

Ambients of Persistent Concurrent Objects

Suad Alagić
 Department of Computer Science
 University of Southern Maine
 Portland, Maine, USA
 alagic@usm.maine.edu

Akinori Yonezawa
 Department of Computer Science
 University of Tokyo
 Tokyo, Japan
 yonezawa@is.s.u-tokyo.ac.jp

Abstract—This paper develops a typed object-oriented paradigm equipped with message-based orthogonal persistence. Messages in this paradigm are viewed as typed objects. This view leads to a hierarchy of types of messages that belong to the core of typed reflective capabilities. Unlike most persistent object-oriented models, this model is equipped with general integrity constraints that also appear as a hierarchy of types in the reflective core. A transaction is naturally viewed as a sequence of messages and it is equipped with a precondition and a postcondition. The presented framework is motivated by ambients of persistent concurrent and mobile objects. The most important practical results supporting the developed model are verification techniques and a virtual platform for constraint management.

Keywords-Object databases; constraints; reflection; transactions.

I. INTRODUCTION

The current object technology has nontrivial problems in specifying classical database integrity constraints, such as keys and referential integrity [10][13][14]. 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, and the integrity requirements for complex objects obtained by aggregation [1]. 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 [2].

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. The constraints must fit into type systems of object-oriented languages and they should be integrated with reflective capabilities of those languages [15]. Most importantly, all of the above is not sufficient if there is no technology to enforce the constraints, preferably statically, so that expensive recovery procedure will not be required when a transaction violates the constraints at run-time [1][2].

The object-oriented database model presented in this paper integrates message-based orthogonal persistence, object-oriented schemas equipped with general integrity constraints accessible by reflection, and transactions that are required to satisfy the schema integrity constraints. The model is based on a type system and it offers a significantly different view of messages in comparison with the mainstream object-oriented languages. The model applies to ambients of persistent and concurrent objects.

A message in mainstream object-oriented languages such as Java or C# is specified in a functional notation. This functional view fits messages that cause no side-effects and report the properties of the hidden object state. The functional view also fits queries. Other categories of messages do not fit the functional notation. An update message is a message that changes the state of the receiver and possibly other objects as well. An update message does not have a result and its semantics does not fit the functional notation.

An asynchronous message [18] in general does not have a result either and hence the functional notation is not appropriate. A particular type of an asynchronous message (a two-way message) has a result, but this result is not necessarily immediately available at the point of the message send. Asynchronous (remote) queries would fit this pattern. A transient message has a limited lifetime and a sustained message does not have this limitation. A message may be one-to-one with a single receiver or a message may be a broadcast message sent to a set of receiver objects. Many messages naturally combine the features of the above mentioned message types. For example, a two-way transient message, a one-to-one query message, a one-to-many sustained update message etc. [9].

Further development of this approach leads to an orthogonal model of persistence [5] that is based on a special message type that promotes the receiver object to persistence. A transaction is defined as a sequence of messages of different types. Concurrency control and recovery protocols can now be implemented in the object-oriented style. Indeed, serialization protocols require knowledge of types of messages (queries versus updates) and impose an appropriate ordering of conflicting messages. Similar comments apply to recovery protocols that are in our view sequences of do, undo and redo

messages.

Object-oriented constraints are a key feature of the presented model. Specifying the behavior of objects of a message type is naturally done using an object-oriented assertion language. Object-oriented assertion languages allow specification of database integrity constraints as class invariants, declarative specification of transactions with pre and post conditions, and queries whose filtering (qualification expression) is specified as an assertion predicate. The assertion languages used to express constraint-related features of the model presented in this paper are JML (Java Modeling Language) [11] and Spec# [12].

Two critical pieces of the technology that supports this model are extended virtual platform for constraint management and verification techniques that apply to constraints. The extended virtual machine integrates constraints into the run-time type system, allows their introspection and enforcement [15]. Verification techniques apply to object-oriented transactions written in Java or C#. The verification technologies are based on PVS [2] and automatic static techniques of Spec# [1].

We first present in Section II a motivating application based on ambients of concurrent and service objects. The fundamentals of the view of messages as typed objects is developed further in Section III, along with the hierarchy of message types. The model of persistence is described in Section IV. Queries and transactions are discussed in Section V. Type safe reflection which includes run-time representation of types (including message types) and assertions is the subject of Section VI.

II. MOTIVATING APPLICATION: AMBIENTS OF CONCURRENT OBJECTS

In this introductory section, we describe the environments that lead to the view of messages as typed objects. An ambient [9] is a dynamic collection of service objects. The types of service objects are assumed to be derived from the type `ServiceObject`. This is why the class `Ambient` is parametric and its type parameter has `ServiceObject` as its bound type as follows:

```
abstract class Ambient
  <T extends ServiceObject> { . . . }
```

When a message is sent to an ambient object, one or more service objects is selected depending upon the type of the message, and the message is sent to those service objects. Messages sent to an ambient are in general asynchronous, hence they are of the type `Message`. When such a message object is created, it has its identity, a lifetime, and behaves according to one of the specific subtypes of the type `Message`. For example, a transient message has a limited discovery time and a sustained message does not. Moreover, messages can be sent to message objects. For example, if a message is a two-way message, a message that refers to the

future method may be sent to the two-way message object to obtain the result when it becomes available [18].

An ambient has a filter which selects the relevant service objects that belong to the ambient. This predicate is defined for a specific `Ambient` class, i.e, a class that is obtained from the class `Ambient` by instantiating it with a specific type of service objects. An ambient has a communication range which determines a collection of service objects that are in the ambient's range. The reach of an ambient object is then the collection of all service objects of the given type that satisfy the filter predicate and are within the communication range of the ambient object.

The class `Ambient` is equipped with a scheduler which selects the next message for execution according to some strategy. So the `Ambient` class looks like this:

```
abstract class Ambient
  <T extends ServiceObject> {
  abstract boolean filter(T x);
  Set<Message> messages();
  Set<T> communicationRange();
  Set<T> reach();
  invariant (forall T x)
    (x in this.reach() <=>
      this.filter(x) and x in
      this.communicationRange());
}
```

An example of a specific ambient class is

```
class StockBroker extends ServiceObject {
  int quote(String stock);
  int responseTime();
  . . .
}
class StockBrokerAmbient
  extends Ambient<StockBroker> {
  String[] displayStocks(){ . . . };
  requestQuote(String stock){ . . . };
  boolean filter(StockBroker x)
    {return x.responseTime() <=10;};
};
StockBrokerAmbient stockbrokers =
  new StockBrokerAmbient();
```

In a more general concurrent setting [18], a concurrent object is equipped with its own virtual machine. A virtual machine is equipped with a stack, a heap, a queue of messages, and a program counter (PC) as shown in Figure 1.

```
interface ConcurrentObject { . . . }
class ConcurrentObjectClass
  implements ConcurrentObject {
  private VirtualMachine VM();
}
```

In a concurrent paradigm of [18], a concurrent object executes messages that it receives by invoking the corresponding methods. In order to be able to do that, the heap of the object's virtual machine must contain reflective classes such as `Class`, `Method`, `Message` etc. These classes are

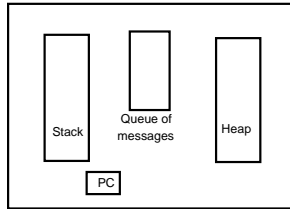


Figure 1. A concurrent object

stored on the heap of the object’s virtual machine. The heap also holds the object state. Execution of a method is based on the object’s stack according to the standard stack-oriented evaluation model.

A concurrent object gets activated by receiving a message. If a concurrent object is busy executing a method, the incoming message is queued in the message queue of the object’s virtual machine. Messages in the queue will be subsequently picked for execution when an object is not busy executing a method. So at any point in time an object is either executing a single message or else it is inactive (i.e., its queue of messages is empty).

In the extreme case, all objects are concurrent objects, i.e., the class `ConcurrentObjectClass` is identified with the class `Object`. A service object is now defined as a concurrent object:

```
interface ServiceObject
    extends ConcurrentObject { . . . }
```

We can now redefine an ambient in this new setting as a concurrent object which represents a dynamic collection of concurrent service objects:

```
class Ambient <T extends ServiceObject>
    extends ConcurrentObject {
    . . . }
```

Since an ambient is a concurrent object, it has its own virtual machine with a queue of messages sent to the ambient object and not serviced yet.

A mobile object is a concurrent object that is equipped with a location:

```
interface MobileObject
    extends ConcurrentObject{
    Location loc();
}
```

A region is an ambient that captures the notion of locality. It consists of all concurrent objects within the region as well as the service objects in that region, as illustrated in Figure 2.

```
class Region <T> extends Ambient<T> {
    Set<ConcurrentObject> objects();
    boolean withinRegion(MobileObject x);
    invariant (ForAll MobileObject x)
        (this.withinRegion(x) =>
```

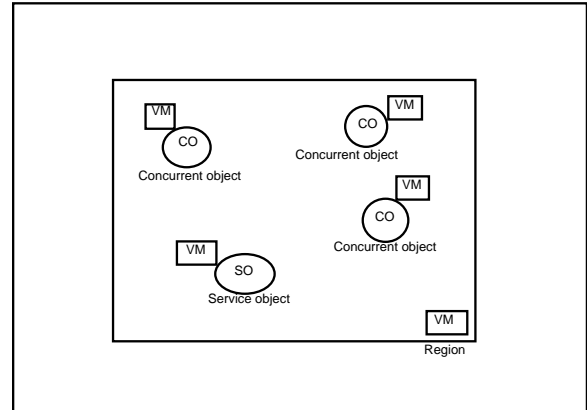


Figure 2. Regions of concurrent and service objects

```
x in this.objects());
}
```

For example, if class `Server` extends `ServiceObject` { . . . } then `Region<Server>` would be an example of a region type. Since a region is a concurrent object, it is equipped with its own virtual machine. Also, since a region is an ambient, it receives messages that are queued in the message queue of the region’s virtual machine to be serviced. Servicing a message sent to a region amounts to selecting a server object and sending the message to that server.

III. TYPES OF MESSAGES

Non-functional messages in this paradigm are objects. A message is created dynamically and it has a unique identifier like any other object. In the concurrent architecture described in Section II object identifiers must be global. The attributes of a message are the receiver object and the array of arguments along with a reference to a method. Messages of specific subtypes will have other attributes. This produces a hierarchy of message types that are subtypes of the type `Message`:

```
interface Message {
    Method m();
    Object receiver();
    Object[] arguments();
    int timeStamp();
}
```

When a message object is created its time stamp is recorded. The implementing class would have a constructor:

```
class MessageObject implements Message {
    MessageObject(Method m, Object receiver,
        Object[] arguments);
    int timeStamp();
    Method m();
    Object receiver();
    Object[] arguments(); }
```

Creating a message could be done just like for all other objects:

```
Message msg =
    new MessageObject(Method m, Object receiver,
                      Object[] arguments)
```

This implies message send in the underlying implementation. However, Message and MessageObject belong to the reflective core along with Class, Method, and Constructor. These types should be final in order to guarantee type safety at run-time. So an alternative is to have a special notation to create an asynchronous message. A functional (and hence synchronous) message is denoted using the usual dot notation:

```
x.m(a1, a2, . . . , an).
```

A non-functional (asynchronous etc.) message would be created as follows:

```
Message msg = x<=m(a1, a2, . . . , an).
```

In general, an asynchronous message does not have a result. The basic type of a message is point-to-point, one-way, and immediately executed. This type of a message could be expressed in a traditional notation

```
receiver.m(arguments)
```

In the new paradigm, the result of an asynchronous message send is a reference to the created message object. An example is:

```
Method requestQuote =
    getClass(``StockBrokerAmbient``).getMethod(
        ``requestQuote``, getClass(``String``));
Message requestQuoteMsg =
    new MessageObject(requestQuote,
                      stockBrokers, stock);
```

An alternative notation looks like this:

```
Message requestQuoteMsg =
    stockBrokers <= requestQuote(stock);
```

An update message is a message that mutates the state of the receiver object and possibly other objects as well. An update message does not have a result, hence we have:

```
interface UpdateMessage extends Message { . . . }
```

A special notation for an update message is

```
x<:=m(a1, a2, . . . , an)
```

The type of this expression is UpdateMessage.

A two-way message requires a response which communicates the result of a message. The result is produced by invoking the method future on a two-way message [18]. This method has a precondition which is that the future is resolved, i.e., that it contains the response to the message.

```
interface TwoWayMessage extends Message{...}
```

The implementing class would contain a constructor which takes the reply interval as one of its parameters.

```
class TwoWayMessageObject
```

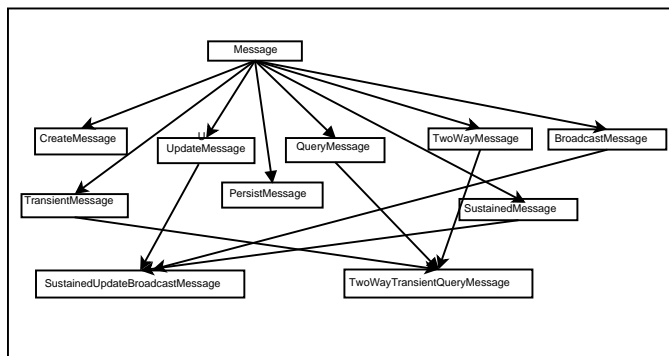


Figure 3. Message type hierarchy

```
implements TwoWayMessage {
TwoWayMessageObject(Method m,
    Object receiver, Object[] arguments,
                      int replyInterval);
boolean futureResolved();
boolean setFuture();
Object future()
    requires this.futureResolved();
}
```

An example of a two way message is:

```
TwoWayMessage requestQuoteMsg =
    new TwoWayMessageObject(requestQuote,
                            stockBrokers, stock, 20);
```

A suggestive notation for a two way message is:

```
TwoWayMessage requestQuoteMsg =
    stockBrokers <=> requestQuote(stock, 20);
```

A one-to-many message is of the type BroadcastMessage and it is sent to multiple objects. Using a suggestive notation for a one-to-many message, we would have:

```
Message requestQuoteMsg =
    stockBrokers <<=> requestQuote(stock);
```

A transient message has a discovery time specified as a finite time interval. If a message is not discovered and scheduled for execution before its discovery time has expired, the message will be regarded as expired and will never be scheduled for execution. The discovery time will be specified in the constructor of the implementing class. A suggestive notation for a transient message is <=|. A sustained message (i.e. a message whose discovery time is not limited) denoted as <=~ is specified by a special message type SustainedMessage.

IV. PERSISTENT OBJECTS

An object is promoted to persistence by executing a message persist which specifies a user name and a name space. This message binds the object to the given user name in the given name space. The root class Object is equipped

with a method `persist` which means that the model of persistence is orthogonal, i.e., objects of any type may be promoted to persistence.

```
class Object { . . .
void persist(NameSpace scope,String userID);
}
```

A name space consists of bindings of user names to objects. Name spaces can be nested. A name space is equipped with methods for establishing such a binding and for looking up an object in a name space bound to a given user id. Typically, name spaces are persistent.

```
interface NameSpace extends ConcurrentObject{
boolean bind(Object x, String name);
Object lookup(String name);
}
```

The type `PersistMessage` is now defined as follows:

```
interface PersistMessage extends Message {
NameSpace scope();
Object userID(String name);
}
```

Creation of a persist message is denoted by a special notation using the symbols `<=?persist`.

A schema extends a name space with additional methods. One of them is the method `select` that returns a set of objects in the schema that satisfy a given assertion.

```
interface Schema extends NameSpace { . . .
Set<Object> select(Assertion a);
}
```

The integrity constraints of a schema are specified in its invariant as illustrated in the example below. The schema `StockMarket` is equipped with a key constraint and a referential integrity constraint.

```
interface Stock {
String code();
float price();
}
interface Broker {
String name();
Set<Stock> stocks();
}
interface StockMarket extends Schema {
Set<Stock> stocks();
Set<Broker> brokers();
invariant:
(forAll s1,s2 in this.stocks():
s1.code()==s2.code() => s1.equals(s2));
(forAll b in this.brokers():
(forAll sb in b.stocks():
(exists s in stocks():
(sb.code() == s.code()))));
}
```

As for a specific assertion language, our previous results such as [2][4] are based on JML and more recent experiments are based on Spec# [1][6]. In fact, our extended

virtual platform [15] accommodates a variety of assertion languages.

V. QUERIES AND TRANSACTIONS

A query message is specified below as an asynchronous message. Its type is a subtype of `TwoWayMessage`. So the result of a query may not be immediately available. When it is, it will be available by sending a functional message future to the query message object.

```
interface QueryMessage
extends TwoWayMessage {
Schema scope();
Assertion query();
}
```

Creation of a particular query object is illustrated below using a special notation with the symbol `<=?select`:

```
StockMarket sch; QueryMessage q;
q <=?select(
forAll b in sch.brokers():
(exists s in b.stocks():
s.code()=='`SNP500'`));
```

A database server is a specific subtype of a service (and hence concurrent) object. It implements a schema:

```
interface DbServer
extends ServiceObject, Schema {
Sequence<Message> log();
}
```

Since a database server is a concurrent object, it is equipped with its own virtual machine. Typically, a database server is a persistent concurrent object. Hence by reachability, its schema (which includes persistent objects and integrity constraints) and its virtual machine will also be persistent.

A database server is equipped with a log of received messages. Here the view of messages as typed objects is critical. Committing a transaction requires extraction of the update and persist messages to reflect those changes in database collections. Implementing serializability protocols requires distinguishing update and query messages and controlling the order of their execution. All of this is possible because these messages are objects belonging to different types so that their properties can be inspected by sending functional messages to those objects.

Unlike the ODMG model, a transaction type is parametric. Its bound type specifies that the actual type parameter must be derived from the interface `Schema`.

```
interface Transaction<T extends Schema> {
boolean commit();
boolean abort();
}
```

Another distinctive feature of the notion of a transaction with respect to ODMG and other persistent object models is that a transaction is naturally equipped with a precondition

and a postcondition and it is defined as a sequence of messages of different types (such as query, update and persist messages). An illustrative example is:

```
interface StockUpdate
    extends Transaction<StockMarket> {
    StockMarket schema();
    void update(String stockCode, float value)
    requires (exists s in this.schema().stocks():
              s.code()==stockCode);
    ensures (forall s in this.schema().stocks():
            s.code()==stockCode =>
              s.price()==value);
}
```

The implementing class of the interface `Transaction` would have the following form:

```
class TransactionObject<T>
    implements Transaction<T extends Schema>{
    TransactionObject(T dbSchema);
    Sequence<Message> body();
    boolean commit();
    boolean abort();
}
```

Taking this approach one step further, a transaction is a concurrent object defined as follows:

```
class ConcurrentTransactionObject<T>
    implements ConcurrentObject
    implements Transaction<T extends Schema>
{ . . . }
```

VI. REFLECTION

Just like in Java Core Reflection (JCR), reflection in a language that supports messages as typed objects includes classes `Class`, `Method`, and `Constructor`. The main differences in comparison with JCR are:

- Reflection includes the interface `Message` with its various subtypes.
- Reflection includes the interfaces `Assertion` and `Expression` with their various subtypes.

The core reflective class `Class` has the following abbreviated signature. A distinctive feature is an assertion representing a class invariant.

```
class Class { . . .
    String name();
    Method[] methods();
    Method getMethod(String name,
                     Class[] arguments);
    Assertion invariant();
}
```

The reflective class `Method` is defined as follows. Its distinctive features are a pre condition and a post condition expressed as assertions. Their type is `Assertion`.

```
class Method { . . .
    String name();
    Class declaringClass();
    Assertion preCondition();
```

```
Assertion postCondition();
Class[] arguments();
Class result();
Expression body();
Object eval(Object receiver,
            Object[] args);
}
```

The body of a method is an expression evaluated by the function `eval`. Just like `Assertion`, the type `Expression` belongs to the reflective core. The method `eval` evaluates the method body after binding of variables occurring in the expression representing the method body is performed. The variables to be supplied to `eval` are the receiver and the arguments.

Availability of assertions in the classes `Method` and `Class` is a major distinction with respect to the current virtual machines such as JVM or CLR. This is at the same time a major difference with respect to the assertion languages such as JML or Spec#. Full implementation of this distinction is given in our previous work [15].

VII. RELATED RESEARCH

The orthogonal model of persistence implemented in [5] and the ODMG model of persistence [8] are based on promoting an object to persistence by either binding it to a name in a persistent name space or making it a component of an object that is already persistent. Message-based model of persistence presented in this paper is a further significantly different development after these initial approaches.

In the ODMG model queries and transactions are objects, and so are in our model, with additional subtleties. In our approach messages are objects, and queries and updates are particular types of messages. A transaction is a concurrent object which consists of a sequence of messages. The fact that messages are objects makes it possible to construct a transaction log as a sequence of messages of different types (queries and updates, checkpoints, commits etc.).

General integrity constraints are missing from most persistent and database object models with rare exceptions such as [1][4][7]. This specifically applies to the ODMG model, PJama, Java Data Objects, and just as well to the current generation of systems such as Db4 Objects [10], Objectivity [14] or LINQ [13]. 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# [6][11].

Constraints in the form of object-oriented assertions are a key component of our approach. Database integrity constraints are specified as class invariants, transactions are specified via pre and post conditions, and queries come with general filtering (qualification) predicates. In comparison with object-oriented assertion languages, such as JML [11] and Spec# [6][12], a major difference is that in our approach assertions are integrated in the run-time type system and

visible by reflection. This makes database integrity constraints accessible and enforceable at run-time. Reflective constraint management, static and dynamic techniques for enforcing constraints, and transaction verification technology are presented in [2][4][15].

Our sources of motivation for the view of concurrent, distributed and mobile objects were the languages ABCL [17][18] and AmbientTalk [9]. The core difference is that both of the above languages are untyped, whereas our approach here is based on a type system. A further distinction is that ABCL and AmbientTalk are object-based and our approach is class based. Other related work is given in [16]. Unlike ABCL reflective capabilities, reflection in this paper is type-safe. A major distinction is the assertion language as a core feature of the approach presented in this paper.

Verification techniques of object-oriented transactions with schemas and transactions specified in either JML or Spec# are presented in [1][2].

VIII. CONCLUSION

Object-oriented assertions allow specification of object-oriented schemas equipped with database integrity constraints, transactions and their consistency requirements, and queries. The view of messages as typed objects leads to a typed reflective paradigm equipped with a message-based orthogonal persistence.

Integrating the above features into the reflective capabilities of the virtual platform leads to static and dynamic techniques for enforcing database integrity constraints. Reflection in this paradigm is much more general than reflection in main-stream typed object-oriented languages as it includes message and assertion types that are integrated into the run-time type system.

The presented approach requires more sophisticated users that can handle object-oriented assertion languages such as JML or Spec#. Those languages and their underlying technologies come with nontrivial subtleties as they are still in the prototype phase. Integrating these technologies into existing object database systems presents a significant challenge yet to be addressed in our future research.

One the other hand, the benefits of the availability of general constraints and static verification of transactions with respect to those constraints are very significant. Data integrity as specified by the constraints could be guaranteed, runtime efficiency and reliability of transactions is significantly improved, and expensive recovery procedures will not be required for constraints that were statically verified. 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.

REFERENCES

- [1] S. Alagić, P. Bernstein, and R. Jairath, Object-oriented constraints for XML Schema, Proceedings of ICODDB 2010, *Lecture Notes in Computer Science 6348*, pp. 101-118.
- [2] S. Alagić, M. Royer, and D. Briggs, Verification technology for object-oriented/XML transactions, Proceedings of ICODDB 2009, *Lecture Notes in Computer Science 5936*, pp. 23-40.
- [3] S. Alagić, The ODMG object model: does it make sense?, Proceedings of OOPSLA, pp. 253-270, ACM, 1997.
- [4] S. Alagić and J. Logan, Consistency of Java transactions, Proceedings of DBPL 2003, *Lecture Notes in Computer Science 2921*, pp. 71-89, Springer, 2004.
- [5] M. Atkinson, L. Daynes, M. J. Jordan, T. Printezis, and S. Spence, An orthogonally persistent JavaTM, ACM SIGMOD Record 25, pp. 68-75, ACM, 1996.
- [6] M. Barnett, K. R. M. Leino, and W. Schulte, The Spec# programming system: an overview, Microsoft Research 2004, <http://research.microsoft.com/en-us/projects/specsharp/> [retrieved, November 16, 2010].
- [7] V. Benzaken and X. Schaefer, Static integrity constraint management in object-oriented database programming languages via predicate transformers, Proceedings of ECOOP '97, *Lecture Notes in Computer Science 1241*, pp. 60-84, 1997.
- [8] R. G. G. Cattell, D. Barry, M. Berler, J. Eastman, D. Jordan, C. Russell, O. Schadow, T. Stanienda, and F. Velez, *The Object Data Standard: ODMG 3.0*, Morgan Kaufmann, 2000.
- [9] T. Van Cutsem, Ambient references: object designation in mobile ad hoc networks, Ph.D. dissertation, Vrije University Brussels, 2008.
- [10] Db4 objects, <http://www.db4o.com> [retrieved, November 16, 2010].
- [11] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cook, P. Muller, and J. Kiniry, JML Reference Manual, <http://www.eecs.ucf.edu/~leavens/JML/> [retrieved, November 16, 2010].
- [12] K. R. Leino and P. Muller, Using Spec# language, methodology, and tools to write bug-free programs, Microsoft Research, <http://research.microsoft.com/en-us/projects/specsharp/> [retrieved, November 16, 2010].
- [13] LINQ: Language Integrated Query, <http://msdn.microsoft.com/en-us/library/bb308959.aspx> [retrieved, November 16, 2010].
- [14] Objectivity, <http://www.objectivity.com/> [retrieved, November 16, 2010].
- [15] M. Royer, S. Alagić, and D. Dillon, Reflective constraint management for languages on virtual platforms, *Journal of Object Technology*, vol 6, pp. 59-79, 2007.
- [16] J. Schafer and A. Poetzsch-Heffter, JCoBox: Generalizing active objects to concurrent components, Proceedings of ECOOP 2010, *Lecture Notes in Computer Science 6183*, pp. 275-299.
- [17] T. Watanabe and A. Yonezawa, Reflection in an object-oriented concurrent language, Proceedings of OOPSLA, pp. 306-315, ACM Press 1988.
- [18] A. Yonezawa, J.-P. Briot, and E. Shibayama, Object-oriented concurrent programming in ABCL/1, Proceedings of OOPSLA, pp. 258-268, ACM Press 1986.