Meta-level Programming with CodA

Jeff McAffer

Department of Information Science
The University of Tokyo
and
Object Technology International
jeff@acm.org

Abstract. Meta-levels are complex pieces of software with diverse demands in both the computation and interaction domains. Common techniques using just code to express behaviour fail to clearly assign responsibility for a particular behaviour's definition or to provide support for the reuse or integration of existing behaviour descriptions. The techniques of fine-grained decomposition of meta-level behaviour into objects and their subsequent composition into object models provides a framework for creating, reusing and integrating complex object behaviours. Using such a framework, we show that users can develop and integrate quite different object models while retaining a high degree of abstraction and fostering meta-level component reuse.

1 Introduction

Meta-levels are potentially complex pieces of software. They have diverse requirements both for computation and for interaction. Building open meta-level architectures is particularly challenging because of the diversity of behaviours we may wish to describe while maintaining a uniform base-level view of object behaviour.

Many of the current architectures are open but in a restricted sense. They reify various aspects of a particular language or object model and provide infrastructure for change within the scope of that domain. If this is called a 'top-down' approach, we have taken a 'bottom-up' approach. Rather than starting with and then opening a particular object model, we start by describing various notions or views of generic object behaviour and then provide infrastructure for composing these behaviours into specific object models.

This is somewhat related to a basic concept of object-oriented software engineering — decompose a specific problem into generic components and then compose the pieces to solve the problem. From this we get both a solution to our problem and a set of reusable components. In addition, the fine-grained decomposition approach fosters a clear separation between, and definition of, the various components.

Using these techniques at the meta-level, we have developed CodA, a meta-level architecture for describing a wide range of object behaviour models. CodA can be thought of as a generic object engine framework in which users define, on a per-object or even per-use basis, how object's behave computationally. To demonstrate this we present three object models from different computing domains, specifically; concurrency, distribution, and communication.

The abstraction of behaviours into objects encourages reuse and simplifies the combination of object behaviours. While in CodA, object model combination is still a somewhat manual process, we propose that the architecture inherently provides for; the easier identification of points of conflict, isolation of the effects of changing a component, increased reusability of components derived from the combination of object models and easier management of the behaviour space. The addition of some simple annotations for describing component *properties* further enhances these qualities.

The remainder of the paper is organized as follows. The next section details the CodA architecture and in particular, our approach to meta-level factoring. Sections 3, 4 and 5 detail three object models implemented in CodA and section 6 discusses the combination of object models. We relate CodA to other work throughout the paper and summarize these relations in section 7. A further section concludes the paper and points out some directions for future work. Appended to the paper are the standard interface and default implementations for many of the meta-components discussed.

2 The CodA Meta-Architecture

The key concept in the design of the CodA meta-level architecture is the 'decomposition' or 'factoring' of the meta-level into fine-grained objects or *meta-components*. When programming base-level object-oriented systems we typically factor out behaviour, create objects for each factor and then compose these objects into applications. The only difference here is that our 'applications' are *object models* which describe the operation of objects. That is, the meta-level is just an application whose domain happens to be the behaviour[1] of objects.

In factoring the meta-level we developed a relatively generic model of object execution. The meta-level is defined as playing a number of *roles* in the description of base-level object behaviour. Each role is filled by a meta-component and corresponds to some behaviour such as; object execution (both mechanisms and resources), message passing, message to method mapping and object state maintenance. Roles may be filled by many different components and components can sometimes fill several roles. An object's behaviour is changed by explicitly redefining components or by extending the set of roles. Examples in later sections highlight this process.

The CodA meta-level architecture is largely run-time oriented. It does not provide integral support for language constructs like classes which are required for the static description of object behaviour. Rather, these constructs are borrowed from whatever language is used to implement CodA. For example, in the

[1] We use the term *behaviour* to denote *how* an object acts as opposed to *what* it does and so by nature, *behaviour* is a meta-level concept.

Smalltalk implementation, Smalltalk metaclasses are used to define a nice interface for particular object models (i.e., sets of meta-components) which are shared by instances. These interfaces determine which components can be modified and what configurations the meta-level can take. CodA is perhaps 'lower-level' than other systems but this approach allows us to gain a certain measure of language independence while retaining the potential of the architecture.

This is in contrast to the meta-level facilities found in systems like CLOS[7] and ClassTalk[4, 5]. The focus of these systems is different in that they intend to open or extend the functionality of particular language facilities or constructs. As such, they deal with somewhat more static issues and it is natural that their capabilities and constructs be language specific.

CodA differs in two other related ways; granularity and decomposition into objects. A common approach to factoring object execution behaviour (e.g., message sending or method execution) is to create public interface methods on a small number of meta-level objects [8, 6, 7]). That is, object behaviour is decomposed into code at the meta-level. Changes to a behaviour are made by modifying its related interface methods.

Unfortunately, code is inherently more difficult to deal with than objects. Method objects have little support or infrastructure for interaction, change or extension. Without this, describing complex interactions between several behaviours (i.e, groups of interface methods) is confusing. Behaviours are not encapsulated into atomic units and overall responsibility for a behaviour is not clear. Unanticipated behaviours have no clear home for their description and/or state.

Decomposition into objects gives us a higher-level view of behaviour. Objects abstract code, define points of interaction and ease integration. They remove us from the details of implementation code. The boom in micro-kernel operating systems is also witness to these ideas. Rather than creating one large, all encompassing kernel with many functions, we define a small and simple infrastructure and build/use the OS components as needed.

Forming meta-levels by composition has some advantages over techniques such as multiple inheritance (e.g., ClassTalk[4, 5]). The issue here is similar to the "parameterization vs. subclassing" issue — Should we create a generic object which we parameterize by plugging together pieces, or do we create a class hierarchy covering all possible parameter configurations? Unfortunately, the number of possible configurations is a combinatorial function with explosive potential. Just the small number of behaviours we will discuss in this paper translates into hundreds of possible combinations. Apart from the class/metaclass name-space explosion (which can be partially covered up using anonymous metaclasses), we also encounter a method name-space problem where methods from metaclasses which describe different behaviours, collide.

To put this design discussion in more concrete terms, below we give descriptions of the major elements in the CodA object system. In addition to the meta-level infrastructure, there is the set of seven components each of which describes some behaviour in the basic execution model.

2.1 The Meta-level

CodA specifies that every object has a conceptual meta-level. The meta-level is not a single object but rather a *set* of meta-components each of which describes some aspect of base-level object behaviour. The implementation of the meta-level (i.e., these sets of components) is not defined except to say that a particular set, and thus a particular meta-level, can optionally be: *fixed* and allow no changes, *changeable* in that existing components can be replaced or *extensible* by allowing new roles and components to be added to the meta-level.

Meta-level programmers need some way of shifting to the meta-level and of accessing an object's meta-components. This can be done using language constructs as in ABCL/R2's ↑ (up arrow) [8] or by making explicit meta-level objects which are accessed via normal message passing (e.g., CLOS). In CodA we adopt the technique best suited to the base-level language. For example, in the Smalltalk implementation an object's meta-level as a whole is 'represented' by the result of sending meta to some object (e.g., anObject meta). The meta is used for both shifting and accessing. All messages sent to a meta are executed at the meta-level and metas 'broker' (i.e., provide access to) meta-components.

In the implementation, the meta may take many different forms. It may store internally the meta-components it brokers, it may simply fetch them from somewhere or it may create new ones for each request. This is transparent to the user/programmer. When asked, a meta will return the actual component which fills the specified role. Similarly, a meta can be requested to use a particular component for a particular role.

2.2 Meta-level Components

As a basis, CodA defines a default set of seven components which are present at the meta-level of all objects; Send, Accept, Queue, Receive, Protocol, Execution and State. While they do not cover every possible aspect of object behaviour, the set is extensible and the standard seven cover the behaviours essential to common object models. Readers should note that these meta-components are *logically* distinct. In reality, one entity may fill multiple roles. For example, the Queue and Receive components may, in a particular case, be implemented as one physical object. Meta-components can also be shared between objects.

In our discussion below, details such as message selectors are taken from the Smalltalk implementation of CodA. A simplified specification of these meta-component's execution interface is provided in an appendix.

Send. A Send's main role is to manage the potentially complex series of interactions between message sender and receiver ensuring proper transmission and synchronization. This includes protocol negotiation, synchronization and resource management. For example, when sending a synchronous message, the sender's Send must inform the receiver that a completion signal (e.g., reply) is required, how and to where the signal is to be transmitted, and also block the sender until the completion signal is received. The PortedObject model discussed below contains a Send which diverges substantially from the default model.

An object's Send is accessed using anObject meta send{:}. It invokes the message receiver's Accept.

Accept. Accepts define the receiver side of the message passing protocol negotiation and synchronization. They are also responsible for determining if a message is valid and how it should be handled (e.g., queued, processed immediately). Note that *accepting* a message is different from *receiving* a message. Acceptance concerns the interaction between the sender and receiver while receiving is the internal act, by the receiver, of picking the message for processing.

An object's Accept is accessed using anObject meta accept{:}. It is invoked by the message sender's Send and invokes the message receiver's Queue.

Queue. Queuing is the main mechanism of decoupling the execution of message senders and message receivers. Messages which have been accepted but cannot yet be processed must be queued. Once queued, the message's sender can be released to continue executing if the message's protocol allows. There are a great variety of possible queuing policies using a variety of factors to determine in which queue a message should be stored (e.g., by sender or type) and the message's place in that queue (e.g., FIFO, priority).

An object's Queue is accessed using anObject meta queue{:}. It is invoked by the message receiver's Accept and by an object's Receive.

Receive. As noted above, receiving and accepting are different operations. Receiving refers to the actual fetching of the next message for execution. In other words, while Accepts are concerned with how objects synchronize and interact with each other (i.e., inter-object synchronization), Receives deal with intra-object synchronization. When a Receive is asked for the next message to process, it may consider many different physical queues and consult various constraint specifications before determining the next appropriate message. The PortedObject model discussed in section 5 details an example of such a situation.

Note that many architectures implicitly combine the operations of Accept, Queue and Receive into the same object with quite a narrow interface. As such, the implementation or integration of a new scheme for one of these behaviours necessarily impinges on the others. Making them explicit and concrete simplifies the construction of complex behaviours.

An object's Receive is accessed using anObject meta receive{:}. It is invoked by objects when they are looking for the next message to process and invokes the object's Queue.

Protocol. A message, having been received, is translated into a method for execution. This is the primary responsibility of an object's Protocol. The most common mapping is an exact message selector to method name match where methods are examined according to some inheritance scheme. Protocols define both the selection criteria (e.g., exact match) and the search scheme (e.g., single/multiple inheritance). In more complex cases, Protocols may maintain multiple method tables and determine which to use based on some aspect of the base-level or system state.

An object's Protocol is accessed using anObject meta protocol{:}. It is invoked by objects when they need to map a message to a method to execute. That is, typically from an object's Execution.

Execution. For an object to execute methods, it must interact with some system resources (e.g., virtual machines, processes). Executions describe how this interaction occurs. By manipulating its Execution, a programmer can control where and when an object runs as well as its overall importance (e.g., priority) and independence.

Having an explicit execution model also enables methods to be somewhat more abstract and to be executed in different ways depending on the situation. For example, if we are debugging an object, we may wish to execute its methods on a special debugging virtual machine or interpreter whereas normally methods are executed as native machine code. It is the Execution's role is to determine how to execute methods and then execute them.

An object's Execution is accessed using anObject meta execution{:}. Executions are generally invoked by either the Accept or Queue (in the passive case) or by the explicit or implicit invocation of a receive operation (in the active case).

State. Though not directly involved in execution, state is an essential part of an object. The role of a State is to organize and maintain object state. It defines both what slots the object has as well as how the data in those slots is stored. The State *does not* actually hold the data. It simply knows how it can be accessed.

An object's State is accessed using anObject meta state{:}. States are invoked whenever one of the object's slots is accessed.

2.3 Example Meta-level

In Figure 1 we depict the events, meta-components and interactions involved in the sending of a message M from object A to object B (as indicated by the heavy dashed arrow). The shaded areas contain meta-components. Each light arrow is an interaction event (dashed for A's execution thread, solid for B's). The heavy solid arrows indicate the base/meta relationship and go from base-level to meta-level. We have labeled only those meta-components relevant to this particular interaction.

We see that A sends M by interacting with it's Send (1). The Send then transfers M to B's Accept (2) which queues it with the Queue (3). At some point, B will execute a receive operation which invokes the Receive (4) and fetches the next message from the Queue (5). The message is mapped to a method by the Protocol (6) and finally, the message is processed by executing the found method (7). In this way, every aspect of basic execution is reified.

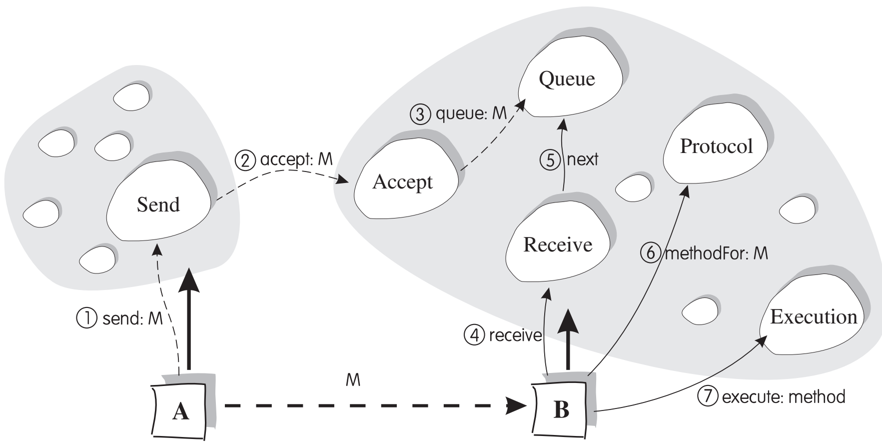

Fig. 1. Sample meta-level configuration and interaction

2.4 Implementation

CodA has been implemented in Smalltalk and much of the remaining discussion draws from that experience. Because CodA deals largely with execution rather than language issues, we have been able to fully integrate it into Smalltalk. For each Smalltalk object we transparently and lazily add the meta-level infrastructure and default meta-components which describe the standard Smalltalk object model. In this implementation, a standard CodA object behaves just like a standard Smalltalk object. Having opened the implementation of a Smalltalk object, we can adjust its behaviour as needed. Objects generally retain their base-level semantics but gain some additional behaviour such as concurrency or distribution.

Using the CodA framework as described above, we have created a library of components with which we have built a number of object models. The following sections describe the design and implementation of three models; ConcurrentObject, DistributedObject and PortedObject as demonstrations of the CodA concepts and design. An appendix contains example implementations of the default behaviours for some meta-components. Our discussion is set in terms of changes to these behaviours.

3 Concurrent Objects

Passive objects are reactive in that they simply respond to external stimulus or input and 'borrow' processing resources from message senders. In the ConcurrentObject model, objects have their own internal *activity* and processing resources (threads). This behaviour is described by a ConcurrentExecution component which fills the Execution role.

A ConcurrentExecution's idling execution behaviour (i.e., what objects do when they are not driven by user code) is similar in intent to that of Actors as seen in [1, 3, 10]. While formal Actors redefine their execution behaviour after every execution, in practice the replacement behaviour is the same; receive and process a message. Our basic activity model is similar; an endless loop, receiving and processing messages. For passive objects, the activity loop is implicit in the runtime system. For ConcurrentObjects, the loop runs explicitly in the threads associated with an object's ConcurrentExecution. The following is an example of such a loop for an object base.

```
| message result |
[true] whileTrue: [
    message := base meta receive receiveFor: base.
    result := self process: message for: base.
    base meta send reply: result to: message for: base]
```

When a message arrives, a passive object's Queue actually calls the object's Execution and directly triggers the processing of the message. That is, there is no queuing, only immediate processing. The Receive is never called explicitly as objects are always implicitly receiving incoming messages. Adding explicit thread(s) and an activity to an object both invokes its Receive and raises the possibility the sender and receiver of a message may be disjoint with respect to execution threads.

In addition to the activity loop, ConcurrentObjects change the Queue to ensure that messages are actually queued rather than passed on to the Execution. StandardQueue (shown below) is an example of such a Queue. It maintains an internal queue structure on which it implements the Queue interface. The actual queuing model used (e.g., FIFO) depends entirely on how we want incoming messages to be ordered (i.e., the object's queuing policy). This is specified by the user in the creation of the StandardQueue.

```
StandardQueue>>nextFor: base
    ^queue next

StandardQueue>>enqueue: message for: base
    queue add: message
```

The ConcurrentObject model does not need to define new message sending mechanisms as the default Send components already include the notions of synchronous, asynchronous and future messages. In the default, passive object case, synchronous sending is the default, future messages represent a promise to compute similar to closures or blocks, and asynchronous messages are mapped to synchronous messages where the result is ignored.

These ideas are included in the default behaviour for two reasons; they are useful in normal object behaviour description and they are relevant to system parallelism, not object concurrency. For example, a distributed system can contain no ConcurrentObjects but still require asynchronous sends.

4 Distributed Objects

The DistributedObject model is a somewhat larger change to object behaviour. In the model, objects live in *spaces*. An object's Execution and State can live in different spaces and can be independently copied, replicated or moved between spaces. Inter-space messaging fits naturally into the normal object model through the use of RemoteReferences or Proxies [14]. The model contains a sophisticated, uniform mechanism for describing how objects are transmitted from space to space (marshaling). The model is equally applicable to passive and active objects and is built largely out of new components/roles (e.g., Marshaling, Replication and Migration) and infrastructure objects (e.g., RemoteReference and Space). Here we present a few of these new structures. More detailed coverage of the DistributedObject model can be found in [9].

Spaces are places in which objects exist (store their state) and execute. A Space is known to every other Space and can be addressed (i.e., sent messages) directly from anywhere in the distributed machine. They manage the mapping between global object ids and local representatives (e.g., RemoteReferences, replica).

A RemoteReference is a local representative or *Proxy*[14] for some remote object. Locally they are just like any other object. They can be stored in instance slots, assigned to variables, passed as arguments, etc. When they are sent a message, the simplest RemoteReference just forwards it to the space containing the real object — the *target*. More sophisticated RemoteReferences process some messages locally while forwarding others to the target.

RemoteReferences are themselves implemented using modified CodA meta-components. According to the CodA execution model, when a message is sent to an object, the sender's Send and the receiver's Accept interact to effect the message transfer. In the DistributedObject case, these meta-components are in different spaces. Local to the sender, the receiver is a RemoteReference and the receiver's Accept is an intelligent RemoteReference to the target's Accept. Rather than performing the normal accept operation, the local Accept *marshals* the message into a stream of bytes and transmits it to the remote space. Once there, it is reconstructed and accepted by the target's Accept. In this way, the DistributedObject model is uniformly applied to all objects in the system, even those at the meta-level.

4.1 Replication

The basic idea of replication is that an object's state can exist in multiple Spaces at the same time. Furthermore, through the use of some distributed consistency schemes, we can maintain the proper semantics of our programs. Schemes for replication range from simple one-off copying (not technically considered replication here) to fully coherent replication. Since this is an entirely new behaviour for objects, it is a new role (Replication) for the meta-level and requires new meta-components for its implementation (The addition of new roles is covered in Section 6). As we will see, the role of the Replication is quite independent of the object's execution and the actual structure of an object's state variables.

To demonstrate replication we develop the partial replication scenario shown in Figure 2. The figure shows two objects, original (in Space 0) and replica (in Space 1). Though not shown, original is actually a 2D N-Body [2] problem solver which calculates the forces exerted by, and movements of, a collection of bodies or *particles* in a 2D plane. N-Body solvers arrange a set of particles in a Quad tree structure according to their physical location and then process each particle individually. Overall, processing consists of a couple tree scans and iterations over the collection of particles.

To distribute this algorithm we divide the particles into subsets which are worked by different solvers, one per Space. The sets however, are not entirely independent since all particles potentially exert forces on all others. As the tree is the central data structure for relating particles to one another, it must be globally known and unique. The solver is a prime candidate for partial replication.

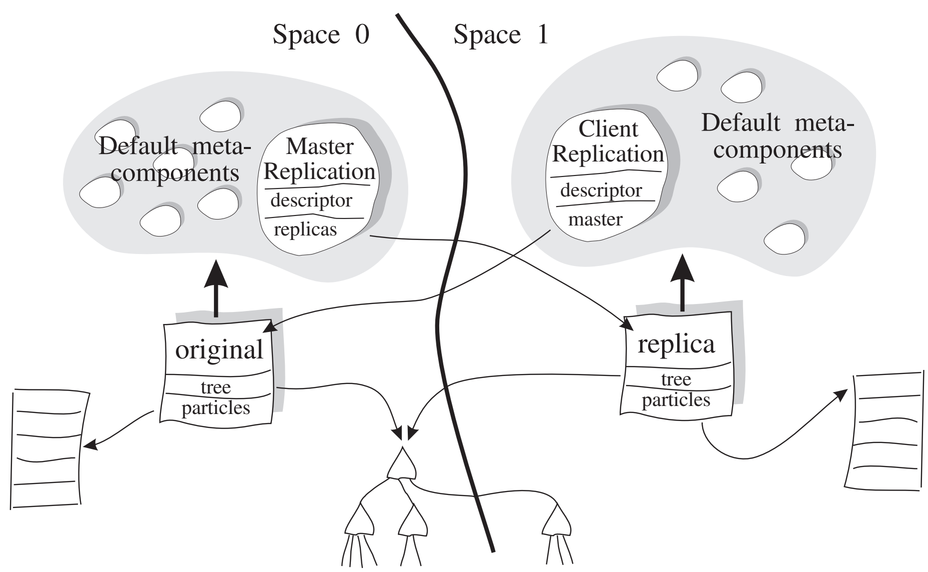

Fig. 2. Distributed object layout

As shown in Figure 2, original, the solver, has two slots; particles and tree. replica is a partial replica of original where the tree slot is consistency managed and the particles slot is not. Every replica in the system shares the same tree but has an independent particle set. The replication of original is done in a series of six steps. Figure 3 shows the required code while the discussion below explains each step.

```
1)  original meta replication asMasterUsing: #('tree') for: original.
2)  original meta replication replicateIn: (Spaces at: 1) for: original

MasterReplication>>replicateIn: space for: base
3)  space replicate: base using: descriptor for: base.
6)  replicas add: space

Space>>replicate: copy using: descriptor for: master
4)  copy meta replication asClientOf: master using: descriptor for: copy.
5)  master become: copy
```

Fig. 3. The making of a replica

1. Ensure that the original's Replication is compatible with the behaviour described by the MasterReplication component. It should be able to detect state changes in the appropriate slots and maintain a list of replicas. The first argument is a marshaling descriptor which specifies how the slots of original are to be copied to the remote space and as a result, how original is to be replicated. Simply giving a slot name indicates that the slot is to be replicated using whatever marshaling technique is appropriate at the time (i.e., the default).
2. Invoke the replication operation and specify which spaces are to receive replica. In keeping with our example, only Space 1 is specified.
3. Copy the relevant slots of original to all of the specified Spaces. The replicate:using:for: message has three arguments. Though the first and third appear redundant, they are not — they are marshaled differently. The first argument is marshaled according to the specification in descriptor while the third is marshaled as a reference. This difference is critical for the next two steps. When the message gets to the remote space, the first and third arguments will no longer be identical. The first will be a copy of base while the third will be a reference to base. Note that though marshaling descriptor specification is a simple addition to the messaging syntax, the details are omitted from this example to improve clarity.
4. Make the remote copy into a replication *client* of original. This is similar to step 1 and executes in the remote Space which will contain replica. copy's meta-level is modified such that all state changes are delegated to master and the Replication knows the identity of its master for future reference.

5. Convert any preexisting remote references to original to be local references to replica. Remote spaces may contain references to master prior to replication. To maintain a consistent view of the world, these remote references should be changed into local references to the newly created replica.
6. Invoke consistency management on the replicated slots of original by adding the Space to the list of consistency controlled replica locations.

In step 1 we hooked the relevant state change operations for original. Note that we do not require a new State component. The existing component's meta-level is manipulated to hook state accesses. This both isolates replication from representation and reduces the possibility of object model conflict. When original's replicated state is changed, its Replication's update:with:for method (shown below) is invoked by the hook. The method simply broadcasts the change in slot to all of original's clients. Typically this would be done by multi-cast messaging for efficiency though here we specify an iterative approach for clarity.

```
MasterReplication>>update: slot with: value for: base
    replicas do: [:space | | rep |
        rep := (base in: space).
        rep meta replication update: slot with: value for: rep]
```

In this example we have shown a relatively lax model of consistency. To implement *strict consistency* requires only the addition of a two phase update protocol between masters and clients and the hooking or delegation of both read and write state accesses on masters and clients rather than just writes. Both of these changes are straightforward and are done using existing meta-level structures and mechanisms.

4.2 Summary

The DistributedObject model, and this example in particular, highlight a central theme to our work — the addition to, or modification of, an object's computational behaviour without changing its base-level code. The result is the ability to use standard class libraries in many different environments. For the N-Body application, the original uniprocessor sequential version required a code change only where a new tree node was created and we wished to explicitly direct its location. Other changes were done to take advantage of the newly introduced concurrency but were not essential.

In terms of distributed object-oriented computation, our model also highlights something which we feel is essential — the ability to talk about distribution on a *per-use* basis rather than just on a per-class or even per-object basis. In the N-Body application (replica creation step 3) we demonstrate the need for use-based marshaling. Also, while our distribution scheme calls for partial replication of the solver objects, someone else's might use a different scheme. Using this architecture, all they need program is the meta-level of individual objects as they are used.

5 Ported Objects

PortedObjects are objects which communicate and behave in a dataflow-like way. They have *ports* or channels over which data flows and when data is available for processing, processing is done. This style of behaviour is interesting in a number of areas. People working in concurrency formalisms like the π-calculus [11] have found channels and ports to be useful in specifying object communication. Most popular data visualization and analysis systems like AVS [16], IRIS/Explorer [15] and parallel system analysis tools like Pablo [13] have dataflow or analysis graph architectures. The model is interesting to us in both regards.

Ported behaviour should be as transparent as possible to the base-level code. For example, in an analysis system we developed, we used a set of generic objects which describe various analysis operations (e.g., filters, collectors, expert systems, DSP processors) and added a set of meta-components which gave these *analysis objects* ported behaviour. The idea was that users (analysts) could then build their own analysis tools by simply connecting existing analysis objects to form the desired analysis graphs.

The addition of porting is done by identifying the parameters and results of each analysis object. Each parameter or result is made into a port on the surface of the object. Users program with PortedObjects by building connections between these ports. 'Programs' are run by feeding data to some of the free parameter ports. Values put in a port are automatically broadcast to all objects connected to that port. When some object in the graph has sufficient input, it processes the data and stores the results in its result ports and so, passes it to the next object. This process continues and data flows through the graph.

5.1 Meta-level Design

The PortedObject meta-level design in CodA is done entirely via modifications to the following five meta-components; Send, Accept, Queue, Receive and Execution.

Send. At the inter-object level, PortedObjects cannot explicitly send messages. They can only store values in their logical output (result) ports. An object cannot tell whether or not storing a result will cause the value to be transmitted to some other object. The meta-level however, can detect the result setting operations and trigger the broadcasting of the new value to all objects connected to the modified port. So, while base-level PortedObjects have no explicit send operations, they implicitly use message sending in their implementation.

A PortedObject's Send behaviour is defined by a generic MultiSend object which provides infrastructure for multi-casting messages to a known set of receivers. For PortedObjects these receivers are represented by ports and connections.

Accept. PortedObjects use MultiAccepts whose behaviour differs from that of normal Accepts only in their support for manipulating ports and connections. This consists mostly of add/remove and connect/disconnect methods in various forms. Also, as messages arrive (via accept:for:), the MultiAccept marks them with the port over which they came and queues them as per normal operation.

Queue. The PortedObject model uses MultiQueues for their Queue component. A MultiQueue supports the sorting of elements into one of many logical queues as defined by some discriminator, in this case, the arrival port. The default Queue interface is augmented with duplicate operations which take an additional parameter, a port identifier.

Receive. A PortedObject's Receive is concerned more with parameter coordination than ports and connections. Some PortedObjects require several inputs to be present before processing can take place. In some cases, processing only makes sense if some set of these parameters are reset from iteration to iteration. In others, a change of one parameter is cause for recalculation. To manage these constraints, PortedObjects use CoordinatedReceives.

When a CoordinatedReceive is asked to receiveFor: by an Execution, it produces the next available message which satisfies the current set of coordination constraints, cSet (see code below). Here cSet represents a very simple system of constraints based on a collection of port identifiers from which it is valid to take a value. As values arrive, their port is removed from cSet. When the set is empty, we know that we have received all the required values and so the object is ready for processing. That is, the receiver is *coordinated*. The initial values for the cSet are derived from information supplied by the programmer as part of the PortedObject definition scheme.

```
CoordinatedReceive>>receiveFor: base
    | message |
    message := base meta queue nextSatisfying: cSet for: base.
    cSet remove: message arrivalPort.
    ^message
```

Execution. Since PortedObjects do not have explicit message passing, we draw a distinction between the implementation receiving and executing a message, and the base-level object itself actually being evaluated. PortedObject evaluation can only happen when the object is coordinated. The messages handled by the Sends and Receives are infrastructure related and serve to transfer data (i.e., parameters and results) and determine coordination.

```
CoordinatedExecution>>process: message for: base
    | method |
    method := base meta protocol methodFor: message for: base.
    self execute: method with: message for: base.
    base meta receive isCoordinated ifTrue: [
        self evaluate: base.
        base meta receive resetCoordinationSet]
```

The main change in a PortedObject's Execution is highlighted by the modified process:for: method shown above. After executing an infrastructure message, the Execution tests for coordination. If the object is coordinated, it is evaluated. After evaluation, the coordination set is reset.

5.2 Compound PortedObjects

In complex PortedObject graphs we would like to be able to think of and manipulate a group of PortedObjects as one. The encapsulation should be completely transparent to objects both inside and outside the group. By taking a generic analysis object and reusing some of the meta-components already described, we can create a *compound* PortedObject as shown in Figure 4.

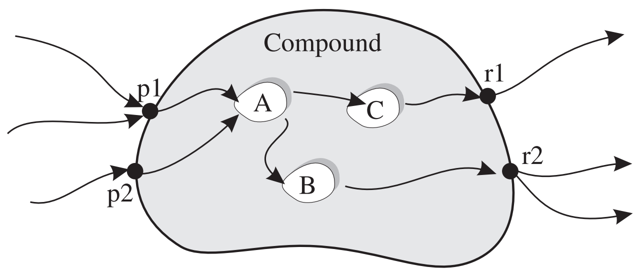

Fig. 4. Compound object example

In the diagram we see three objects (A, B and C) encapsulated in Compound. Compound is itself just a generic analysis object which by default has no ports or particular evaluation behaviour. We have added parameters p1 and p2, and results r1 and r2. The parameters and results are logically linked, as appropriate, to those of the contained objects.

In accordance to the PortedObject model, data values coming to A should come from some PortedObject's Send (e.g., a MultiSend). Compound's Send fits those requirements but it manages the *external* connections for Compound and has no facilities for managing a separate set of *internal* connections. The situation is similar for Compound's result ports and Accept.

An obvious solution is to implement new Send and Accept components which keep two connection lists, one internal and one external. But this would just be duplicating existing behaviour and adding special cases in connection management. An alternative is to use two PortedObjects instead of the single Compound. One would handle the group's parameters and one its results. This however goes against our goal of having the group act as one object.

We take a somewhat novel approach and extend Compound's meta-level to have two new roles, InternalSend and InternalAccept. These roles are actually filled by normal MultiSend and MultiAccept components. Compound's original Accept and Send components remain unchanged and continue to handle all external connections while InternalSend and InternalAccept handle the internal connections. Figure 5 shows the configuration for the parameter side of Compound from Figure 4. Note that p1 and p2 in the two figures are the same.

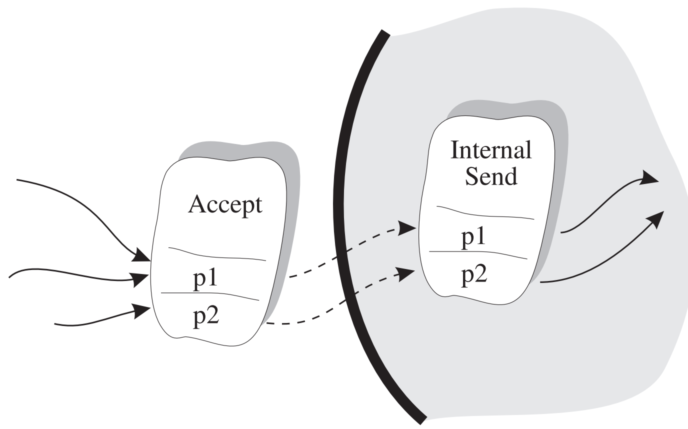

Fig. 5. Compound object parameter handling

Compound's Accept has two ports, p1 and p2, corresponding to its two parameters. Values arriving at those ports are tagged as described above and then passed, at the meta-level, to the corresponding port of Compound's InternalSend. From there they are, as per normal operations, broadcast over the appropriate port's connections to the objects contained in Compound. The structure of the result side is analogous though reversed.

This model is simple and appealing. From a porting and communication viewpoint, all objects have a consistent and uniform model. From a meta-level architecture point of view, it demonstrates how the meta-level is completely extensible and how meta-component defined behaviour is reused. In our original architecture design we never imagined a requirement for having multiple Send or Accept components. In this situation however, it is not only convenient and reusable but is aesthetically pleasing.

6 Putting the Pieces Together

As we have seen, in building object models, we may need to extend the set of roles that the meta-level plays. To do this, we generally create the role and develop at least two components; one which defines some default behaviour and one which defines the new behaviour we specifically want to add to objects. The default component is used when an object's role is accessed but no behaviour (e.g., component) has been explicitly provided. Using this technique, as soon as it is added to the system, all objects have some defined behaviour for the role. Developers are then free to create variations on this default behaviour and substitute them for the default component on individual or groups of objects. For example, the DistributedObject model defines several new roles and we have shown that one of them, Replication, can be filled by three different components; DefaultReplication, MasterReplication and ClientReplication.

Until now we have only discussed how to build object models from scratch. We have not looked at composing new object models from others. The combination of object models in CodA is still largely a manual process but the architecture itself reduces considerably the work to be done.

Combining disjoint (i.e., non-overlapping) object models is straightforward. The new model simply contains the union of the non-default components from the originals. As long as all the components provide the standard CodA interfaces, the new model will run fine.

Combining overlapping object models requires programmer intervention. The general approach is to build new components which merge the behaviour of those being combined. CodA's fine-grained decomposition into objects helps in several ways here. First, the finer granularity gives a more precise indication of where the models collide. Second, the object-orientedness of the decomposition limits both the scope of the conflict and the spread of the change required for its resolution.

Objects also give us an abstraction of behaviour which is easier to use and reuse. Consider two object models X and Y which both redefine the Send component. If we wish to create a model XY (the combination of X and Y) we have to resolve the conflicts between XSend and YSend. If we assume that this resolution creates XYSend, a Send with the properties of both the X and Y models, then the conflict between XSend and YSend has been resolved and need never be resolved again. If the conflict is encountered in the combination of some other models, we can simply reuse XYSend. As we build a library of components, conflict resolution will become more a problem of identifying the existing component with the correct properties than of actually writing code.

The fine-granularity of our design is double edged however. Ad hoc groups of meta-components do not present as nice a package of object behaviour as a single 'meta-object'. Users are faced with a potentially large choice of possible components to fill a particular role. We address this in a number of ways. Relatively simple object models like ConcurrentObjects are represented by methods which configure meta-levels. To use that model, a user just applies the method to the object in question (see the example code below). This simply overwrites any preexisting behaviours.

```
configureAsConcurrentObject: base
    base meta queue: (StandardQueue for: base).
    base meta execution: (ConcurrentExecution for: base)
```

More sophisticated models use base-level language constructs (e.g., classes and metaclasses) as object model representatives. This has the benefit of being integrated with the environment but the drawback of still requiring user-written code.

Using the notion of *properties*, we address both the composition and combination problems. A property is a simple declarative token which points out one way in which a component is different from the default. For example, some of the PortedObject model components are *multi*. Comparing property lists allows us to even more precisely identify conflicts in components. Properties are also used in object model specification. Rather than hardcoding the use of particular components in a model, programmers declaratively specify that, for example, they want a Send with a certain set of properties. Whether a change is required and which actual Send is used is determined dynamically.

Properties, like many categorization systems, suffer from naming problems. Defining and guaranteeing the semantics of a particular property is difficult at best. So, while they do not solve the composition or combination problems, these operations, in a sufficiently rich and consistent component environment, are reduced to property constraint satisfaction.

A completely different approach is component generalization and parameterization. Looking at the models we have defined here, only ConcurrentObjects and PortedObjects overlap in the component domain. Initially we applied the above techniques with success. Then, as we developed other models which also overlapped, we found ourselves generalizing and parameterizing the various components to be more reusable. For example, Executions were changed to take a user supplied code block to define their execution activity. The result was a library of general components which can be setup in many different ways and so can be used in many different situations.

7 Related Work

There are several projects related to our work. We have already mentioned some and relate to a few more below. In general, most previous efforts have either a different focus or different approach with respect to the issues in object behaviour description.

RbCl [6] is similar to CodA in that explicitly supports active objects and factors the meta-level into objects. However, it factors out only a limited set of components and does not provide a framework for their composition and interaction.

The Apertos operating system [17, 18] differs mainly as a result of a different target domain than of the overall architecture. Apertos reifies aspects of object behaviour at the operating system level (e.g., memory management, page faults and device drivers). This level is mostly orthogonal to the current CodA meta-components. It would be interesting to combine the two domains in one framework to get a more complete and far-reaching reification of object behaviour.

Recent work in AL-1/D with distribution control at the meta-level [12] is also similar to ours. They focus on a set of meta-level concepts directly related to distribution requirements. We feel that in fact, in real systems, the issues related to distribution are more far-reaching. They involve heterogeneous state representations and update policies, and demand mechanisms for the control of the intra-object concurrency implicitly introduced by remote referencing. Furthermore, all of this should be possible on a per-use basis. As such, it is appropriate to use a meta-architecture with wider scope. It also appears that the CodA infrastructure and interface is more clearly defined.

Over the years we have been influenced by Actalk [3]. It provides a testbed for describing object behaviours in areas relating primarily to concurrent execution and message passing. Though recent versions are more and more component-based at the meta-level, it is still somewhat monolithic and code-based. Having said that, we are very interested in implementing in CodA, many of the object models available in Actalk.

8 Conclusions and Future Work

By treating the meta-level as "just another application" and applying typical software engineering practices, namely fine-grained decomposition into objects, we have created an extensible, uniform framework for object behaviour description. Object model definition by composition and extension was demonstrated through the development of three object models; ConcurrentObject, DistributedObject and PortedObject. These models show that in CodA, quite diverse object behaviours can be created with relatively minor changes to the meta-level and almost no changes to the base-level.

We have found that the decomposition of behaviour into objects as opposed to code gives meta-level users a higher-level abstraction in which to program. Responsibility for the definition of particular behaviours rests with identifiable, discrete objects rather than being spread through the meta-level code. Individual meta-components have a narrower domain and so are more easily reused. The unruliness of groups of meta-components (as opposed to single 'meta-objects') is addressed via the use of *properties* and the constructs (e.g., classes and meta-classes) available in the implementation environment.

While we have not solved the general object model composition and combination problem, the CodA architecture helps to identify and isolate points of conflict between models. Meta-components provide *firewalls* which limit both the scope of potential conflict and the spread of the changes resulting from its resolution. Components developed to resolve a particular conflict can be reused wherever that conflict occurs so the conflict need only be resolved once. In addition, we propose some simple but potentially useful concepts such as *properties* which ease the burden on meta-level users.

An interesting avenue of future work is to look for techniques which allow the dynamic *compression* and *expansion* of meta-levels. While the separation of meta-level roles into individual objects is logically efficient, it may be less than optimal in implementation. By providing some sort of declarative description of each meta-component's behaviour and expected interactions (e.g., properties), we can automatically combine several components into one. This is analogous to some problems in typing, partial evaluation and code generation. Similarly, by remembering something of their original structure, compressed meta-structures can be expanded into their original form. Using a combination of expansion and compression, monolithic (e.g., compressed) structures can be made open. To change a monolithic meta-level structure, it is first expanded, then changed and finally re-compressed. We believe that this direction holds great promise in the battle to make systems more open and to make open systems more efficient.

9 Acknowledgements

We gratefully thank Jean-Pierre Briot, Pierre Cointe, Nick Edgar and Laurent Thomas for discussions and helpful comments on drafts of this paper.

References

1. G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* Series in Artificial Intelligence. MIT Press, 1986.
2. J. Barnes and P. Hut. A hierarchical $O(N \log N)$ force-calculation algorithm. *Nature*, 324:446–449, 1986.
3. Jean-Pierre Briot. Actalk: A testbed for classifying and designing actor languages in the Smalltalk-80 environment. In S. Cook, editor, *Proceedings ECOOP '89*, pages 109–129, Nottingham, July 1989. Cambridge University Press.
4. Jean-Pierre Briot and Pierre Cointe. Programming with explicit metaclasses in Smalltalk-80. In *Proceedings of OOPSLA '89*, pages 419–431, October 1989.
5. Pierre Cointe. CLOS and Smalltalk: A comparison. In Andreas Pæpcke, editor, *Object-oriented programming: The CLOS perspectives*, pages 215–250. MIT Press, 1993.
6. Yuuji Ichisugi, Satoshi Matsuoka, and Akinori Yonezawa. RbCl: A reflective object-oriented concurrent language without a run-time kernel. In *Proceedings of the International Workshop on Reflection and Meta-level Architecture*, pages 24–35, November 1992. Tokyo, Japan.
7. Gregor Kiczales, Jim des Rivières, and Daniel Bobrow. *The Art of the Metaobject Protocol.* MIT Press, Cambridge, Massachusetts, 1991.
8. Hidehiko Masuhara, Satoshi Matsuoka, Takuo Watanabe, and Akinori Yonezawa. Object-oriented concurrent reflective languages can be implemented efficiently. In *Proceedings OOPSLA '92, ACM SIGPLAN Notices*, pages 127–147, October 1992. Published as Proceedings OOPSLA '92, ACM SIGPLAN Notices, volume 27, number 10.
9. Jeff McAffer. Meta-level architecture support for distributed objects. In preparation.
10. Jeff McAffer and John Duimovich. Actra - An industrial strength concurrent object-oriented programming system. *ACM SIGPLAN OOPS Messenger*, 2(2):82–85, April 1989. Proceedings of the ACM SIGPlan OOPSLA Workshop on Object-Based Concurrent Programming.

11. Robin Milner. The polyadic π-calculus: A tutorial. In *Logic and Algebra of Specification*. Springer Verlag, 1992.
12. Hideaki Okamura and Yutaka Ishikawa. Object location control using meta-level programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, LNCS 821, pages 299–319. Springer Verlag, July 1994.
13. Daniel A. Reed. *An overview of the Pablo performance analysis environment*. Department of Computer Science, University of Illinois, 1992.
14. Marc Shapiro, Yvon Gourhant, Sabine Habert, Laurence Mosseri, Michel Ruffin, and Celine Valot. SOS: An object-oriented operating system – Assesment and perspectives. *Computer Systems*, 2(4):287–337, Fall 1989.
15. Silicon Graphics Inc. *Explorer User's Guide*, 1992.
16. Stardent Computer Inc. *Application Visualization System, User's Guide*, 1989.
17. Yasuhiko Yokote. The Apertos reflective operating system: The concept and its implementation. In *Proceedings OOPSLA '92, ACM SIGPLAN Notices*, pages 414–434, October 1992. Published as Proceedings OOPSLA '92, ACM SIGPLAN Notices, volume 27, number 10.
18. Yasuhiko Yokote, Fumio Teraoka, and Mario Tokoro. A reflective architecture for an object-oriented distributed operating system. In S. Cook, editor, *Proceedings ECOOP '89*, pages 89–106, Nottingham, July 1989. Cambridge University Press.

A Meta-component Specification

The following is a description of the essential meta-components in CodA. Only the protocols needed for actual execution are specified. Others such as those required for configuration are not included as they will vary depending on the capabilities or properties of the component. All specifications are presented in terms of the Smalltalk implementation. The interface also supports a number of convenience methods which combine frequently used operations under one simpler protocol. These are not generally included in the specification. Readers are referred to the body of the paper for explanations of the component's roles.

Note that throughout the interface, the base-object (typically referred to as base) is explicitly specified as an argument. This is done for two main reasons: It allows for meta-components which have a one-to-many relationship with the base-level and it removes the requirement for implicit assumptions regarding the behaviour and arguments associated with an interface. For example, it may not be the case that the *sender* field of a message being sent is actually the object doing the send operation. As a general rule, if a meta-component maintains a one-to-one relationship to the base-level then the base argument is ignored.

A.1 MetaComponent

All meta-components respond to the following messages:

isDefaultBehaviour Answer true if the receiver represents the default behaviour for the role in which it was cast. Answer false if it has been explicitly set by the user.

A.2 Send

Fundamentally there are three different kinds of message sending: synchronous, asynchronous and future. These are realized as different protocols. Variations on the message argument introduce orthogonal concepts such as; express and system messages. Sends also explicitly support the transmission of reply messages as their requirements may be different from that of other message sends.

send: message for: base Send message for base. Defines the default sending behaviour and is typically, though not necessarily, mapped to one of the send operations given below.

send{Async/Sync/Future}: message for: base Send message for base. The sender and receiver are synchronized according to the specified mode (i.e., Async, Sync or Future).

reply: result to: message for: base Reply result to the reply destination listed in message for base. Replies are normal messages but may need to be treated differently to facilitate synchronization and other schemes.

A.3 Accept

accept: message for: base Determine if message can be accepted by base. To accept a message is to promise to consider performing computation based on its contents. It is not an implicit guarantee that the message will be processed but rather that the message has arrived at the destination. The act of accepting a message also involves a preliminary determination of what is to be done with the message. For example, if the message is marked as *express* then it should be considered for immediate execution.

acceptReply: message for: base Replies are normal messages but may need to be treated specially to facilitate synchronization and other schemes.

A.4 Queue

There are a great variety of possible queuing policies and factors in determining in which queue a message should be put and where it should be placed. These policies and factors are generally established via setup parameters on the Queue. The Queue protocol supports methods for enqueuing and dequeuing messages and various forms of message retrieval.

dequeue: message for: base Remove message from the receiver.

enqueue: message for: base Add message to the receiver.

nextFor: base Remove and answer the next available message from the receiver. This defines the default dequeuing behaviour and is typically, though not necessarily, mapped to one of the next operations given below.

blockingNextFor: base Remove and answer the next available message from the receiver. An answer is not given until a message is available.

nonBlockingNextFor: base Remove and answer the next available message or nil if none is available. An answer is always returned immediately.

nextSatisfying: constraints for: base Remove and answer the next available message from the receiver which satisfies the constraints. An answer is not given until such a message is available.

peekFor: base Answer the next available message from the queue or nil if none are available. No messages are removed from the receiver. An answer is always returned immediately.

A.5 Receive

receiveFor: base Answer the next available queued message. This defines the default receiving behaviour and is typically, though not necessarily, mapped to one of the receive operations given below.

nonBlockingReceiveFor: base Answer the next available queued message or nil if none are available. Subsequent calls will not return the same message. An answer is always returned immediately.

blockingReceiveFor: base Answer the next available queued message. Subsequent calls will not return the same message. An answer is not given until a message is available.

A.6 Protocol

methodFor: message for: base Answer the method best suited to processing message. If a method cannot be found then answer some method which will handle the error condition.

A.7 Execution

Executions describe the basic processing activity of an object. How and when they receive, lookup and execute messages. For passive objects this is determined largely by the external thread of control and when other objects send messages to the Execution's base-object(s).

For active objects, the Execution has complete control over these aspects. It must also define what the object does when it is not processing some received message as well as how the object's execution maps onto physical computational resources (e.g., processes and processors). In short, the Execution provides an encapsulation of processing power for the exclusive use of its base-level object(s).

execute: method with: arguments for: base Execute method with arguments on receiver base.

process: message for: base A convenience protocol which combines message to method mapping and method execution. message is processed by first sending methodFor:for: to the relevant Protocol and then execute:with:for: to the receiver.

processImmediately: message for: base Similar to process:for: but the any normal execution currently underway is interrupted with the processing of message.

activityFor: base Answer an evaluable description of base's activity loop.

A.8 State

States describe the physical storage and structure of objects. It is important to note that they do not actually contain the base-level state but simply know how to store and retrieve it. State slots can be named or numbered.

at: id for: base Answer the current value of slot id in base.
at: id put: value for: base Store value in slot id of base.
slotIdsFor: base Answer a list of all the ids for the slots available in base.

B Default Meta-component code

The following are the default implementations for many of the methods mentioned in the body of the paper. They are given as a point of reference so readers can judge the amount of change required to effect the behaviours described.

```
DefaultSend>>send: message for: base
    ^message receiver meta accept
        accept: message for: message receiver

DefaultSend>>reply: result to: message for: base
    | reply |
    reply := message asReply.
    reply arguments: (Array with: result).
    ^reply receiver meta accept
        acceptReply: reply for: reply receiver

DefaultAccept>>accept: message for: base
    ^base meta queue enqueue: message for: base

DefaultAccept>>acceptReply: message for: base
    ^base meta execution processImmediately: message for: base

DefaultQueue>>enqueue: message for: base
    ^base meta execution process: message for: base

DefaultQueue>>nextFor: base
    ^nil

DefaultReceive>>receiveFor: base
    ^base meta queue nextFor: base

DefaultProtocol>>methodFor: message for: base
    ^self lookupTable at: message selector

DefaultExecution>>execute: method with: arguments for: base
    ^method executeFor: base withArguments: arguments
```

```
DefaultExecution>>process: message for: base
    | method |
    method := base meta protocol methodFor: message for: base.
    ^self execute: method with: message args for: base

DefaultExecution>>processImmediately: message for: base
    ^self process: message for: base

DefaultState>>at: id for: base
    ^self slots at: id

DefaultState>>at: id put: value for: base
    ^self slots at: id put: value
```

This article was processed using the LaTeX macro package with LLNCS style