The CodA MOP

A position paper for the OOPSLA '93 Reflection workshop

Jeff McAffer

Department of Information Science
The University of Tokyo
and
Object Technology International
jeff@is.s.u-tokyo.ac.jp

July 27, 1993

1 Introduction

CodA is a framework for creating, controlling and understanding concurrent computing environments. It fosters integration and seamlessness between the meta- and base-levels, between meta-levels and between the different languages which may be used throughout a system. Its essential component is a simple, clear and consistent interface protocol specification for basic system objects (e.g., scheduler, object, metaobject). CodA uses an *operational decomposition* method of defining the meta-level. That is, the meta-level objects are compositions of other objects each of which describes some operation or behaviour. For example, metaobjects are decomposed into objects which describe message sending, message receiving, method lookup, execution and so forth. This is similar in concept to the metaobjects found in RbCl [1] but in CodA the concepts are more clearly specified and are generalized and applied to other meta-level objects such as the scheduler.

CodA embodies a complete system model (i.e., objects, evaluators,...) which is independent of any implementation environment or language. It defines a protocol for inter-object message passing and object execution. We think not in terms of meta-interpretation but of meta-definition. A metaobject is responsible for defining the behaviour of its denotation, not necessarily executing it. That is, there is no meta-interpreter. How the behaviour is described, stored and executed is not explicitly defined by CodA.

In this paper we present the metaobject protocol (MOP) for CodA. Only meta-level components and protocols relating to message passing and concurrency are described. Reification of other aspects such as object structure and name spaces is deferred to future work. In an attempt to make our discussions more concrete, we will, without loss of generality, assume that the underlying implementation environment is Smalltalk. This will also help to show how CodA concepts can be integrated with existing systems. The term *object* is used to refer to both Smalltalk and CodA objects since in most cases they can be used interchangeably. The properties attributed to non-CodA objects are not relied upon for the completeness or robustness of the CodA system.

2 CodA object/metaobject model

CodA specifies that every object have a metaobject which provides definitions of the metaobject protocol (MOP). By default, an object's metaobject is similar to a typical class — an object which defines all possible operations and which is the metaobject for many other objects (i.e., instances). For lack of a better term, we will call these default metaobjects, *classes*, as they provide the object's default behaviour. In reality, the class can be any object including a regular class object from some underlying environment. For example, CodA and Smalltalk are integrated by ensuring that all Smalltalk classes meet the MOP specifications and thus can be used as CodA metaobjects. An important property of such classes is that they not contain unprotected shared state as it is assumed they will be used by many, possibly concurrent, base-objects (instances).

Conceptually, a *metoabject* is a collection of objects each of which defines a particular facet of the base-object behaviour. Here we identify four particular *meta-components* which correspond to our areas

of interest: SendDescription, ReceiveDescription, ProtocolDescription and ExecutionDescription. There may be more or fewer components depending on particular system requirements. Regardless, each provides a definition of an operation (the one specified in the name) required by the base-object. The MOP is a collection of the protocols defined by all possible such components (see Appendix A for the complete MOP specification). Note that components need not be physically distinct from each other but their separation in the protocol allows for this possibility.

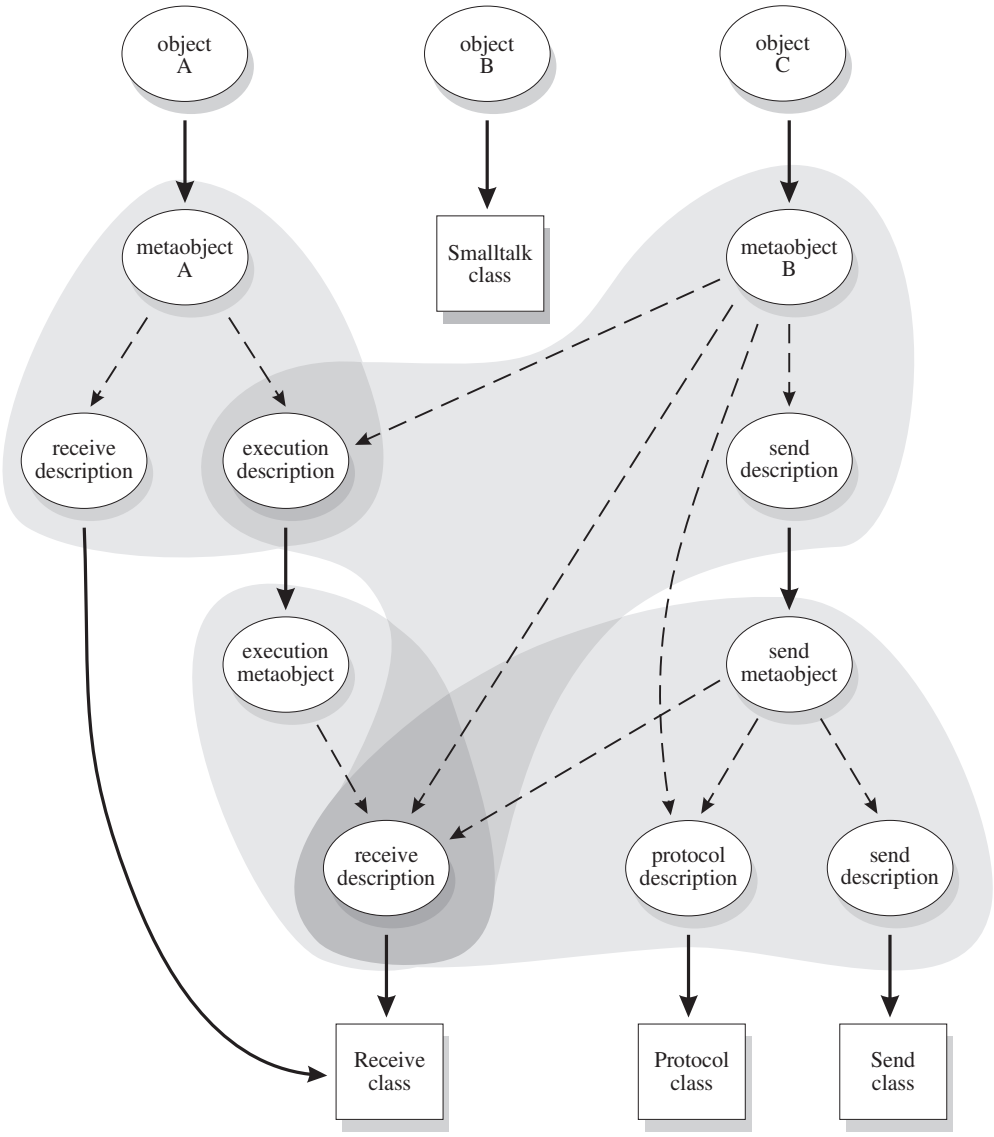

Figure 1: Example meta-component structure

Object behaviour is modified by explicitly associating meta-components with an object. The operations defined by these meta-components override the specifications contained in the object's default metaobject (class). Any object can be a meta-component as long as it responds to the protocol required of its role. Figure 1 shows an example meta-component structure. In the figure, solid lines indicate a base/meta relationship, dashed lines depict meta-component relations and shaded areas delimit the extent of individual meta-component groups (commonly thought of as meta-levels). The boxes are objects which terminate the meta-component chain and implement all require behaviour (i.e., classes).

There are several points to note in the figure. Object B is a regular Smalltalk object, having its normal Smalltalk class as its metaobject. Object A has the behaviour defined by its default metaobject (metaobject A) but with the receive and execution operations overridden. All of C's default metaobject behaviours have been overridden by specialized meta-components. In the diagram there are several places where meta-components are shared between objects. For example, A's executiondescription and C share

their receive description. Meta-components can be at several different *meta-levels* at the same time (e.g., the same receive description is used at C's meta- and meta-meta-levels). All of this (and more) is possible in CodA. The semantic definition of such structures is left to the user.

The CodA metaobject structure eliminates the explicit level-shifting and reflective procedures seen in other reflective systems [3, 4]. Meta-components are the same as any other object in the system — it just happens that their domain is the description of object behaviour. Reflective or meta-level processing is done by communicating directly with the desired meta-components. As a result, the distinction between *levels* is somewhat blurred and is of less importance. In theory there are distinct meta-levels across a given, fully reified, object set but in practice it does not make sense to speak of levels outside a particular *meta-component chain* — what may be a meta-component in one chain may be a meta-meta-meta-component in another. Even within chains, a given meta-component may be at multiple levels. As we see in Figure 1, the resulting reflective structure is more of a *graph* than a linear *tower*. Only the required meta-components are reified and computation descends the chains only as far as, and in the direction prescribed by, the reification.

While in theory, CodA admits an infinite chain of metaobjects, in practice, a metaobject chain will *bottom-out* at a default metaobject which is implemented by some underlying machine. Any such class can be replaced by some decomposed metaobject — essentially growing the reflective chain but only where needed. Since here we assume the underlying machine is Smalltalk, the primitive meta-components are typically Smalltalk classes.

2.1 Example: Future objects

Futures are a useful tool in concurrent systems. They provide an interesting behaviour set which can be implemented trivially using CodA. The following are the steps required to make an arbitrary object into a future. Note that since anObject is just a placeholder it can be a trivial object.

1. Reify the object: anObject reifyMetaobject
2. Set the object's protocol description to describe the protocol of a future:
   anObject metaobject protocolDescription: FutureProtocol new

A FutureProtocol object defines the base-object to have only two methods: #trap: and #setValue:, the code for both is given below. #trap: traps all incoming messages and causes the senders to wait on a semaphore. When some object has a value for the future, it sends #setValue: with the value to the future. This causes the future to change into (become) the value and all of the waiting senders to be signalled. Upon awakening from the wait, the senders resend their messages but this time they are sent to the real value rather than the future object. Note that in this situation, the object (future) is very simple an need not have a fully featured (in terms of potential) metaobject. In fact, the entire behaviour of the object can be defined by one very simple metaobject.

```
trap: message
    gate wait.
    ^self perform: message
```

```
setValue: value
    self become: value.
    [gate isEmpty] whileFalse: [gate signal]
```

2.2 Example: Concurrent objects

CodA objects are sequential or concurrent depending on their execution description. Similarly for an object's meta-components. To make a sequential object into a concurrent object requires that its execution description object be changed. This is accomplished as follows.

1. Reify the object: anObject reifyMetaobject
2. Set the object's execution description to be a concurrent execution description:
   anObject metaobject executionDescription: ConcurrentExecution new

The only difference between the original SequentialExecution and the new ConcurrentExecution is that the new description defines a default activity for the base-object (e.g., what is it to do when it is not processing some received message?). Below is an example of a simple concurrent object's activity. The activity is *forked* as a new thread of control and loops infinitely waiting for messages, looking them up and executing the resolved methods. If incoming messages are to be buffered then the object's receive description object should be modified to perform the required actions.

```
| message method |
[true] whileTrue: [
    message := object metaobject receiveDescription blockingReceiveFor: object.
    method := object metaobject protocolDescription methodFor: message for: object.
    ^self execute: method withArguments: method arguments for: object
```

3 CodA scheduling model

Most of the current meta-architecture research has ignored the potential of reflective scheduling. CodA makes this potential accessible by reifying the scheduler in much the same way as it does the metaobject — by decomposing it on an operational basis. Typical schedulers may have some user-defineable parameters but they are not fully open. CodA schedulers are logically composed of four components: QueuingFunction, Queue, ExecutionFunction and Executor. The scheduling model makes no assumptions about the types of scheduling policies which may be implemented.

Most any scheduler will have some sort of queue and some sort of evaluator, and it is not unusual to find these constructed of discrete objects [2]. However, CodA specifies a minimal protocol set for these components and reifies the determination of the queuing and execution discriminators. Discriminators are objects used by, in this case, the queue and executor to determine how to handle a particular object. A common queuing discriminator is priority. Traditional queues are designed to automatically access the *priority* field of their elements and sort them in some order based on its value. Another common discriminator is arrival time (e.g., FIFO queuing) where where objects are differentiated by their time of arrival.

Unfortunately, such simple discriminators restrict scheduling policies to using only local, instantaneous and typically numerical data. More sophisticated scheduling techniques require knowledge of system- or group-wide state, synthesized representations of object state or event historys. It is the discriminator functions's job to provide this information in a useful form. Discriminator functions are similar to hash functions in that they take an object and produce another object which is a distillation of the input's and system's salient state — the discriminator (hash value). This value *annotates* the object and helps the operator (e.g., queue, hashed set) know what to do with it. Discriminator functions can perform any operations including preventing the object from being queued or executed. They can be fully autonomous and pro-active in their calculation of discriminators.

Systems usually have just one scheduler. CodA provides for an arbitrarily complex hierarchical directed acyclic graph of schedulers. When a scheduler determines that an object should be run, it passes it to one of its *processors* — an object which takes responsibility for executing the scheduled object's code. This processor can be another scheduler or some actual processing resource. Schedulers may control several physical processing resources or only part of one. Note the similarity between this design and the meta-component reification structure.

3.1 Example: Individual and group scheduling

As with metaobjects, schedulers need not be heavyweight objects. They provide a convenient place from which to control the execution of objects relative to other system components. Objects specify, in their ExecutionDescription, the scheduler by which they wish to be scheduled. Individual objects can have their own scheduler which dynamically controls aspects of their execution (e.g., migration and preemption periods). Groups of objects controlled by one scheduler cooperate (in some way defined by the scheduler) to use processing resources and can be coordinated, stopped, started, migrated, etc. by manipulating their scheduler.

3.2 Example: Run-to-completion and Preemptive Scheduling

Within the scheduler architecture we must also be able to implement various scheduling policies in a straightforward and modular way. Below we show the code differences between a scheduler implementing a run-to-completion (RTC) policy (on the left) and one implementing a preemptive policy (on the right). In both cases only the scheduler's Executor need be changed. Here, Executors run a separate thread of control in a loop defined by the #activity method and the Executor>>#stop: method is executed when an object is stopped (say by waiting on a semaphore). The main difference between the two implementations is in how the Executors wait to #run: the next object. The RTC executor waits for the running object to stop (by waiting on the gate semaphore). The preemptive executor waits for either the preemption period to timeout or the current object to stop and detects the difference using the prematureStop flag.

So, with relatively few changes to the code and none to the infrastructure, we can implement schedulers with quite different behaviours.

```
Executor>>activity
    [[true] whileTrue: [
        gate wait.
        self run: scheduler next]]

Executor>>run: execution
    active add: execution.
    processor run: execution

Executor>>stop: execution
    active remove: execution.
    gate signal.
    processor stop: execution
```

```
Executor>>activity
    [[true] whileTrue: [
        self run: scheduler next]]

Executor>>run: execution
    active add: execution.
    prematureStop := false.
    processor run: execution.
    delay time: preemptionPeriod.
    prematureStop ifFalse: [
        processor stop: execution.
        scheduler reschedule: execution]

Executor>>stop: execution
    active remove: execution.
    prematureStop := true.
    processor stop: execution.
    delay disable
```

3.3 Example: Adaptive scheduling

Masuhara proposes an interesting experimental adaptive scheduler in [5]. Very roughly, this preemptive scheduler attempts to run first those scheduled objects with the most waiting messages. An object's *priority* is the scaled difference between its current receive queue size and the standard deviation of the system-wide average queue size for the past M received messages (where M is some fixed constant). The scheduler periodically scans the objects and calculates the new system-wide queue size average and standard deviation and updates all object's priorities. The scheduler's ready-to-run queue is then reorganized to take into account the new priority values. The queue is really two FIFO queues — one for objects with priority over some threshold and one for all others. Objects in the former are run before objects in the latter and are given longer preemptive time periods in which to run.

In CodA the bulk of such a scheduler is implemented with the standard objects discussed so far. We use the preemptive Executor from above and have the preemptionPeriod set dynamically for each object according to which queue it came from (i.e., its priority). The Queue has an obvious implementation as two coordinated FIFO subqueues. The ExecutionFunction is not particularly needed though we could do a more sophisticated calculation of the preemptionPeriod there. The real work of this scheduler is done in the QueuingFunction.

The QueuingFunction must do two things; calculate the *priority* of newly queued objects and periodically scan the queued objects recalculating their priorities. The first operation is implemented directly in #discriminatorFor: as shown below. The periodic update is implemented by forking a new thread in the QueuingFunction which is run (scheduled) by some scheduler closer to the actual processing resources to avoid the scheduler deadlocking on itself. The activity loop of this new *daemon* is shown in the #activity method below. We also need to modify the ReceiveDescription object meta-component to supply the #averageQueueSize method which returns an (estimated) average size of the base-object's receive queue size over the past M messages. Note that #averageAndDeviationUsing: just iterates over the receiver calculating the average and standard deviation of the values produced by evaluating the argument block with each element.

```
priorityFor: anObject using: stats

    ^(anObject metaobject receiveDescription queueSize - stats average) / stats deviation

discriminatorFor: anObject

    ^self priorityFor: anObject using: self mostRecentStats
```

```
activity

    [true] whileTrue: [
        stats := scheduler queue waitingObjects averageAndDeviationUsing: [:object |
            object metaobject receiveDescription averageQueueSize].
        scheduler queue waitingObjects do: [:object |
            priority := self priorityFor: object using: stats.
            object metaobject executionDescription priority: priority].
        scheduler queue reorganize.
        Delay waitFor: someTimePeriod]
```

4 Conclusions and future work

The major contributions of this work to date are:

- A clearly defined MOP which deals with issues of concurrency and message sending
- A plausible and flexible scheduler architecture and protocol
- Demonstrated integration of one object system with another through meta-level manipulation

The remaining work with CodA, as with most meta-architectures, is not so much with the model itself, though CodA is sure to be incomplete at this stage, but in determining how to put it to good use. CodA provides a platform, fully integrated with sophisticated programming tools (Smalltalk), for investigating the use of reflection in controlling and understanding concurrency.

While the examples given above are quite simplistic — describing only ways of implementing behaviours which are not difficult to do without this technology — we are using CodA to explore meta-control techniques which depart radically from those currently available. Fine grained ubiquitous schedulers, object interaction contracts and relations, and message/method resolution strategies are of particular interest.

Another use for CodA is in our work on concurrency monitoring, debugging and analysis. Its open architecture provides access to all the components of system concurrency not only for monitoring but for dynamic behaviour modification using on-the-fly analysis. CodA is also used in the construction and running of the monitoring tools themselves. We have found it to very clean and conceptually simple.

References

[1] Ichisugi, Y., Matsuoka, S. & Yonezawa, A. RbCl: A Reflective Object-Oriented Concurrent Language without a Run-time Kernel. In Proc. of IMSA'92 International Workshop on Reflection and Meta-level Architecture, Tokyo, Nov.4–7, 1992.

[2] Okamura, H., Ishikawa, Y. & Tokoro, M., "AL-1/D: A Distributed Programming System with Multi-Model Reflection Framework," In Proc. of IMSA'92 International Workshop on Reflection and Meta-level Architecture, Tokyo, Nov.4–7, 1992.

[3] Rivières, J. & Smith, B. C., "The Implementation of Procedurally Reflective Languages," Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming, 1984.

[4] Smith, B. C., "Reflection and Semantics in Lisp", In Conference Record of ACM POPL '84, pp. 23-35, 1984.

[5] Masuhara, H., "A Meta-Level Scheduling Mechanism in ABCL/R2", In the Proceedings of the Workshop on Object-Oriented Computing '93, Hakone, Japan, March, 1993.

A Protocols

The following descriptions separate the protocols by CodA component. Notice that throughout the meta-component protocols, the client object is explicitly specified as a argument. This is done for two main reasons — it allows for meta-components which have a one-to-many relationship with their denotations and it removes implicit assumptions regarding the behaviour and arguments associated with a protocol. 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.

A.1 Object protocol

metaobject{:} Get or set the receiver's metaobject. #metaobject is a fixed, primitive, sequential method for all objects.

{de}reifyMetaobject Destruct/build a CodA metaobject for the receiver.

A.2 Metaobject protocol

object{: anObject} Get or set the receiver's denotation if the metaobject is capable of a one-to-one relationship with its denotations. Smalltalk metaobjects by default not capable of this.

sendDescription{: description} Get or set the receiver's send description.

receiveDescription{: description} Get or set the receiver's receive description.

protocolDescription{: description} Get or set the receiver's protocol description.

executionDescription{: description} Get or set the receiver's execution description.

isReified Answer true if the receiver is a CodA (i.e., reified) metaobject. Answer false if it is a default (i.e., Smalltalk) metaobject.

A.3 Send description protocol

send: message for: anObject Send message for anObject. Valid for both sequential and concurrent objects. This defines the default sending behaviour for anObject.

send{Async/Sync/Future}: message for: anObject express: flag Send message for anObject in the mode (e.g., Sync) specified. If flag is true, send the message as an express message. If no mode is specified then the default sending behaviour is used. Valid only for concurrent objects.

send: selector withArguments: args to: receiver from: sender Build a message using the provided parameters and then send it using the default sending behaviour. Valid for both sequential and concurrent objects.

send{Async/Sync/Future}: selector withArguments: args to: receiver from: sender replyTo: replyDest with: replySelector express: flag Build a message using the given parameters and then send it in the mode (e.g., Sync) specified. If no mode is specified then the default sending behaviour is used. If flag is true, send the message as an express message. Valid only for concurrent objects.

buildMessage: selector withArguments: args to: receiver from: sender {replyTo: replyDest with: replySelector} Build and answer a message based on the given arguments. Valid for both sequential and concurrent objects. Sequential objects cannot specify reply information.

A.4 Receive description protocol

A ReceiveDescription defines two sets of behaviours; one which is seen from the outside and one from the inside of an object/metaobject. That is, protocols for *accepting* incoming messages and protocols for *receiving* or fetching accepted messages are required.

accept: message for: anObject Accept the incoming message. This may or may not result in some execution in the receiver's denotation. Valid for both sequential and concurrent objects.

acceptExpress: message for: anObject Accept an incoming express message. This may or may not result in some execution in the receiver's denotation. message is handled with higher *priority*. Valid only for concurrent objects.

receiveFor: anObject Answer the next available accepted message. Valid only for concurrent objects.

blockingReceiveFor: anObject Answer the next available accepted message. Block the sender #blockingReceiveFor: until a message is available. Valid only for concurrent objects.

selectiveReceive: messageSpec for: anObject Answer the next available accepted message which matches messageSpec. Block the sender of #selectiveReceive:for: until such a message is available. Valid only for concurrent objects.

A.5 Protocol description protocol

methodFor: message for: anObject Answer a method suitable for use in processing message. Answer a default method if no other can be found.

methodFor: message for: anObject ifAbsent: absentHandler Answer a method suitable for use in processing message. Answer a method which evaluates absentHandler if no other can be found.

A.6 Execution description protocol

execute: method with: arguments for: anObject Execute method with arguments. Valid for both sequential and concurrent objects.

process: message for: anObject Process message. This form cooperates with the protocol description to find the appropriate method which is then executed. Valid for both sequential and concurrent objects.

processExpress: message for: anObject Process message as an express message. This form cooperates with the protocol description to find the appropriate method which is then executed. Generally speaking, the current activities of the receiver's denotation are interrupted in order to execute the desired method. Valid only for concurrent objects.

A.7 Scheduler protocol

stop: Remove any potential of the indicated object being executed.

run: Create potential that the indicated object will be executed.

restart: Re-create potential that the indicated object will be executed.

next Answer the next runnable object. This removes the object from the queuing mechanism.

peek Answer the next runnable. This does not remove the object from the queuing mechanism.

active Answer the objects which are currently running (i.e., active)

interrupt:with: Interrupt the execution of a scheduled (may be active) object with an executable object to be evaluated immediately. The object's position in any queues does not change.

start/stop Start or stop the operation of the receiver.

A.8 QueuingFunction and ExecutionFunction protocol

discriminatorFor: Answer a description of how the argument should be queued/executed.

potentialDiscriminatorFor: Answer the same information as #discriminatorFor: but only in a *what-if* context. That is, do not change any internal state of the receiver based on this request.

A.9 Queue protocol

dequeue: Remove the indicated object from the receiver.

enqueue: Add the indicated object to the receiver.

next Answer the next object available. This removes the object from the receiver.

peek Answer the next object available. This does not remove the object from the receiver.

A.10 Executor protocol

run: Run the indicated object's code immediately (from the view point of the receiver). That is, the object is passed to the processing resource for execution.

stop: Stop the execution of the indicated object indefinitely with direct potential of it running again.

interrupt:with: Interrupt the execution of an active object with an executable object to be evaluated immediately. The object's position in any queues does not change.