# Adding distribution to general applications: The meta-level approach

**Jeff McAffer**
Department of Information Science
The University of Tokyo
and
Object Technology International

Email: jeff@acm.org

## Abstract

Complex applications may contain elements from several different computational domains (e.g., concurrent, distributed). Class library developers cannot anticipate the domains in which their objects will be used and application developers often do not know how their applications will behave in these environments. To handle such situations we must be able to prototype and modify the behaviour of objects and applications. Traditional adaptation and prototyping techniques currently require intrusive changes to an object's code to describe new behaviour. By making an object's computational behaviour descriptions explicit and distinct from its application domain behaviour we can adapt it to many different computational styles without intrusive code changes. This increases its (re)usability and allows safer and easier prototyping. We present CodA, a *meta-level architecture* which supports such a separation and detail its use to add concurrency and distribution to an *unsuspecting*, non-trivial commercial application with relatively few changes to its code.

## 1 Introduction

With the widening acceptance of object-oriented design and computing, the demands placed on software developers are changing. One of the promises of object-oriented computing is highly modular and reusable packages of functionality (e.g., class libraries). The traditional strategy for realizing this is the *black box*. The black box view holds that, class libraries should be opaque to the user/client who should not need to see inside the box to solve their problems. In practice, this is not always possible or desirable.

It is impossible for library designers to know or guess all possible uses of the classes they build. Clients often want to view and use the same class in several different ways. By enforcing the black box paradigm on our software modules we are forcing users to conform to the designer's view of the world. This often requires them to leap through very high hoops to do relatively simple things. Most often we want to reuse object semantics but not necessarily their *computational* behaviour. If object library designers are to meet this demand, they must be able to create objects which work in several computing domains (e.g., sequential, distributed, concurrent)[1] with relatively little change.

It may seem exotic to assume that objects will really be placed in such situations but as system scope grows wider and deeper, the nature of the computational landscape changes. It becomes more diverse and dynamic. Requirements for the same objects to run on mainframes, large-scale multiprocessors,

[1] The architecture is by no means limited to just these three. We have experimented with and reported on other computational models in (McAffer 1995a).

workstations, embedded controllers and personal digital assistants (PDAs) exist today and the spectrum will only widen in the future.

Many software development environments support the design and prototyping of an object's domain behaviour. In this new, larger scoped world, designers of both class libraries and applications need support for prototyping their objects' computational behaviour. The major problem in this area is that the computational behaviour of objects is typically hidden. It is either implicit in the language environment or is defined inside an object's black box.

If an object's computational aspects were distinct from its domain-specific aspects, they could be manipulated separately. The same object could be used in many different ways, adapted to changing requirements and interaction between objects using disparate models could be facilitated. Moreover, if we had a generalized framework for modeling these behaviours, much of the work of their combination and interaction would be eliminated.

This is one of the goals of researchers in meta-level architectures, reflective computing and more generally, *open implementations* (The FOIL working group 1994). A fundamental tenet of this work is that the so-called *base*-level (domain semantics) and *meta*-level (computational semantics) descriptions of an object are distinct but causally connected. They run different code, have different environments and different subject matters but changes at the meta-level affect the behaviour (i.e., execution) of the base-level. The base-level defines the domain-specific behaviour of the object while its meta-level defines the computational behaviour. This causal connection is what helps us maintain sanity in these systems and is the basis for the distinction between the levels.

Many people have been unknowingly using this approach for years. Recently there have been attempts to codify these ad hoc practices and the work in meta-level architectures and reflection. The result is the emergence of open implementations as a new field of research. Open implementations trade-off the clarity and simplicity of the black-box model and the freedom and flexibility of completely unprotected systems. They allow designers to expose, in a controlled way, the underlying implementation of their objects. The implementation can then be manipulated by the user to better conform to specific situations.

Existing formalisms of these ideas (i.e., meta-level architectures) are founded on base-level language structures such as classes and inheritance. Since they are limited to the scope of the base-level language, they do not support a broad range of behaviour descriptions. This issue becomes more and more important as systems get larger and more complex. In such systems programmers need to mix various computational models but yet have the parts of the system interact and interoperate.

We have developed the CodA meta-level architecture to help remove these limitations. CodA's basic design is language model independent. The architecture can be used to describe object behaviours from many different languages as well as behaviours which are not language specific (e.g., distribution).

The use of these techniques to provide an environment for prototyping object and application behaviour in a wide range of domains is described in the following section. In Section 3 we detail the application of this architecture to the reuse of the domain behaviour of a commercial sequential uniprocessor expert system tool (and some example knowledge bases) in the concurrent and distributed computational domains. A final section summarizes our work and points to some directions for future work.

## 2 The architecture

We have developed CodA(McAffer 1995c; McAffer 1995a), 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 objects behave computationally. Unfortunately, full disclosure of the architecture's low-level details is beyond the scope of this paper. Instead we summarize its relevant features and properties and refer interested readers to the the more complete descriptions in (McAffer 1995c; McAffer 1995b; McAffer 1995a).

The main goal of this work is to separate *what* an object does (its *base-level*) from *how* it does it (its *meta-level*). If we can do this then we can alter, and thus prototype, the computational characteristics of objects in isolation of their base-level semantics. Sequential objects can be made concurrent, state storage strategies changed, messaging mechanisms abstracted, etc. Our goal is to allow these changes to be done with relatively low impact on an object's base-level code.

The CodA architecture opens or exposes an ob-

ject's implementation and provides a framework for its manipulation. While other systems such as the CLOS MOP (Kiczales, des Rivieres, and Bobrow 1991), ABCL/R2 (Masuhara, Matsuoka, Watanabe, and Yonezawa 1992) and ClassTalk (Briot and Cointe 1989; Cointe 1993) take a *structural* approach to meta-level decomposition, CodA takes an *operational* approach. In CodA, the meta-level is decomposed into a number of *behaviours* each of which corresponds to a particular *operation* required for an object to execute. For example, it includes behaviours for; method execution (both mechanisms and resources), message passing, message to method mapping and state maintenance. Each of these behaviours is described by a *meta-component*, an explicit and distinct object.

Meta-components are typically small and simple objects which define a clear *programmer's interface* (what some might call a MetaObject Protocol or MOP). They provide specific definitions or implementations of the behaviours they describe. An object's characteristics are changed by explicitly redefining or modifying the meta-components which describe its behaviours or by extending its set of behaviours. A system may contain many different components for the same behaviour but only one can be used on a particular object at a particular time. It is also possible for objects to share components.

As a basis, CodA defines a default set of seven behaviours. These seven define all of the operations needed for basic generic object execution. It is fully expected that programmers will add to this list while implementing new models of object execution and CodA provides an extensible framework for doing this. The default behaviours are:

Send Defines how an object handles outgoing messages. This includes protocol negotiation and synchronization.

Accept Implements the external interface for incoming messages. Accepts interact with the sending object's Send to determine if a message is valid and concerns the base-object. Note that this is different from receiving.

Queue Organizes and holds messages which have been accepted but not yet received or processed. Queues are the main mechanism for decoupling senders and receivers.

Receive Defines the operations related to fetching accepted and queued messages for processing. To receive a message is to consider processing it. Even though a message has been accepted by the Accept, it can later be ignored by the Receive. When asked for the next message to process, a Receive may consider messages from many queues and various synchronization constraints before choosing a particular message.

Protocol Maps messages onto methods for execution using some scheme (e.g., inheritance).

Execution Describes how an object executes methods both in terms of the execution model and the processing resources.

State Defines what state slots an object has and how the slots are accessed. Note that a State *does not* actually hold the data, it simply knows how it is manipulated.

### 2.1 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 (i.e., the meta-level) for a particular object. 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. Note that this example assumes $B$ is a concurrent object (see below).

We see that $A$ sends $M$ by interacting with its 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). During the execution of the method, the receiver's ($B$'s) state slots may be accessed via its State component (8). In this way, every aspect of basic execution is reified.

### 2.2 Object models

This approach to decomposition provides a highly refined and expressive environment for defining object behaviour but it is too low-level for programmers to

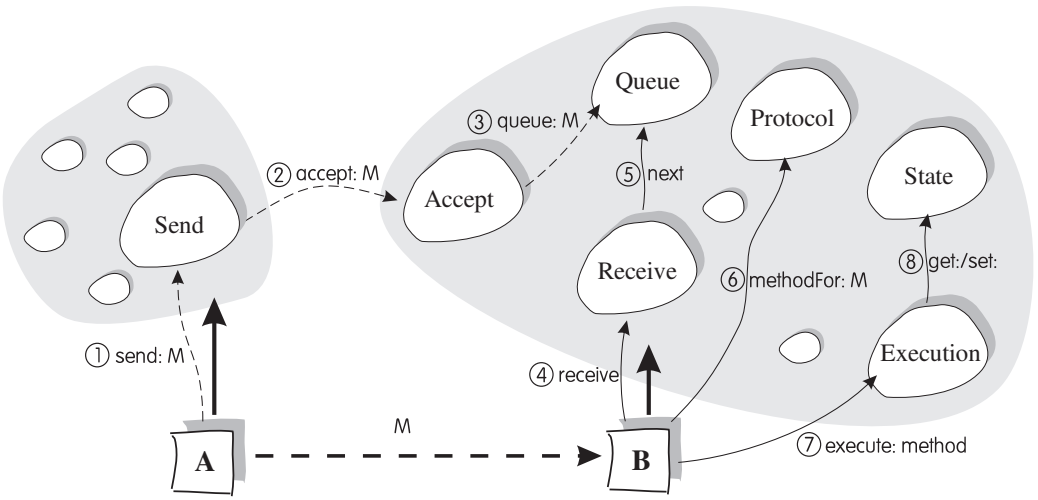


Figure 1: Sample meta-level configuration and interaction

use. It requires too much of users in terms of definition and configuration of the meta-level. Normal systems have language-based structures (e.g., classes) which are used to organize the meta-level (e.g., metaclasses). This is too restrictive as it limits meta-level defined behaviour to the scope of the base-level language. Instead, we adopt a more generic approach, *object models*, in which users can abstract logical behaviours and manipulate them as concrete entities at the meta-level.

Sets of meta-components are grouped together into object models. Object models represent higher-level concepts such as classes, concurrency and distribution. Their main role is to specify configurations of inter-related meta-components which form some logical or coherent behaviour. Some models have descriptions of one or two behaviours while others specify many. The ConcurrentObject model is a good example.

ConcurrentObjects are objects which have their own processing resources or thread(s) of control. Object *B* in Figure 1 is an example. A general object's processing resources are described in its Execution. By defining a new Execution component which allocates the appropriate OS process resources, an object is made *active* (as opposed to *passive*). ConcurrentObjects actively choose which message to handle next and when to handle it. They process messages autonomously with respect to other objects. Since messages may not be handled immediately, ConcurrentObjects must also have a way of buffering waiting messages. Rather than using the default Queue behaviour which immediately triggers message processing (i.e., does not queue messages), we use a shared Queue component which stores messages from multiple senders for later retrieval.

With changes to these two behaviours a passive object can be converted to an active object. The ConcurrentObject model itself is a realization of the relationship between the Execution and Queue for active objects. It specifies the constraints that the Execution be *active* (i.e., allocate a thread) and that the Queue be *shared* (i.e., be protected from multiple accesses).

Models can of course specify constraints more precisely but the more general the specification, the more easily the model can be applied and combined. Complex meta-levels are constructed by combining several object models which describe different aspects of object behaviour. The architecture provides some implicit and explicit mechanisms for managing the combination of models whose domains overlap but this will not be discussed here. Interested readers are referred to (McAffer 1995c; McAffer 1995a) for more details.

## 2.3 Distributed objects

Using the CodA architecture we have developed a distributed object system called *Tj*. *Tj* is a series of object models related to distributed systems (e.g., DistributedObject, ReplicatedObject and MigrantObject). In *Tj*, objects live and execute in

*spaces*. Due to CodA's separation of state and execution, an object's Execution and State can live in different spaces and can be independently copied, replicated and moved. Objects communicate via inter-space messages.

In a particular space, remote objects are represented by RemoteReferences which are similar to Proxies (Shapiro, Gourhant, Habert, Mosseri, Ruffin, and Valot 1989). An object's location is transparent to itself and its clients. Local to their reference, RemoteReferences 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 the message to the space containing the real receiver — the *target*. More sophisticated RemoteReferences process some messages locally while forwarding others to the target.

RemoteReferences are themselves implemented using modified CodA meta-components as shown in Figure 2. 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 distributed case, these meta-components are in different spaces. Local to the sender, the receiver is a RemoteReference (e.g., $B'$) and its Accept is an intelligent RemoteReference to the Accept of the target. Rather than performing the normal accept operation, the local Accept (RemoteAccept) *marshals* the message into a stream of bytes and transmits it to the remote space. Once there, the message is reconstructed and accepted by the real Accept. In this way, the DistributedObject model is uniformly applied to all objects in the system, even to those of the meta-level architecture in which it is implemented!

*Tj* contains a uniform and extensible mechanism for marshaling objects to be transmitted between spaces. Every object having the DistributedObject model has a Marshaling behaviour which defines the transmitted shape of the object, a *marshaling descriptor*. The shape is based on the characteristics of the object's use, state, class and the system as a whole. The marshaling framework allows messaging code to be declaratively annotated to specify how arguments and return values should be passed (e.g., by value, reference, move, replica, ...).

In addition to the basic DistributedObject model, the *Tj* system contains models for ReplicatedObjects and MigrantObjects. These models add *new* behaviours to the meta-level of an object. ReplicatedObjects offer a sophisticated mechanism for replicating object state in any number of spaces and applying a variety of update and coherency schemes. MigrantObjects dictate the location of a object's state and execution. They can monitor object activity (execution and accessing) and dynamically determine the best location according to some user-supplied metric. Alternatively, objects can be explicitly moved under user guidance. The models are integrated with the marshaling framework mentioned above allowing objects to be passed as replicas or migrants. These models are orthogonal and can be combined freely. They are largely independent of other object behaviours and so can be applied to any object. They are also transparent to the base-level. Objects are unaware of their location and the location of others.

## 2.4 Implementation

CodA and *Tj* have been implemented in Smalltalk and run on clusters of workstations and large-scale (1024 processor) multiprocessor machines such as the Fujitsu AP1000. 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 CodA meta-level infrastructure. All Smalltalk objects can have their behaviours redefined using CodA.

In the default case, objects operate under CodA's DefaultObject model which matches the semantics of the objects in the implementation environment (e.g., Smalltalk). Programmers use the system by building application objects normally and then modifying their meta-levels as desired. Objects generally retain their base-level semantics but gain some additional or different behaviour such as concurrency or distribution.

Using *Tj*, objects are transparently made available to other objects on remote processors. Most existing code continues to run though perhaps with some additional overhead due to remote messaging. Users can tune the object distribution graph and mechanisms to reduce or eliminate the overhead for their particular use-cases.

It is this mode of operation which we wish to support. Applications and class libraries are written independently of the environment in which they will run and users adapt the objects to fit their use-cases. The CodA architecture eases the adaptation and thus makes objects more flexible and (re)usable.

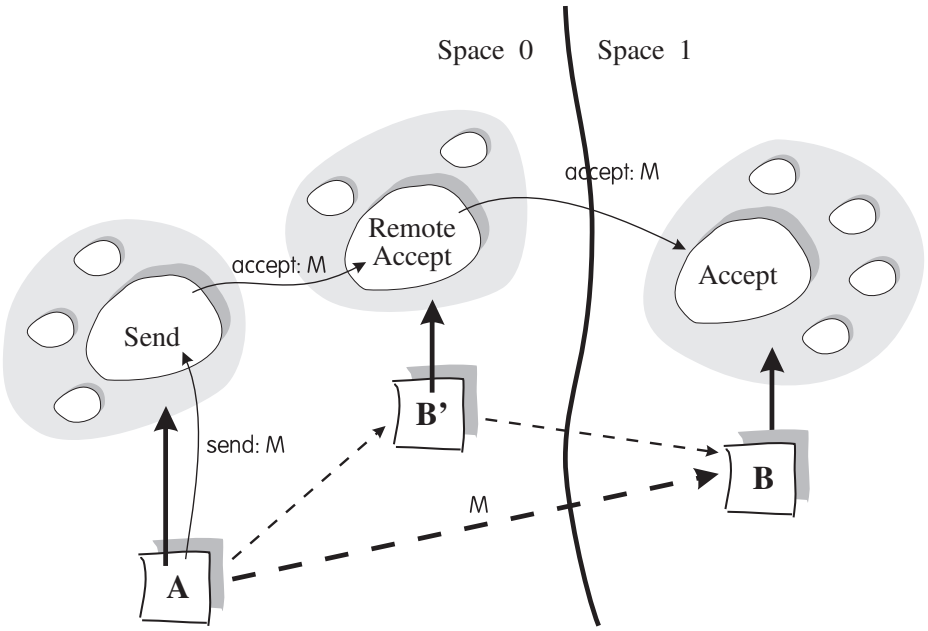


Figure 2: RemoteReferences and the meta-level

## 3 Example: An expert system

We have found that CodA and *Tj* are easily and usefully applied to real-world systems. In particular, we selected a forward chaining rule-based production tool based on the Rete pattern matcher(Forgy 1982) as a test application. The implementation we chose, **ENVY**[2]*/Expert*, is a commercial product written by a third-party. This 'arms-length' implementation affords us an opportunity to investigate the impact of our changes to base-level code in a more objective fashion.

The basic layout of the application is shown in Figure 3. An *expert system* consists of a collection of objects or facts (*factbase*), a collection of rules (*rulebase*) and an *inference engine* which processes this knowledge. The rulebase is a Rete pattern matching graph with one input node and many output and internal nodes. The internal nodes represent the individual tests in the clauses of the rules. Each output node represents a single path through all of the predicates for one rule.

When new *facts* are discovered, they are wrapped in *tokens* and injected into the root of the pattern matching graph. At each node the token is optionally merged with other waiting tokens and tested. Tests are done by executing some Smalltalk code on the facts in the token. If the tests fail, that branch of matching is terminated. If the tests succeed, the token is passed on to following node(s). Tokens making it to an output node represent a set of facts or a *context* which satisfy all the predicates for one path through the clauses of a rule. That rule is activated (added to the *agenda*) with its satisfying context.

The inference engine is somewhat disjoint from the pattern matching. Its job is to iteratively take activated rules off the agenda and execute their conclusions in their activation context. Doing this may add or remove facts, activate or deactivate rules and execute arbitrary Smalltalk expressions. Execution of the inference engine continues until there are no more activations to be processed.

### 3.1 Adding concurrency and distribution

To introduce concurrency and distribution behaviours to this system, the first step is to enable the assertion of facts which are not local. This is straightforward with *Tj*. Since object location is transparent we simply create fact objects in, or move them to, several different processors. While the rulebase and inference engine remain centralized, rule antecedent satisfaction is partially distributed. Since antecedents are tested by sending messages to facts, fact distribution implies predicate execution distribution. The matcher and its infrastructure however are still centralized and continue to work as before.

Unfortunately, when a fact is introduced, the Rete

[2]**ENVY** is a registered trademark of Object Technology International Inc.

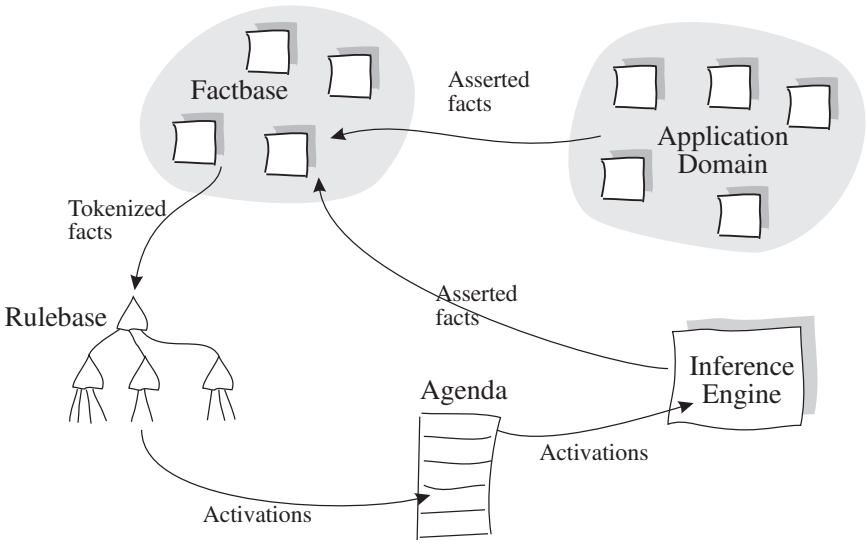


Figure 3: Overview of **ENVY**/*Expert*

pattern matcher does some initial sorting based on the fact's class. This induces many class messages. As most interesting facts are mutable objects, they only ever exist remotely as references. Since these class messages are actually remote messages, they contribute a significant amount of messaging overhead.

Using CodA's flexibility we create a variation of the RemoteReference object which caches the class of a remote object in its local reference — A sort of intelligent proxy. When an object is asserted as a fact, its Marshaling behaviour is modified to specify that the object should be exported as a ClassedReference (i.e., class cached remote reference). This is a simple change which eliminates all of these class messages at the cost of just 4 bytes per remote reference object. Note that since RemoteReferences are unique within a space, this is just $4 * N$ bytes where $N$ is the number of spaces in the system.

The next step is to add distribution to the expert system's inferencing mechanisms (e.g., engine, Rete nodes and agenda). Doing this decouples the nodes of the pattern matching network from each other and from the inference engine. This increases the available concurrency and enables behaviours such as migration of entire or partial knowledge bases.

Structurally there are no barriers to running all of these elements on different processors. However, this implicitly adds parallelism and raises the possibility of concurrent access problems. The most effective means of protecting an object from concurrency is to make it concurrent (active) itself. Active objects have explicit control over which threads execute their code and so are implicitly protected. An object is made active by adding the ConcurrentObject model to its meta-level. Variations on this allow the specification of the object's protected (execution controlled) methods and default activity. The changes required for this are discussed in a later section.

Once the application elements are decoupled and protected, they can be freely distributed. The form of this distribution depends on the nature of the knowledge. For example, paths through the node graph represent paths through individual rule antecedents. One rule may have many paths to satisfaction due to OR operations. If the corresponding Rete nodes are clustered on the same processor, techniques such as caching are more effective. Matching graphs may also have distinct subgraphs of related rule antecedents which should be clustered or distributed. Similarly, if particular kinds (e.g., classes) of facts are relevant only to a certain subset of rules, facts of those types should migrate or be created near the nodes which test them.

The list of possible optimizations is long. The key point is that these techniques are transparent in *Tj*. They can be applied independently of the base-level (e.g,. pattern matcher) code and varied to find the best behaviour for a given situation.

### 3.2 Changes to original code

A main thrust of this paper is to investigate the nature of the code changes required when adding and varying concurrency and distribution as described above. To increase the objectivity of our experiment we chose an *unsuspecting* implementation — one which was not designed with concurrency, distribution or CodA/*Tj* in mind. The expert system used here is a commercial product, **ENVY**/*Expert*, implemented in Smalltalk. It was written strictly as a uniprocessor, sequential package taking no account of the situation in which we use it. In total it consists of approximately 100 classes and 1500 methods. Of these, some 80 classes and 1000 methods are directly related to the execution and support of inferencing.

One of our goals in building our architecture was to enable and foster the clear separation of base- and meta-levels. Success or failure in this respect can be seen by looking at the number and type of modifications to the original base-level code needed to accommodate changes at the meta-level. We classify these changes into the four categories shown below.

**Annotations** are modifications which do not alter the base-level semantics of a method or the messages (contents or order) it sends. The annotations add information which can be used by the meta-level to better adapt the actual behaviour of the object. For example, marking a message as asynchronous is noting that the result is not used and so there is no need to wait for its arrival. In some cases, the passing of additional arguments may be considered an annotation. For example, object creation methods in distributed systems often require an additional location argument. This information does not change the semantics of the sender or receiver. It just adds additional information typically used in an annotation change to the receiver. Note also that meta-level code often takes the form of an annotation. It does not execute at the base-level and does not in general alter the order/semantics of the base-level code itself.

**Semantic** changes are changes to the content or ordering of the messages sent by a method. These are required when the original design precludes the addition of our new behaviour (e.g., concurrency and distribution).

**Structural** changes are typically done to expose operations which are embedded in methods. For example, assume that some method, as one part of its implementation, iterates over a set of objects. In a distributed implementation it may be necessary to separate out this iteration for remote use. One could argue that these changes are only required because the original designers somehow failed to adequately factor their code. In reality however, architects cannot design for all possible uses and so we should expect to make some structural changes.

**Additional** methods contain completely new semantics. Typically these are needed to handle infrastructure added in support of new behaviours. If an object's representation is changed, it may require additional methods in support of its original interface. Methods which provide new interfaces which are combinations of existing methods are also counted as additional.

Ideally most changes are made as annotations or additions since these have the least impact on the base-level. Structural changes can often be done with relatively little impact but may introduce programmer errors or incompatibilities between versions or configurations of the application. Semantic changes are not necessarily bad but they do affect the portability of objects between models and may change the correctness of the object in its domain.

The changes and additions required for the final concurrent and distributed version of **ENVY**/*Expert* are summarized in Table 1. As shown, the only changes required were annotations and additions. The bulk of the annotations were declarations of meta-level structures used automatically by the CodA architecture. For example, the CodApublicMethods method shown in Figure 4 which declares a class' public interface. Although this type of annotation is implemented as a new (i.e., added) method, it is not part of the object's call chain (i.e., it is only used by the architecture) and is counted as an annotation to the class.

Figure 5 gives an example of a change to an existing system annotation. The upper method shows the annotation which defines the default activity block (i.e., what an object does) for all objects. The lower method defines the activity for an InferenceEngine. The changes (indicated by an arrow (->) introduce a new execution branch for when the engine is enabled for activation execution. The net effect is that while normal objects suspend when there are no more messages to process, InferenceEngines pull the next available activation off the agenda and run its conclu-

| | Total | Annotation | Semantic | Addition | Structural | Summary |
|---|---|---|---|---|---|---|
| Classes | 80 | 14 | 0 | 7 | 0 | 15 |
| Methods | 1000 | 18 | 0 | 16 | 0 | 34 |

Table 1: Summary of expert system changes and additions

```
InferenceEngine class»CodApublicMethods
  "Declare the public interface for InferenceEngines"
  ^super CodApublicMethods
    addAll: #(resume run step stop suspend stepEngineWithActivations:
      acceptNewObject: modifyObject: removeObject:);
    yourself
```

Figure 4: An inference engine's public methods

sions (i.e., runOneStep). The normal execution path is essentially duplicated in both branches.

The changes discussed here are representative of those required throughout the application. Some are simple declarative specifications while others are subtle modifications default behaviours. Since they mostly execute at the meta-level, there is little chance for conflict with the base-level.

### 3.3 Experiments

Having applied our techniques to the expert system tool itself, we also experimented with the application of the new concurrent and distributed tool to several representative knowledge bases. In particular, the Waltz line labeling system (Brant, Grose, Lofaso, and Miranker 1991).

Waltz is a system for aiding in the 3-dimensional interpretation of 2-dimensional line drawings. It transforms sets of 2D lines into sets of vertices by injecting some knowledge or meaning into the line topology. This knowledge is contained in a set of rules. The rulebase has some 33 rules which manipulate 9 different types of fact objects (e.g., lines, junctions) and their components.

The calculations and comparisons done by Waltz are relatively simple. As lines are analyzed, Waltz creates intermediate and junction objects (facts) representing the convergence of two or three edges and adding a dimension. A variation of this system, Waltzdb, generalizes the Waltz solver to handle junctions of more than three lines. It uses a more abstract representation of the knowledge of junction construction and contains slightly more rules. Both Waltz and Waltzdb are standard benchmark programs for rulebased production systems (Brant, Grose, Lofaso, and Miranker 1991).

While Waltz, from a domain point of view, is not particularly motivating, it is representative of applications which execute a known set of queries interleaved with data update transactions (e.g., databases). Execution begins with one set of facts (lines) and produces another set (junctions). Along the way, intermediate facts such as edges and flags are manipulated.

Distribution is added to the domain by making a simple modification to the initialization mechanism. This change distributes the initial factbase over the machine topology. The inference engine however is still centralized. Since the engine runs all the rule conclusions, which in turn create new facts, and since new objects are by default created local to their creator, the distribution is less than even.

To increase the distribution of new facts, we annotate the rules which create them to explicitly specify their location. So for example, intermediate edges are created on the processor which contains the majority of its associated lines. Since rule conclusions are just Smalltalk code, these changes have the same form as those done to the inference engine itself. Figure 6 show an example of such a modification. The modified statements are indicated by an arrow (->). Note that using a normal new message creates the edge on the processor executing the method. That is, the one running the inference engine. The newIn: allows us to specify the location of the new edge to be the same

```
DefaultExecution»CodAactivityBlockFor: anObject
   "Describe the default execution pattern for active objects"
   ^[| message result |
   [true] whileTrue: [
      message := anObject meta receive receiveFor: anObject.
      result := self process: message for: anObject.
      anObject meta send reply: result to: message for: anObject]

InferenceEngine class»CodAactivityBlockFor: anObject
   "Describe the basic execution pattern for active InferenceEngines"
   ^[| message result |
    [true] whileTrue: [
->      anObject isInRunningState
->         ifTrue: [
              message := anObject meta receive nonBlockingReceiveFor: anObject.
->             message isNil
->                ifTrue: [anObject runOneStep]
->                ifFalse: [
                    result := anObject meta execution process: message for: anObject.
                    anObject meta send reply: result to: message for: anObject]]
           ifFalse: [
              message := anObject meta receive receiveFor: anObject.
              result := anObject meta execution process: message for: anObject.
              anObject meta send reply: result to: message for: anObject]]]
```

Figure 5: An inference engine's activity

```
   makeEdges
      "Construct an Edge fact for every WaltzLine."
      | WaltzLine aLine |
      aLine notNil.
   PRIORITY 10
   ACTIONS
      |edge1 edge2|
      edge1 :=
   ->    (WaltzEdge newIn: (aLine meta state spaceFor: aLine))
            startPoint: aLine point1 endPoint: aLine point2.
      edge2 :=
   ->    (WaltzEdge newIn: (aLine meta state spaceFor: aLine))
            startPoint: aLine point2 endPoint: aLine point1.
      edge1 assert.
      edge2 assert
```

Figure 6: Example modified rule

as that of aLine.

## 4 Conclusions and future work

We have presented an overview of the CodA meta-level architecture and the *Tj* distributed object system and detailed their roles in prototyping the behaviour of objects and applications. CodA uses a fine-grained operational decomposition of the meta-level into objects. This approach, in addition to fostering a clear separation of the base- and meta-levels, allows the description of a wide range of object models (e.g., passive, active and distributed objects) within the same framework. Though we have discussed only concurrency and distribution, the CodA architecture is by no means limited to just these. We have developed other models in the areas such as inter-object communication, intra-object synchronization and language construct modeling to name but three. Interested readers can find information on these in (McAffer 1995c; McAffer 1995a; McAffer 1995b).

We have also detailed an example showing the steps required to introduce concurrency and distribution to a third-party expert system tool. This application was interesting because the original base-level program was large and not specifically designed to run in this environment (CodA and *Tj*) or these computational domains (concurrent and distributed). With relatively few modifications we were able to introduce both concurrency and distribution. The notions added were quite straightforward in nature but demonstrate the viability of our approach. More sophisticated mechanisms are added in an analogous fashion.

These strategies and others are made possible by using a uniform, transparent, non-invasive mechanism for introducing distribution and concurrency. Sophisticated and deep modifications of object computational behaviour can be added to unsuspecting applications with relatively minor changes to their code. Complete integration with its implementation environment allows CodA to be applied to tertiary systems which function with entirely different base-level paradigms (e.g., rulebased production systems).

The architecture, models and applications we have outlined are real and running but are not ready for 'prime time'. There are still many issues to be addressed in terms of meta-level architecture implementation efficiency and safety. However, the current performance levels are sufficient to allow the use of behaviours involving moderate amounts of meta-level activity. We believe that our architecture inherently supports a range of optimization and implementation techniques which will reduce the overhead of the flexibility we provide. This is one of the major directions for future work.

## 5 Acknowledgements

We gratefully thank Jean-Pierre Briot for discussions and helpful comments on drafts of this paper.

## References

Brant, D., T. Grose, B. Lofaso, and D. Miranker (1991). Effects of Database Size on Rule System Performance:Five Case Studies. In *Proceedings of the 17th International Conference on Very Large Data Bases (VLDB)*.

Briot, J.-P. and P. Cointe (1989, October). Programming with explicit metaclasses in Smalltalk-80. In *Proceedings of OOPSLA '89*, pp. 419–431.

Cointe, P. (1993). CLOS and Smalltalk: A comparison. In A. Pæpcke (Ed.), *Object-oriented programming: The CLOS perspectives*, pp. 215–250. MIT Press.

Forgy, C. (1982). Rete: A fast match algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence* (19), 17–37.

Kiczales, G., J. des Rivieres, and D. Bobrow (1991). *The Art of the Metaobject Protocol*. Cambridge, Massachusetts: MIT Press.

Masuhara, H., S. Matsuoka, T. Watanabe, and A. Yonezawa (1992, October). Object-oriented concurrent reflective languages can be implemented efficiently. In *Proceedings OOPSLA '92, ACM SIGPLAN Notices*, pp. 127–147. Published as Proceedings OOPSLA '92, ACM SIGPLAN Notices, volume 27, number 10.

McAffer, J. (1995a, September). *A Meta-level Architecture for Prototyping Distributed Object Systems*. Ph. D. thesis, Department of Information Science, University of Tokyo, Tokyo, Japan. Available by FTP at ftp://camille.is.s.u-tokyo.ac.jp/pub/members/jeff/docs/phd.thesis.ps.gz.

McAffer, J. (1995b, August). Meta-level architecture support for distributed objects. In *Proceed-*

*ings of the International Workshop on Object-Orientation in Operating Systems (IWOOOS)*, pp. 232–241. IEEE TCOS.

McAffer, J. (1995c, August). Meta-level programming with CodA. In *Proceedings of the European Conference on Object-Oriented Computing (ECOOP)*, LNCS 952, pp. 190–214. Springer Verlag.

OI (1994). The open implementations workshop proposal and responses. Available on internet at: http://www.parc.xerox.com/OI/.

Shapiro, M., Y. Gourhant, S. Habert, L. Mosseri, M. Ruffin, and C. Valot (1989, Fall). SOS: An object-oriented operating system – Assesment and perspectives. *Computer Systems 2*(4), 287–337.