reduction rules

$$\text{[DIS1]} \qquad \mathbf{forall}\, y\, \mathbf{in}\, (\mathbf{select}\, e\, (x)\ \mathbf{from}\, x\, \mathbf{in}\, A\ \mathbf{where}\, p(x)) \,:\, \phi(y) = \mathbf{forall}\, x\, \mathbf{in}\, A \,:\, p(x)\, \mathbf{implies}\, \phi(e(x))$$

$$\text{[DIS2]} \qquad \mathbf{exists}\, y\, \mathbf{in}\, (\mathbf{select}\, e\, (x)\ \mathbf{from}\, x\, \mathbf{in}\, A\ \mathbf{where}\, p(x)) \,:\, \phi(y) = \mathbf{exists}\, x\, \mathbf{in}\, A \,:\, p(x)\, \mathbf{and}\, \phi(e(x))$$

The above rules will distribute functional replacements over quantifier bodies. This enables the Simplifier to also identify combinations of transactions and constraints where — although the same class extents are involved — integrity is

trivially preserved because different attributes of the objects are accessed. For instance, application of the first rule [DIS1] to (4), will distribute the functional replacement over the quantifier body, thus resulting in (5). Now, simplification can proceed using standard reduction rules. Using [REC3], the Simplifier destroys the remaining record-update operation, and we are left with a formula that closely matches the original assumption (6). The remaining term will be solved since the Simplifier automatically asserts the original assumption

$$\textbf{forall}\, x\, \textbf{in}\, os \cdot \texttt{PERS} \;:\; x \cdot \textbf{age} \leq 65$$

as an additional rewrite rule while simplifying the consequent. □

5.2 A Detailed Analysis using the ND-Solver

Unfortunately, not all goals are as easily solved as the one that is discussed in the previous example. Often, when a transaction and constraint operate on the same parts of the database, it becomes difficult to completely solve the goal by simplification. There are many possibilities of how the final proof term may look like and there hardly seems to be a general pattern that would allow further simplification. For instance, the *Hire* transaction updates the **applications**-field from the PERS-table, which is exactly the same field that is also accessed by the integrity constraint oc_3. In this case, simplification alone cannot prove the entire goal and the following goal is left after simplification:

$$\begin{aligned}&(\textbf{forall}\, x\, \textbf{in}\, \texttt{PERS} \;:\\ &\quad \textbf{forall}\, y\, \textbf{in}\, x \cdot \textbf{applications} \;:\; \mathit{get\_Job}(os\; y) \cdot \textbf{req\_skills} \leq x \cdot \textbf{abilities})\\ &\quad \Rightarrow (\textbf{forall}\, x\, \textbf{in}\, \texttt{PERS} \;:\; \textbf{forall}\, y\, \textbf{in}\, x \cdot \textbf{applications} - \textbf{set}(j) :\\ &\qquad\qquad \mathit{get\_Job}(os\; y) \cdot \textbf{req\_skills} \leq x \cdot \textbf{abilities}) \quad (7)\end{aligned}$$

Do we need to derive another rewrite-lemma that will allow further simplification of this term? In the approach taken by Sheard & Stemple [2], further simplification would be employed by adding the following rule

$$(\textbf{forall}\, x\, \textbf{in}\, A \;:\; \phi(x)) \Rightarrow (\textbf{forall}\, x\, \textbf{in}\, (A - B) \;:\; \phi(x))$$

to their knowledge base. Indeed, by adding the above rule to the Isabelle Simplifier we could also solve the remaining proof goal. However, many of such rules can be added and one may doubt whether they would apply more frequently in other proofs. This is one of the shortcommings of their approach as mentioned in [2].

Fortunately, Isabelle largely eliminates the need for adding an extensive amount of knowledge to the Simplifier. The simplifications discussed in the previous section are usually sufficient to already yield a proof goal that can be further processed by the Natural Deduction Solver, which only employs standard lemmas by means of introduction and elimination properties for HOL. In the case of formula (7), Isabelle will invoke the introduction and elimination properties of

universal quantification and set-membership, eventually proving the validity of (7). By interleaving the slightly customized Isabelle Simplifier with the Natural Deduction Solver, a powerful transaction verifier is provided for: most of the examples can be solved in just a few seconds time.

	C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8	C_9	C_{10}	C_{11}
Hire	10.4	7.7	3.9	2.4	4.9	5.4	3.2	2.1	2.3	2.6	2.5
Fire	10.3		3.5	2.5	4.5	2.7	3.7	1.6	2.3	5.5	2.4
Subsribe	5.9	2.7	1.0	1.1	3.4	1.4	1.7	1.6	1.5	1.5	2.9

Table 1. Proof Timings for the Job Specification (in seconds)

Table 5.2 shows the proof-times for our example specification. All timings are obtained running Isabelle on an ordinary SPARC-5 workstation with 80MB of internal memory. Horizontally alligned are the constraints, while the transactions are vertically alligned. Including the implicit schema constraints for referential integrity and object identity a total of 11 constraints is listed. This generates a total of 33 proof goals, one for each combination of transaction and constraint. These are put in a ML-text file and on loading the specification, the file will be automatically processed by the Isabelle theorem prover. Only one of the goals (for C_2 and *Fire*) could not be solved automatically; constraint C_2 should be tested at run-time after the *Fire*-transaction commits.

6 Comparison with Related Work

Our work follows the line of research set out by Sheard & Stemple ([2]). In this approach, the Boyer-Moore theorem prover is used to implement a compile-time mechanism to verify constraint invariance with respect to update operations on a relational database. The initial database specifications are given in a language called ADABTPL, which are then mapped to the Boyer-Moore theorem prover for automatic proof assistance. To that end, the Boyer-Moore theorem prover is enriched with higher-order functions, and a basic theory about tuples, finite sets and natural numbers is defined, in which databases can be represented. The actual transaction safety verifier component is implemented using a *term-rewriting* system. The term-rewriter uses a large knowledge base, which stores general knowledge about the transaction and constraint language. This includes basic theorems, such as a rule asserting the commutativity of the set-union operation. Much of the power of the Sheard & Stemple system derives from adding more problem-specific rules (so-called meta-lemmas) which enable the simplification of terms that frequently appear during transaction safety analysis. Our approach using Isabelle/HOL differs in that it uses the object-oriented rather

than the relational framework. Also, we employ a novel more general verification strategy that uses *natural deduction* in addition to *term-rewriting*. This has the benefit of offering a more general proof strategy for transaction verification. Initially, we tried to follow the approach of [2], but we soon ended up adding many new non-standard rewrite-rules to the Simplifier. Often, it was doubtful if they were relevant in the context of other database specifications; the knowledge base approach of Sheard & Stemple [2] tends to tune the transaction verifier to specific example databases, rather than offering a verifier which is more broadly applicable.

In [12] another related approach is described, as employed in the DAIDA-project, which also allows for proof assistance in demonstrating constraint invariance with respect to operations on a database. The main topic of [12] did not concern constraint invariance, but incremental refinement of initial database specifications to actual database programs; the work on proof asssitance for constraint invariance is more or less a spinn-off of the actual topic of the DAIDA-project. The initial database specification is given in a language called TDL, and the TDL specification is then mapped to Abrial's language of Abstract Machines. By employing the B-tool, interactive proof assistance is offered for checking constraint invariance. The most notable difference with our approach employing Isabelle/HOL is that our system offers automatic, rather than an interactive, proof assistance. Another difference is that TM/ODMG employs an object-oriented style and is purely functional, whereas TDL has less object-oriented features and uses an explicit pre-/post-conditional style based on predicates and sets.

The later work of [3–5] follows a different approach. It exploits several techniques related to *abstract interpretation* for the task of compile-time transaction verification in an O2 database system. Their analysis starts with a simple compilation technique to identify those combinations of transaction and constraint that are certainly not in conflict because the transaction and constraint access different attributes or class extents. The same analysis is actually implemented in our system using term-rewriting with the Isabelle Simplifier tool. For those combinations of transaction and constraint that could not be proved safe in the first step, Benzaken *et al* use a second more detailed analysis. This analysis takes some details of the semantics into account. It can, for instance, prove that deletion of an object from a set does not affect a constraint that universally quantifies over it. It is not clear, however, what the exact limitations are of taking only small portions of the semantics of the application into account. In principle, the line of research set out by Sheard & Stemple [2] (and our extension of it) offers more potential: since the full semantics of the application is taken into account, we should eventually increase the amount of proofs that can be performed. Furthermore, we use a functional rather than an imperative programming language for transaction specification. It is well-known that functional languages offer a relatively clean logical structure which is more suitable for verification; in imperative languages the simple structure is destroyed by constructions such as assignment and aliasing.

7 Conclusions & Future Work

In this paper, we have outlined a framework for compile-time verification of transaction safety in an object-oriented database. It is a first attempt at generalizing the ideas of Sheard & Stemple as presented in [2] to the object-oriented data model, using modern theorem proving technology. The higher-order logic Isabelle theorem prover is used to automatically verify which constraints might, or might not be violated by a given transaction. An improved verification strategy is presented, that involves *natural deduction* in addition to *term-rewriting*. This eliminates the need for extensive customized proof strategies, and our system largely builds on general purpose proof algorithms supplied by the Isabelle/HOL package.

Tests have been done using a prototype system for a realistically large example specification, which we believe is representative of many real-world OO database applications. The example includes several complex transactions and constraints which are potentially in conflict because the same extensions, or often even the same attributes, are accessed. For instance, the constraint C_2 mentions the `placed`-field from the `JOB`-table, and the `placements`-field from the `PERS`-table. Although the same fields are updated by the *Hire* transaction, the system proves that there is actually no conflict. Such a proof can only be done using a sophisticated *semantic* analysis. Typically, these are the harder cases where our approach should offer more potential than an analysis based on an *abstract interpretation* as outlined in [3–5] which only takes some very global properties of the semantics into account.

In this paper we have highlighted some of the difficulties found in the mapping of an object-oriented database schema to HOL, but many issues remain open and full ODMG is not yet supported by our first prototype. For instance, our system does not yet support the concept of relationships, nor do we fully support the important notions of polymorphic sets and late-binding. Embedding of these language features – whose semantics is known to be difficult [13–15] — in the HOL framework remains a future challenge, but is a topic of ongoing research. At present we are experimenting using disjoint sum-types to represent polymorphic sets in the HOL-setting. Obviously, this will further complicate the proofs as additional case-splits are needed.

On the other hand, it seems that there are some ways that reasoning about the OO case is easier than for the relational case. The relational data model does not provide support of nested-sets and other complex (nested) data structures as already available in HOL. As a consequence, when mapping a relational database language to HOL, we do not fully benefit from the power of the HOL language and rather inefficient input is generated for the theorem prover.

An interesting feature of our system is that it is more broadly applicable than transaction verification; since it largely builds on standard Isabelle proof algorithms, the system should be fairly easily customized to different domains. Preliminary test results using the bank-account example of [16,17] have shown that the same proof algorithms are applicable to several forms of transaction commutativity analysis as well.

A topic that was not discussed in this paper is the generation of feedback to database designers. At present, the system only reports a 'yes', could prove, or 'no', could not prove, but eventually we would like to support some more advanced modes of feedback to database designers. For instance, designers would typically like to know why a proof actually failed or how a transaction might be corrected such that integrity will be preserved. An overview of the different modes of feedback can be found in [18] and we plan to study the implementation of a similar feedback component for our system.

References

[1] Piero Fraternali & Stefano Paraboschi, "A Review of Repairing Techniques for Integrity Maintenance," in *Proceedings First International Workshop on Rules in Database Systems, Edinburgh, Scotland, 30 August–1 September, 1993*, Norman W. Paton & M. Howard Williams, eds., Springer-Verlag, New York–Heidelberg–Berlin, 1994, 333–346.

[2] Tim Sheard & David Stemple, "Automatic verification of database transaction safety," *ACM Trans. Database Syst.* 14 (Sept., 1989), 322–368.

[3] Veronique Benzaken & Doucet, "Themis: a database programming language with integrity constraints," in *Database programming languages (DBPL-4): Proceedings of the 4th International Workshop on Database Programming Languages, Object Models and languages*, Springer-Verlag, 1994, 243–262.

[4] Veronique Benzaken & Doucet, "Themis: a Database Programming Language Handling Integrity Constraints," *VLDB Journal* 4 (1995).

[5] Veronique Benzaken & Xavier Schaefer, "Ensuring efficiently the integrity of a persistent object store via abstract interpretation," in *Proceedings of the 7th International Workshop on Persistent Object Systems*, Morgan Kaufmann, May, 1996.

[6] H. Balsters, R. A. de By & R. Zicari, "Typed sets as a basis for object-oriented database schemas," in *ECOOP 1993 Kaiserslautern*, 1993.

[7] René Bal, Herman Balsters, Rolf A. de By, Alexander Bosschaart, Jan Flokstra, Maurice van Keulen, Jacek Skowronek & Bart Termorshuizen, "The TM Manual; version 2.0, revision e," Universiteit Twente, Technical report IMPRESS / UT-TECH-T79-001-R2, Enschede, The Netherlands, June 1995.

[8] Luca Cardelli, "A semantics of multiple inheritance," *Inf. and Comput.* 76 (1988), 138–164.

[9] Herman Balsters & Chris C. de Vreeze, "A semantics of object-oriented sets," in *The Third International Workshop on Database Programming Languages: Bulk Types & Persistent Data (DBPL–3), Aug. 27–30, 1991, Naflion, Greece*, Paris Kanellakis & Joachim W. Schmidt, eds., Morgan Kaufmann Publishers, San Mateo, CA, 1991, 201–217.

[10] R. G. G. Cattell, *The Object Database Standard: ODMG–93*, Morgan Kaufmann Publishers, San Mateo, CA, 1994.

[11] Lawrence C. Paulson, *Isabelle: A Generic Theorem Prover*, Lecture Notes in Computer Science #828, Springer-Verlag, Berlin, 1994.

[12] Alexander Borgida, John Mylopoulos & Joachim W. Schmidt, *Database Programming by Formal Refinement of Conceptual Designs*, IEEE Data Engineering, Sept., 1989.

[13] Luca Cardelli, "Amber," *Combinators and Functional Programming*, New York–Heidelberg–Berlin (1986).

[14] G. Castagna, "Object-Oriented Programming: A Unified Foundation," *Progress in Theoretical Computer Science* (1996,).

[15] Peter Buneman & Atsushi Ohori, "A Type System that Reconciles Classes and Extents," in *The Third International Workshop on Database Programming Languages: Bulk Types & Persistent Data (DBPL–3), Aug. 27–30, 1991, Nafplion, Greece*, Paris Kanellakis & Joachim W. Schmidt, eds., Morgan Kaufmann Publishers, San Mateo, CA, 1991, 191–202.

[16] Man Hon Wong & Divyakant Agrawal, "Context-specific synchronization for atomic data types in object-oriented databases," *TCS* (1995).

[17] William E. Weihl, "The Impact of Recovery on Concurrency Control," *Journal of Computer and System Sciences* (1993).

[18] David Stemple, Subhasish Mazumdar & Tim Sheard, "On the modes and meaning of feedback to transaction designers," in *Proceedings of ACM-SIGMOD 1987 International Conference on Management of Data, San Francisco, CA, May 27–29, 1987*, Umeshwar Dayal & Irv Traiger, eds., ACM Press, New York, NY, 1987, 374–386, (also appeared as ACM SIGMOD Record 16, 3, Dec., 1987).