Towards the Implementation of a Uniform Object Model

F. J. Hauck

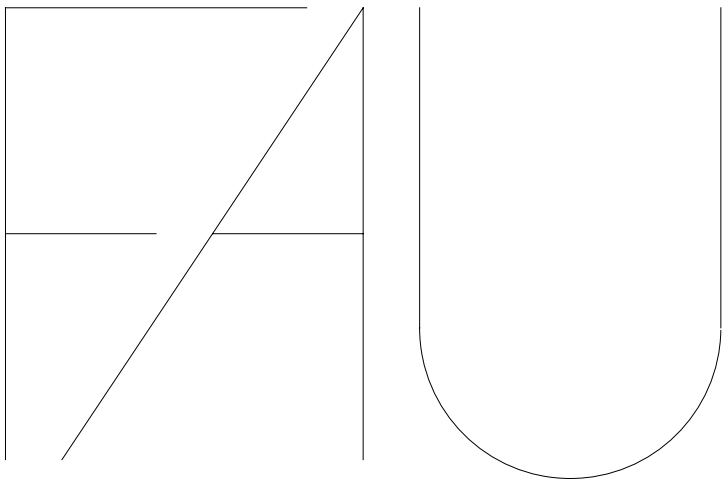
Oktober 1992 TR-I4-5-92

Technical Report

Computer
Science Department

Operating Systems — IMMD IV

Friedrich-Alexander-University
Erlangen-Nürnberg, Germany



This paper was also published as:

F. J. Hauck: “Towards the implementation of a uniform object model”; *Parallel Comp. Architectures: Theory, Hardware, Software, and Appl.* – SFB Colloquium SFB 182 and SFB 342; A. Bode, H. Wedekind [Eds.], (Munich, Oct. 8-9, 1992); Lecture Notes in Comp. Sci.; Springer, Berlin; to appear 1993

Towards the Implementation of a Uniform Object Model

Franz J. Hauck
hauck@immd4.informatik.uni-erlangen.de

University of Erlangen-Nürnberg
IMMD 4, Martensstraße 1
D-W 8520 Erlangen, Germany

Abstract. In most cases the programming models of distributed object-oriented systems have a two-stage or even a three-stage object model with different kinds of objects for values, distributable and non-distributable objects. This allows an efficient implementation for at least non-distributed objects, because traditional compilers can be used. This paper discusses some aspects of the implementation of a uniform object model that does not know of any distinction between distributable and non-distributable objects and allows an independent application description of the distribution of objects.

Instead of integrating distribution later into a non-distributed language our method takes the opposite approach. For the time being a distributed object model is implemented in a general, distributed, and, of course, inefficient way. With some additional information derived by user supplied declarations or automatically by a compiler the general implementation becomes more and more optimized. With assertions like *immutable* or *constantly* and *initially bound* objects the implementation can be optimized such that the non-distributed case is not worse than in traditional object-oriented languages.

1 Introduction

Usually distributed object-oriented systems are derived from non-distributed object-oriented languages, i.e. *Argus* from *CLU* [1], *Clouds* from *C++* – called *DC++* [2] – and from *Eiffel* – called *Distributed Eiffel* [3]. These approaches introduce a new kind of object which is distributable, all other objects of the base language used being non-distributable. The result is a two-stage object model with different semantics for each stage. The parameter-passing semantics depend on the kind of object passed, i.e. *call-by-copy* semantics (corresponds to *call-by-value*) for non-distributed objects and *call-by-reference* for distributed objects. Often, values form their own kind of objects resulting in a three-stage object model, e.g. in *DC++*.

This is an advantage because those object models are easy and quite efficient to implement, as the compiler of the base language can be used for all non-distributed kinds of objects and produces efficient code as well. Distributed objects are implemented by stub objects, which are non-distributed objects. These are passed by *call-by-copy* semantics which is in fact *call-by-reference* semantics for the distributed object.

The different semantics for the different kinds of objects are the great disadvantage of these approaches. Often the kinds of objects are fixed by their respective classes, e. g. in *Argus*. There is a need for two classes, one for distributed and one for non-distributed objects. This impairs the reuse of code and it is not easy to see which semantics apply for a certain method call.

Another problem remains. The distribution of the objects is explicitly described in the program, thereby also affecting reusability. Yet, distribution is orthogonal to the programming of objects, with the exception of different failure models and different time behavior. Thus, we require a distribution description separate from the program description. To achieve this we need a uniform object model, as that is the only way to abstract completely from the distribution of objects.

Emerald is one approach with a uniform object model [4], but it has no separate description of distribution. Distribution is described inside an *Emerald* program with explicit statements.

This paper shows some aspects of implementations for uniform object models with a separate distribution description. Instead of extending a non-distributed language we choose the opposite method. In chapter 2 we introduce an object model for arbitrarily distributed objects. There is a separate description language for distribution. Chapter 3 presents a simple distributed implementation which is not very efficient. The efficiency is improved by suitable optimizations. Different aspects of the possible optimizations are shown. Chapter 4 contains a conclusion and ends with an outlook on future work.

These thoughts are originated from within the PM project of the IMMD 4, University of Erlangen-Nürnberg. The distributed object model is used there to model distributed operating systems.

2 A Distributed Object Model

This chapter introduces a distributed uniform object model for programming distributed applications or parts of operating systems. It defines *objects, methods, classes, types,* and *requests*.

2.1 Objects and Classes

Objects are the smallest distributable entities. They have behavior and state. The behavior is expressed with *methods*; operations which can be invoked by clients of an object. The state of an object *A* consists of references to other objects. Object *A* can be a client of all these objects and invoke any of their methods. References are bound to variables which can be accessed by the objects' methods only[1]. A graphical representation is shown in fig. 3.1. The rectangles represent objects, boxes represent variables and arrows represent references from variables to objects.

1. The access from a client to a variable can be allowed by using some special methods which can be derived automatically by a compiler.

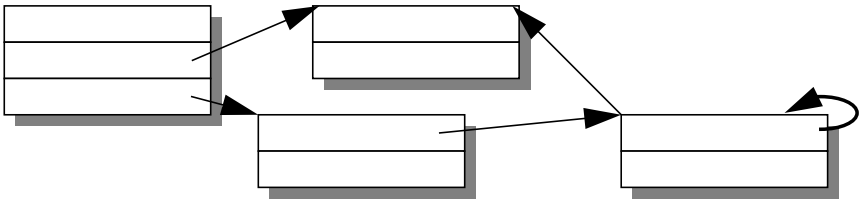


Fig. 2.1 A graphical representation of objects, variables and references

With the invocation of a method an independent virtual thread of activity is created executing the code of the method. After having done all computing and sending back the results to the client the thread of activity is destroyed. The thread of activity is called a *request*. During the execution of a method invocation all requests are seen as part of the state of the object which defines the method. The general parameter-passing semantics is *call-by-reference*. The *request* gets some references as parameters from the client and sends back some references as a result.

Classes are defined as sets of objects with the same behavior and identical structure of state. They constitute equivalence classes according to an equal relation of state and behavior. This extensional definition does not directly correspond to the definition of the term *class* in most programming languages. In a language the term *class* is almost intensionally defined. *Classes* are directly described and their objects are created from this description.

Languages like *Self* do not know classes as a concept of the language itself [5], but it is possible to identify the extensionally defined classes in these languages. For this paper it does not matter whether classes are concepts of a language or not. We refer in the next sections to the extensionally defined term.

2.2 Types

Types are equivalence classes of *classes* with the same abstract behavior. In most languages, like *C++* and *Eiffel*, the notion of type and class is the same, [6] and [7]. These languages do not know any mechanisms to declare two classes to be of the same type; not even when these classes are identically declared.

Type conformance is a subset relationship between types. This corresponds to the definition of Palsberg and Schwartzbach, [8] and [9]. Type conformance is also possible in *C++* and *Eiffel*, but only by using inheritance between classes. Inheriting classes are type-conforming to their base classes.

We propose that the notion of type is to be represented by some language concepts. A class[2] has to decide to which type it belongs. This results in an explicit separation of types and classes. It is necessary to make an explicit decision as to which type is supported by which class, because the compiler cannot decide which abstract behaviors are conforming.

2. Or an object (depending on if classes are defined as a language concept).

2.3 Distribution

As mentioned above objects are the smallest entities for distribution. References can point to a distributed or to a local object. The method invocation on distributed objects is implemented by an RPC style communication. The semantics of a program is therefore independent of the possible distribution of the objects with the exception of different failure models and possible run-time differences. Thus, distribution is described in a separate programming model based on relative properties of distribution between the objects, called *collocations* and *dislocations*. Further information can be found in [10] and [11].

Objects can migrate arbitrarily in the system. This means that they can change their physical location at run-time, e. g. for an optimization of the total run-time of a program. The migration of objects is controlled by so called *cooperations*. These are part of the programming model of the distribution system and can be described there. Cooperations may dynamically create and destroy relationships of collocation or dislocation between objects and indirectly cause object migrations – see again [11].

3 Aspects of Implementation

One simple approach towards implementation of the described object model is exemplified in the following:

- Objects are implemented as continuous parts of memory containing all variables and methods.
- Each object has a unique identification called *OID* (object identifier), which has a system-wide uniqueness.
- References to other objects are stored in variables. These references are represented as *OID*s.
- Method invocations are handled by a run-time system which determines the actual position of an object by its *OID*.

An implementation like this is possible but certainly not efficient. The identifiers need a size of 96 to 128 bits to support a world wide distributed system. A variable needs 12 to 16 bytes of memory to store a reference. The search for objects will be very expensive, as objects may be located anywhere. Searching is also done for often used objects like integers or booleans.

It is immediately obvious that smaller identifiers would increase this efficiency. They would need less memory and allow faster searches for objects. The best case of an implementation of an OID is a pointer. There is no need for a search if the OID is a pointer to the memory space of an object. But in this case the object is not migratable without restrictions, as it must be located in the same address space.

As mentioned already an efficient implementation is possible with a multi-stage object model. Non-distributed objects are addressed by pointers and distributed objects are addressed by stub objects containing a large OID. These stub objects are only part of the implementation. This kind of implementation should be rejected be-

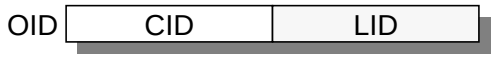


Fig. 3.1 Splitting the OID

cause it cannot support a uniform object model. The following sections will show some aspects of optimizing the addressing of objects under certain circumstances.

Generally a more efficient implementation can be achieved by using smaller representations of the OIDs. The identifiers are to be stored in variables. But a variable is only bound to a subset of all objects during its lifetime. Thus, only the identifiers of those objects are to be stored in that variable. One approach could be to assign each variable a specific and different set of identifiers for all objects possibly bound to that variable. This is disadvantageous because assigning one variable to another is a more complex operation; the identifier of one object stored in one variable is different from the identifier of the same object stored in another variable. Trying to avoid all conversions of identifiers generally leads to a representation of identifiers which is as large as the unique OID.

Another drawback to using different identifiers for the same object is complex identity comparisons. They are needed as user level statements or as run-time statements for all kinds of aliasing-detection.

3.1 Constant Collocation

When the user specifies the distribution of an application in the distribution system there are objects which are constantly collocated; meaning their collocation relationship does not change during run-time. Using this information groups of objects can be derived which are all in a transitive collocation relationship. These groups are called *clusters* [11].

A simple way of using smaller identifiers is to split the OID into a Cluster ID (*CID*) and an identification local to a cluster (*LID*), see fig. 3.1. Inside a cluster local objects can be addressed by the LID only. The same objects are addressed from outside using the full OID. The CID allows a more efficient address resolution because there are fewer clusters than objects in the system.

Identity comparisons are simpler, because LIDs are part of the OIDs. References from inside a cluster assigned to variables outside are converted by simply adding the CID part to the LID.

The LID itself can be implemented by a pointer into the continuous memory space of a cluster. This pointer can even be an offset for some segmentation mechanism used to implement a position-independent addressing-scheme inside a cluster.

This kind of optimization can only be used for constantly collocated objects. Mobile objects form a cluster of their own and can only be addressed by a full OID. Variables inside a cluster which can be bound to objects of their own cluster and to objects outside of that cluster cannot use the efficient local addressing-scheme. They have to use the full OID address. The distinction between LID and a full OID would cause more overhead than the optimization of the local addressing would increase efficiency.

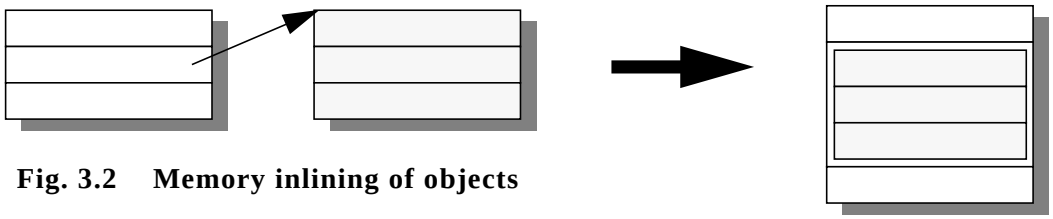


Fig. 3.2 Memory inlining of objects

3.2 Immutable Objects

Immutable objects only have constant bindings to other immutable objects. This implies that all their requests do not interfere. The result of a request depends only on its parameters. A request cannot store any information in an immutable object.

Immutable objects are easily shared. Shared immutable objects can be copied and each client can have its own copy without noticing it. The semantics of a program stay the same with shared or with copied immutable objects. Only correct identity comparisons between the copies must be realized. Immutable objects are candidates for a per cluster copy. Thus, they can always be addressed by a small LID. In our distribution system immutable objects can be collocated to two dislocated objects. This conflict is not solvable in the case of a mutable object.

This kind of optimization is very suitable for all kinds of value objects. Previously, in *CLU* and *Emerald*, values were modelled by immutable objects, [12] and [4]. There is no need to share integer values all over the system, as there can be copies for at least each cluster.

As the identifier of an immutable object has enough information about the object itself, there is no need to implement the object in the memory space in all cases. For further optimizations special identifiers, called *VID*s (value IDs), can be used for binding immutable objects. These identifiers do not address certain objects (in the implementation), they stand for them. VIDs can be made unique within the system and thus there is no need for a conversion when assigning a reference from one cluster to another. The representation of a VID can be very special, e.g. for integers it may be a bit-pattern. This bit-pattern allows for direct integer arithmetic operations.

3.3 Constantly and Initially Bound Variables

A variable is initially bound when a reference is bound at creation time. Constantly and initially bound variables are bound at the object creation. This binding cannot be changed. Such a situation can be used for an optimization called memory inlining. The variable does not contain an identifier of the bound object but the object itself (fig. 3.2). In the distributed object model this optimization is only possible with an additional constantly collocated relationship between the objects.

In *C++* all non-pointer variables are initially and constantly bound references. In *Eiffel* these kinds of objects are called *expanded objects*. In both languages the problem lies with the different kinds of objects having different parameter-passing semantics for non-pointers or expanded objects respectively; *call-by-copy* instead of *call-by-reference*. In our approach the memory inlining is a matter of an implementation and is independent of all semantics of the object model. There is no need to

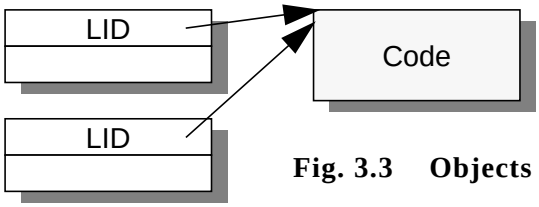


Fig. 3.3 Objects of one class and their code object

declare initially and constantly bound references at all, because they may also be derived by a compiler.

Inside a cluster inlined objects can be addressed by a LID. Outside of a cluster the usual OID is used. Initially and constantly bound, immutable objects can be treated like plain immutable objects; they can be copied freely. In this special case the new properties can be used to place the VIDs directly in the code and not necessarily in the variables. This procedure is the same as that done by compilers which implement constants integrated into machine instructions and which are not loaded from memory. Variables initially and constantly bound to immutable objects may not be represented in the memory space of the object when this property is known to all clients. This is the case, at least, for the code of the object's methods.

3.4 Classes

Classes are defined extensionally in chapter 2. Thus, they do not need a representation at run-time. Each object has its own methods, because only its own methods can access its local variables. In an implementation it does not make sense to have all objects with their own method code. The grouping of equal methods is recommended for efficient memory usage. The access of the method code to the variables of an object is implemented by segmentation mechanisms like index registers or hidden parameters.

The method code of the objects belonging to one class can be implemented by a special object visible to the implementation only. Each object of the class has a reference to that code object. It represents the class (fig. 3.3).

The code of a class is normally immutable. Thus, the code can easily be copied in each address space and be addressed locally. This is a must for executable code. The code object has a strongly collocated relationship to all the objects using the code. Because it is immutable, dislocation conflicts between these objects can be solved by different copies of the code.

The reference from each object to the code object is initially and constantly bound. Thus, there is no need for the reference when all clients know which class an object belongs to. Then the code of a method can be directly addressed by a client. The reference to the code object can be omitted when all clients know the class of the object.

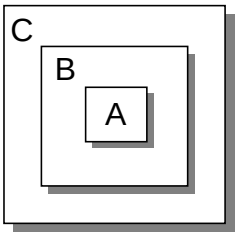


Fig. 3.4 Class *C* inheriting from *B* and *A*

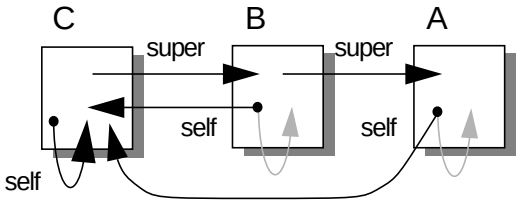


Fig. 3.5 Bindings between differential classes

3.5 Inheritance

Inheritance is a concept of composing classes. As types are separated from classes we need some kind of inheritance for both of them. Inheritance for types may be a concept of creating conforming types. This does not lie within the scope of this paper.

This section deals with class-based inheritance, but not subtyping. Inheritance between classes is used for refining and reusing the behavior of some base classes into subclasses. In most languages the subclass is seen as one unit containing the inherited properties of the base classes. See fig. 3.4 for a graphical representation of class *C* inheriting from class *B*, which inherits from class *A*.

We propose another view to inheritance. We identify subclasses as differential classes which describe all the changes of and refer to the base classes. Thus, the subclasses do not include properties of the base classes, they only refer to them. In fig. 3.5 the same situation seen in fig. 3.4 is presented, but the boxes represent differential classes. The objects of the classes in an inheritance hierarchy refer to each other by a reference. These references correspond to keywords like *self* and *super*[3]. *Self* refers to the last subclass which inherits no further. *Super* refers to the base class. These references are represented in fig. 3.5 by arrows. Grey arrows are *self* references which are rebound by the inheritance mechanism.

Using references of the objects of inheriting classes to model inheritance has its advantages for implementation. There is no need for a special implementation mechanism to deal with inherited classes. The implementation treats differential classes as normal classes. The creation of objects of these classes also causes the creation of objects of the base classes with certain bindings for their *self* and *super* variables. The bindings are initially and constantly bound and all objects are collocated, i.e. the bindings are addressed by LIDs and memory-inlining can be done. This is quite the same implementation as in traditional languages, e.g. *C++*.

In our model there is no need to collocate the objects. A dislocation leads to a distributed inheritance mechanism, which allows the distribution of some parts of an object. This is possible because there is not only one object of an inheriting class, but many objects of many base or differential classes respectively.

3. We adopted these names from the *Smalltalk* terminology [13].

3.6 Type Checking

Like classes types do not necessarily need a run-time representation. Types are introduced to get programs type checked, i.e. all method calls are legal and the well-known *Smalltalk* error message, “message not understood”, will not occur.

Compilers can do most of the type checking at compile-time, but there are several circumstances which need a run-time type checking, e.g. for *type-case* and *type-of* statements and for binding references to a variable with larger type. In a distributed environment there may be objects with types which are not even known at compile-time. To initialize a distributed application the run-time system needs to type check all references to objects with these unknown types. For all these cases of run-time type checking a representation of types at run-time is necessary.

Therefor we introduce one more set of auxiliary objects for types. All objects have a reference to a type-object. This is constantly and initially bound and immutable. Thus, it is subject to the above-mentioned optimizations.

The type objects have a method for type comparisons as in *Emerald* [4]. These methods and the unique identification of type-objects are used for run-time type checking.

4 Conclusion

We have shown that in some circumstances the implementation of a uniformly distributed object model can be optimized. This optimization is as efficient as non-distributed language implementations when the application is not distributed, because the implementation places all objects into one cluster and collocates them. Internal pointers are avoided by memory-inlining when objects are initially and constantly bound. Immutable objects allow the inlining of object references in the method code. This corresponds to compiled programs in a non-distributed language.

A distributed application is obviously not as efficient as a non-distributed application when we only look at the addressing of objects and memory usage. The above-mentioned optimizations are a step towards an optimal implementation.

To validate the practical use of the optimizations the construction of a prototypical compiler is planned. This compiler will output, with specific distribution descriptions, the possible implementation optimizations of specific example applications.

5 References

1. B. Liskov, “The Argus language and system”; In: *Distributed systems, methods, and tools for specification*; M. Paul, H. J. Siegert [Eds.], Lecture Notes in Comp. Sci. 190; Springer, Berlin; 1985 – pp. 343-430
2. P. Dasgupta, R. Anathanarayanan, S. Menon, A. Mhindra, R. Chen: *Distributed programming with objects and threads in the Clouds system*; Tech. Report GIT-CC 91/26, Georgia Inst. of Techn., Atlanta GA; 1991

3. L. Gunaseelan, Richard J. Jr. LeBlanc: *Distributed Eiffel: a language for programming multi-granular distributed objects on the Clouds operating system*; Georgia Inst. of Techn., Tech. Rep. GIT-CC-91/50; Atlanta GA, 1991
4. N.C. Hutchinson, R.K. Raj, A.P. Black, H.M. Levy, E. Jul: *The Emerald Programming Language*; Univ. of Washington, Seattle WA, Tech. Report 87-10-07; Oct. 1987, revised Aug. 1988
5. D. Ungar, R.B. Smith: "Self: The power of simplicity", In: *Proc. of the Conf. on Obj.-Oriented Progr. Sys., Lang., and Appl.*; N. Meyrowitz [Ed.], (Orlando FL, Oct. 4-8, 1987); SIGPLAN Notices 22(12); ACM, New York NY; Dec. 1987 – pp. 227-242
6. M.A. Ellis, B. Stroustrup: *The annotated C++ reference manual – ANSI base document*; Addison-Wesley, Reading MA, USA; 1990
7. B. Meyer: *Eiffel: the language*; Prentice Hall, New York NY; 1992
8. N. Oxhøj, J. Palsberg, M.I. Schwartzbach: "Making type inference practical"; In: *Proc. of the 6th Eur. Conf. on Obj.-Oriented Progr.*; O. Lehrmann Madsen [Ed.], (Utrecht, June 29-July 3, 1992); Lecture Notes in Comp. Sci. 615; Springer, Berlin; June 1992 – pp. 329-349
9. J. Palsberg, M.I. Schwartzbach: "A note on multiple inheritance and multiple subtyping", In: *Multiple inheritance and multiple subtyping* – Pos. Papers of the ECOOP '92 Workshop W1; M. Sakkinen [Ed.], (Utrecht, June 29,1992); Tech. Rep., Dep. of Comp. Sci. and Inf. Sys., Univ. of Jyväskylä, Finland; May 1992 – pp. 3-5
10. M. Fäustle: *Beschreibung der Verteilung in objektorientierten Systemen*; Diss., IMMD, Univ. Erlangen-Nürnberg; 1992
11. M. Fäustle: "An orthogonal optimization language for uniform object-oriented systems"; *Parallel Comp. Architectures: Theory, Hardware, Software, and Appl.* – SFB Colloquium SFB 182 and SFB 342; A. Bode, H. Wedekind [Eds.], (Munich, Oct. 8-9, 1992); Lecture Notes in Comp. Sci.; Springer, Berlin; to appear 1992
12. B. Liskov, J. Guttag: *Abstraction and Specification in Program Development*; MIT Press, Cambridge MA; 1986
13. A. Goldberg, D. Robson: *Smalltalk-80: the language and its implementation*; Addison-Wesley, Reading MA, USA; 1983