1

AN OBJECT MODEL FOR DISTRIBUTED AND CONCURRENT PROGRAMMING BASED ON DECOMPOSITION

Matthias Radestock, Susan Eisenbach

Department of Computing
Imperial College of Science, Technology and Medicine
180 Queen's Gate, London SW7 2BZ
E-mail: {mr3,se}@doc.ic.ac.uk

ABSTRACT

Many of today's object models represent parallelism and distribution inadequately because these models grew out of sequential object oriented languages. In this paper we develop a model that is directly derived from an abstract but very simple and compact definition of the object-oriented paradigm. It is shown how *inter-object and intra-object concurrency* arise naturally from this paradigm.

The *decomposition hierarchy* plays a central role in the model. It is the logical communication system topology. The position of an object in the hierarchy determines a *unique object identifier*. This enables us to refer to objects prior to their existence and to store object references in the persistent database. Clustering of objects based on the hierarchy allows the efficient implementation of persitent storage mechanisms.

The methods of an object and its instance variables are represented as subobjects in the decomposition hierarchy. Parameters to methods and local variables are in turn subobjects of the method-objects. This unified representation simplifies the model and has a significant impact on the *message invocation process*. We develop a scheme for our model that allows recursive and concurrent invocations of methods.

1 INTRODUCTION

The main motivation for devising concurrent object models is the advantages that are achieved by unifying the abstraction for processor and memory (cf. [KL93]). The *object-oriented paradigm* provides a means for specifying composite data structures and associating operations with those structures. It thus provides a data-oriented view of functionality. All aspects of the *abstract data type paradigm*[Par72] are incorporated while at the same the emphasis is put on the functional rather than the structural part of the definition. Consequently the paradigm includes a variety of concepts and mechanisms related to the functional part, e.g. inheritance, defaults, polymorphism. By contrast, the *process paradigm*[Mil89] provides a control-oriented view of functionality. Decomposition of functionality is achieved by decomposition of control – a process describes the interaction between its subprocesses, i.e. their communication, synchronisation, creation and destruction. Thus the overall functionality of a process is determined by the functionality of its subprocesses and the specified interactions. Hence, the object-oriented paradigm is a model for data abstraction, whereas processes are models for control abstraction. The concurrent object-oriented paradigm combines the data-oriented view of functionality in the traditional object-oriented paradigm with the control-oriented view of functionality in the process paradigm. This is achieved by viewing objects as processes. We can define objects as follows:

Definition 1 *Something is an object if it:*

- *has a state, and*
- *can receive messages from objects, and*
- *may change its state due to the arrival of a message, and*
- *can send messages to other objects due to the arrival of a message*

This definition is a combination of static (state) and dynamic (messages) aspects and the interaction between them. Concurrency deals with the dynamic aspects. Control flow in the concurrent object-oriented paradigm is modelled as *message passing*. The equivalent in the traditional object-oriented paradigm is the notion of a procedure call.

From the above definition we can derive the notion of an object *boundary* and *encapsulation*. The system state is divided by object boundaries which encapsulate the state of individual objects. Normally objects only have access to this part of the system state. Access to other objects' state information is only possible via message transfer and it is up to the receiver whether and how the state information is made accessible. Hence the above definition allows objects to exist and act concurrently – each operating on its own state and communicating by exchanging messages. Two notions of concurrency emerge:

Intra-object concurrency The concurrent reception and interpretation of messages within one particular object.

Inter-object concurrency The interpretation of arriving messages by several objects concurrently.

Thus both intra-object and inter-object concurrency arise naturally from the definition of the term 'object' and can be viewed as intrinsic features of the object-oriented paradigm and systems based on it (cf. [W$^+$91]). Concurrency, parallelism and distribution should thus be easy to represent in concurrent object-oriented models. Yet many of today's object models suffer from an inadequate representation of these aspects that results in considerable conceptual difficulties (cf. [YM93, Mes93, KL93]). The main reason for this is that the *models* were derived from sequential object-oriented languages (and later concurrent extensions of such languages). Only in few cases the *languages* were derived from the models and even then most models had a less purist view of the object-oriented paradigm than the one outlined above. In this paper we derive an object model *directly* from the above paradigm. Our model deals with concurrency in an equally natural way as the paradigm itself.

2 THE ROLE OF THE DECOMPOSITION HIERARCHY

In our model we exploit an important property of the object-oriented design method[CY91, R$^+$91]: objects are incorporated in a *decomposition hierarchy*. This hierarchy is a tree with the root node representing the entire system. It plays a central role in our model – it is used to obtain object references and serves as the logical communication infrastructure. Figure 1 describes the decomposition relation in terms of an axiom system. The relation is defined over *object identifiers*(OID), so names of all graph nodes must be unique. The second axiom ensures that any node has at most one predecessor. The top node in the hierarchy is called the `Universe`. It doesn't have a predecessor and is unique, ie. it's the only node which doesn't have a predecessor (third axiom). There is an implicit $\forall$-quantification in all axioms. The fourth axiom thus guarantees that the graph is fully connected and includes all elements of the domain.

Sorts

OID

Constants

`Universe` : OID

Predicates

contains(OID, OID)

Axioms

$$\mathsf{contains}(x, y) \longrightarrow x \neq y \tag{1.2}$$

$$\mathsf{contains}(x_1, y) \wedge \mathsf{contains}(x_2, y) \longrightarrow x_1 = x_2 \tag{1.3}$$

$$\neg\mathsf{contains}(x, \texttt{Universe}) \tag{1.4}$$

$$\exists x.\mathsf{contains}(x, y) \vee y = \texttt{Universe} \tag{1.5}$$

Figure 1 The contains Relation

2.1 Object References and the Decomposition Hierarchy

In order to communicate with each other, objects must have references to other objects (the receiver of a message in particular). Since potentially any object in the system can be referred to by any other object, or even by itself, we need some kind of system-wide *object identity.* One way of identifying objects is to assign an identifier to them at the time they are created. As this identifier has to be unique the creation of it has to take place centrally. For a distributed system this means a huge amount of communication taking place at a central location. Furthermore there is no way of referring to objects before they come into existence. This imposes major constraints on the design of any implementation based on such a model.

In our model we exploit the structure of the decomposition hierarchy in order to derive object identifiers from it. That the hierarchy can be used in such a way has been recognised in recent research (cf. [PB94]). However, we first need to eliminate the 'global identifiers' property from the relation in Figure 1. How this can be done is shown in Figure 2. Provided that unique names are assigned to the immediate successors of a graph node we can devise a function

Sorts

$\text{SEQ}_{\text{NAME}}, \text{NAME}$

Constants

$\texttt{Universe} : \text{NAME}$

Predicates

$\mathsf{path}(\text{SEQ}_{\text{NAME}})$

Axioms

$$\mathsf{path}(s < x >) \longrightarrow \mathsf{path}(s) \tag{1.2}$$

$$\mathsf{path}(< x > s) \longrightarrow x = \texttt{Universe} \tag{1.3}$$

Figure 2 Using *Paths* as Identifiers

which returns a unique name for any node in the graph. This could be done by simply taking the path from the root of the tree to the node. We cannot describe this representation in terms of a relation. What we have to do instead is to specify an abstract data structure for this type of graph. Then we can show that the set of unique identifiers that are returned by the function, and the relation between these identifiers as obtained from the graph, obey the axioms defined in Figure 1. As the *path* relation is unary its models can be represented as a set (of sequences of names). It turns out that this set can be mapped to the set of *object identifiers* from Figure 1. We can also get an appropriate mapping for the constants and predicates:

$$\mathsf{path}(s) \longleftrightarrow s \in \text{OID}$$
$$\mathsf{path}(s < x >< y >) \longleftrightarrow \mathsf{contains}(s < x >, s < x >< y >)$$

In [BC87] the importance of the decomposition hierarchy is stressed. However the approach presented there implements *part hierarchies* (as they are called in that paper) on top of the abstraction hierarchy. We make it the *basis* of our approach.

2.2 Communication in a Hierarchical Model

One of the aims of object-oriented design in hierarchical systems is to reduce communication between objects in the hierarchies to communication between neighbouring nodes. Then the hierarchies provide a means for the design of a distributed system – the nodes can be distributed across a network of computers and the needed connections are known in advance. By using path names as described above the unique identifiers can also be used to locate the object.

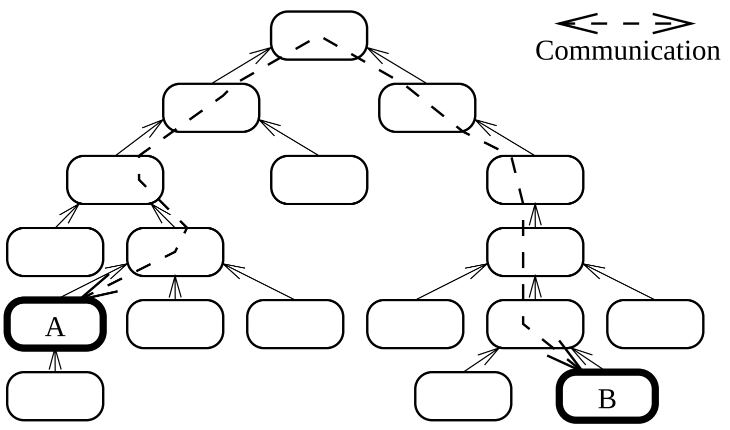

Figure 3 Communication Between Objects

As the graph is fully connected, point to point communication (i.e. communication between any two objects) can be modelled wholly in terms of communication between neighbouring nodes (Figure 3). But this is not always a desired feature as it concentrates communication in the top nodes in the graph. If the physical configuration of the distributed system matches the decomposition hierarchy, i.e. the system itself is a tree structure, this would be most efficient. However, most distributed systems are either flat or their hierarchy doesn't match the decomposition hierarchy. In case of a flat structure this would mean that although a point to point connection between two objects is possible the information flow is channelled through many objects. It seems therefore reasonable to use the described method only for the connection setup: If an object wishes to send a message to another object it finds that object's location (and consequently the path to it) through the decomposition hierarchy. Then a private communication channel between these two objects is set up which is independent from the hierarchy. Messages are sent along this channel. Fig-

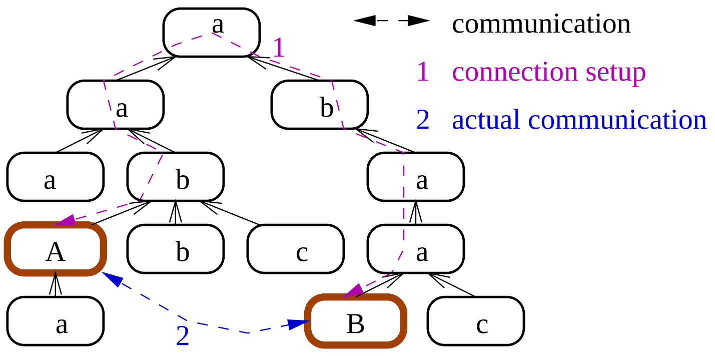

Figure 4 Connection Setup

ure 4 shows an example of the connection setup and the actual communication between two objects in such a system.

The problem of concentration of communication activities in the top nodes of the hierarchy, actually appears to be less serious when we deal with the decomposition hierarchy of an object-oriented system. Objects mostly communicate with their subobjects, container objects and siblings (ie. fellow subobjects) and references to these are thus logically short links. However, modelling data types as objects results in a high number of references to the objects representing the data types. These objects can usually be found in a library that forms a part of the decomposition hierarchy. Thus the references from inside the part of the decomposition hierarchy that represents the actual application program correspond to logically long links.

A communication always involves two parties. When objects are stored in a persistent database both of them need to be loaded into memory. The relation between object identifiers and paths in the hierarchy allows the implementation of efficient caching mechanisms that reduce the overhead involved in from communication. From the object identifier it is easy to determine which objects are the subobjects, container object and siblings. Due to our hierarchical model of communication these objects form a cluster of communication activities. An implementation of the model using a database for persistent storage[Cat94] of objects, can treat these clusters in a special way. For instance, whenever an object is requested from the database it could be loaded into memory together with all the other objects in the same cluster. The database itself can store the clusters' objects in a way that makes their combined retrieval and update efficient. Clusters overlap, because we can identify a cluster around every object. The unique object identifier can be used to determine which objects are cached in memory, thus minimising the access to persistent store. The clustering doesn't have to stop at the level of immediate subobjects, immediate parent and immediate siblings. We could also compute a *distance* metric between objects based on the hierarchy and derive from it access probabilities. Further objects may thus be included in a cluster, while still ensuring the transitive closure of the cluster, ie. whenever two objects A (the center of the cluster) and B are elements of a cluster then all objects on the path in the decomposition hierarchy between A and B will also be in the cluster (Figure 5). The access

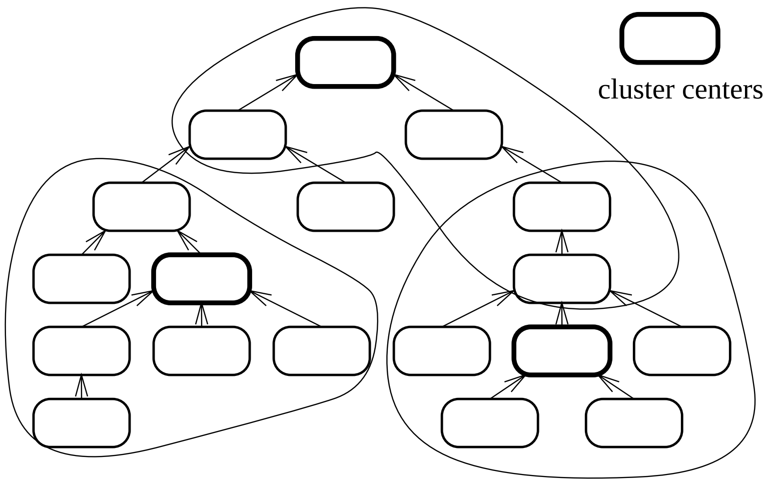

Figure 5 Clusters of Objects

probabilities of these objects will thus be at least as high as the access probability for B. This way of extending the cluster allows us to adjust the mean cluster size to be close to the optimum cluster size for the database. The membership of objects to clusters can change dynamically as objects are added or removed from the hierarchy. Because of the transitivity property of clusters only the clusters for objects that are close in the hierarchy to the added/removed object need to be reconsidered in such a case.

As the previous discussion reveals, the decomposition hierarchy is the basis for a convenient way of modelling communication and so it needs to be inbuilt.

It can be explicitly represented in the model by providing each object with a *set of parts*. This is a set of pairs (*object name, hard link*). *Hard links* are links of the underlying communication system. The set of parts of an object includes links to exactly those objects which are part of it. An additional hard link exists to the container object. Using these links all the communication can take place. Note that, although we used the term 'hard' for the description, the links are set up and destroyed dynamically during the evolution of the system, when subobjects are being added or removed.

Conceptually the modifications to the model gives objects the new role of *switches* for messages. Each object has to have the capability to analyse any received message, determine its destination and to either forward it to the container object, one of its subobjects or process it itself. This decision procedure can be based entirely on object identifiers, as they indicate the position of objects in the hierarchy. The model also makes objects *managers* of their subobjects, being responsible for their creation and destruction. This is feasible because an object's existence requires the existence of its container object. The container object is also close to an object with respect to the communication system, which allows efficient implementation of management operations.

3 STATE AND BEHAVIOUR OF OBJECTS

The behaviour of an object can be determined by a *behaviour script*. When a message is received by the object (and not forwarded to another object) the script is interpreted. This gives objects the role of *processors* of messages. During this interpretation the object's state is accessed. Thus the current object state affects the behaviour of the object. We shall now investigate how we can model this state and how we can associate a behaviour script with objects.

3.1 State as Set of Attributes

From the designer's point of view the state of an object is an assignment of values to a set of attributes[Boo94, R^+91]. This seems to enforce the separate modelling of attributes and values. The attributes have a type. Normally this type is specified in terms of a set of possible values, e.g.

attribute colour={red, green, blue}

This approach enforces the atomicity of values – we cannot decompose them. The specification of the attribute types introduces yet another category in our model that also requires some means of defining sets. But sets are just an abstract data type and our intuition tells us that therefore they should be modelled as an object, i.e. we would model the type of an attribute as an object. However, we actually don't need the explicit notion of type in the first place. We can always model it with the means we already have when it is required – an approach taken in languages like *SmallTalk*[GR89]. We thus only have one domain of values and we allow the values of attributes to range over all values in the domain. However, we don't want to model a domain (even though it's just a single one) either. So we just equate the domain of values with the domain of object references. Another way of viewing this is that attributes only have a single type – object reference. In our example we would then have three object references – red, green and blue. The set of attribute values then becomes a set of object references (e.g. if the colour is green the attribute has a reference to the object *green*). Again *SmallTalk* is an example for such an approach, as is the *Actor model*[Agh86].

The entries in the set of attributes are pairs of the type (*attribute name, object reference*). The entries in the set of parts are pairs of the type (*object name, hard link*) (cf. section 5). What we want is the representation of the set of attributes in terms of the set of parts. For this we associate a *value* field with every object, which is of the fixed type *object reference*. An attribute can now be represented as a part of the object with the name derived from the *attribute name* and the associated object reference stored as the *value* of the attribute object, eg. we could have a *colour* subobject with a value of */colours/green*. This greatly simplifies our model: there is no longer a distinction between the parts of an object and its attributes. Hence no special operations need to be defined for attributes.

3.2 Behaviour Description

In order to associate a behaviour with an object we could insert yet another slot into our object model. Fortunately, this turns out to be unnecessary, since we can view a behaviour script as a function returning a value. This value is an object reference. We can thus replace the *value* slot introduced above with the behaviour script. In case of an attribute, the script would just return the value of that attribute. In order to dynamically update attributes we therefore need to be able to convert object references to behaviour scripts (returning references). Furthermore we need the capability to dynamically associate behaviour scripts with objects.

We arrive at an object model that is extremely simple in its structure yet very powerful. The internal structure of objects based on this model is shown in

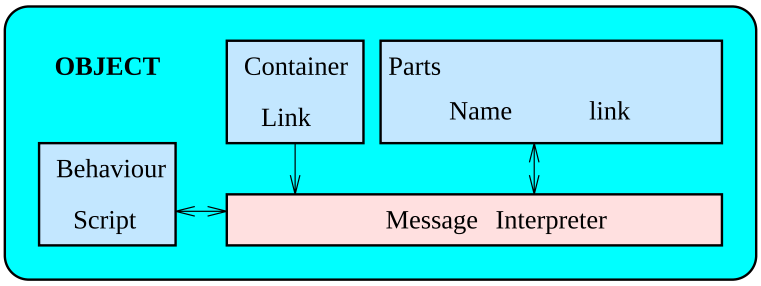

Figure 6 Object Model

Figure 6. The following is a summary of the key features of the model:

- Objects exist and act concurrently.
- Objects are nodes in a decomposition hierarchy.
- An object has a link to its container object in the hierarchy and to its subobjects. All logical communication with other objects takes place along these links, whereas physical communication may happen via direct links between objects, with the logical communication structure being used to establish the connection.
- Objects are addressed by *references*.
- References are lists of names, denoting paths in the decomposition hierarchy.
- An object has an associated *behaviour script*.
- The behaviour script of an object can be set, retrieved or evaluated.
- The evaluation of a behaviour script yields a reference.
- References can be converted to behaviour scripts.

The model unifies the concepts of class, instance, instance variables, class variables and methods into the single concept of an object. In this aspect it is similar to the actor model. There are numerous advantages of such an approach. The simplicity of the model makes its formal analysis and definition simpler. Also the functionality required from the kernel of any system based on the model is reduced. The uniform representation of all concepts in terms of objects makes their representation in databases easier (cf. [AB87, Kro93]). This is important if information needs to be stored persistently.

4 MESSAGES AND RECURSION

The commonly used statement to send a message is

$$result = object.method(arg_1, arg_2, \ldots, arg_n)$$

where *result*, *object*, $arg_1 \ldots arg_n$ are all references and *method* is the name of a method. Statements of this form occurs in the behaviour description of an object. When executed the following happens:

1. A connection from the current object (ie. the object whose behaviour script is being evaluated) to *object* is established.
2. The message is transferred along the connection.
3. A result is received along the connection and assigned as a value to the object referred to by *result*.

The above functionality is not available as primitive to the model. We shall therefore investigate how a similar behaviour can be achieved by other means.

For every message that an object can receive, we create a subobject. We call these subobjects *method-objects* or *methods* as their meaning is similar to methods as we know them from languages like *SmallTalk*. Figure 7 shows how methods are translated into method-objects. Instead of sending a message to an object we invoke the interpretation of the behaviour script of the appropriate method – the method-object whose name is identical with the method identifier in the message. The set of methods associated with an object does not have to be constant. Method-objects may be destroyed, thereby depriving the object

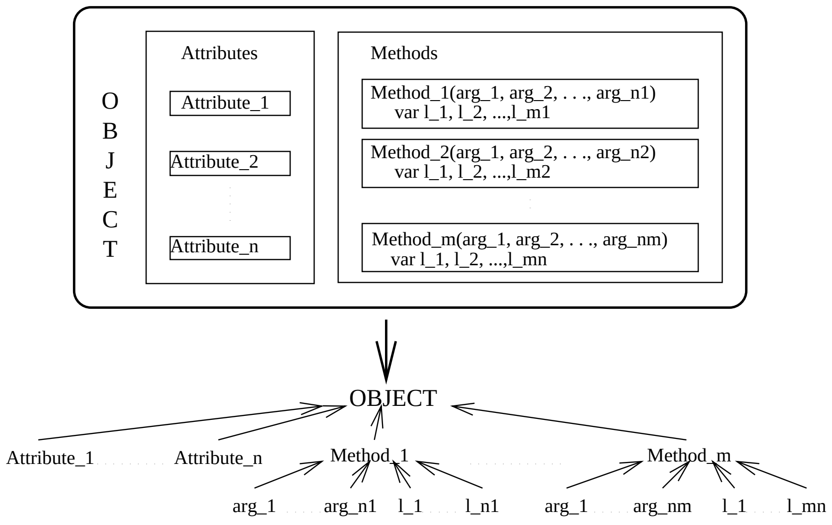
OBJECT
Attributes
Attribute_1
Attribute_2
Attribute_n
Methods
Method_1(arg_1, arg_2, . . ., arg_n1) var l_1, l_2, ...,l_m1
Method_2(arg_1, arg_2, . . ., arg_n2) var l_1, l_2, ...,l_m2
Method_m(arg_1, arg_2, . . ., arg_nm) var l_1, l_2, ...,l_mn
OBJECT
Attribute_1
Attribute_n
Method_1
Method_m
arg_1
arg_n1
l_1
l_n1
arg_1
arg_nm
l_1
l_mn

Figure 7 Methods as Part of the Decomposition Hierarchy

of receiving particular messages, or new methods may be created, extending the set of receivable messages. It is also possible to change the behaviour script of a method-object.

The arguments in a message need to be handled in a special way in our model. Viewing a method as a function the arguments in a message represent local variables of that function. The way to express local variables in our model is to make them parts of the method-object. The original statement for sending a message can be translated into the following sequence of statements (assuming that the formal parameters of the method are named $par_1 \dots par_n$):

$$\begin{aligned}&object/method/par_1 = arg_1\\&object/method/par_2 = arg_2\\&\dots\\&object/method/par_n = arg_n\\&result = object/method\end{aligned}$$

First the arguments are transferred to the parameter-objects of the method. Then the method is invoked.

4.1 Recursion

What happens if the method is invoked recursively? In our model the same script can be interpreted more than once at the same time, so deadlock doesn't occur when we have a recursion. A method-object can accept requests for evaluation (e.g. interpreting the behaviour script) at any time. The problem with recursion lies somewhere else – in our translation of local variables. The local variables of a method (e.g. the arguments in a message) have become subobjects of the method-object. So when the arguments of the recursive invocation are assigned to these objects the previously assigned arguments (of the earlier invocation) are overwritten. To prevent this an object creates a copy of itself (ie. it is *cloned* and all its subobjects when it receives a request for evaluation. The newly created object then evaluates the behaviour script. Accesses to subobjects are received by the copies of those. Objects outside the tree are accessed as usual. We believe that this approach is a novelty in the design of object models. It is essentially the ordinary procedure invocation mechanism applied to our model where the procedure invocation corresponds to the evaluation of a method-object. The one layer of decomposition in procedures comprises the arguments of the invocation and the static and dynamic variables. Unlike procedures our method-objects have more than one layer of decomposition. The

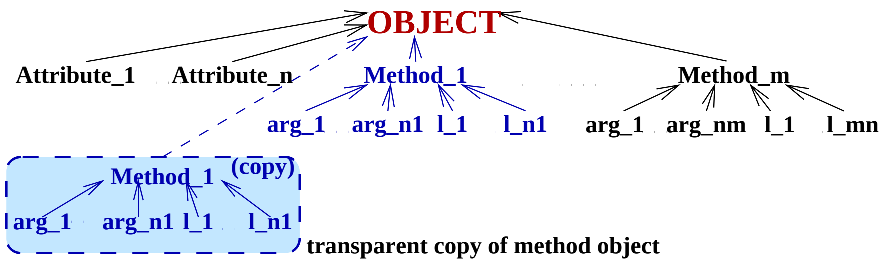

Figure 8 Method Invocation

example in Figure 8 demonstrates the invocation of a method. It can be seen how the additional subtree is created and which objects are accessed. The dotted arrow indicates that this part of the hierarchy is invisible to the other objects further up in the hierarchy.

If objects are stored in a persistent database then the cloning can be carried out within the database as it is a recursive operation over the *subobject* relation. The cloning thus becomes a computational less expensive task. We could have a less eager strategy when creating the copy of the method-object. We might delay the creation of the copies of the subobjects until they are being accessed by the copy of the method object. However, in order to prevent the modification of arguments of the method invocation we would also have to create the copies when subobjects of the original method-object are modified.

The subtree that is created when evaluating a method-object is part of the decomposition hierarchy. For all the objects in it the whole process of creating copies is transparent. Objects outside this subtree cannot access the objects in it. The subtree is invisible to them and their requests go to the original objects, making it possible to start further evaluations at the same time. When the interpretation of a script finishes the subtree is removed. During this process the removal of any subsubtrees is delayed if there are still scripts being interpreted in objects that reside in those parts of the tree, preventing the destruction of the environment in which those script interpretations are taking place. Once the subtree is removed all changes to the objects in it are lost. Fortunately, that changes to objects further up in the hierarchy remain. This is important as otherwise the attributes of an object couldn't be changed permanently. The attributes are parts of the object, they reside on the same level of decomposition as the method-objects. Hence when a method-object wishes to access an attribute of the object it is part of this request is handled by the original attribute-object.

4.2 Synchronisation

In any concurrent system synchronisation plays an important role. Without means of synchronising activities any such system would behave chaotically. Methods for synchronisation are not specified in our object model. We could attempt to implement synchronisation methods on top of the presented model. But it turns out that without any low-level synchronisation this is impossible. As in the case of recursion the problems are again caused by the modelling of methods. When we transfer the arguments to a method (as it was described above) we require that no other object interferes with that process. We can address this problem by introducing the database concept of *locks*[KS92]. But we cannot model locks with the means we have. If we for instance tried to coordinate the access to the method-object by a scheme granting permissions we would only shift the problem to the object which grants the permission. It is therefore necessary to make locks part of the model. We modify the method invocation from section 7 by enclosing it between a `lock` and `unlock` action:

```
lock $object/method$
$object/method/par_1 = arg_1$
$object/method/par_2 = arg_2$
...
$object/method/par_n = arg_n$
$result = object/method$
unlock $object/method$
```

Any object is either in a locked or unlocked state. If an object is locked it delays the execution of all lock requests. If it is unlocked a lock request is answered and the object goes into the locked state. Locks are usually supported by databases. This fact can be utilised for efficient implementations.

We have to be careful with recursion again. When a method-object is invoked in its locked state (as in the example) recursion would yield to deadlock. However, from the preceding section we know that the invocation is actually sent to a copy of the method-object and hence all the messages that are sent to the object from lower levels of the hierarchy (i.e. from inside the method) are being processed by the copy. This copy thus must be set into an unlocked state after its creation. This approach by default prevents other objects from invoking the method for the duration of the message interpretation as the original method-object is still in its locked state. This can easily be overcome though. The method itself just has to explicitly unlock the original method-object. Of course the *unlock* statement in the caller script (see above) is now redundant and must be ignored. This approach to synchronisation is quite elegant as the control over the degree of concurrency is determined by the method-objects themselves.

5 INHERITANCE

Inheritance is an important concept in object-oriented design[CY91, Boo94] and it would therefore be advantageous if we were able to express it in our model. It turns out that there exists a method of modelling inheritance that can be easily added to our model – *delegation*. It is the most suitable approach in the kind of system we look at. Other authors ([BY87, KL89]) have come to the same conclusion when investigating inheritance in highly dynamic object-oriented systems. Since we don't have the explicit notion of classes inheritance can occur between any objects. It turns out that some of our model's features make it particularly suitable for incorporating delegation without great difficulty. How does delegation work?

> If an object receives a message it doesn't understand it forwards this message to it's parent object. If this doesn't understand it either it forwards the message to its parent. Eventually the message either arrives at the top object of the abstraction hierarchy and is rejected or one of the parents in the chain accepts it. In this case it executes the associated method. The execution takes place in the context of the object the message was send to originally. In case of a multiple inheritance the message is only rejected if all alternatives fail.

Since in our model the methods are subobjects of the receiver object, if a method corresponding to a message doesn't exist the subobject with the name of the method doesn't exist. Hence 'not understanding a message' means that a container object cannot forward the message to one of its parts because this part doesn't exist. With *delegation* we then forward the message to the parent object instead (in case of multiple inheritance there can be several). We must distinguish however between normal messages and delegated messages – if a delegated message is accepted the action has to take place in the context of the original object. This requires a notion of inheritance that is slightly different from delegation and was introduced by A. Yonezawa in [BY87] – the *Recipe-Query Scheme*. In order to execute a method in the context of another object (that is precisely what has to happen when a parent object reacts to the message on behalf of the original receiver) the body of the method (the *recipe*) is transferred to the original object. So actually the original message is not forwarded to the parent object but replaced by a query for the body of the method (i.e. the behaviour script).

In order to add inheritance by delegation we just need an additional slot in the object model containing the reference to the parent object (or several references in case of multiple inheritance) and modify the object lookup and message forwarding procedures to incorporate the functionality of delegation as described above. If we allow dynamic access to the parent slot we get a very dynamic notion of inheritance in which objects can change their parents at run-time. Note also that this notion of inheritance does not require any special support from a persistent storage database.

6 SUMMARY AND FUTURE WORK

It has been shown that concurrency is an inherent feature of objects and follows directly from very simple definitions of the object-oriented paradigm. The unification of common concepts in object-oriented design such as class, instance, instance variable, class variable and method, yields a powerful object model. Concurrency is expressed naturally in the model. Persistent storage of data can be achieved via a small interface to a database.

We have made the decomposition hierarchy a central element of the model. It serves as a logical communication structure and to create object identifiers. Objects communicate along links derived from the *subobject* relation. Communication is concentrated in clusters around objects in the hierarchy. This can be exploited to implement efficient caching mechanisms for persistent storage databases. Object identifiers are derived from paths in the hierarchy. The route of communication between two objects can be determined by examining their object identifiers. It also means that we can refer to objects prior to their existence and that object references can be stored in persistent databases. The globally unique object identity ensures that caching mechanisms in a persistent storage database can operate efficiently.

Our model turns instance variables and methods into subobjects. Method-objects have their parameters and local variables as subobjects. As instance variables and methods are frequently accessed by the object they belong to, and similarly parameters and local variables of a method are only accessed by the method-object they belong to, the clustering of communication is improved even further. The representation of methods as objects affects the way those methods are invoked. We have developed a scheme that permits the recursive and concurrent invocation of methods. Method-objects, together with their subobjects, are cloned upon invocation and *locks* are employed for synchronisation purposes. Method-objects have full control over the degree of concurrency in their invocation. In the cloning of objects we see another utilisation of the decomposition hierarchy, as it is essentially a recursive function over the subobject relation. It can therefore be implemented entirely inside a persistent storage database. *Locks* can be modelled as locks in the database.

In our recent research we formally specified the described object model in terms of first order logic[RE94]. The simplicity of the model yields a very compact specification. We also devised a language for behaviour scripts and defined the semantics of that language in terms of the π-calculus[MPW92, Mil91, Wal91], which allowed an elegant representation of the various degrees of concurrency in the model. A prototype of a system based on the model has been implemented on the distributed system *Darwin*[MDEK95, CDF$^+$95, MEK95], as well as a prototype compiler for the script language. Our current research investigates the relation between the object-model and databases with the aim to implement a persistent storage mechanism.

REFERENCES

[AB87] M P Atkinson and P Buneman. Types and persistence in database programming languages. *ACM Computing Surveys*, 19(2):105–190, June 1987.

[Agh86] G.A. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[BC87] E. Blake and S. Cook. On including part hierarchies in object-oriented languages, 1987.

[Boo94] G. Booch. *Object-Oriented Analysis and Design.* The Benjamin/Cummings Series in Ada and Software Engineering. Benjamin/Cummings, Redwood City, California, 1994.

[BY87] J.-B. Briot and A. Yonezawa. Inheritance and synchronisation in concurrent oop. In O. Nierstrasz, editor, *ECOOP'87 Proceedings*, Lecture Notes In Computer Science. Springer-Verlag, 1987.

[Cat94] R Catell. *Object Data Management: Object-Oriented and Extended Relational Database Systems.* Addison Wesley Publishing, New York, 1994.

[CDF$^+$95] S. Crane, N. Dulay, H. Fossa, J. Kramer, and M. Sloman. Configuration management for distributed software services. In *Proc. of the Int. Symposium on Integrated Network Management, Santa Barbara, USA.* Chapman & Hall, May 1995.

[CY91] P. Coad and E. Yourdon. *Object-Oriented Design.* Yourdon Press Computing Series. Prentice Hall, Englewood Cliffs, 1991.

[GR89] A. Goldberg and D. Robson. *SmallTalk-80: The Language.* Addison-Wesley, 1989.

[KL89] D. Kafura and K. Hae Lee. Inheritance in actor based concurrent object-oriented languages. In O. Nierstrasz, editor, *ECOOP'89 Proceedings*, Lecture Notes In Computer Science. Springer-Verlag, 1989.

[KL93] D. Kafura and R. Lavender. Concurrent object-oriented languages and the inheritance anomaly. In *ISIPCALA'93 Proceedings*, 1993.

[Kro93] P Kroha. *Objects and Databases, Object Oriented Databases.* McGraw-Hill International (UK) Limited, Berkshire, England, 1993.

[KS92] H Korth and A Silberschatz. *Database System Concepts.* McGraw-Hill International (UK) Limited, Berkshire, England, 1992.

[MDEK95] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Fifth European Software Engineering Conf.*, Barcelona, 1995.

[MEK95] J. Magee, S. Eisenbach, and J. Kramer. System structuring: A convergence of theory and practice? In K.P. Birman, F. Mattern, and A. Schiper, editors, *Theory and Practice in Distributed Systems, Proc. of the Dagstuhl Workshop*, volume 938 of *LNCS*. Springer Verlag, 1995.

[Mes93] J. Meseguer. Solving the inheritance anomaly in concurrent object-oriented programming, 1993.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[Mil91] R. Milner. The polyadic π-calculus: a tutorial. Technical Report ECS-LFCS 91-180, University of Edinburgh, October 1991.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I and II. *Information and Computation*, 100:1–77, 1992. Also as Tech. Rep. ECS-LFCS 89-85/86, University of Edinburgh.

[Par72] D Parnas. On the criteria for decomposing systems into modules. *CACM*, 15(2):1053–1058, 1972.

[PB94] R. Pandey and J. Browne. A compositional approach to concurrent object-oriented programming. In *ICCL'94 Proceedings*, 1994.

[R$^+$91] J. Rumbaugh et al. *Object-Oriented Modeling and Design.* Prentice-Hall International, London, 1991.

[RE94] M. Radestock and S. Eisenbach. Towards a minimal object-oriented language for distributed and concurrent programming. In *PODC'94 Proceedings.* Springer-Verlag, 1994.

[W$^+$91] P. Wegner et al. Panel: What is an object? In O. Nierstrasz, editor, *Object-Based Concurrent Computing; Proc. of the ECOOP'91 Workshop*, volume 707 of *Lecture Notes In Computer Science*. Springer-Verlag, 1991.

[Wal91] D. Walker. π-calculus semantics of object-oriented programming languages. In *Conf. on Theoretical Aspects of Computer Software*, Tohoku University, Japan, September 1991.

[YM93] A. Yonezawa and S. Matsuoka. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Object-Based Concurrency.* MIT Press, 1993.