Towards a Taxonomy of Distributed-Object Models

Arno Bakker Ihor Kuz Maarten van Steen
Vrije Universiteit Amsterdam

Keywords: distributed objects, object models, taxonomy, distributed computing

Abstract

Different ideas about object-orientation and distributed computing have resulted in a large number of distributed-object models. Use of the same terminology with different meanings makes these models hard to compare. What is currently missing is a framework for describing object models which can be used to compare and classify them. An attempt at defining such a framework is presented in this paper.

1 Introduction

Many researchers have recognized the potential benefits of using the object-oriented paradigm in the field of distributed computing. However, they have different opinions on how the paradigm should be used in this field. This lack of consensus is reflected in the large number of models for distributed objects that exist today. The variety of models and the differences in terminology used make it hard to compare them, and to assess their suitability for a particular application domain. What is needed is a framework for describing distributed-object models, enabling us to classify the different models and to make in-depth comparisons. Such a framework does not currently exist.

In this paper we present such a framework. Our framework is a result of studying a small, but representative set of object models and recording in which areas those models show notable differences. We focused on aspects which are important in distributed computing, such as distribution, communication, security and fault tolerance [7].

The framework presented here is a down-sized version of a more complete framework which is still under construction. For example, in this version we do not address security. A future version of the framework will be used to do a comprehensive comparison of distributed-object models and to create a taxonomy of such models.

The framework is explained in Section 2. Closing remarks, related work and the issues not yet addressed in this version can be found in Section 3.

2 The Framework

Our current framework describes a distributed-object model in four parts. The first part focuses on the basics of an object: the logical parts it consists of (state, method, interfaces, ...), if there are multiple kinds of objects, how objects are identified and referenced, etc. The other three parts focus on aspects which are important in a distributed context, notably *distribution of* (parts of) *the object*, *object interaction*, and if and how the occurrence of *partial system failures* is taken into account.

These last three aspects correspond to three requirements for programming languages for distributed applications identified by Bal et al. [1]. These requirements are: (1) the ability to assign different parts of a program to be run on different different processors, (2) the ability to communicate and synchronize, and (3) the ability to detect and recover from partial failure of the system.

By describing distribution, interaction, and fault tolerance we are able to form an idea as to what extent, and how an object model fulfills these requirements. More importantly, we will be able to compare the different approaches taken by the various object models.

Each of the four parts (object basics, interaction, distribution, fault tolerance) is further subdivided, creating a framework which is hierarchically structured. In this paper instead of describing all facets of the framework in detail we choose to concentrate on the rationale behind its structure and contents.

2.1 Objects

2.1.1 Overview

The first part of the framework describes the basics of the objects in an object model. It consists of three subparts: Object Characteristics and Structure, Naming and Interface. This division is shown in Figure 1.

1. Characteristics and Structure
   (a) Object Granularity
   (b) Kinds of Objects
   (c) Object Structure
   (d) Activity Model
   (e) Object Classes
   (f) Object Composition
2. Naming
3. Interface
   (a) Inheritance
   (b) Polymorphism
   (c) Typing

Figure 1: Object Basics

2.1.2 Object Characteristics and Structure

In some object models, objects typically encapsulate whole applications, while in others an object represents smaller things such as records or lists. In the extreme case, everything is considered an object, even integers. The typical size of an object in an object model is its *granularity*.

A common definition for an object is one in which an object consists of state and methods for accessing and modifying that state. The methods can be grouped together to form one or more interfaces. In a number of object models objects have a different *structure*, i.e. consist of different parts. For example, in the object model of the *Obliq* language [4] an object is perceived as a collection of named fields. In *Emerald* [3] the parts of an object as seen by the user are its name, its state, its methods, and optionally a process.

The *activity model* refers to how processes or threads are related to the objects in the object model. In an object model in which objects are *active*, threads are always bound to an object and an object always has a thread of control bound to it. In a *passive* model, processes and objects can exist and be created independently. Some object models use a hybrid model in which an object can be active or passive.

Many distributed-object models support the concept of a *class*. The definition of a what a class is differs between models, however. A class can be the definition of an object (as in C++) or it can be a factory for creating objects. Classes sometimes are objects themselves or are code repositories which are shared by the objects created from them. Most models supporting the class concept also define a number of rules for them, for example that a class should always be co-located with its instances. These are important distinguishing parameters because in this example the rule can have impact on an object's mobility [2].

2.1.3 Naming

Object models have different ways of identifying and referencing objects. Objects may be identified by a symbolic name or have some universally unique identifier (UUID). Object models may allow different symbolic names to be assigned to the same object or require that there is a one-to-one mapping between objects and names. In other models, object identifiers are not universally unique, but are only valid in a particular context.

2.1.4 Interface

The interfaces of an object define the possible interactions with that object. In some models objects have exactly one interface, while others permit multiple interfaces per object.

Most object models allow objects to inherit, i.e. receive properties or characteristics of other objects [16]. There are two kinds of inheritance: *interface inheritance* and *implementation inheritance*. Object models supporting the first type of inheritance allow an object's interface or interfaces to be defined in terms of interfaces of other objects. In models supporting the second type, we can use the implementation of existing objects to implement new objects. There are object models that allow inheritance from more than one object (*multiple inheritance*) and models that allow inheritance at run-time, i.e. *dynamic inheritance*.

Object models greatly differ in the way they support type. This makes it an area in which clear distinctions can be made between object models. Typing is important in a distributed system. Strongly typed interfaces can help ensure consistency of communication and also document some of the behaviour of the object [15] (see also Section 2.2.4).

2.2 Interaction

2.2.1 Overview

The second part of the framework provides a description of how objects interact in a distributed-object model. In particular, it concentrates on important aspects of method invocation, the different types of interaction (other than method invocations), the description of object behaviour, and the way concurrency and synchronization are handled. Distinguishing these aspects has shown to be important in order to classify the various models. This part of the framework is summarized in Figure 2.

1. Method Invocations
   (a) Static / Dynamic Invocation
   (b) Parameter Passing
   (c) Dynamic Binding
2. Other Types of Interactions
3. Behaviour
4. Concurrency
   (a) Concurrent Access to a Single Object
   (b) Serializing Concurrent Actions To Multiple Objects
5. Synchronization
   (a) Synchronizing Actions on a Single Object
   (b) Synchronizing Actions Involving Multiple Objects

Figure 2: Interaction

2.2.2 Method Invocations

In some object models method invocations are viewed as messages to other objects that have to be explicitly accepted; in others the programmer sees them as direct invocations on the object.

A method usually has a number of parameters and a return value. The collection of permitted datatypes for these parameters differs from model to model. For example, most systems do not allow pointers as parameters since they are valid only in the current address space. Object models also differ in the way parameters are passed (e.g., by value, by reference or by copy-restore) and the set of *parameter passing modes* (e.g., `in`, `out`, `inout`).

Parameter passing modes in particular are interesting in a distributed context. For example, in the object model used in the *Spring* operating system [13] the programmer can specify the "direction" in which a parameter is going. When a method is invoked on an object with another object as a `consume` parameter, the latter object is moved to the location of the called object. Using parameter passing modes such as `consume`, `produce`, `copy`, etc., the programmer can provide more information about the distribution behaviour of the application. This information can be used by the underlying system to do optimizations (at run- or compile-time).

Dynamic binding, i.e. determining at run-time which method should be executed is a feature of many object-oriented languages. Distributed-object models use this mechanism for various things. To see why it is important to discuss dynamic binding, consider the following example. Dynamic binding is an integral part of the *Distributed Smalltalk* model [2]. To determine which method code should be executed, the system has to traverse the class hierarchy to find the parent class that implements the method. If the class hierarchy is distributed over the system[1] the look-up process can become rather expensive since it might have to cross the network several times. Alternatively, the class hierarchy (or actually the classes on the path up in the hierarchy) has to be replicated on the local machine, which might become a substantial waste of resources when dealing with a large number of classes.

2.2.3 Other Types of Interactions

Method invocations are not the only way of interacting with objects. There are other types of interactions which some object models support and which others do not. *Events* are particularly important in system management applications where they are used to signal that a certain condition has arisen, for example, that a host has gone down. Especially object models designed to build such applications have extensive support for events. Other models support *interprocess* (or *interobject*) *byte streams*, which are useful when writing distributed multimedia applications.

2.2.4 Behaviour

Most object models allow the programmer to specify the *behaviour* of the object. Often, it can be specified only as comments to the definition of the object's methods, but in a number of models it is a formal part of the object definition. Making behaviour an explicit part of an object definition is important.

To illustrate, consider a producer and a consumer using a shared buffer object to communicate. Using some formal notation we can specify the behaviour of the buffer object, i.e. a producer can only add to the buffer when it is not full and a consumer can only remove items when it is non empty. This definition of the behaviour of the object can be used by the compiler and run-time system to control access to the object. When the buffer is full it will not allow the producer to call the `add` method on the buffer.

[1] Classes are also objects in Smalltalk and can therefore be distributed.

The programmer is therefore relieved from including explicit access control code in the bodies of the buffer methods [17].

By making object behaviour explicit, the system can make decisions about when to execute a method and when not to. This does not only make programming easier, it also creates safer communications because methods are never executed when they are not supposed to be.

Another reason why making behaviour explicit is important, is that formally specified behaviour of an object is documentation, making the application in which the object is used easier to understand.

2.2.5 Concurrency

Concurrency and object-oriented programming do not go well together: the mechanisms for handling concurrency interfere with the object-oriented features of the object-oriented programming language [12]. This makes it an interesting part of the framework because it shows us the different approaches and solutions the distributed-object model designers have taken, and in what way these approaches and solutions affect the object-oriented-Ness of the object model. These different approaches and solutions are again a discriminating factor.

We divide the discussion on concurrency in our framework into two parts: concurrent access to a single object and concurrent operations that involve more than one object. Papathomas [12] designed a classification scheme for concurrent object-oriented programming languages based on the way they handle concurrency. We use this scheme to classify distributed-object models.

Orthogonal / Non-Orthogonal An object model is *orthogonal* if objects in that model have no special support for concurrent operations. A common property of *non-orthogonal* object models is that an object's state is protected from concurrent execution of the object's methods.

Uniform / Non-Uniform A non-orthogonal object model is *non-uniform* if it supports two kinds of objects: Objects of which the execution of methods is *serialized* and objects whose method execution is not.

Integrated / Non-Integrated A uniform object model is *integrated* if threads of control are always associated with objects and are created only as a result of the creation of objects.

This division will need to be further refined. For example, there are severals ways to serialize the execution of methods. Some models simply disallow concurrent execution of methods, while others support transactional mechanisms. Such differences will be incorporated into our framework as well.

2.3 Distribution

The third part of the framework focuses on distribution of objects or parts of objects in the distributed-object model. The way distribution is handled is probably the most important aspect of a distributed-object model. Distribution greatly affects performance and availability. The particular approach an object model has taken can also tell us something about its scalability.

The third part of the framework is summarized in Figure 3.

1. Unit of Distribution (What)
2. Act of Distribution (Who & When)
   (a) Mobility
3. Expression of Distribution (How and Where)
   (a) Level/grain of Location Transparency

Figure 3: Distribution

2.3.1 Unit of Distribution (What)

The *unit of distribution* refers to what is being distributed in a particular object model. In most object models one distributes the state of the object. If the state is distributed as a whole, the state is the unit of distribution. If it is distributed in parts, the unit of distribution is a part. In other object models one distributes objects. Objects can also be distributed in whole or in parts. Note that distributing an object as a whole is not the same as distributing its state. For example, in *Clouds* [5] one distributes objects. The state is brought to the object's location when needed (using a Distributed Shared Memory system).

An example of a model in which parts of objects are distributed is the *Fragmented Objects* model [8]. In that model an object consists of multiple fragments which are individually distributed over the system. In this model the unit of distribution is the *object fragment*.

Objects and state may be replicated, again in whole or in part, in some models. Those models often also support *fragmented replicas* (multiple fragments constitute one "full" replica) and *replicated* *fragments*. In those cases the unit of distribution is a fragment.

2.3.2 Act of Distribution (Who & When)

The assignment of location to a unit of distribution (i.e. *distributing* that unit) can take place at several moments in time. It can be done at compile-time, at startup-time or during run-time. The assignment can furthermore be permanent, i.e. one which cannot be changed afterwards, or one which is mutable. Assigning units to a particular location or changing the location of a unit is done either by the system or by the programmer.

We can classify object models according to when and by whom location assignment is done and whether or not that assignment can change. The models in which it can be changed can be further classified on the basis of two transparencies which they may or may not provide. If an object (or other unit of distribution) can change location without other objects noticing it, the object model is said to be *relocation transparent*. When the object or unit itself does not notice the change of location, the object model is said to be *migration transparent* [6].

2.3.3 Expression of Distribution (How & Where)

Once we know what the unit of distribution is and by whom and when distribution takes place, we can look at how and where the distribution is expressed. Object models take different approaches here.

Being able to express the distribution of units over the distributed system implies that there is some way to express location in the object model. At first glance this might seem to conflict with the desire to have a *location-transparent* distributed-object model, but it does not. For example, the programmer may be able to program the application without taking location into account, because the distribution of the objects in the distributed-object model is specified in a separate *distribution map* which is read by the system when the application is started.

A distinction can be made between the level of location transparency a model provides to the programmer. We distinguish four levels or *grains* of location transparency:

- **totally transparent**
- **partially transparent** (e.g., there is a distinction between local and remote objects, but the remote location is not known)
- **logically transparent** (e.g., object X is on virtual node N, co-locate object Y with X)
- **not transparent** (e.g., locations of resources are explicitly used)

This, in effect gives us the answer to the question how location is specified, if visible at the programming level. A second question we wish to answer is how we can obtain the location of a particular unit of distribution, if it can be explicitly manipulated. This will help us discriminate between the various object models.

2.4 Failure and Fault Tolerance

The fourth and final part of this version of the framework is focused on how the occurrence of system failures is incorporated into the object model. Examples of system failures are processor crashes, network failures, temporary lack of resources, etc. This part is summarized in Figure 4.

1. Failure Transparency
2. Failure Semantics
3. Failure Detection / Notification
4. Mechanisms for Working in the Presence of Failures
   (a) Atomic Transactions
   (b) Persistent Objects
   (c) Replication

Figure 4: Fault Tolerance

2.4.1 Failure Semantics, Detection and Notification

Some object models assume that the system that implements the model can handle all faults in a way transparent to the programmer. In other models, failures are explicitly taken into account in the model. In the latter case we are interested in what an object model guarantees when a failure occurs. For example, the OMG (CORBA) object model [11] guarantees *exactly once* semantics for successful method invocations and *at-most once* semantics if an error occurred and the call raised an exception. A distributed-object model could also make guarantees about, for example, the availability of objects.

Another important and distinctive aspect is how failures are detected by or signaled to the programmer. As mentioned above, the OMG object model uses exceptions, but other object models support different mechanisms.

Object models often provide mechanisms which the programmer can use to increase the reliability and availability of the distributed applications. The three most common mechanisms are: *Atomic Transactions*, *Persistent Objects* and *Replication*.

3 Closing Remarks

We presented our framework for distributed-object models. Developing a universal framework for comparing and classifying object models is not an easy exercise. Small aspects, specific to an object model (such as dynamic binding and parameter passing modes) can have great impact on their performance in a distributed environment. Recognizing all these aspects will require a lot of time and effort. Until now we have tried to discover those aspects empirically. In the future we will take a more theoretical, top-down approach.

The version of our framework presented here is not complete. First of all, a number of important aspects of distributed-object models have not yet been addressed, such as security, object management (how are objects defined, created, destroyed, etc.), programming language support (from what language or languages can we access and implement objects), and object versioning (an important aspect in a wide-area context).

Furthermore, this version contains a number of classifications which need to be worked on. An example of this is the classification of object models based on when and by whom distribution of (parts of) objects is specified (see Section 2.3.2). Drawing a graph of the possible combinations proves to be non-trivial, because there are subtle but distinct differences between combinations that might seem equivalent at first glance.

Related Work

Earlier work in this area was done by Bal et al. [1]. They developed a similar type of framework for classifying programming languages for distributed computer systems. Some interesting work on a meta-model for describing distributed-object models using a restricted set of concepts was done by Manola [10]. He was furthermore involved in developing the ANSI *Object Models Features Matrix* [9], used to compare object models with the intention of finding ways to make them interoperate.

A joint XOpen/NM taskforce used a common language to compare object models [14]. This is in particular interesting because different uses of the same terminology is one of the things that make distributed-object models hard to compare.

References

[1] BAL, H., STEINER, J., AND TANENBAUM, A. Programming Languages for Distributed Computing Systems. *ACM Computing Surveys 21*, 3 (Sept. 1989), 261–322.

[2] BENNETT, J. K. The Design and Implementation of Distributed Smalltalk. In *Proceedings OOPSLA* (Orlando, Fl., Oct. 1987), ACM, pp. 318–330.

[3] BLACK, A., HUTCHINSON, N., JUL, E., AND LEVY, H. Object Structure in the Emerald System. In *OOPSLA '86 Conference Proceedings* (Sept. 1986), N. Meyrowitz, Ed., ACM. published as SIGPLAN Notices 21(11), Nov. 1986.

[4] CARDELLI, L. Obliq, A Language with Distributed Scope. Research report 122, Systems Research Center, Digital Equipment Corporation., June 1994.

[5] DASGUPTA, P., ANANTHANARAYANAN, R., MENON, S., MOHINDRA, A., AND CHEN, R. Distributed Programming with Objects and Threads in the Clouds System. Tech. Rep. GIT-CC-91/26, Georgia Institute of Technology, Atlanta, GA, 1991.

[6] ISO. Open Distributed Processing - Reference Model - Part 3: Architecture. International Standard / ITU-T Recommendation 10746-3 / X.903, ISO/IEC, 1995.

[7] KUZ, I. A Framework for Describing Distributed Object Models. Master's thesis, Department of Mathematics and Computer Science, Vrije Universiteit Amsterdam, Sept. 1996.

[8] MAKPANGOU, GOURHANT, LE NARZUL, AND SHAPIRO. Structuring distributed applications as fragmented objects. Rapport de Recherche 1404, INRIA, Jan. 1991.

[9] MANOLA, F. *X3H7 Object Features Matrix*. ANSI, Feb. 1995. URL: http://info.gte.com/ftp/doc/activities/x3h7.html.

[10] MANOLA, F., AND HEILER, S. A "RISC" Object Model for Object Systems Interoperation: Concepts and Applications. Tech. Rep. TR-0231-08-93-165, GTE Laboratories Inc., Aug. 1993.

[11] OBJECT MANAGEMENT GROUP. *The Common Object Request Broker: Architecture and Specification. Revision 2.0*. Object Management Group Publications, July 1995.

[12] PAPATHOMAS, M. Concurrency Issues in Object-Oriented Programming Languages. In *Object Oriented Development*, D. Tsichritzis, Ed. Centre Universitaire d'Informatique, University of Geneva, July 1989, pp. 207–245.

[13] RADIA, S., HAMILTON, G., KESSLER, P., AND POWELL, M. The Spring Object Model. In *Proceedings of the USENIX Conference on Object-Oriented Technologies (COOTS)* (Monterey, California., June 1995).

[14] RUTT, T. Comparison of the OSI management, OMG and Internet management Object Models. Report, Joint XOpen/NM Forum Inter-Domain Management (JIDM) Taskforce, Mar. 1994.

[15] SCHMIDT, D., AND VINOSKI, S. Object Interconnections: Introduction to Distributed Object Computing (Column 1–8). *SIGS C++ Report* (1995–1996).

[16] TAIVALSAARI, A. On the Notion of Inheritance. *ACM Computing Surveys 28*, 3 (Sept. 1996), 438–479.

[17] VAN DEN BOS, J., AND LAFFA, C. PROCOL: A concurrent object-oriented language with protocols, delegation and constraints. In *Acta Informatica* (Mar. 1991), no. 28, pp. 511–538.