



# Next-Generation Object Database Standardization

Date: 27-September-2007

mars/2007-09-13

Send Comments or Inquiries to <mailto:objectdb@omg.org>

## Object Database Technology Working Group White Paper

**Abstract:** Following the dissolution of the Object Data Management Group (ODMG) in 2001, standardization efforts for object databases languished. What has emerged since is a fractured marketplace where each vendor has developed a unique set of programming interfaces and features and no truly portable way of interacting with an object database exists. In 2005, the OMG's Object Database Technology Working Group was formed as the successor to the ODMG, and our first effort has been to create the object equivalent of the relational calculus. We believe that the foundation for this "object calculus" can be found in the research done by Prof. Kazimierz Subieta and his students at the Polish-Japanese Institute of Information Technology. We have prepared this white paper to serve as an introduction to Prof. Subieta's "stack-based architecture" (SBA) and to define the OMG version of it. The definitions and semantics of SBA will, we believe, allow the construction of a complete and correct object model that supports a powerful object query language as well as a complete and correct set of equivalent native programming language bindings.

*This paper presents a discussion of technology issues considered in the Object Database Technology Working Group of the Object Management Group. The contents of this paper are presented to create discussion in the computer industry on this topic; the contents of this paper are not to be considered an adopted standard of any kind. This paper does not represent the official position of the Object Management Group.*

## Table of Contents

|   |    |
|---|----|
| Introduction.....                                 | 3  |
| 1 The basic model.....                            | 4  |
| 2 Extending the model to include inheritance..... | 9  |
| 3 Extending the model to include roles.....       | 10 |
| 4 Programming language access.....                | 13 |
| 5 Updating from AOQL.....                         | 17 |
| 6 So what does all this mean?.....                | 17 |
| 7 A new object database standard.....             | 19 |
| 8 References.....                                 | 19 |

## Introduction

The OMG's Object Database Technology Working Group (ODBTWG) was formed at the December 2005 Technical Working Group meeting in Burlingame, California<sup>1</sup>. This working group was created to "re-ignite" object database standardization, which has languished since the dissolution of the Object Data Management Group (ODMG). In support of this goal, the OMG acquired the rights to the ODMG's intellectual property 8 (the ODMG 3.0 specification) and the ODBTWG 8 conducted a survey of object database vendors on areas of interest for standardization.

The survey responses indicated that vendors were most interested in standardization efforts focused on:

- XML/XSD (standard XSD for data import/export)
- Standardized programming language bindings, especially for new languages like C#
- Standardization of advanced features like replication, e.g. define expected behaviors etc.

The ODBTWG discussed 3 possible ways to pursue standardization:

1. Develop a theoretical underpinning for object databases and base all standards on it (object database equivalent of the relational calculus)
2. Quickly produce a standard on a vendor-neutral technology, like a standard XML representation for object database contents
3. Quickly develop a standard for C#/.NET programming language bindings, working through vendor-specific issues ("we already have a C# API that our customers use and we don't want to change our code")

Approach 1 was chosen from the above list. The catalyst for its selection was a paper submitted by Professor Kazimierz Subieta of the Polish-Japanese Institute for Information Technology (PJIIT) which provided a thorough and complete set of definitions for an abstract object data store and a corresponding stack-based abstract query engine ( <http://www.sbql.pl/> ). This is work that had not been done for object databases in the past, and it was generally recognized as a weakness for object databases in general and the ODMG 3.0 specification in particular (e.g. incompleteness and inconsistency in the object model given in Chapter 2). Approach 2 was briefly considered but since the W3C was already standardizing object representation in XML we decided to wait until their work was completed to avoid duplication of effort. Approach 3 was considered also but was thought to hold as little promise for adoption as the original ODMG 3.0 language binding efforts given the absence of a sound theoretical foundation upon which to build the bindings.

---

<sup>1</sup> Formation of this WG in 2005 should not be interpreted to mean that OMG's interest in object databases started then. It actually started in September 2003 with a single presentation and continued for the next 2 years during which time OMG obtained the rights to the ODMG3.0 Specification from Morgan Kaufman.

The following sections of this paper explain the abstract object store and its associated abstract query engine. To avoid being overly tied to Professor Subieta's work we have chosen to use different names and acronyms for the abstract store models et. al., but the influence of his research on our thinking will be obvious. It is our hope that object database vendors will see the potential in this work and join us in developing the full specification of this abstract object database, which should serve as a platform-independent model upon which specific implementations can be successfully built.

## 1 The basic model

The basic abstract store model, referred to here as Abstract Store model 0 (AS0) corresponds to Professor Subieta's "M0" store model. The AS0 store model has the features required to persist objects having a minimalist object model. In the AS0 model, everything is an object, and objects resolve to a single value. As such, there is no concept of "object attributes," where object attributes are analogous to fields in a record. In AS0, each "object attribute" would itself be an object, and the aggregations of these "object attributes" into an "object" would be accomplished using an aggregation object. Each AS0 object has a unique identifier referred to as its object ID, and an external name that is assigned to the object. The external name is a developer-chosen domain name applicable for the type of object (i.e. a class name or the name of a "field"), so unlike the object ID it is not guaranteed to be unique. In addition to the ID and external name, each object has a value that is returned when the object is referenced. The value can be a number, a string, the ID of another object, or a set of objects. Therefore, in the AS0 model all objects can be represented by tuples (triplets) as follows:

- **Atomic object:**  $\langle i, n, v \rangle$  where  $i$  is the ID of the object,  $n$  is an external name assigned to the object, and  $v$  is the value of the atomic object (e.g. an integer, a string, etc.)
- **Pointer (reference) object:**  $\langle i_1, n, i_2 \rangle$  where  $i_1$  is the ID of the reference object,  $n$  is an external name assigned to the object, and  $i_2$  is the ID of the object referred to.
- **Aggregation object:**  $\langle i, n, T \rangle$  where  $i$  is the ID of the object,  $n$  is an external name assigned to the object, and  $T$  is a set of objects comprising the aggregate.

The AS0 store itself can be represented by a tuple as well:

- **AS0 store:**  $\langle S, R \rangle$  where  $S$  is a set of objects and  $R$  is the set of "root" object IDs (that is, the set of object IDs that an external application would use as a "starting point" to navigate to all of the other objects in the store)

The AS0 model does not include static inheritance, and its simplicity allows concrete realization in simple tabular data, XML, relational tables, or persisted objects. Consider the following instance of an AS0 store:

```

aStore = <S = {< i1, Emp, { < i2, name, "Doe" >,
                    < i3, sal, 2500 >,
                    < i4, worksIn, i17 > } >,
             < i5, Emp, { < i6, name, "Poe" >,
                    < i7, sal, 2000 >,
                    < i8, worksIn, i22 > } >,
             < i9, Emp, { < i10, name, "Lee" >,
                    < i11, sal, 900 >,
                    < i12, address, {<i13, city, "Rome" >,
                                    <i14, street, "Boogie">,
                                    <i15, house#, 13 > } >,
                    < i16, worksIn, i22 > } >,
             < i17, Dept, { <i18, dname, "Trade" >,
                    < i19, loc, "Paris" >,
                    < i20, loc, "London" >,
                    < i21, employs, i1 > } >,
             < i22, Dept, { < i23, dname, "Ads" >,
                    < i24, loc, "Rome" >,
                    < i25, employs, i5 >,
                    < i26, employs, i9 > } > } >,
R = {i1, i5, i9, i17, i22} >

```

**Figure 1: Example of AS0 store**

The AS0 store shown in Figure 1 contains 5 “root” objects identified by the set R. These objects have object IDs *i1*, *i5*, *i9*, *i17*, and *i22*. Object *i1* is an “Emp” (Employee) object, which is an aggregation object. Referencing object ID *i1* will return a set of objects which correspond to what we would consider to be the “object attributes” of an Emp object, in this case a name, a salary, and a reference object contained in a field/attribute named “worksIn.” The employee name is contained in the object with ID *i2*, which has external name “name,” and which resolves to the string value “Doe.” Likewise, the salary is contained in the object with ID *i3*, which has external name “sal,” and which resolves to the numeric value 2000. The department the employee works in is returned in the object with ID *i4*, which has external name “worksIn,” and which resolves to object ID *i17*. You can see that the object with ID *i17* is an object with external name “Dept,” which is also an aggregation object that resolves to a set of objects containing the the department name, its locations, and reference objects pointing to the employees of the department (the objects with the external name “employs”). A graphical depiction of the contents of the store in Figure 1 is shown in Figure 2.

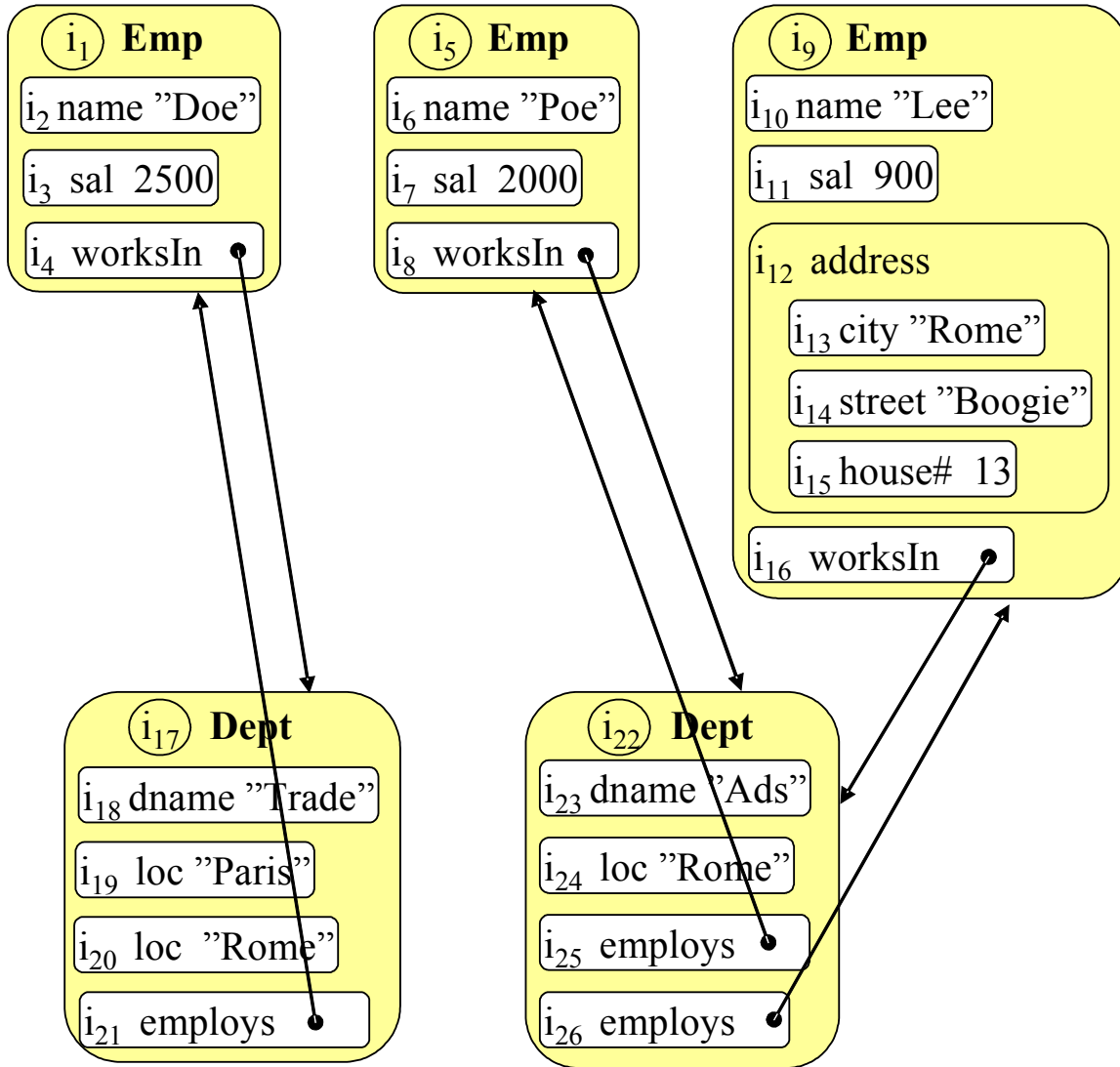


Figure 2: Graphical representation of AS0 store in Figure 1

There is nothing remarkable about this simple store model and it seems quite intuitive. Its significance lies in the precise definition of an object as one of the 3 allowed kinds of tuples and of the store itself as a pair of sets (the set of all objects in the store and the set of all “root” object IDs).

Using this simple data store model as a foundation, we can now define an abstract stack-based query processor that can query the contents of an AS0 store. Stack-based machines will be familiar to anyone who has used a Hewlett-Packard Reverse Polish Notation (RPN) calculator, so this paper will not go into depth on stack-based machines in general but will instead focus on the stack-based query processor itself.

In order to query the contents of an AS0 store, we need a query language to describe what we want from the store and a query processor to interpret the query language and return the desired items from the store. The query language we will use is the Abstract Object Query Language (AOQL), which is (at this point) identical to Prof. Subieta's Stack-Based Query Language (SBQL). The syntax rules for AOQL for an AS0 store are as follows:

**Table 1: AOQL syntax rules**

| Syntax Rule   | Notes  |
|---|--|
| query ::= literal   | The set L  |
| query ::= name  | The set N  |
| query ::= unaryAlgOperator query  | Unary algebraic operators                        |
| unaryAlgOperator ::= <i>count</i>   <i>sum</i>   <i>max</i>   -   <i>sqrt</i>   <b>not</b>   <i>avg</i>   ... |  |
| query ::= query binaryAlgOperator query   | Binary algebraic operators                       |
| binaryAlgOperator ::= = < > +  - * /  <b>and</b>   <b>or</b>   <i>intersect</i>  ...                          |  |
| query ::= query NonAlgOperator query  | Non-algebraic operator                           |
| NonAlgOperator ::= <b>where</b>   .   <b>join</b>   $\forall$   $\exists$                                     |  |
| query ::= $\forall$ query query   $\exists$ query query   | Alternative (traditional) syntax for quantifiers |
| query ::= query <b>as</b> name  | Name definition                                  |
| query ::= query <b>group as</b> name  | Grouping and name definition                     |
| query ::= <b>if</b> query <b>then</b> query   | Conditional query                                |
| query ::= <b>if</b> query <b>then</b> query <b>else</b> query   | Another conditional query                        |
| querySeq ::= query   query, querySeq  | Sequence of queries                              |
| query ::= <b>struct</b> ( querySeq )   (querySeq)   | Structure constructor                            |
| query ::= <b>bag</b> ( )   <b>bag</b> ( querySeq )  | Bag constructor                                  |
| query ::= <b>sequence</b> ( )   <b>sequence</b> ( querySeq )  | Sequence constructor                             |

Note that as an abstract query language, a concrete implementation could use whatever operator names it desires as long as these general rules are followed. Likewise, in the AS0 store a concrete implementation would not have to actually include the class name in every object in the store, but the equivalent functionality must be provided. The AS0 store model is not meant to constrain a concrete implementation, and likewise AOQL syntax is not intended to constrain an actual implementation.

Table 2: AOQL Examples

| Example Query   | Features illustrated  |
|---|---|
| 2000  | Literal   |
| <i>Emp</i>  | Name  |
| <i>Sal</i>  | Name  |
| 2+2   | algebraic operator  |
| <i>sal</i> > 2000   | algebraic operator  |
| <i>Emp where</i> ( <i>sal</i> > 2000)   | Non-algebraic and algebraic operator  |
| <i>Emp where</i> ( <i>sal</i> > 2000) . ( <i>name</i> , ( <i>worksIn.Dept</i> ))              | As before, plus projection and structure composition                                |
| (( <i>Emp as e where</i> (( <i>e.sal</i> ) > (2000 + <i>x</i> + <i>y</i> ))). <i>e.name</i>   | Auxiliary name <i>e</i>   |
| (( <i>Emp as e join</i> (( <i>e.worksIn.Dept as d</i> )) . ( <i>e.name</i> , <i>d.dname</i> ) | Dependent join with auxiliary naming, path expression, projection and <b>struct</b> |
| $\forall$ ( <i>Emp where sal</i> < 1000) (( <i>worksIn.Dept.dname</i> ) = “Ads”)              | Universal quantifier with no “variable”   |
| $\exists$ ( <i>Emp as e</i> ) <i>count</i> ( <i>e.address</i> ) = 0                           | Existential quantifier with a “variable” <i>e</i> and counting function             |
| <b>bag</b> ( 1, 1, 2, 3, 5, 8, 3, 13)   | Bag constructor   |
| 2000  | Literal   |

Table 2 above shows examples of AOQL queries that illustrate features of the syntax defined in Table 1.

Consider the following AOQL query applied to the example AS0 store depicted in Figure 1 and Figure 2:

### Dept join avg((employs.Emp).sal)

This query returns each department along with the average salary of its employees. The abstract query processor evaluates the query using two stacks. One stack is the environment stack (abbreviated ENVs in Prof. Subieta’s papers) and the other is the query results stack (abbreviated QRES in Prof. Subieta’s papers). The query results stack is analogous to the stack in a Hewlett-Packard RPN calculator as it holds intermediate results of the query evaluation and at completion contains the final result. The environment stack is analogous to a program call stack and it contains the set of objects upon which the various stages of the query are performed.

In the query above, the environment stack initially will contain all of the root object IDs (set R) in the store. The evaluation of the string “Dept” will cause a set containing the objects IDs of all objects having external name “Dept” (the set would be  $\{i17, i22\}$ ) to be pushed onto the query results stack. When the “join” string is encountered, the query processor will evaluate the first object ID ( $i17$ ) and, since  $i17$  refers to an aggregation object, the associated set of objects (in this case the objects corresponding to  $\{i18, i19, i20, i21\}$ ) is pushed onto the environment stack. As processing continues, intermediate results are placed onto the query results stack while sets of objects that the query operators are to access are placed on the environment stack. To see all of the steps the abstract query processor takes to evaluate this expression, see the accompanying Flash movie [omg.swf](#).<sup>2</sup> This Flash movie was put together by Prof. Subieta to show how the stack-based query processor works.

To illustrate the power of the AS0 and AOQL models, the ODBTWG received a demonstration from Prof. Subieta of a concrete AOQL implementation developed at the PJIIT that uses an XML file as an AS0 store. A GUI allows the user to query the store contents using a concrete syntax that follows the rules defined in Table 1. Using this demonstration program, it becomes obvious that AOQL’s syntactical rules in conjunction with the definition of an AS0 store makes it possible to perform complex queries very easily, some of which are difficult to express in standard SQL. For example, if you want a report with the name of each department and the sum of the salaries of its employees which are not bosses, this can be written as:

```
deptemp.(((Dept as d) join ((sum(d.employs.Emp.sal._VALUE)
- (d.boss.Emp.sal._VALUE)) as s)).(d.dname._VALUE, s));
```

**Figure 3: Sample concrete query with XML store from demo program**

In Figure 3, “deptemp” is a name for the XML data store that is a concrete implementation of an AS0 store. The concrete query language shown here is very close to the abstract syntax in Table 1. The query joins a department object (renamed to “d”) with the sum of the values of the departments’ employees’ salaries less the salary of the department boss, where this total is renamed to “s.” The query then returns a pair containing the department name and the total “s.”

## 2 Extending the model to include inheritance

Extending the AS0 store model to include static inheritance is easily done by changing the definition of the AS0 store as follows to create the AS1 store model, which corresponds exactly to Prof. Subieta’s M1 store model:

- **AS1 store:**  $\langle S, C, R, CC, SC \rangle$  where S is a set of objects and R is the set of “root” object IDs just as in AS0. C is the set of all classes in the store, and classes themselves are objects. The class objects in an AS1 store are aggregation objects which contain only *class invariants* such as methods; they do not contain variants

<sup>2</sup> Link opens up a zip file; open the .swf file inside.

such as “data members” or “fields” (e.g.  $\langle i1, \text{“EmpClass”}, \{ \langle i2, \text{“changeSal”}, (\text{method code}) \rangle, \langle i3, \text{“changeDept”}, (\text{method code}) \rangle \} \rangle$ ). CC is a relation (a table) that defines which classes are related by inheritance, e.g. if  $\langle i1, i2 \rangle \in CC$ , then the class identified by object ID  $i1$  inherits from the class identified by object ID  $i2$ . SC is a relation that defines class membership for the non-class objects in the store, e.g. if  $\langle i1, i2 \rangle \in SC$ , the object identified by object ID  $i1$  is a member of the class identified by object ID  $i2$ . Note that the tuples (triplets) in the store are the same as for the AS0 store.

The abstract query language (AOQL) itself does not need to be modified to support the AS1 store. The only change required is the evaluation process which pushes objects onto the environment stack. When an object identifier is evaluated, the object’s contents are pushed onto the environment stack. Afterward, any invariants for the object’s class and its superclasses are also pushed onto the environment stack. This gives the query processor access to all of the object’s “fields” and inherited methods.

### 3 Extending the model to include roles

Static inheritance is useful but is sometimes insufficient for modeling real-world situations. For example, a Person class might include basic attributes for a person, but these might not include the attributes necessary to define an Employee class. With only static inheritance, the schema typically will have the Employee class inherit from the Person class. The difficulty comes when a Person object is created and that object must later be made to model an Employee (e.g. the real-world person is hired by a company). In this case, the Person object might be used to construct a new Employee object and the original Person object might be discarded. This introduces complexity in an application, since you don’t really need to create a new “person” when someone gets a job. The alternative to using inheritance is to use *composition*, where an Employee object contains only attributes not found in a Person object in conjunction with a pointer/reference to a Person object. Using this system, you wouldn’t have to “throw away” the original Person object when the Person needed to model an Employee, as the newly-created Employee object could point to your original Person object. Of course, this approach breaks the Liskov Substitution Principle<sup>8</sup> since you can’t use an Employee object wherever you might use a Person object since Employee would no longer be a subtype of Person.

The solution for this common dilemma is contained in Prof. Subieta’s concept of *dynamic inheritance*, or object roles. With dynamic inheritance, an application can define an Employee class, a Parent class, and other such classes all of which are subtypes of Person. When a person object requires an Employee role, the Employee object can be created and be associated with the Person object as needed and later be disassociated or destroyed when the Person is no longer acting in that role.

While this concept would allow a database schema to match the “real world” much more closely than static inheritance allows, it is not a concept that is currently realized in object-oriented programming languages. This does not mean, however, that this concept could not be supported for objects resident in an AS2 store through the use of database-

supplied APIs and query semantics. The AS2 store model includes the static inheritance features of AS1 plus dynamic inheritance:

- **AS2 store:**  $\langle S, C, R, CC, SC, SS \rangle$  where S, C, R, CC and SC are as defined for AS1. The relation SS is added to define the dynamic inheritance between objects and roles. That is, if  $\langle i1, i2 \rangle \in SS$ , the role object identified by object ID  $i1$  is dynamically inherited by the object identified by object ID  $i2$ .

The following is an example of an AS2 store:

```
aStore = <S = { <i1, Person,    { <i2, name, "Doe" >,
                          <i3, born, 1948 > } >,
              <i4, Person,    { <i5, name, "Poe" >,
                          <i6, born, 1975 > } >,
              <i7, Person,    { <i8, name, "Lee" >,
                          <i9, born, 1951 > } >,
              <i13, Emp,      { <i14, sal, 2500 >,
                          <i15, worksIn, i127 > } >,
              <i16, Emp,      { <i17, sal, 1500 >,
                          <i18, worksIn, i128 > } >,
              <i19, Student, { <i20, studentNo, 223344 >,
                          <i21, faculty, "Physics" > } >
              } >,
  <C = { <i40, PersonClass , { <i41, age, (age code)> } >,
        <i50, EmpClass ,    { <i51, changeSal,
                          (changeSal code) >,
                          <i52, netSal, (netSal code)> } >,
        <i60, StudentClass , { <i61, avgScore,
                          (avgScore code) > } >
        } >,
  <R = { i1, i4, i7, i13, i16, i19 } >,
  <CC = { } >,
  <SC = { < i1, i40 >, < i4, i40 >, < i7, i40 >, < i13, i50 >,
        < i16, i50 >, < i19, i60 >
        } >,
  <SS = { < i13, i4 >, < i16, i7 >, < i19, i7 > } >
```

**Figure 4: Example of AS2 store**

Notice in Figure 4 that there is no static inheritance between the classes in set C since relation CC is empty. That is, the EmpClass object (ID  $i50$ ) does not inherit from the PersonClass object (ID  $i40$ ). The instances of the EmpClass object contain only data members (fields) applicable to employees (“sal” for salary, “worksIn” for pointer to department (not modeled)) and do not include data members of PersonClass objects (“name” for name and “born” for year of birth), so instances of EmpClass cannot be “substituted” for PersonClass objects, though of course queries for employees will return person objects that have the Emp role.

The relation SC in Figure 4 shows that the objects with identifiers  $i1$ ,  $i4$ , and  $i7$  are members of the class with class object ID  $i40$  (PersonClass). Likewise, the objects with identifiers  $i13$  and  $i16$  are members of the class with class object ID  $i50$  (EmpClass) and

the object with identifier *i19* is a member of the class with class object ID *i60* (StudentClass).

The relation SS in Figure 4 shows which objects have which roles. Note that unlike the relations CC and SC, SS changes over time. SS shows that the role object with ID *i13* is associated with the object with ID *i4* (that is, Person “Poe” is currently acting as an Employee with salary 2500 working in the department with object ID *i127*).

A graphical depiction of the AS2 store shown in Figure 4 is shown below in Figure 5.

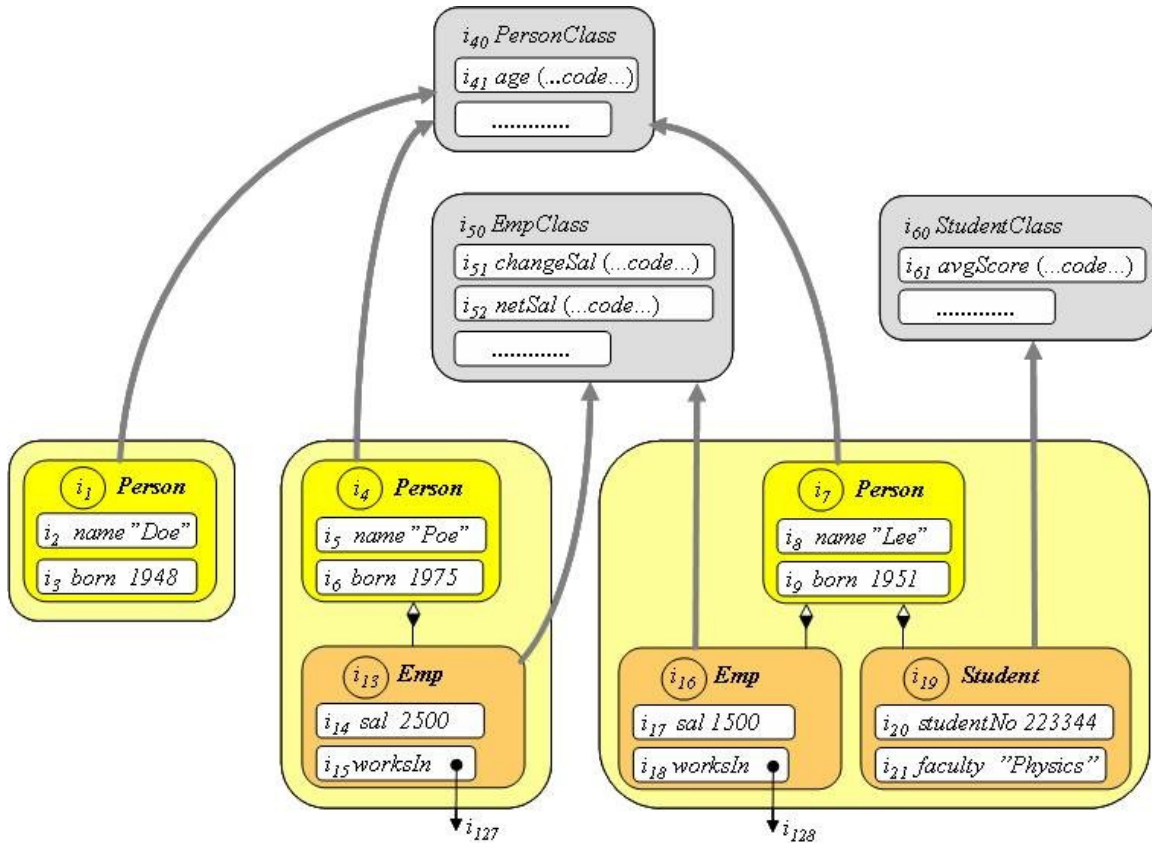


Figure 5: Example of AS2 store

The abstract query language (AOQL) syntax adds 2 new rules for supporting the AS2 store as shown below in Table 3:

Table 3: New AOQL syntax rules to support AS2

| Syntax Rule                               | Notes  |
|---|--|
| query ::= (role_name) query               | Dynamic cast to return a role from result of a query, useful for objects which have multiple roles |
| query ::= query <b>has role</b> role_name | Existential test to see if result has the specified role   |

The modifications to the abstract query processor itself are also minor, namely that the initial state of the environment stack should include all of the objects and role objects present in the store prior to query evaluation. Note here that this means an actual implementation *should have access to* these objects, as opposed to pushing all of these objects onto a call stack. Again, since we are talking here about an abstract machine there is no constraint for stack size, etc. that would constrain an actual implementation. Otherwise, the invariants from the role objects' superclasses as well as the invariants from the "regular" objects' superclasses are pushed onto the environment stack during query evaluation as is done for the AS1 store model.

## 4 Programming language access

Typically, object database developers want to access the contents of an object database from their development language of choice (Java, C++, et. al.) without having to construct and parse a query string. AOQL is in itself a language that is distinct from the developer's native programming language, and it is natural to question how we might map the AOQL syntax to an object programming language so that the expressive power of AOQL could be used via programming language APIs<sup>3</sup>.

In considering a programming language mapping to AOQL, it is tempting to construct an abstract programming language to go with the abstract query language. This would keep our formal definitions of the SBA concepts all in the abstract without "polluting" them with platform/language-specific concepts. However, the task of defining an abstract object-oriented programming language is difficult and there are actual programming languages available which are platform and operating system neutral, which is the chief benefit one would get from an abstract language. For the purposes of this white paper, we will focus on a mapping from the AOQL to Ruby, which will serve as our "abstract" object programming language. As an interpreted language that embodies both object-oriented and functional programming concepts, Ruby is an excellent choice for an AOQL-to-language mapping. Other platform-neutral language choices were considered for this document (Java and Scala to name two), but Ruby was chosen for its popularity, clarity and concise syntax.

Note that this white paper does not attempt to define the programming language APIs that an application developer would use to query an object database. The intent is to show how AOQL operators can be mapped to Ruby so that such APIs could later be created. If AOQL's operators and syntax can be mapped to a programming language, the theoretical underpinnings that make AOQL a complete and correct query language can be used to define a set of APIs that are also correct and complete for application developers.

---

<sup>3</sup> A concrete implementation of AOQL would require a parser that would translate the query string into a series of native programming language API calls, so formalizing this mapping is helpful even for the construction and/or specification of a parser.

**Table 4: Mapping of AOQL syntax to Ruby**

| <b>Num</b> | <b>AOQL Syntax Rule</b>   | <b>Ruby equivalent</b>  |
|------------|---|---|
| 1          | query ::= literal   | Any Ruby literal, e.g. 2000, “abc” ...  |
| 2          | query ::= name  | Any Ruby variable name, e.g. <i>Emp</i>   |
| 3          | query ::= unaryAlgOperator query  | Unary algebraic operators   |
|            | unaryAlgOperator ::= <i>count</i>   <i>sum</i>   <i>max</i>   -   <i>sqrt</i>   <b>not</b>   <i>avg</i>   ... | <i>\$container<sup>4</sup>.size</i>   <i>\$container.inject(0) { cum_sum,item  cum_sum + item }</i>   <i>\$container.max</i>   -   <i>Math.sqrt</i>   !   <i>\$container.inject(0) { cum_sum,item  cum_sum + item }</i> / <i>\$container.size</i>   ... |
| 4          | query ::= query binaryAlgOperator query   | Binary algebraic operators  |
|            | binaryAlgOperator ::= = < > + - * /  <b>and</b>   <b>or</b>   <i>intersect</i> ...                            | = < > + - * /  <i>and</i>   <i>or</i>   <i>&amp;</i>   ...  |
| 5          | query ::= query NonAlgOperator query  | Non-algebraic operator  |

<sup>4</sup> NOTE: Throughout table 4, *\$container* refers to any Ruby container that has *Enumerable* as a mixin, including arrays and anonymous arrays, hashes, sequences, etc.

|    |   |   |
|----|---|---|
|    | NonAlgOperator ::= <b>where</b>   .   <b>join</b>   $\forall$   $\exists$ | <p>Implemented through:</p> <ul style="list-style-type: none"> <li>predicate boolean result: <b>where</b>   <math>\forall</math><br/>e.g. <b>where</b> employee's salary is more than 55000 could be done as:<br/><code>result = \$container.find_all { employee  employee.salary &gt; 55000}</code></li> <li>e.g. <math>\forall</math> employees with salary &gt; 55000 could be done as:<br/><code>(\$container.find_all { employee  employee.salary &gt; 55000}).each { s  puts s}</code></li> <li>predicate boolean result and result set analysis: <math>\exists</math><br/>e.g. <math>\exists</math> employee with salary &gt; 55000 could be done as:<br/><code>(\$container.find_all { employee  employee.salary &gt; 55000}).nil?</code></li> <li>direct object references: <b>join</b></li> <li>built-in operator: .</li> </ul> |
|    | query ::= $\forall$ query query   $\exists$ query query                   | See above.  |
| 6  | query ::= query <b>as</b> name  | Name definition: using Ruby variables   |
| 7  | query ::= query <b>group as</b> name                                      | Grouping and name definition: using variables and structuring elements (collections, classes)   |
| 8  | query ::= <b>if</b> query <b>then</b> query                               | <i>if (expr) then &lt;statements&gt; end</i>  |
| 9  | query ::= <b>if</b> query <b>then</b> query <b>else</b> query             | <i>if (expr) then &lt;statements&gt;<br/>else &lt;statements&gt; end</i>  |
| 10 | querySeq ::= query   query, querySeq                                      | Sequence of queries: could be in a method or could be an anonymous function declaration with multiple queries within  |

|    |   |   |
|----|---|---|
| 11 | query ::= <b>struct</b> ( querySeq )  <br>(querySeq)            | <p>Here, a structure containing an employee's salary and department is created from multiple query sequences:</p> <pre>EmpSalDept = Struct.new("EmpSalDept", :salary, :dept) result = EmpSalDept.new( (\$container.find{ employee  employee.SSN == 123456789}).salary, ((\$container.find{ employee  employee.SSN == 123456789}).worksIn )</pre>                            |
| 12 | query ::= <b>bag</b> ( )   <b>bag</b> ( querySeq )              | <p>Here, a bag (simple unordered collection) is created from multiple query sequences, the first object in the "bag" is the employee's salary and the second object is what department the employee works in:</p> <pre>Array[].( (\$container.find{ employee  employee.SSN == 123456789}).salary, ((\$container.find{ employee  employee.SSN == 123456789}).worksIn )</pre> |
| 13 | query ::= <b>sequence</b> ( )  <br><b>sequence</b> ( querySeq ) | <p>See above, in Ruby the sequence and the bag would both be created from an array, which can be iterated like an unordered list or accessed by cell number (subscript)</p>   |

Programming language access to an object database is a paramount concern for most developers, so it may seem unnatural not to mention it first. The concepts of the Stack Based Architecture, however, such as the definition of an object and the structure of a store or the operation of the stack-based abstract machine, are easier to describe in terms of the AOQL. If the contents of the query results stack (QRES) can be returned in a "container" (equivalent of Ruby type with Enumerable `mix in`), then developing a mapping between the AOQL and its native language equivalent is fairly straightforward as shown in Table 4.

## 5 Updating from AOQL

Everything described in this paper has been about reading the contents of an object database. What if you need to update the contents of an object database? Prof. Subieta and his team at the PJIIT have developed extensions to their Stack-Based Query Language (SBQL) that allow one to update the contents of a database. These changes are still being worked on and the syntax is still being finalized.

## 6 So what does all this mean?

The precise and complete definitions of the AOQL and abstract stores that have been defined by Prof. Subieta's research allow us to build object databases with very powerful query capabilities. Concrete implementations of these abstract machines would be quite welcome in the object database marketplace, and there are reference implementations that have been developed at the PJIIT. The ODBTWG believes an OMG specification could be developed that features several compliance levels, thereby allowing market differentiation depending on what features customers wish to pay for. Table 5 shows a possible conformance level/feature matrix that could be used to indicate what level of support for these features an object database could provide. A vendor could then indicate, for example, that they conform at level 0 and include optional features SY and QL, so their product could be said to be 0+SY+QL conformant.

**Table 5: Possible conformance level/feature matrix**

| <b>Conformance Level</b> | <b>Store Model</b> | <b>Compatible Data</b>   | <b>Features</b>   | <b>Optional Features</b>  |
|--------------------------|--------------------|--|---|---|
| 0                        | AS0                | One or more of the following:<br>-tabular/CSV<br>-relational tables<br>-XML (w/o inheritance)<br>-simple persisted objects (no inheritance)  | All of the following:<br>-native PL APIs providing equivalent of basic AOQL functionality<br>-storage of basic objects (no inheritance)<br>-full transaction semantics          | -replication (RP)<br>-multi-versioning (MV)<br>-synchronize over non-reliable network (SY)                                    |
| 1                        | AS1                | Zero or more of the above plus one or more of the following:<br>-XML files with static inheritance<br>-persisted objects which have static inheritance   | All of the above features plus all of the following:<br>-native PL APIs providing equivalent of inheritance-aware AOQL<br>-store of objects with static inheritance             | -hot-spare fault tolerance (HS)<br>-concrete query language binding to AOQL for query string access to database contents (QL) |
| 2                        | AS2                | Zero or more of the above plus one or more of the following:<br>-XML files which include dynamic inheritance (roles) in their schema<br>-persisted objects which have dynamic inheritance (object roles) | All of the above features plus all of the following:<br>-native PL APIs providing equivalent of role-aware AOQL (e.g. the 2 additional syntax rules included in AOQL for roles) |   |

Initially the OMG would have to issue RFPs to establish the full and complete definitions of the abstract query language and the semantics of the abstract store models. The semantics and behavior of the standardized optional features would also have to be defined. Subsequently, a second series of RFPs could be issued for concrete implementations in Java, C++, etc.

## 7 A new object database standard

What we would like to have in the end is a new standard for object databases that is based on a sound theoretical framework with precise and complete definitions. Prof. Subieta's work is a great starting place because it shows what must be available in an object store in order to support an advanced query language. What we imagine is a new standard which could be fashioned after the ODMG 3.0 specification, something like this:

### OMG "Next Generation" Object Database Standard ("ODMG 4.0")

Chapter 1 – Introductory material

Chapter 2 – New object model based on the abstract store model and an abstract stack-based object query language (AOQL), includes definitions and detailed semantics of all optional features to be standardized

Chapter 3 – XML/XSD specification for data import/export as replacement for "ODL", provided for all conformance levels

Chapter 4 – Full syntax of abstract query language, provided for all conformance levels

Chapter 5, 6, ... - Programming language APIs for specific language bindings

Our new specification would not have to be done in this way, but such an organization would be familiar to those who have used ODMG 3.0 in the past. Of prime importance would be clear definitions and explicit semantics (especially in chapter 2), even including the use of state diagrams or Petri nets as needed to convey the semantics of how certain features are supposed to work in conformant products.

Looking forward, the plan is (assuming there is sufficient vendor interest) for the ODBTWG to prepare (and then issue from its parent task force) one or more RFPs that together will establish the full and complete definitions of the abstract query language (AOQL) and the semantics of the abstract store models, as well as, the semantics and behaviors of optional features (compliance points). This will be the Platform Independent Model (PIM) for a conformant object database. When the PIM is defined, it will serve as the basis for a series of RFPs to develop specifications for concrete implementations of the "Next Generation Object Database" in Java, C++, etc., i.e., Platform Specific Models (PSMs).

## 8 References

- [1] Letter from Morgan Kaufman Publishers to OMG re: "License for use of the Object Data Standard" dated 3 June 2004, OMG document: [omg/2004-06-04](http://omg.org/2004-06-04).
- [2] The Object Database Standard: ODMG 3.0, Cattell et. al., Morgan-Kaufman 2000, ISBN 978-1558606470
- [3] Liskov, B. and Wing, J. 1993 FAMILY Values: a BEHAVIORAL NOTION of SUBTYPING. Technical Report. UMI Order Number: TR-562b., Massachusetts Institute of Technology.