Verifiable Object-Oriented Transactions

Suad Alagić and Adnan Fazeli

Department of Computer Science,
University of Southern Maine, Portland, Maine, USA
alagic@usm.maine.edu, adfazeli@gmail.com

Abstract. Unlike the existing object-oriented and other database technologies, database schemas in the technology developed in this research are equipped with very general integrity constraints specified in a declarative, logic-based fashion. These declarative specifications are expressed in object-oriented assertion languages and they apply to transactions that are implemented in a full-fledged, mainstream object-oriented programming language. The model of transactions is based on more advanced features of object-oriented type systems, the ownership model, and very general constraints. The main distinction in comparison with other database technologies is that transactions can be verified to satisfy the schema integrity constraints. The two main contributions of this paper are object-oriented schemas equipped with integrity constraints and static verification of transactions with respect to the integrity constraints. Solutions to these open problems have been out of reach so far. Furthermore, transaction verification is not only largely static, but it is also automatic, so that the subtleties of the underlying verification technology are hidden from the users. In addition to static verification, the technology offers dynamic enforcement of the integrity constraints when necessary. The overall outcome is a significant increase in data integrity along with run-time efficiency and reliability of transactions.

1 Introduction

This paper is addressing a major limitation of the current generation of object-oriented database systems. In fact, other widely used database technologies exhibit the same problem. The solution to this problem developed in this paper is based on recent developments in assertion languages and verification technologies. This represents a major departure from the technologies and tools that are commonly used in database systems.

The key issue in object-oriented database systems is management of persistent objects. Object-oriented languages have no support for persistent objects that would be suitable for databases. An object-oriented database schema specifies (collections of) persistent objects, and their types in particular. Complex actions on persistent objects are expressed as programs in an object-oriented programming language. A transaction is expected to start in a consistent database state and if successfully completed it must leave the database in a consistent state.

G. Agha et al. (Eds.): Yonezawa Festschrift, LNCS 8665, pp. 251–275, 2014.

The current generation of object-oriented systems is based on typed object-oriented programming languages. This is a source of a major discrepancy: data languages are declarative and mainstream object-oriented languages by themselves do not have such capabilities. Database schemas, the consistency requirements, and queries should be specified in a declarative style.

The current object technology has nontrivial problems in specifying even classical database integrity constraints, such as keys and referential integrity [10,17,20]. No industrial database technology allows object-oriented schemas equipped with general integrity constraints. In addition to keys and referential integrity, such constraints include ranges of values or number of occurrences, ordering, constraints that apply to inheritance, and the integrity requirements for complex objects obtained by aggregation [2]. More general constraints that are not necessarily classical database constraints come from complex application environments and they are often critical for correct functioning of those applications [3].

Object-oriented schemas are generally missing database integrity constraints because those are not expressible in type systems of mainstream object-oriented programming languages. Since the integrity constraints cannot be specified in a declarative fashion, the only option is to enforce them procedurally with nontrivial implications on efficiency and reliability. In a typed constraint-based database technology, the constraints would fit into the type systems of object-oriented languages and they should be integrated with reflective capabilities of those languages [22] so that they can be introspected at run-time. Most importantly, all of that is not sufficient if there is no technology to enforce the constraints, preferably statically, so that expensive recovery procedures will not be required when a transaction violates the constraints at run-time [2,3].

The idea of static verification of transaction safety with respect to the database integrity constraints has been considered in previous research [8,23,25,6] but it has not been implemented at a very practical level so that it can be used by typical object-oriented database programmers. The first problem is that object database technologies such as ODMG [9], Db4 [10], and Objectivity [20] are not equipped with general constraints, and even have difficulties in specifying keys and referential integrity [17]. This problem is resolved in this research by using an object-oriented assertion language such as JML [13] or Spec# [19]. An assertion language allows specification of schemas with general database integrity constraints (invariants) and transactions can be specified in a declarative fashion with preconditions and postconditions.

The ability to statically verify that a transaction implemented in a mainstream object-oriented language satisfies the database integrity constraints has been out of reach for a long time. Some of our previous results were based on a higher-order interactive verification system which is so sophisticated that it is unlikely to be used by database programmers. A pragmatic goal has been static automatic verification which completely hides the prover technology from the users. Automatic static verification of the object-oriented constraints is a major distinction with respect to our previous work [3,4] as well as with respect to other

work [8,23,25,6]. Our goals are object-oriented schemas with general integrity constraints, transactions written in a mainstream object-oriented language, and their static verification that guarantees ACID properties in an implementation based on an object-oriented database management system. These goals represent a significant advancement of our previous results reported in [2].

A key observation is that if it is not possible to verify that transactions satisfy the schema integrity constraints, then it is not possible to truly guarantee the ACID properties of the transaction model. ACID stands for atomicity, isolation, consistency and durability. A serializable concurrent execution of a set of transactions has the property that it will maintain the schema integrity constraints only as long as the individual transactions by themselves (i.e., in isolation) satisfy those constraints [11]. This explains the relationship between the research reported in this paper and other research on object-oriented transactions. Most recent research on object-oriented transactions, such as [14,26,24], has been directed toward an apparatus for providing properties such as atomicity, isolation, and serializability that would replace the existing inadequate concurrent apparatus in object-oriented programming languages with respect to transactions. The integrity constraints (the C component) are not considered. Our research does exactly that.

The contributions of this paper are:

- Specification of object-oriented database schemas equipped with classical as well as more general integrity constraints not available in the existing database technologies.
- Schema modeling techniques based on abstraction, specification inheritance, and aggregation including the ownership model.
- A model of object-oriented transactions equipped with declarative specifications and techniques for automatic static and dynamic verification of transaction safety with respect to the schema integrity constraints.
- A complex object-oriented application that demonstrates the above techniques, verification of complex transactions in particular.
- A model of ACID transactions implemented on top of an object-oriented database management system that guarantees all ACID properties, the C component in particular.

This paper is organized as follows. We first specify (section 2) the basic features of our model of object-oriented transactions. General issues of concurrent transactions and the integrity constrains are discussed in section 3. In section 4 we present some key semantic concepts for modeling complex application environments and the associated transactions. Levels of consistency as they relate to the model of transactions and the ownership model are discussed in section 5. In section 6 we consider complex schemas equipped with a variety of general integrity constraints, including classical database constraints such as keys and referential integrity. This is followed by sample transactions with respect to schemas equipped with integrity constraints in section 7. In section 8 we elaborate the relationship between declarative specification of constraints and database queries. The impact of inheritance on the schema integrity constraints

is discussed in section 9. Abstraction techniques for object-oriented schema are given in section 10. Dynamic constraint checking is the subject of section 11. The implementation issues related to the underlying database platform are discussed in section 12. Related research is summarized in section 13 and conclusions are given in section 14.

2 Transaction Model

Our main contribution is an implemented model of automatic verification of object-oriented transactions with respect to the object-oriented schemas equipped with constraints. To our knowledge this is the first time such a verification was possible for transactions written in a full-fledged mainstream object-oriented language and object-oriented schemas and transactions extended with very general constraints.

The components of our transaction model are more sophisticated features of the type system such as bounded parametric polymorphism, the ownership model, specification of the schema integrity constraints, pre and post conditions for transactions, and their automatic verification.

In our transaction model there exists an interface *Schema* and a class *Transaction*. Specific schemas implement the interface *Schema*.

A schema is a complex object, an aggregation of its components. A schema is the owner of its components. The ownership model of our transaction model allows constraints that apply to complex objects and their components. A particular case are integrity constraints that apply to complex schema objects. Objects with the same owner are modeled as peers. This in particular applies to components of a schema object.

Our model of transactions allows controlled updates of persistent objects in such a way that the constraints associated with complex objects, and schema objects in particular, are enforced. Independent updates of components of a complex object that might violate the integrity constraints that apply to the whole complex object are not allowed. Situations in which a transaction necessarily and temporarily violates the schema integrity constraints are carefully controlled in this model.

The relationship between a transaction object and a schema object is also modeled as a peer relationship. A transaction is not a component of a schema, and a schema is not a component of a transaction. There are multiple transactions accessing the same schema object and all of them cannot own the schema object. An object can have at most one owner. In addition, there are constraint-related reasons for modeling a transaction and its schema as peers to be elaborated in section 4. The basic features of the transaction model are represented in figure 1.

In our approach, the class *Transaction* is bounded parametric, where the bound type is the type of schema to which a specific transaction type is bound. This makes it possible for a particular transaction class to be compiled with respect to a specific schema type. The notation in the code given below follows

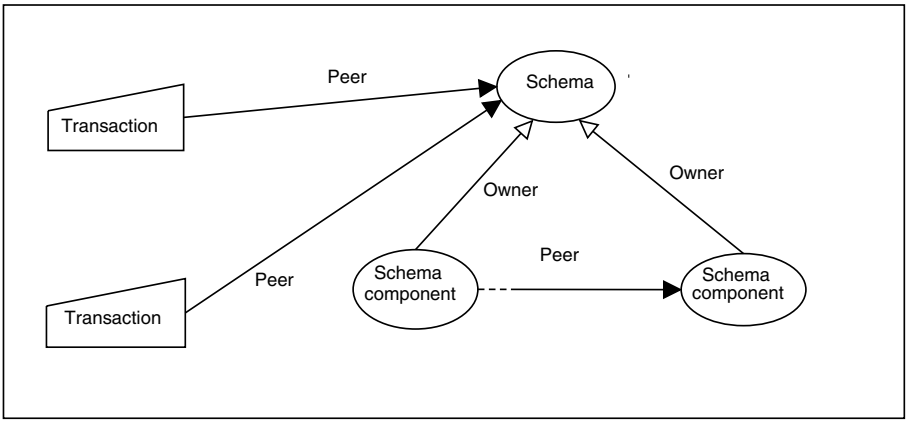

Fig. 1. Transactions and schemas

Spec#. T! denotes a non-null object type, i.e., an object type that does not allow null references. The attribute [Peer] indicates that the relationship between a transaction object and its associated schema object is specified as the peer relationship. The attribute [SpecPublic] denotes private components that can be used as public only in specifications.

public interface Schema {...}

public class Transaction <T> **where** T: Schema {
[**SpecPublic**][**Peer**] **protected** T! schema;
public Transaction(T! schema){ this.schema = schema;}
}

Both schemas and transactions are equipped with very general logic-based constraints to be elaborated throughout the paper starting with sections 6 and 7.

3 Concurrent Transactions and Integrity Constraints

Our view is that a database schema should be equipped with explicitly specified integrity constraints that transactions acting on the database must satisfy. The database is acted upon by a set of concurrent transactions. A well-known fact is that concurrent executions of transactions may violate the database integrity constraints even if individual transactions do not.

If individual transactions respect the integrity constrains, then obviously their *serial execution* will as well. That is, if the integrity constraints hold initially, they will hold after completion of the first transaction, and likewise they will hold after each subsequent transaction in a serial execution.

However, *serial executions* are unacceptable for performance and database availability reasons. Database technologies are naturally based on concurrent

transactions. Since concurrent transactions may violate the integrity constraints, they must be managed by a technology that allows only those concurrent executions that do not violate those constraints. From the viewpoint of database integrity, those concurrent executions are equivalent to serial executions. Such concurrent executions are called *serializable executions* [11].

A *concurrent execution* is *serializable* if it has an equivalent *serial execution*. Two executions are said to be equivalent from the viewpoint of integrity if they have the same ordering of conflicting actions. Two actions are conflicting if they are executed on the same object and at least one of them is an update. Various locking protocols have been invented and implemented to guarantee the serializability condition. The classical and the best known is *two phase locking*. Two phase locking is a pessimistic strategy and it is provably a sufficient condition for seralizability [11]. There are optimistic alternatives.

The beauty of the classical results on serializability is that they do not depend upon a particular form of the integrity constraints. But the underlying assumption is that whatever the integrity constraints are, each individual transaction in a concurrent execution is required to satisfy those constraints in isolation. Research on object-oriented transactions or the current generation of object-oriented database systems do not address this fundamental requirement. The reason is that there has been no technology to deal with more complex integrity constraints. This is precisely the main point of the research reported in this paper. We develop verification techniques that would guarantee that an object-oriented transaction satisfies the database integrity constraints.

The verification techniques for object-oriented transactions that we investigated belong to one of the following categories:

- *Dynamic enforcement of constraints*
  A representative of this type of technology is JML [13]. In this case schema constraints and transactions are specified in JML, and transactions are full-fledged Java programs. Database technologies enforce a few classical database constraints such as keys and foreign keys. JML allows much more general constraints. The main disadvantage is that violations are detected at run-time, where the implications may be non-trivial, such as invocation of expensive recovery procedures to maintain data integrity. In addition, dynamic enforcements of constraints in database systems comes with a siginificant cost. But very general constraints are specifiable and enforceable.
- *Static interactive reasoning*
  A representative of this type of technology is PVS [21]. The main advantage of this technology is that it is very general. PVS is a higher-order verification system that allows specifications of specialized logics suitable for transaction verification. Careful investigation of the transaction model shows that it is actually temporal in nature. This is why we developed transaction verification techniques in PVS that are based on a suitable temporal logic [3]. The main disadvantage of this technology is that it requires very sophisticated users and hence it is not likely to be directly used by ordinary database programmers. This technology has a complementary role in our transaction

verification environment. It is used to verify more general integrity constraints (such as temporal) that technologies based on object-oriented assertion languages cannot handle.

– *Automatic static verification*
This is the most appealing technology from the viewpoint of users. A representative of this technology is Spec# [15]. In our approach, schemas and transactions are specified in Spec# and transactions are full-fledged C# programs. In this technology the subtleties of the underlying prover technology are hidden from the users. Static automatic verification is attempted, with runtime checks generated as well. In addition, this particular technology comes with the ownership model that it is essential for a sophisticated model of object-oriented transactions. It also comes with features that allow controlled updates that might violate the integrity constraints. This technology is the focus of this paper because it is precisely what has been out of reach for many years: static verification of very general integrity constraints for transactions written in a mainstream, preferably object-oriented, programming language.

In the transaction verification technology presented in this paper static verification is complemented with dynamic checks. Dynamic checks are in fact necessary even if static verification succeeds. Static verification guarantees that the transaction code is correct with respect to the schema integrity constraints and the transaction specification in terms of its pre and post conditions. But if the schema integrity constraints or the transaction precondition do not hold at the transaction start, the results of static verification do not apply. In many situations checking the transaction precondition is possible only at run-time. For example, inserting an object into a database collection equipped with a key constraint requires checking that the key of the inserted element does not already exist in the database collection. This is why the transaction code must be written in such a way that it handles run-time exceptions caused by dynamic checks of constraints. In the absence of such exceptions, or if those exceptions are correctly managed, the transaction postcondition and the schema integrity constrains will hold at the transaction completion (commit) point. The key point about static verification is that if a transaction fails a static check, it should never be executed. This avoids major problems related to violation of the integrity constraints running a transaction that provably does not satisfy those constrains. The penalties of executing such a transaction are aborting a transaction and invoking expensive recovery procedures to restore database consistency.

4 Owners and Peers

In this section we consider the semantic modeling techniques for object-oriented database schemas explicitly supported in the technology presented in this paper. These techniques have not been considered in object database technologies such as ODMG [9], Db4 [10], and Objectivity [20], and hence have no proper support in those technologies.

In addition to inheritance, the key abstraction technique for modeling complex applications in this paper is aggregation. This abstraction is well understood in semantic data models, but in the object-oriented model it has specific implications. A complex object defined by aggregation is represented by its root object called the owner along with references to the immediate components of the owner specified as its representation fields. References to other objects do not represent components of that object. This way a complex object is defined as a logical unit that includes all of its components. Constraints that apply to objects defined by aggregation may now be specified in such a way that they refer both to the owner object and to the components defined by its representation fields. In a flight scheduling application developed in this paper, a flight scheduling object is defined as an aggregation of flights, planes and airports, as illustrated in figure 2.

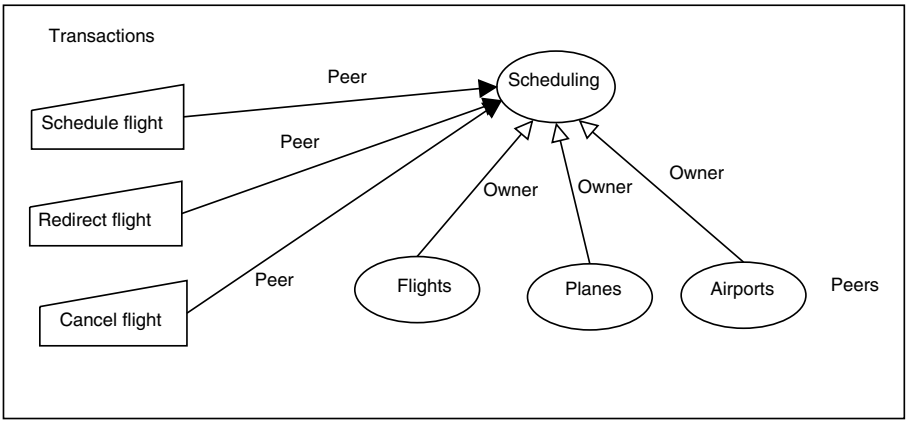

Fig. 2. Owners and peers in flight scheduling

The notion of ownership comes with a related semantic modeling notion. Objects that have the same owner are called peers. The relationship among objects flights, planes and airports is clearly not the ownership relationship. These objects are peers as they have the same owner, the scheduling object.

The peer relationship has a role that may be even independent from the notion of ownership. Consider the relationship of a transaction object and its associated schema object. As we already explained, a transaction and its associated schema are modeled as peers. Of course, we may view the overall application as the owner of the schema object and all the associated transaction objects.

In addition to the above abstractions, inheritance is naturally an essential modeling abstraction which we do not show in the above diagram. The model of this application includes inheritance hierarchies of different aircraft types and different airport types, as well as an inheritance hierarchy of different transaction types. The interplay of inheritance and constraints is discussed in section 9.

5 Levels of Consistency

The schema integrity constraints are typically violated during transaction execution and then the constraints are reinstated when the transaction is completed, so that the constraints should hold at commit time. The mechanism for handling correctly these situations is illustrated below by the structure of a transaction that closes an airport:

expose(flight scheduling){
close airport;
cancel all flights to or from the closed airport;
}

After the first action the referential integrity constraints are temporarily violated to be reinstated after the second action of cancelling all flights to or from the closed airport. The purpose of the **expose** block is to indicate that the schema object invariants may be violated in this block. Otherwise, the verifier will indicate violation of the schema invariants. In the **expose** block the object is assumed to be in a mutable state and hence violation of the object invariants is allowed. Outside of the **expose** block, assignments that possibly violate the invariants will be static errors. Different situations that may occur with respect to the object state and its satisfaction of the object invariants are summarized below:

- **Valid** object state – object invariants hold, updates must satisfy the invariants.
- **Mutable** object state - object invariants are not required to hold, updates are allowed to violate them
- **Consistent** object state – the object is in a **valid** state and
  - the object does not have an owner or
  - the owner is in a **mutable** state
- **Committed** object state – the object is in a **valid** state and
  - the object has an owner
  - the owner is also in a **valid** state.

When a transaction operates on an object, the implicit assumption is that the object is in a consistent state. This means that either the object does not have an owner to put restrictions on the object, or that the object has an owner, and the owner is in a mutable state, hence it allows update actions on the object. Since the object is in a consistent state, its state is valid and its components are thus in a committed state. In order to update the receiver and the states of its components, the receiver state must be changed to a mutable state using the *expose* block. This will also change the state of the components of the object from committed to consistent, so that methods can be invoked on them. The notions of valid, mutable, consistent and committed objects, and the effect of the expose statement, are illustrated in figure 3.

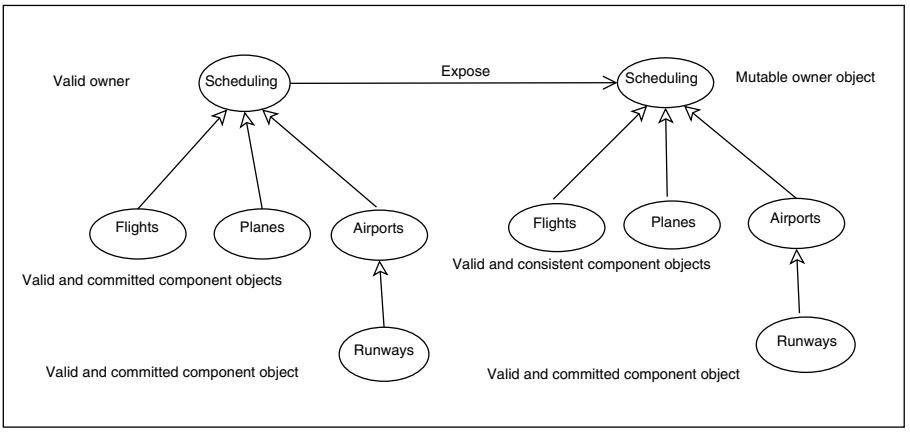

Fig. 3. Flight scheduling consistency states

There is an obvious alternative to viewing a transaction and its associated schema as peers: just omit any ownership or peer attributes. But in fact, using the peer relationship has important implications for transaction verification. A transaction is verified under the assumption that the schema integrity constraints hold when the transaction is started. If this condition is not satisfied, a transaction cannot be verified even if it is in fact correct with respect to the schema integrity constraints. So we really have to guarantee this condition.

Spec# adds an implicit precondition for peer consistency so that a transaction can assume this condition in its verification. This applies to in-bound parameters and the receiver of any method. The implicit postcondition for peer consistency also applies to all out-bound parameters and return values. The caller of a method is required to satisfy the peer consistency requirement. This means that an object and its peers must be valid, and their owner must be exposed first before an update is performed.

6 Constraints for Schemas

We now consider a specific schema in which the core object type is defined using the aggregation abstraction and the ownership model along with the associated integrity constraints. The *FlightScheduling* schema contains specification of three database collections: a list of airplanes, a list of airports, and a list of flights.

The schema *FlightScheduling* exhibits two cases of the aggregation abstraction as supported by the Spec# ownership model. The attribute [Rep] indicates that the lists of flights, airports and airplanes are components of the flight scheduling object which becomes their owner. The attribute [ElementsRep] indicates that list elements are also components of the flight scheduling object. These elements are then peers according to the Spec# ownership model. This has implications on invariants that can now be defined to apply to entire complex objects,

i.e., including their components determined by the [Rep] and [ElementsRep] fields. These constraints are called ownership-based invariants.

```
public class FlightScheduling: Schema {
[SpecPublic][Rep] [ElementsRep] private List<Airplane!>! airplanes;
[SpecPublic][Rep] [ElementsRep] private List<Airport!>! airports;
[SpecPublic][Rep] [ElementsRep] private List<Flight!>! flights;
// constraints
}
```

In the collection of airplanes the key is *Id*, in the collection of airports the key is *Code*, and in the collection of flights the key is *FlightId*. The first referential integrity constraint specifies that each flight in the collection of flights refers to a unique airplane in the collection of airplanes. The remaining (omitted) referential integrity constraints specify that each flight in the collection of flights refers to a unique airport as its origin and a unique airport as its destination.

For presentation purposes, the notation in this paper is more mathematical than the Spec# notation. However, there is a direct correspondence between this notation and the Spec# notation.

```
invariant ∀{int i ∈ (0: flights.Count), int j ∈ (0: flights.Count);
  flights[i].FlightId = flights[j].FlightId ⇒ flights[i].Equals(flights[j])};
invariant ∀{int i ∈ (0: flights.Count);
   ∃ unique {int j ∈ (0: airplanes.Count); airplanes[j].Equals(flights[i].Airplane)}};
```

A class is in general equipped with an invariant which specifies valid object states. The schema integrity constraints are specified above as class invariants. These assertions allow usage of universal and existential quantifiers as in the first-order predicate calculus, as well as combinators typical for database languages such as min, max, sum, count, avg etc. These constraints in the above schema refer to private components of the schema object. As explained earlier, the attribute [SpecPublic] means that these private components can be used as public only in specifications. Typically, such components will also be defined as public properties with appropriately defined *get* and *set* methods so that access to them can be controlled.

Spec# constraints limit universal and existential quantification to variables ranging over finite intervals. The above constraints contain specifications of half open intervals. The limitation that quantifiers are restricted to integer variables ranging over finite intervals was a design decision to sacrifice expressiveness in order to allow automatic static verification. This limitation is no problem in the application considered in this paper as the above schema shows.

The above schema contains non-null object types (indicated by the symbol *!*) that capture a very specific object-oriented integrity constraint. A frequent problem in object-oriented programs is an attempt to dereference a null reference. If

this happens in a database transaction, the transaction may fail at run-time with nontrivial consequences. The Spec# type system allows specification of non-null object types. Static checking will indicate situations in which an attempt is made to access an object via a possibly null reference.

7 Sample Transactions

Each class that a schema refers to is also equipped with its constraints as illustrated below for the class *Flight*. The relationship between a flight object and the associated airplane object is defined as a peer relationship for the reasons explained in section 4. The invariants include the obvious ones: the origin and the destination of a flight must be different and the departure time must precede the arrival time. If the current time is greater than the arrival time or the current time is less than the departure time, the status of the flight must be idle. If the current time is greater than the departure time and less than the arrival time the flight status must be either flying, landing or takeoff.

invariant to $\neq$ from;

invariant departureTime $<$ arrivalTime;

invariant DateTime.Now $>$ arrivalTime $\Rightarrow$ this.flightStatus = FlightStatus.Idle;

invariant DateTime.Now $<$ departureTime $\Rightarrow$ this.flightStatus = FlightStatus.Idle;

invariant DateTime.Now $\geq$ departureTime $\wedge$ DateTime.Now $\leq$ arrivalTime $\Rightarrow$
this.flightStatus = FlightStatus.TakeOff $\vee$
this.flightStatus = FlightStatus.Flying $\vee$
this.flightStatus = FlightStatus.Landing;

The constraints specified in this section include some classical database integrity constraints such as keys and referential integrity, and in addition constraints that are not typical for the existing database technologies, object-oriented in particular. In fact, we are not aware of a database technology that allows constraints of the above variety.

To make the job of the verifier possible, specification of methods that change the object state, such as database updates, requires specification of the frame conditions. This is done by the *modifies* clause, which specifies those objects and their components that are subject to change. The frame assumption is that these are the only objects that will be affected by the change, and the other objects remain the same. An attempt to assign to the latter objects will be a static error.

Sample instantiations of the class *Transaction* by the flight scheduling schema are given below.

```
public class ScheduleFlightTransaction:
                    Transaction<FlightScheduling> {
public Flight? scheduleFlight (string! flightId,
            string! toAirportCode, string! fromAirportCode,
            DateTime departure, DateTime arrival, Airplane! plane)
```

```
// constraints
{// transaction body }
}
```

Flight? in the above code is an explicit notation for a type that may contain a null value. The preconditions of the transaction *scheduleFlight* are that the flight id does not exist in the list of flights, that its origin (denoted *fromAirportCode*) and its destination (denoted *toAirportCode*) must refer to existing (valid) airport codes, and that the departure time precedes the arrival time. Valid airport codes are kept in a table *ValidCodes*. The transaction *scheduleFlight* modifies only the list of flights as specified in its **modifies** clause. The postcondition guarantees that the newly scheduled flight exists in the list of flights.

```
requires toAirportCode ≠ fromAirportCode;
requires ∀ {int i ∈ (0: schema.Flights.Count);
              schema.Flights[i].FlightId ≠ flightId };
requires ∃ unique {int i ∈ (0: schema.Airplanes.Count);
              schema.Airplanes[i].Equals(plane)};
requires ∃ unique {string code ∈ ValidCodes.airportsCodes;
                   code = toAirportCode};
requires ∃ unique {string code ∈ ValidCodes.airportsCodes;
                   code = fromAirportCode };
requires departure < arrival;
modifies schema.flights;
ensures ∃ unique {int i ∈ (0: schema.Flights.Count);
                   schema.Flights[i].FlightId = flightId };
```

The first precondition of the transaction *cancelFlight* specifies that there exists a unique flight in the collection of flights with the given id of the flight to be deleted, denoted *flightId*. The second precondition specifies a requirement that the flight departure time is greater than the current time. The **modifies** clause specifies that this method modifies the collection of flights. The postcondition specifies that the cancelled flight does not appear in the list of flights.

```
requires ∃ unique {Flight flight ∈ schema.Flights;
                   flight.FlightId = flightId};
requires ∀ {Flight flight ∈ schema.Flights;
              flight.FlightId = flightId ⇒
              flight.departureTime > DateTime.Now };
modifies schema.flights;
ensures ∀ {Flight! flight ∈ schema.Flights;
              flight.FlightId ≠ flightId };
```

The precondition for the transaction *redirectFlight* are that the id of the flight to be redirected, denoted *flightId*, must exist in the list of flights, and that its status must not be landing. This transaction modifies just the list of flights. The

postcondition ensures that the destination of the redirected flight has indeed been changed in the list of flights to *newDest*.

```
requires ∃ unique {Flight flight ∈ schema.flights;
    flight.FlightId = flightId ∧ (flight.FlightStatus ≠ FlightStatus.Landing)};
requires ∀ {Flight flight in schema.flights;
        flight.FlightId = flightId ⇒ flight.from ≠ newDest };

modifies schema.flights;

ensures ∀ {Flight! flight in schema.Flights;
        flight.FlightId = flightId ⇒ flight.to = newDest };
```

8 Constraints and Queries

Queries are pure methods. Pure methods are functions that have no impact on the state of objects, database objects in particular. Interplay of constraints and queries is illustrated below. The attribute [Pure] indicates a pure method and *result* refers to its result.

An example of a query (hence pure) method is *flightsDepartureBetween* which returns a list of flights whose departure time is within a given interval. The preconditions require that the time interval is not empty (i.e. the initial time is less than the end time) and that the initial time is greater than the current time. The postcondition ensures that the flights that are returned by this method have the departure times within the specified bounds.

```
[Pure] public List<Flight!>? flightsDepartureBetween
                (DateTime beginDateTime, DateTime endDateTime)
requires beginDateTime < endDateTime;
requires beginDateTime > DateTime.Now;
ensures ∀ {Flight! f ∈ result;
              f.departureTime ≥ beginDateTime ∧
              f.departureTime < endDateTime };
{// method body }
```

The body of this method is specified as a LINQ query given below:

```
// open db
IEnumerable<Flight> flights =
from Flight flight ∈ db
where flight.departureTime ≥ beginDateTime ∧
        flight.departureTime < endDateTime
select flight;
// close db;
```

A native query in Db4 Objects (details omitted) has the following form:

```
// open db
IList<Flight!>? flights =
db.Query<Flight!>(delegate(Flight! f) {
return (f.departureTime ≥ beginDateTime ∧
        f.departureTime < endDateTime); };
// close db;
```

9 Specification Inheritance

Specifications of constraints in a class are inherited in its subclasses. In addition, method postconditions and class invariants may be strengthened by additional constraints. Method preconditions remain invariant. This discipline with respect to inheritance of constraints is a particular case of behavioral subtyping [16]. It guarantees that an instance of a subtype may be substituted where an instance of the supertype is expected with no behavioral discrepancies.

Consider the class *Airport* given below in which an airport object is the owner of its list of runways, as well as of the specific runways in that list.

```
public class Airport {

[SpecPublic] private string code;
[Additive] protected int numRunways;
[SpecPublic] [Rep] [ElementsRep] protected List<Runway!>! runways;
// methods and constraints
}
```

The invariants of this class specify that that the number of runways must be within the specified bounds. In addition, there are ownership based invariants on flights in the take-off and landing queues in the runways. These are invariants that relate properties of the owner and its components and hence apply to the entire complex object of an airport. These constraints include a constraint that one and the same flight cannot be in two different queues belonging to different runways. In order to make it possible for subclass invariants to refer to the field *numRunways*, Spec# requires the attribute [Additive] in the specification of this field.

```
invariant numRunways ≥ 1 ∧ numRunways ≤ 30;
invariant runways.Count = numRunways;
invariant /* No multiple occurrences of the same flight in runways*/
```

Methods *addRunway* and *closeRunway* along with the associated constraints are specified as follows:

```
public virtual void addRunway(Runway! runway)
modifies runways, numRunways;
ensures ∃ {Runway! r ∈ runways; r.Equals(runway)};
{//code }

public virtual void closeRunway (Runway! runway)
modifies runways, numRunways;
ensures numRunways > 0;
ensures numRunways = old(numRunways) - 1;
ensures ∀ { Runway! r ∈ runways; !r.Equals(runway)};
{// code }
```

Consider now a class *InternationalAirport* derived by inheritance from the class *Airport*. The class *InternationalAirport* inherits all the invariants from the class *Airport*. In addition, it adds new invariants that are conjoined with the inherited ones. These additional invariants require that the number of runways is higher than the minimum required by an airport in general. Furthermore, an additional requirement is that there exists at least one runway of the width and length suitable for international flights. This is expressed using a model field *IntRunway*. The notion of a model field is explained in section 10 that follows.

```
public class InternationalAirport: Airport {
invariant numRunways ≥ 10;
invariant ∃ {Runway! r ∈ Runways; r.IntRunway };
// IntRunway is a boolen model field in Runway
// constructor, methods
public override void closeRunway (Runway! runway)
ensures numRunways ≥ 10;
ensures ∃ {Runway! r ∈ runways; r.IntRunway };
{// code}
}
```

Overriding of the method *closeRunway* demonstrates the rules of behavioral subtyping. One would want to strengthen the precondition of this method by requiring that there is more than one international runway at an international airport or else the invariant for the international airport will be violated. But that is not possible by the rules of behavioral subtyping. Otherwise, users of the class *Airport* would see behavior of the method *closeAirport* that does not fit its specifications in the class *Airport*. This would happen if the airport object is in fact of the run-time type *InternationalAirport*. The *modifies* clause cannot be changed either for similar reasons. But the postcondition can be strengthened as in the above specifications. The postcondition now ensures that the number of runways is greater than or equal to ten and that there exists at least one international runway after the method execution. These are specific requirements for international airports.

Specification inheritance has implications on behavioral subyping of parametric types that follow well-known typing rules for such types. For example, if we derive a schema *InternationlFlightScheduling* by inheritance from the schema *FlightScheduling*, *Transaction<InternationlFlightScheduling>* will not be a subtype of the type *Transaction<FlightSchedling>*, and hence not a behavioral subtype either.

A class frame is the segment of the object state which is defined in that class alone. A class frame does not include the inherited components of the object state. An invariant of a class will include constraints that apply to its frame, but it may also further constrain the inherited components of the object state. For example, an object of type *International Airport* has three class frames. These class frames correspond to classes *Object*, *Airport* and *InternationalAirport*.

The notions *valid* and *mutable* apply to each individual class frame. The notions *consistent* and *committed* apply to the object as a whole. So an object is consistent or committed when all its frames are valid. The *expose* statement changes one class frame from valid to mutable. The class frame to be changed is specified by the static type of the segment of the object state to be changed. For example, the body of the method *closeAirport* of the class *InternationalAirport* has the following form:

```
assert runways ≠ null;
[Additive] expose((Airport)this){
    runways.Remove(runway);
    numRunways–;
}
```

10 Abstraction

Typical classes in this application have private fields that are made public only for specification purposes. Examples are fields *code* and *runways* in the class *Airport*. Users of this class would clearly have the need to read the code of an airport, and some users would have the need to inspect the runways of an airport. On the other hand, these fields are naturally made private as users are not allowed to access them directly in order to change them.

The basic mechanism for exposing a view of the hidden object state is to use public pure methods. A related technique is to use public properties. A public property is defined as a pair of *get* and *set* methods. The constraints in the *set* method control correctness of an update to a backing private field. A property *Runways* of the class *Airport* is specified below.

```
public List<Runway!>! Runways {
get { return runways;}
[Additive] set
requires value ≠ null;
ensures runways = value;
```

ensures /*no multiple occurrences of the same flight in runways*/
{/* code */}
}

Property getters are pure methods by default. Properties represent a general abstraction mechanism as the value of a property returned by the method *get* need not just be the value of a backing field, but it may be computed in a more complex way from the hidden (private) components of the object state.

Another abstraction mechanism is based on model fields. A model field is not an actual field and hence it cannot be updated. The model fields of an object get updated automatically at specific points in a transaction. An example of a model field is *IntRunway* of the class *Runway* specified below.

model bool IntRunway {
satisfies IntRunway = (length $\geq$ 80 $\wedge$ width $\geq$ 10);}

Unlike pure methods, model fields do not have parameters. But they often simplify reasoning. The verifier checks that the *satisfies* clause can indeed be satisfied, i.e., that there exists an object state that satisfies this clause. The *satisfies* clause may depend only on the fields of *this* and the objects owned by *this*. The *satisfies* clauses may be weaker in superclasses, and strengthened in subclasses.

11 Dynamic Checking of Constraints

Static verification of a transaction ensures that if the transaction is started in a consistent database state (the schema invariants hold) and the transaction precondition is satisfied, the schema invariants and the postcondition of the transaction will hold at the point of the commit action. The application program that invokes the transaction must satisfy the above requirements at the start of the transaction, and will be guaranteed that the postcondition and the schema invariants will hold at the end of the transaction execution.

Static verification does not say anything about what happens if the schema integrity constraints or the transaction precondition are not satisfied. What it says is that the code of a successfully verified transaction is correct with respect to the integrity constraints. Violation of constraints may still happen at run-time given the actual data. For example, a transaction may be invoked with arguments that do not satisfy the precondition and hence the verification results do not apply. This is why the dynamic checks that Spec# generates are essential. JML does the same, but it does not offer automatic static verification of code. Run-time tests will generate exceptions indicating violation of constraints. The transaction must handle these exceptions properly. Static verification guarantees that in the absence of such exceptions the results of transaction execution will be correct with respect to the integrity constraints. This extends to concurrent serializable executions of a set of transactions that have been statically verified.

Explicit dynamic checks may be used to verify that the constraints hold at run-time. This is illustrated below with a dynamic check of the precondition of the transactions *addAirport* in which *a* denotes the airport that should be added. The precondition of *addAirport* is that an airport with the code of the new airport does not exist in the database. This can be checked only dynamically by querying the database and asserting that this condition is satisfied.

```
IList<Airport!>? airports =
db.Query<Airport!>(delegate(Airport! arp){
return (arp.Code = a.Code);};
assert airports = null;
```

Ensuring that a new airport has been added to the database is accomplished by querying the database and asserting that the list of airports in the database with the new code is not empty and that the newly inserted airport is indeed in the database.

```
IObjectSet? airportsSet =
db.Query(typeof(Airport!));
assert ∃ unique {Airport! arp ∈ airportsSet; arp.Equals(a)};
```

12 Database Platform

The underlying database platform that we used in the implementation of the presented transaction model is Db4 Objects. Db4 is an open source object-oriented database management system. It manages persistent objects, offers multiple query languages, and two programming language interfaces for specifying transactions: Java and C#.

Research presented in this paper addresses precisely the limitations of the current generation of persistent object-oriented systems, and Db4 in particular. Db4 does not have an explicit notion of a schema and it does not have a transaction class. There are practically no constraints, especially of the kind presented in this paper. Consequently, the constraints are not enforced, and hence the fact that Db4 claims support for ACID transactions is not justified because of the C component.

Our research produces a much more sophisticated database technology that offers explicit types of schemas and transactions, very general constraints for both, and transaction verification. These specifications could be expressed in JML or Spec#, and the transaction code could be written in Java or C#. Enforcing constraints is done statically if Spec# is used, and it is dynamic in both JML and Spec#.

The role of PVS in our transaction verification environment is complementary. It is used to reason about more general schema and transaction constraints, such as temporal, which the two assertion languages cannot support. In addition, the

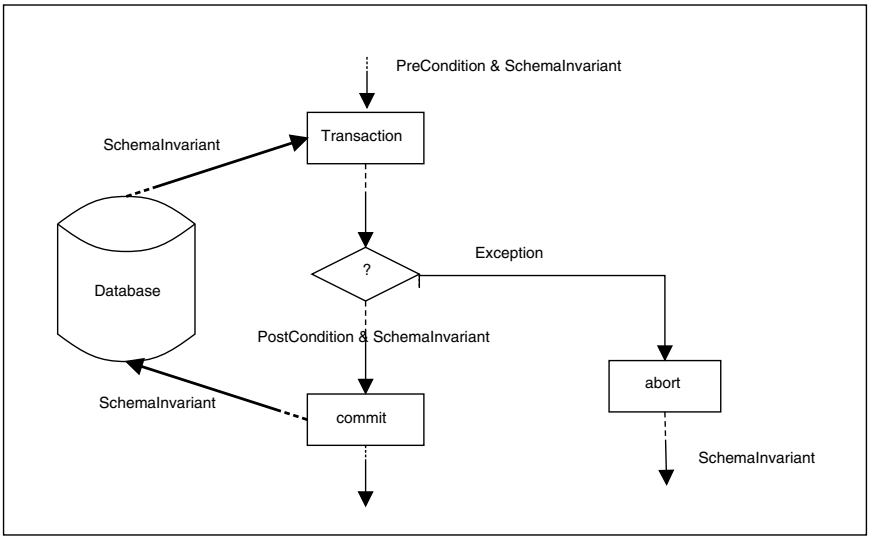

Fig. 4. Transaction execution

role of PVS is to show that some constraints are in fact provable, especially if static verification comes with difficulties.

Our technology truly supports ACID transactions. We rely on Db4 for implementation of atomicity, isolation and durability, and we guarantee consistency. In the actual implementation this is accomplished as follows:

- The precondition for the transaction invocation is that the transaction precondition and the schema invariant hold. This precondition will be the subject of static verification at the point of transaction call and it will be enforced dynamically if static verification is not possible.
- The above constraints are thus the precondition for the actual transaction action. The postcondition of the transaction action that is enforced is that the transaction postconditon and the schema invariant hold. This is the task of static transaction verification. The postcondition will be enforced dynamically as well.
- The above postcondition is the precondition for the transaction commit. The postconditon for the transaction commit is that the schema invariant holds.
- If an exception is raised and it is not handled by the transaction, the transaction will be aborted. The postconditon of the transaction abort is that the schema invariant is reinstated. Figure 4 illustrates the above points.

Isolation of concurrent transactions is implemented as follows. When a transaction opens the database, it creates a private workspace. Accessing persistent objects for the first time brings them into the private workspace of the transaction. Updates of objects affect initially only the objects in the private workspace.

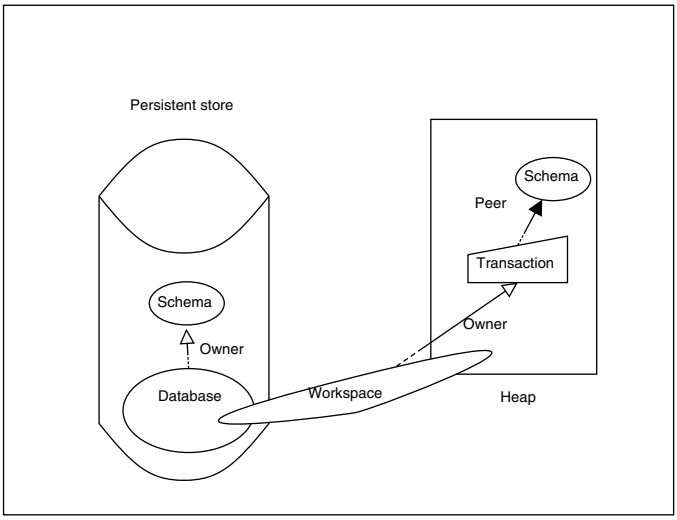

Fig. 5. Architecture

Since workspaces of individual transactions are private, updates that affect one workspace are inaccessible to updates of other transactions. Figure 5 illustrates some of the overall system architecture.

When a workspace is initially populated, the schema integrity constraints should hold for objects that are in the private workspace because they hold for the objects in the database. If that is not the case, the results on serializability and our verification techniques do not apply. During transaction execution the integrity constraints will typically be violated at some points. Static verification guarantees that the schema integrity constraints and the transaction postcondition will hold at the commit point. This is the property of the transaction code and hence the precondition of the transaction commit.

Atomicity is accomplished by installing all the changes recorded in the transaction workspace in the database so that either all of them are performed or none of them are. In the actual Db4 implementation a two phase write-ahead protocol is used to guarantee that all changes are made safely or none of them are.

There are two options supported by Db4 as far as the locking protocols are concerned. In a pessimistic strategy the private workspace consists of database objects and in-memory objects. In-memory objects are kept in separate workspaces and database objects are locked using a protocol that guarantees serializability (such as two-phase locking [11]). The locks will be held throughout transaction execution and released after a successful commit. In an optimistic protocol the private workspace will consist of in-memory objects only. Database objects will be accessed by multiple transactions and at commit time a concurrent execution will be checked for serializability.

The serializability check looks at the ordering of conflicting actions in the transactions log. The ordering of conflicting actions must correspond to some serial execution. If the serializability check fails at commit time, a suitable action must be performed, such as a rollback.

Once the commit is completed successfully, the updated objects in the database become available to other transactions. Successful verification guarantees that the objects in the database satisfy the integrity constraints. The D component of the notion of an ACID transaction will hold because committed changes will persist in the database.

The postcondition of the transaction abort is guaranteed by restoring the database objects and in-memory objects to the state before transaction execution so that the restored objects will satisfy the schema integrity constraints. Db4 has methods used in our implementation that rollback changes to persistent objects and refresh in-memory objects.

Our protocols differ from other similar protocols, and Db4 protocols in particular, in a fundamental way. They guarantee that the database integrity constraints will indeed be satisfied. To our knowledge, this is the only object-oriented technology that truly guarantees the C component of the notion of ACID transactions.

13 Related Research

General integrity constraints are missing from most persistent and database object models with rare exceptions such as [2,4,8]. This specifically applies to the ODMG model [9,5], PJama [18], Java Data Objects [12], and just as well to the current generation of systems such as Db4 Objects [10], Objectivity [20] or LINQ [17]. Of course, a major reason is that mainstream object-oriented languages are not equipped with constraints. Those capabilities are only under development for Java and C# [13,19]. In addition, none of the above technologies has support for the modeling techniques based on the ownership model.

A classical result [23] on the application of theorem prover technology based on computational logic to the verification of transaction safety is relational. Early object-oriented results include [8] and the usage of Isabelle/HOL [25]. A recent result [6] is relational and functional. In comparison with the above results and our own previous results, research reported in this paper produces object-oriented schemas with more general integrity constraints, transactions written in a mainstream object-oriented language, and their static verification that guarantees ACID properties in an implementation based on an object-oriented database management system.

Our previous results include techniques based on JML and PVS [3]. Our most recent results that apply to XML Schema constraints and the associated transactions are based on Spec# [2]. Reflective constraint management, static and dynamic techniques for enforcing constraints, and transaction verification technology are presented in [3,4,22]. The above techniques were applied to ambients of concurrent and persistent objects in [1].

A substantial amount of recent research has been directed toward a correct model and the required apparatus in object-oriented programming languages that would support the notion of a transaction [14,26,24]. These results are meant to resolve the mismatch between the existing concurrent object-oriented features of languages such as Java and C# and those required by the notion of ACID transactions. Unlike that research, research reported in this paper concentrates on the C component of ACID transactions. In addition, we also implement the D component of the ACID model based on the support of an object-oriented database management system [10].

14 Conclusions

The main contributions of this paper are solutions for two related problems that have been open so far:

- Lack of general schema integrity constraints in the existing object-oriented persistent or database technologies.
- Lack of a transaction specification and verification technology that would verify, preferably statically, that a transaction satisfies the schema integrity constraints.

The constraint-based technology allows specification of object-oriented schemas equipped with general database integrity constraints, transactions and their consistency requirements. The verification techniques presented in the paper allow largely static and automatic verification of transactions with respect to the specified constraints. A major advantage is that all the subtleties of the underlying verification and prover technology are completely hidden from the users. The implications on data integrity, efficiency and reliability of transactions are obvious and non-trivial.

Data integrity as specified by the constraints could be guaranteed. Runtime reliability of transactions is significantly improved. Expensive recovery procedures will not be required because the objects that violate the integrity constraints will never be committed to the database. The generated dynamic checks provide a significantly better control over exceptions raised by violation of the integrity constraints. In addition, more general application constraints that are not necessarily database constraints could be guaranteed. All of this produces a much more sophisticated technology in comparison with the existing ones.

The impedance mismatch between data and programming languages is to a great extent caused by different levels of abstraction of these two classes of languages. Data (query in particular) languages are largely declarative, and programming languages are largely procedural. A distinctive feature of the technology presented in this paper is declarative database programming in which the main emphasis is on writing a variety of constraints. The procedural code is in general simple, and thanks to recent extensions of object-oriented languages also largely declarative.

We demonstrated that an object-oriented model of transactions requires more advanced features of object-oriented type systems such as bounded parametric polymorphism and non-null object types. In addition, we showed that the ownership model is also essential for the transaction model. It allows specification of schemas using the aggregation abstraction and specification and enforcement of the integrity constraints that apply to complex objects. To our knowledge, no object-oriented persistent or database technology has the above features.

The model for schemas and transactions presented in this paper has been designed in such a way that it has a direct representation in Spec#. This makes Spec# verification technology directly applicable. Spec# limitations in expressiveness (like those for universal and existential quantification) presented no paricular problem in the application that we developed. But strictly speaking, the chosen application features a variety of temporal constraints that cannot be specified in Spec# in a temporal logic style. We use a different technology for specifying temporal constraints for schemas and transactions that complements the environment presented in this paper. That technology is based on a higher-order interactive verification system. While it is capable of expressing much more general constraints expressed in specialized logics, this technology requires very sophisticated users.

Automatic static verification (as in Spec#) is clearly a preferable verification technology from the viewpoint of the users. At this point that technology is still a prototype. The underlying architecture that separates the users view from the prover technology is very complex. Static verification sometimes comes with difficulties. However, while dynamic enforcement technology (as in JML) allows very general constraints, it comes with run-time penalties that are particularly pronounced in database applications.

References

1. Alagić, S., Anumula, A., Yonezawa, A.: Verifiable constraints for ambients of persistent objects. In: Advances in Software, vol. 4, pp. 461–470 (2011)
2. Alagić, S., Bernstein, P.A., Jairath, R.: Object-oriented constraints for XML Schema. In: Dearle, A., Zicari, R.V. (eds.) ICOODB 2010. LNCS, vol. 6348, pp. 100–117. Springer, Heidelberg (2010)
3. Alagić, S., Royer, M., Briggs, D.: Verification technology for object-oriented/XML transactions. In: Norrie, M.C., Grossniklaus, M. (eds.) Object Databases. LNCS, vol. 5936, pp. 23–40. Springer, Heidelberg (2010)
4. Alagić, S., Logan, J.: Consistency of Java transactions. In: Lausen, G., Suciu, D. (eds.) DBPL 2003. LNCS, vol. 2921, pp. 71–89. Springer, Heidelberg (2004)
5. Alagić, S.: The ODMG object model: does it make sense? In: Proceedings of OOPSLA, pp. 253–270. ACM (1997)
6. Baltopoulos, I.G., Borgström, J., Gordon, A.D.: Maintaining database integrity with refinement types. In: Mezini, M. (ed.) ECOOP 2011. LNCS, vol. 6813, pp. 484–509. Springer, Heidelberg (2011)
7. Benzaken, V., Doucet, D.: Themis: A database language handling integrity constraints. VLDB Journal 4, 493–517 (1994)

8. Benzanken, V., Schaefer, X.: Static integrity constraint management in object-oriented database programming languages via predicate transformers. In: Akşit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 60–84. Springer, Heidelberg (1997)
9. Cattell, R.G.G., Barry, D., Berler, M., Eastman, J., Jordan, D., Russell, C., Schadow, O., Stanienda, T., Velez, F.: The Object Data Standard: ODMG 3.0. Morgan Kaufmann (2000)
10. Db4 objects (2010), http://www.db4o.com
11. Eswaran, K.P., Grey, J.N., Lorie, R.A., Traiger, I.L.: The notions of consistency and predicate locks in a database system. Comm. of the ACM 19, 624–633 (1976)
12. Java Data Objects, Apache, http://db.apache.org/jdo/
13. Java Modeling Language, http://www.eecs.ucf.edu/leavens/JML/
14. Jagannathan, S., Vitek, J., Welc, A., Hosking, A.: A transactional object calculus. Science of Computer Programming 57, 164–186 (2005)
15. Leino, K.R., Muller, P.: Using Spec# language, methodology, and tools to write bug-free programs. Microsoft Research (2010), http://research.microsoft.com/en-us/projects/specsharp/
16. Liskov, B., Wing, J.M.: A behavioral notion of subtyping. ACM TOPLAS 16, 1811–1841 (1994)
17. Language Integrated Query, Microsoft Corporation, http://msdn.microsoft.com/en-us/vbasic/aa904594.aspx
18. Atkinson, M.P., Daynès, L., Jordan, M.J., Printezis, T., Spence, S.: An orthogonally persistent Java. ACM SIGMOD Record 15(4) (1966)
19. Microsoft Corp., Spec#, http://research.microsoft.com/specsharp/
20. Objectivity, http://www.objectivity.com/
21. Owre, S., Shankar, N., Rushby, J.M., Stringer-Clavert, D.W.J.: PVS Language Reference, SRI International. Computer Science Laboratory, Menlo Park, California, http://pvs.csl.sri.com/doc/pvs-language-reference.pdf
22. Royer, M., Alagić, S., Dillon, D.: Reflective constraint management for languages on virtual platforms. Journal of Object Technology 6, 59–79 (2007)
23. Sheard, T., Stemple, D.: Automatic verification of database transaction safety. ACM Transactions on Database Systems 14, 322–368 (1989)
24. Smaragdakis, Y., Kay, A., Behrends, R., Young, M.: Transactions with isolation and cooperation. In: Proceedings of OOPSLA 2007. ACM (2007)
25. Spelt, D., Even, S.: A theorem prover-based analysis tool for object-oriented databases. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 375–389. Springer, Heidelberg (1999)
26. Welc, A., Hosking, A.L., Jia, L.: Transparently reconciling transactions with locking for Java synchronization. In: Thomas, D. (ed.) ECOOP 2006. LNCS, vol. 4067, pp. 148–173. Springer, Heidelberg (2006)